Network Working Group
Internet-Draft
Intended status: Informational
Expires: 5 January 2026

C. Aguado
Amazon
M. Griswold
FullCtl
J. Ramseyer
Meta
A. Servin
Google
T. Strickx
Cloudflare
Q Misell
AS207960 Cyfyngedig

4 July 2025

Peering API
draft-ietf-grow-peering-api-01

Abstract

We propose an API standard for BGP Peering, also known as interdomain interconnection through global Internet Routing. This API offers a standard way to request public (settlement-free) peering, verify the status of a request or BGP session, and list potential connection locations. The API is backed by PeeringDB OIDC, the industry standard for peering authentication. We also propose future work to cover private peering, and alternative authentication methods.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://bgp.github.io/draft-ietf-peering-api/draft-peering-api-ramseyer-protocol.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-grow-peering-api/.

Source for this draft and an issue tracker can be found at https://github.com/bgp/draft-ietf-peering-api.

Status of This Memo

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

This Internet-Draft will expire on 5 January 2026.

Copyright Notice

Table of Contents

1. Introduction

The Peering API is a mechanism that allows networks to automate interdomain interconnection between two Autonomous Systems (AS) through the Border Gateway Protocol 4 ([RFC4271]). Using the API, networks will be able to automatically request and accept peering interconnections between Autonomous Systems in public or private scenarios in a time faster than it would take to configure sessions manually. By speeding up the peering turn-up process and removing the need for manual involvement in peering, the API and automation will ensure that networks can get interconnected as fast, reliably, cost-effectively, and efficiently as possible. As a result, this improves end-user performance for all applications using networks interconnection supporting the Peering API.

1.1. Business Justification

By using the Peering API, entities requesting and accepting peering can significantly improve the process to turn up interconnections by:

* Reducing in person-hours spent configuring peering
* Reducing configuration mistakes by reducing human interaction
* And by peering, reducing network latency through expansion of interconnection relationships

2. Conventions and Terminology

All terms used in this document will be defined here:

Initiator Network that wants to peer

Receiver Network that is receiving communications about peering

Configured peering session that is set up on one side

Established session is already defined as per BGP-4 specification Section 8.2.2 of [RFC4271]

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. These words may also appear in this document in lower case as plain English words, absent their normative meanings.

3. Audience

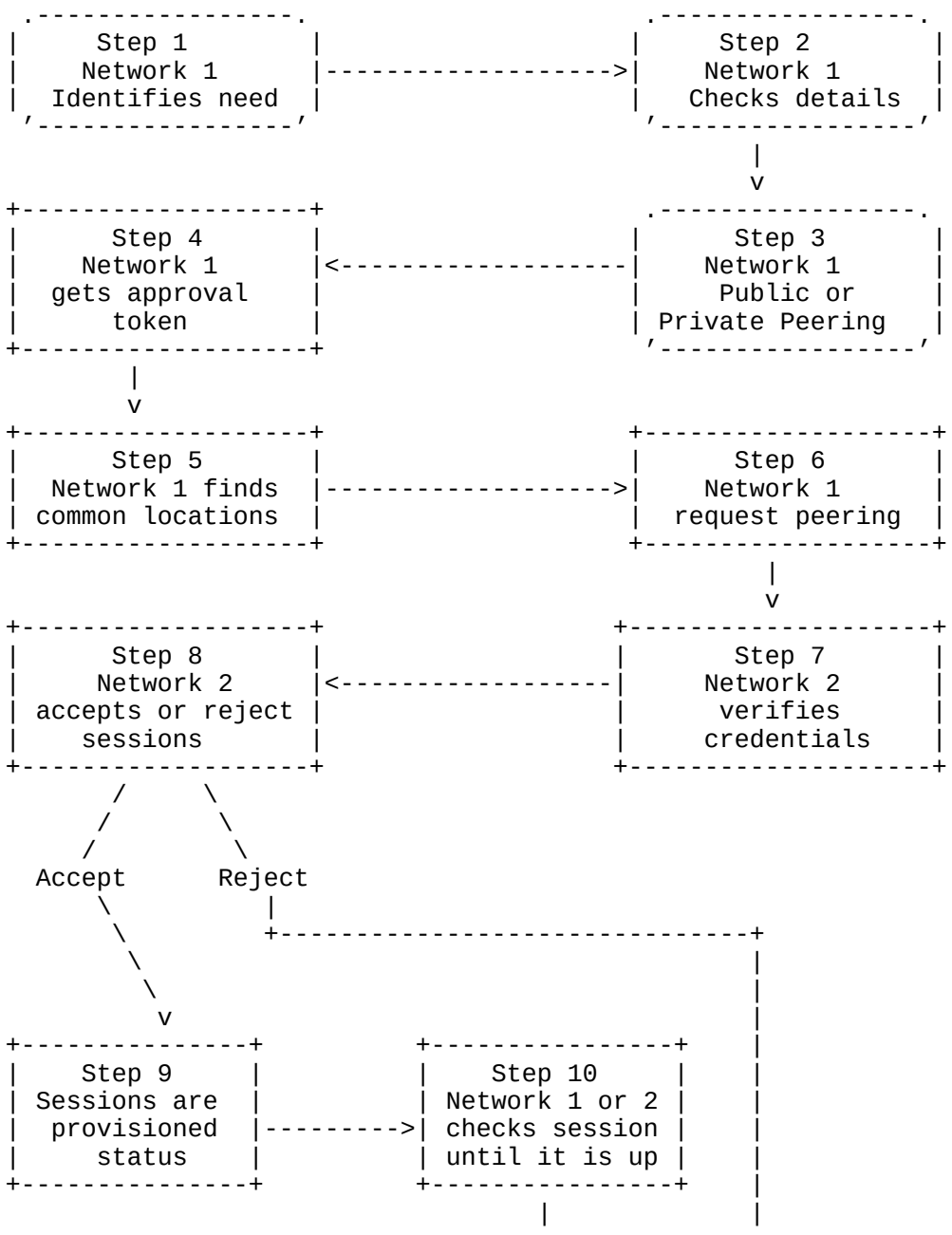
The Peering API aims to simplify peering interconnection configuration. To that end, the API can be called by either a human or some automation. A network engineer can submit API requests through a client-side tool, and configure sessions by hand or through existing tooling. Alternately, an automated service can request BGP sessions through some trigger or regularly scheduled request (for example, upon joining a new peering location, or through regular polling of potential peers). That automated client can then configure the client sessions through its own tooling. For ease of exchanging peering requests, the authors suggest peers to maintain both a client and a server for the API. Toward the goal of streamlining peering configuration, the authors encourage peers to automate their network configuration wherever possible, but do not require full automation to use this API.

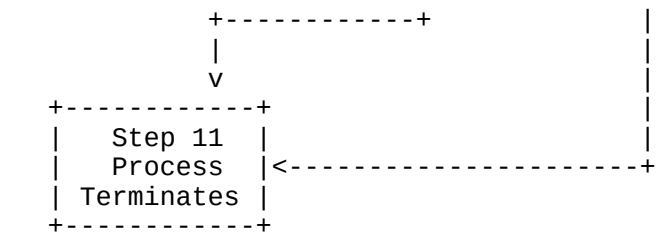
4. Protocol

The Peering API follows the Representational State Transfer ([BCP56]) architecture where sessions, locations, and maintenance events are the resources the API represents and is modeled after the OpenAPI standard [openapi]. Using the token bearer model ([RFC6750]), a client application can request to add or remove peering sessions, list potential interconnection locations, and query for upcoming maintenance events on behalf of the AS resource owner.

4.1. Example Peering Request Negotiation

Diagram of the Peering Request Process





* Step 1 (Human): Network 1 identifies that it would be useful to peer with Network 2 to interchange traffic more optimally.

* Step 2 (Human): Network 1 checks technical and other peering details about Network 2 to check if peering is possible.

* Step 3 (Human): Network 1 decides in type (Public or PNI) of peering and facility.

* Step 4 (API): Network 1 gets approval/token that is authorized to ‘speak’ on behalf of Network 1’s ASN.

* Step 5 (API): Network 1 checks PeeringDB for common places between Network 1 and Network 2; GET /locations

* Step 6 (API): Network 1 request peering with Network 2; POST /add_sessions

* Step 7 (API): Network 2 verifies Network 1 credentials, check requirements for peering.

* Step 8 (API): Network 2 accepts or rejects session(s). API Server gives yes/no for request.

* Step 9 (API): If yes, sessions are provisioned, Networks 1 or Network 2 can check status; API: GET /sessions

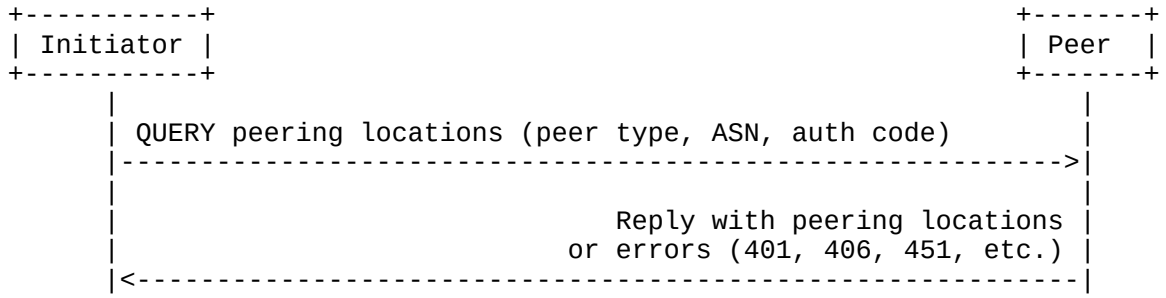
* Step 10 (API): API keeps polling until sessions are up

* Step 11 (API): Process Terminates

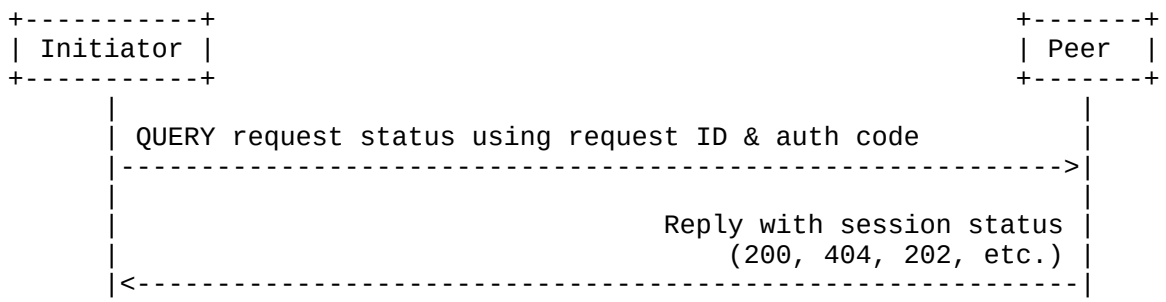
4.2. Example API Flow

The diagram below outlines the proposed API flow.

4.2.1. List Locations



4.2.2. Request session status



4.2.3. REQUEST

1. ADD SESSION (CLIENT BATCHED REQUEST)

* The initiator's client provides a set of the following information, where local always refers to the receiver and peer always refers to the initiator:

  - Structure:

    1. Local ASN
    2. Local IP
    3. Peer ASN
    4. Peer IP
    5. Local BGP Role according to [RFC9234]
    6. Peer BGP Role according to [RFC9234]
    7. Local insert ASN (optional to support route servers)

8. Peer insert ASN (optional to support route servers)

9. Local monitoring session (optional to support monitoring systems)

10. Peer monitoring session (optional to support monitoring systems)

11. Peer Type (public or private)

12. Session Secret (optional with encoding agreed outside of this specification)

13. Location (Commonly agreed identifier of the BGP speaker, e.g. PeeringDB IX lan ID)

* The receiver’s expected actions:

  - The server confirms requested clientASN in list of authorized ASNs.

  - Optional: checks traffic levels, prefix limit counters, other desired internal checks. 2. ADD SESSIONS (SERVER BATCHED RESPONSE)

* APPROVAL CASE

  - Server returns a list with the structure for each of the acceptable peering sessions. Note: this structure may also contain additional attributes such as the server generated session ID.

* PARTIAL APPROVAL CASE

  - Server returns a list with the structure for each of the acceptable peering sessions as in the approval case. The server also returns a list of sessions that have not deemed as validated or acceptable to be created. The set of sessions accepted and rejected is disjoint and the join of both sets matches the cardinality of the requested sessions.

* REJECTION CASE

  - Server returns an error message which indicates that all of the sessions requested have been rejected and the reason for it.

4.2.4. CLIENT CONFIGURATION

The client then configures the chosen peering sessions asynchronously using their internal mechanisms. The client SHOULD pull and use additional information on the new peering from public sources as required to ensure routing security, e.g., AS-SETs to configure appropriate filters. For every session that the server rejected, the client removes that session from the list to be configured.

4.2.5. SERVER CONFIGURATION

The server configures all sessions that are in its list of approved peering sessions from its reply to the client. The server SHOULD pull and use additional information on the new peering from public sources to ensure routing security, e.g., AS-SETs to configure appropriate filters.

4.2.6. MONITORING

Both client and server wait for sessions to establish. At any point, client may send a "GET STATUS" request to the server, to request the status of the session (by session ID). The client will send a structure along with the request, as follows:

* structure (where local refers to the server and peer refers to the client):
  - Session ID
  - Local ASN
  - Local IP
  - Peer ASN
  - Peer IP
  - Local BGP Role ([RFC9234])
  - Peer BGP Role ([RFC9234])
  - Local insert ASN (optional, as defined above)
  - Peer insert ASN (optional, as defined above)
  - Local monitoring session (optional, as defined above)
  - Peer monitoring session (optional, as defined above)

- Peer Type
- Session secret (optional, as defined above)
- Location
- Status

The server then responds with the same structure, with the information that it understands (status, etc).

4.2.7. COMPLETION

If both sides report that the session is established, then peering is complete. If one side does not configure sessions within the server's acceptable configuration window (TimeWindow), then the server is entitled to remove the configured sessions and report "Unestablished" to the client.

5. Authentication

Authentication to this API is performed with Bearer tokens. Two methods are defined below to obtain a bearer token, the choice of which is supported is up to operator preference however the authors recommend the use of RPKI Attested OAuth over reliance on an external OIDC provider.

5.1. OAuth with RPKI Attested OAuth

It is envisioned the primary authentication mechanism for this API will be with RPKI Attested OAuth as defined in [draft-blahaj-grow-rpki-oauth].

When a client wishes to authenticate to a Peering API server using RPKI Attested OAuth, it first makes a POST request to the server's /oauth/client_register endpoint, containing the following JSON body:

```
{
  "idp_base": "https://example.com/idp/"
}
```

The idp_base URL provided MUST host an OAuth Discovery document as per [draft-blahaj-grow-rpki-oauth].

The Peering API server registers with the well-known scope urn:ietf:params:oauth:scope:peering-api.

After successfully registering, the Peering API server stores its OAuth Client ID and Client Secret, along with the ASNs that this IdP is authoritative for. Future Bearer tokens can be checked against this list for which ASNs they are allowed to submit requests on behalf of.

The Peering API server then returns the following JSON body to the client:

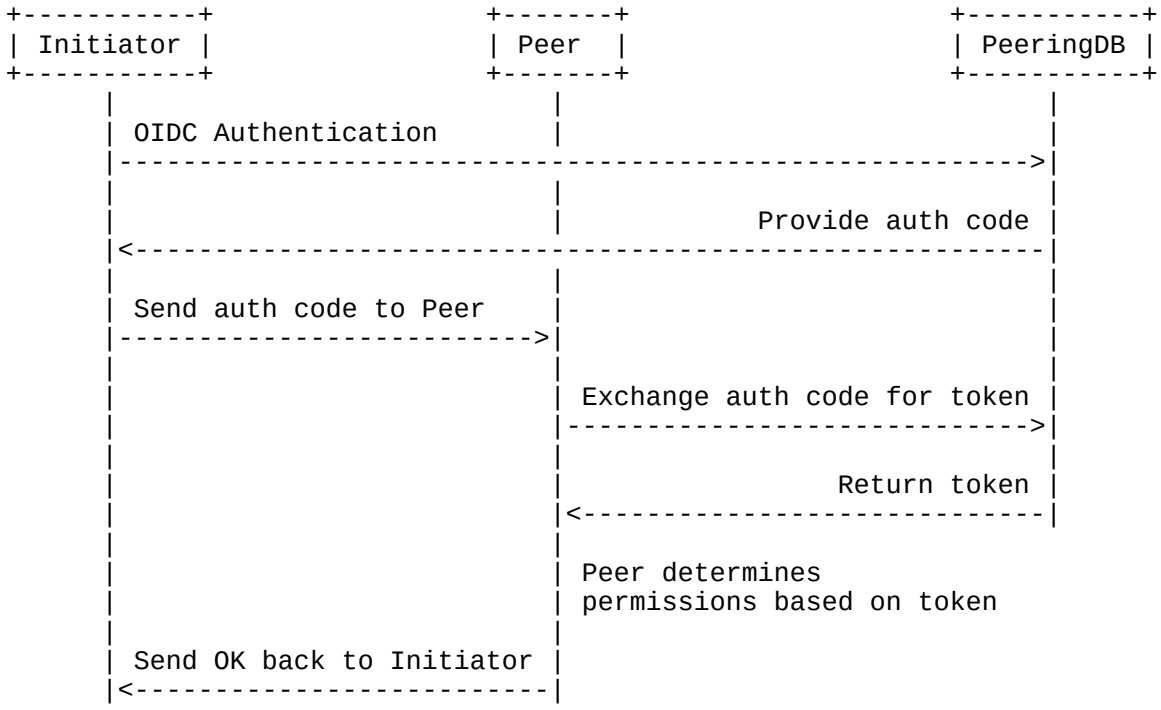
```
{
  "client_id": "example-id"
}
```

A client may then request that its IdP perform an OAuth Token Exchange [RFC8693] towards this Client ID, and use the resulting Bearer token to authenticate to the Peering API server.

5.2. OAuth with Peering DB

As an alternative, this document allows, but does not recommend, the use of PeeringDB OAuth.

First, the initiating OAuth2 Client is also the Resource Owner (RO) so it can follow the OAuth2 client credentials grant Section 4.4 of [RFC6749]. In this example, the client will use PeeringDB OIDC credentials to acquire a JWT access token that is scoped for use with the receiving API. On successful authentication, PeeringDB provides the Resource Server (RS) with the client’s email (for potential manual discussion), along with the client’s usage entitlements (known as OAuth2 scopes), to confirm the client is permitted to make API requests on behalf of the initiating AS.



6. API Endpoints and Specifications

Each peer needs a public API endpoint that will implement the API protocol. This API should be publicly listed in peeringDB and also as a potential expansion of [RFC9092] which could provide endpoint integration to WHOIS ([RFC3912]). Each API endpoint should be fuzz-tested and protected against abuse. Attackers should not be able to access internal systems using the API. Every single request should come in with a unique GUID called RequestID that maps to a peering request for later reference. This GUID format should be standardized across all requests. This GUID should be provided by the receiver once it receives the request and must be embedded in all communication. If there is no RequestID present then that should be interpreted as a new request and the process starts again. An email address is needed for communication if the API fails or is not implemented properly (can be obtained through PeeringDB).

For a programmatic specification of the API, please see the public Github ([autopeer]).

This initial draft fully specifies the Public Peering endpoints. Private Peering and Maintenance are under discussion, and the authors invite collaboration and discussion from interested parties.

6.1. DATA TYPES

Please see specification ([autopeer]) for OpenAPI format.

Peering Location

Contains string field listing the desired peering location in format pdb:ix:$IX_ID, and an enum specifying peering type (public or private).

Session Status

Status of BGP Session, both as connection status and approval status (Established, Pending, Approved, Rejected, Down, Unestablished, etc)

Session Array

Array of potential BGP sessions, with request UUID. Request UUID is optional for client, and required for server. Return URL is optional, and indicates the client's Peering API endpoint. The client's return URL is used by the server to request additional sessions. Client may provide initial UUID for client-side tracking, but the server UUID will be the final definitive ID. RequestID will not change across the request.

BGP Session

A structure that describes a BGP session and contains the following elements:

* local_asn (ASN of requestor)
* local_ip (IP of requestor, v4 or v6)
* peer_asn (server ASN)
* peer_ip (server-side IP)
* local_bgp_role (BGP role according to [RFC9234])
* peer_bgp_role (BGP role according to [RFC9234])
* local_insert_asn (optional, to support route servers, defaults to true)
* peer_insert_asn (optional, to support route servers, defaults to true)

* local_monitoring_session (optional, to support monitoring systems, defaults to false)

* peer_monitoring_session (optional, to support monitoring systems, defaults to false)

* peer_type (public or private)

* session_secret (optional, as defined above)

* location (Peering Location, as defined above)

* status (Session Status, as defined above)

* session_id (of individual session and generated by the server)

As not all elements are reflected in the [autopeer] OpenAPI definition to date, we define the missing fields here to be reflected in [autopeer] in the future.

* local_bgp_role and peer_bgp_role: these field describe the BGP roles of the local and peer side of the session according to [RFC9234] represented by an integer. The roles for both sides MUST be set in a way that does not violate role correctness as defined in Section 4.2 of [RFC9234].

* local_insert_asn and peer_insert_asn: these fields define whether the local or peer side will insert their ASN into the AS path attribute of forwarded BGP routes. They are mostly relevant to route servers. The fields are boolean and optional. If not provided, they default to true.

* local_monitoring_session and peer_monitoring_session: these fields define whether the local or peer side of the session will forward routes to other ASes or not. As the role of monitoring systems is not defined in [RFC9234], we add this role via a boolean, optional flag. If not provided, they default to false. local_monitoring_session and peer_monitoring_sessions MUST NOT be true at the same time for the same session to avoid a role mismatch.

Error

API Errors, for field validation errors in requests, and request-level errors.

The above is sourced largely from the linked OpenAPI specification.

6.2. Discovery

This document does not specify how to discover an ASN's Peering API endpoint. Some possible options are:

* Via the RPKI using [rpki-discovery]
* Publication in WHOIS/RDAP
* Manual configuration

The preferred method is via the RPKI.

6.3. Endpoints

(As defined in [autopeer]). On each call, there should be rate limits, allowed senders, and other optional restrictions.

6.3.1. Public Peering over an Internet Exchange (IX)

* /sessions: ADD/RETRIEVE sessions visible to the calling PEER
  - Batch create new session resources
    - Establish new BGP sessions between peers, at the desired exchange.
    - Below is based on OpenAPI specification: [autopeer].
    - POST /sessions
      + Request body: Session Array
      + Responses:
        * 200 OK:
          - Contents: Session Array (all sessions in request accepted for configuration). Should not all the sessions be accepted, the response also contains a list of sessions and the respective errors.
        * 400:
          - Error
        * 403:

- Unauthorized to perform the operation

* 422:

- Please contact us, human intervention required

- List all session resources. The response is paginated.

  - Given a request ID, query for the status of that request.

  - Given an ASN without request ID, query for status of all connections between client and server.

  - Below is based on OpenAPI specification: [autopeer].

  - GET /sessions

    + Request parameters:

      * asn (requesting client’s asn)

      * request_id (optional, UUID of request)

      * max_results (integer to indicate an upper bound for a given response page)

      * next_token (opaque and optional string received on a previous response page and which allows the server to produce the next page of results. Its absence indicates to the server that the first page is expected)

    + Response:

      * 200: OK

        - Contents: Session Array of sessions in request_id, if provided. Else, all existing and in-progress sessions between client ASN and server.

          o next_token (opaque and optional string the server expects to be passed back on the request for the next page. Its absence indicates to the client that no more pages are available)

      * 400:

        - Error (example: request_id is invalid)

            * 403:

                - Unauthorized to perform the operation

* /sessions/{session_id}: Operate on individual sessions

    - Retrieve an existing session resource

        o Below is based on OpenAPI specification: [autopeer].

        o GET /sessions/{session_id}

            + Request parameters:

                * session_id returned by the server on creation or through the session list operation.

            + Responses:

                * 200 OK:

                    - Contents: Session structure with current attributes

                * 400:

                    - Error (example: session_id is invalid)

                * 403:

                    - Unauthorized to perform the operation

                * 404:

                    - The session referred by the specified session_id does not exist or is not visible to the caller

    - Delete a session.

        o Given a session ID, delete it which effectively triggers an depeering from the initiator.

        o Below is based on OpenAPI specification: [autopeer].

        o DELETE /sessions/{session_id}

            + Request parameters:

        * session_id returned by the server on creation or through the session list operation.

      + Response:

        * 204: OK

          - Contents: empty response as the session is processed and hard deleted

        * 400:

          - Error (example: session_id is invalid)

        * 403:

          - Unauthorized to perform the operation

        * 404:

          - The session referred by the specified session_id does not exist or is not visible to the caller

        * 422:

          - Please contact us, human intervention required

6.3.2. UTILITY API CALLS

Endpoints which provide useful information for potential interconnections.

* /locations: LIST POTENTIAL PEERING LOCATIONS

  - List potential peering locations, both public and private. The response is paginated.

    o Below is based on OpenAPI specification: [autopeer].

    o GET /locations

      + Request parameters:

        * asn (Server ASN, with which to list potential connections)

        * location_type (Optional: Peering Location)

    * max_results (integer to indicate an upper bound for a given response page)

    * next_token (opaque and optional string received on a previous response page and which allows the server to produce the next page of results. Its absence indicates to the server that the first page is expected)

  + Response:

    * 200: OK

      - Contents: List of Peering Locations.

        o next_token (opaque and optional string the server expects to be passed back on the request for the next page. Its absence indicates to the client that no more pages are available)

    * 400:

      - Error

    * 403:

      - Unauthorized to perform the operation

6.3.3. Private Peering (DRAFT)

* ADD/AUGMENT PNI

* Parameters:

  - Peer ASN

  - Facility

  - email address (contact)

  - Action type: add/augment

  - LAG struct:

    o IPv4

    o IPv6

    - o Circuit ID

  - Who provides LOA? (and where to provide it).

* Response:

  - 200:

    - o LAG struct, with server data populated

    - o LOA or way to receive it

    - o Request ID

  - 40x: rejections

* REMOVE PNI

  - As ADD/AUGMENT in parameters. Responses will include a requestID and status.

7. Public Peering Session Negotiation

As part of public peering configuration, this draft must consider how the client and server should handshake at which sessions to configure peering. At first, a client will request sessions A, B, and C. The server may choose to accept all sessions A, B, and C. At this point, configuration proceeds as normal. However, the server may choose to reject session B. At that point, the server will reply back with A and C marked as "Accepted," and B as "Rejected." The server will then configure A and C, and wait for the client to configure A and C. If the client configured B as well, it will not come up.

This draft encourages peers to set up garbage collection for unconfigured or down peering sessions, to remove stale configuration and maintain good router hygiene.

Related to rejection, if the server would like to configure additional sessions with the client, the server may either reject all the session that do not meet the criteria caused by such absence in the client’s request or approve the client’s request and issue a separate request to the client’s server requesting those additional peering sessions D and E. The server will configure D and E on their side, and D and E will become part of the sessions requested in the UUID. The client may choose whether or not to accept those additional sessions. If they do, the client should configure D and E as well. If they do not, the client will not configure D and E, and the server should garbage-collect those pending sessions.

As part of the IETF discussion, the authors would like to discuss how to coordinate which side unfilters first. Perhaps this information could be conveyed over a preferences vector.

8. Private Peering

Through future discussion with the IETF, the specification for private peering will be solidified. Of interest for discussion includes Letter of Authorization (LOA) negotiation, and how to coordinate unfiltering and configuration checks.

9. Maintenance

This draft does not want to invent a new ticketing system. However, there is an opportunity in this API to provide maintenance notifications to peering partners. If there is interest, this draft would extend to propose a maintenance endpoint, where the server could broadcast upcoming and current maintenance windows.

A maintenance message would follow a format like:

* Title: string
* Start Date: date maintenance start(s/ed): UTC
* End Date: date maintenance ends: UTC
* Area: string or enum
* Details: freeform string

The "Area" field could be a freeform string, or could be a parseable ENUM, like (BGP, PublicPeering, PrivatePeering, Configuration, Caching, DNS, etc).

Past maintenances will not be advertised.

10. Security Considerations

This document describes a mechanism to standardize the discovery, creation and maintenance of peering relationships across autonomous systems (AS) using an out-of-band application programming interface (API). With it, AS operators take a step to operationalize their peering policy with new and existing peers in ways that improve or completely replace manual business validations, ultimately leading to the automation of the interconnection. However, this improvement can only be fully materialized when operators are certain that such API follows the operational trust and threat models they are comfortable

with, some of which are documented in BGP operations and security best practices ([RFC7454]). To that extent, this document assumes the peering API will be deployed following a strategy of defense in-depth and proposes the following common baseline threat model below.

10.1. Threats

Each of the following threats assume a scenario where an arbitrary actor is capable of reaching the peering API instance of a given operator, the client and the operator follow their own endpoint security and maintenance practices, and the trust anchors in use are already established following guidelines outside of the scope of this document.

* T1: A malicious actor with physical access to the IX fabric and peering API of the receiver can use ASN or IP address information to impersonate a different IX member to discover, create, update or delete peering information which leads to loss of authenticity, confidentiality, and authorization of the spoofed IX member.

* T2: A malicious actor with physical access to the IX fabric can expose a peering API for an IX member different of its own to accept requests on behalf of such third party and supplant it, leading to a loss of authenticity, integrity, non-repudiability, and confidentiality between IX members.

* T3: A malicious actor without physical access to the IX fabric but with access the the peering API can use any ASN to impersonate any autonomous system and overload the receiver’s peering API internal validations leading to a denial of service.

10.2. Mitigations

The following list of mitigations address different parts of the threats identified above:

* M1: Authorization controls - A initiator using a client application is authorized using the claims presented in the request prior to any interaction with the peering API (addresses T1, T2).

* M2: Proof of holdership - The initiator of a request through a client can prove their holdership of an Internet Number Resource (addresses T1, T3).

* M3: Request integrity and proof of possession - The peering API can verify HTTP requests signed with a key that is cryptographically bound to the authorized initiator (addresses T1, T2).

The Peering API does not enforce any kind of peering policy on the incoming requests. It is left to the peering API instance implementation to enforce the AS-specific peering policy. This document encourages each peer to consider the needs of their peering policy and implement request validation as desired.

10.3. Authorization controls

The peering API instance receives HTTP requests from a client application from a peering initiator. Those requests can be authorized using the authorization model based on OAuth 2.0 ([RFC6749]) with the OpenID Connect [oidc] core attribute set. The choice of OpenID Connect is to use a standardized and widely adopted authorization exchange format based on JSON Web Tokens ([RFC7519]) which allows interoperation with existing web-based application flows. JWT tokens also supply sufficient claims to implement receiver-side authorization decisions by third parties when used as bearer access tokens ([RFC9068]). The peering API instance (a resource server in OAuth2 terms) should follow the bearer token usage ([RFC6750]) which describes the format and validation of an access token obtained from the Oauth 2.0 Authorization Server. The resource server should follow the best practices for JWT access validation ([RFC8725]) and in particular verify that the access token is constrained to the resource server via the audience claim. Upon successful access token validation, the resource server should decide whether to proceed with the request based on the presence of expected and matching claims in the access token or reject it altogether. The core identity and authorization claims present in the access token may be augmented with specific claims vended by the Authorization Service. This document proposes to use PeeringDB’s access token claims as a baseline to use for authorization, however the specific matching of those claims to an authorization business decision is specific to each operator and outside of this specification. Resource servers may also use the claims in the access token to present the callers’ identity to the application and for auditing purposes.

10.4. Proof of holdership

The peering API defined in this document uses ASNs as primary identifiers to identify each party on a peering session besides other resources such as IP addresses. ASNs are explicitly expected in some API payloads but are also implicitly expected when making authorization business decisions such as listing resources that belong to an operator. Given that ASNs are Internet Number Resources assigned by RIRs and that the OAuth2 Authorization Server in use may not be operated by any of those RIRs, as it is the case of PeeringDB or any other commercial OAuth2 service, JWT claims that contain an ASN need be proved to be legitimately used by the initiator. This document proposes to attest ASN resource holdership using a mechanism based on RPKI ([RFC6480]) and in particular with the use of RPKI Attested OAuth ([draft-blahaj-grow-rpki-oauth]).

10.5. Request integrity and proof of possession

The API described in this document follows REST ([rest]) principles over an HTTP channel to model the transfer of requests and responses between peers. Implementations of this API should use the best common practices for the API transport ([RFC9325]) such as TLS. However, even in the presence of a TLS channel with OAuth2 bearer tokens alone, neither the client application nor the API can guarantee the end-to-end integrity of the message request or the authenticity of its content. One mechanism to add cryptographic integrity and authenticity validation can be the use a mutual authentication scheme to negotiate the parameters of the TLS channel. This requires the use of a web PKI ([RFC5280]) to carry claims for use in authorization controls, to bind such PKI to ASNs for proof of holdership, and the use of client certificates on the application.

Instead, this document proposes to address the message integrity property by cryptographically signing the parameters of the request with a key pair that creates a HTTP message signature to be included in the request ([RFC9421]). The client application controls the lifecycle of this key pair. The authenticity property of the messages signed with such key pair is addressed by binding the public key of the pair to the JWT access token in one of its claims of the access token using a mechanism that demonstrates proof of possession of the private key [RFC9449]. With these two mechanisms, the resource server should authenticate, authorize, and validate the integrity of the request using a JWT access token that can rightfully claim to represent a given ASN.

11. IANA Considerations

11.1. OAuth URI

This document adds one new entry to the "OAuth URI" registry:

URN	Common Name	Reference
urn:ietf:params:oauth:scope:peering-api	Well-known scope for the BGP Peering API	This document

Table 1

12. References

12.1. Normative References

[autopeer] "Github repository with the API specification and diagrams", n.d., <https://github.com/bgp/autopeer/>.

[draft-blahaj-grow-rpki-oauth] "Attesting the Identities of Parties in OpenID Connect (OIDC) using the Resource Public Key Infrastructure (RPKI)", n.d., <https://datatracker.ietf.org/doc/draft-blahaj-grow-rpki-oauth/>.

[oidc] "OpenID.Core", n.d., <https://openid.net/specs/openid-connect-core-1_0.html>.

[openapi] "OpenAPI-v3.1.0", n.d., <https://spec.openapis.org/oas/v3.1.0>.

[rest] Fielding, R. T., "Architectural Styles and the Design of Network-based Software Architectures", 2000, <http://roy.gbiv.com/pubs/dissertation/top.htm>.

[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/rfc/rfc2119>.

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

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/rfc/rfc6750>.

[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/rfc/rfc7519>.

[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/rfc/rfc8174>.

[RFC8693] Jones, M., Nadalin, A., Campbell, B., Ed., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", RFC 8693, DOI 10.17487/RFC8693, January 2020, <https://www.rfc-editor.org/rfc/rfc8693>.

[RFC8725] Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", BCP 225, RFC 8725, DOI 10.17487/RFC8725, February 2020, <https://www.rfc-editor.org/rfc/rfc8725>.

[RFC9068] Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens", RFC 9068, DOI 10.17487/RFC9068, October 2021, <https://www.rfc-editor.org/rfc/rfc9068>.

[RFC9421] Backman, A., Ed., Richer, J., Ed., and M. Sporny, "HTTP Message Signatures", RFC 9421, DOI 10.17487/RFC9421, February 2024, <https://www.rfc-editor.org/rfc/rfc9421>.

[RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", RFC 9449, DOI 10.17487/RFC9449, September 2023, <https://www.rfc-editor.org/rfc/rfc9449>.

12.2. Informative References

[BCP56] Best Current Practice 56, <https://www.rfc-editor.org/info/bcp56>. At the time of writing, this BCP comprises the following:

Nottingham, M., "Building Protocols with HTTP", BCP 56, RFC 9205, DOI 10.17487/RFC9205, June 2022, <https://www.rfc-editor.org/info/rfc9205>.

[RFC3912] Daigle, L., "WHOIS Protocol Specification", RFC 3912, DOI 10.17487/RFC3912, September 2004, <https://www.rfc-editor.org/rfc/rfc3912>.

[RFC4271] Rekhter, Y., Ed., Li, T., Ed., and S. Hares, Ed., "A Border Gateway Protocol 4 (BGP-4)", RFC 4271, DOI 10.17487/RFC4271, January 2006, <https://www.rfc-editor.org/rfc/rfc4271>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/rfc/rfc5280>.

[RFC6480] Lepinski, M. and S. Kent, "An Infrastructure to Support Secure Internet Routing", RFC 6480, DOI 10.17487/RFC6480, February 2012, <https://www.rfc-editor.org/rfc/rfc6480>.

[RFC7454] Durand, J., Pepelnjak, I., and G. Doering, "BGP Operations and Security", BCP 194, RFC 7454, DOI 10.17487/RFC7454, February 2015, <https://www.rfc-editor.org/rfc/rfc7454>.

[RFC9092] Bush, R., Candela, M., Kumari, W., and R. Housley, "Finding and Using Geofeed Data", RFC 9092, DOI 10.17487/RFC9092, July 2021, <https://www.rfc-editor.org/rfc/rfc9092>.

[RFC9234] Azimov, A., Bogomazov, E., Bush, R., Patel, K., and K. Sriram, "Route Leak Prevention and Detection Using Roles in UPDATE and OPEN Messages", RFC 9234, DOI 10.17487/RFC9234, May 2022, <https://www.rfc-editor.org/rfc/rfc9234>.

[RFC9325] Sheffer, Y., Saint-Andre, P., and T. Fossati, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 9325, DOI 10.17487/RFC9325, November 2022, <https://www.rfc-editor.org/rfc/rfc9325>.

[rpki-discovery] "A Profile for Peering API Discovery via the RPKI", n.d., <https://datatracker.ietf.org/doc/draft-misell-grow-rpki-peering-api-discovery/>.

Appendix A. Acknowledgments

The authors would like to thank their collaborators, who implemented API versions and provided valuable feedback on the design.

* Ben Blaustein
* Jakub Heichman (Meta)
* Prithvi Nath Manikonda (Amazon)
* Q Misell (Glauca Digital)
* Stefan Pratter (20C)
* Aaron Rose (Amazon)
* Ben Ryall (Meta)
* Erica Salvaneschi (Cloudflare)
* Job Snijders
* David Tuber (Cloudflare)
* Matthias Wichtlhuber (DE-CIX)

Authors' Addresses

Carlos Aguado
Amazon
Email: crlsa@amazon.com

Matt Griswold
FullCtl
Email: grizz@20c.com

Jenny Ramseyer
Meta
Email: ramseyer@meta.com

Arturo Servin
Google
Email: arturolev@google.com

Tom Strickx
Cloudflare
Email: tstrickx@cloudflare.com

Q Misell
AS207960 Cyfyngedig
Email: q@as207960.net