Stream: Internet Engineering Task Force (IETF)
RFC: 9635
Category: Standards Track
Published: September 2024
ISSN: 2070-1721
Authors: J. Richer, Ed. *Bespoke Engineering*
F. Imbault *acert.io*

RFC 9635
Grant Negotiation and Authorization Protocol (GNAP)

Abstract

The Grant Negotiation and Authorization Protocol (GNAP) defines a mechanism for delegating authorization to a piece of software and conveying the results and artifacts of that delegation to the software. This delegation can include access to a set of APIs as well as subject information passed directly to the software.

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/rfc9635.

Copyright Notice

Table of Contents

1. Introduction 9
1.1. Terminology 9
1.2. Roles 10
1.3. Elements 13
1.4. Trust Relationships 13
1.5. Protocol Flow 15
1.6. Sequences 17
1.6.1. Overall Protocol Sequence 18
1.6.2. Redirect-Based Interaction 20
1.6.3. User Code Interaction 22
1.6.4. Asynchronous Authorization 24
1.6.5. Software-Only Authorization 26
1.6.6. Refreshing an Expired Access Token 27
1.6.7. Requesting Subject Information Only 28
1.6.8. Cross-User Authentication 29
2. Requesting Access 31
2.1. Requesting Access to Resources 33
2.1.1. Requesting a Single Access Token 33
2.1.2. Requesting Multiple Access Tokens 35
2.2. Requesting Subject Information 37
2.3. Identifying the Client Instance 37
2.3.1. Identifying the Client Instance by Reference 39
2.3.2. Providing Displayable Client Instance Information 40
2.3.3. Authenticating the Client Instance 40
2.4. Identifying the User 41
2.4.1. Identifying the User by Reference 42
2.5. Interacting with the User 42
2.5.1. Start Mode Definitions 44

2.5.2. Interaction Finish Methods 46
2.5.3. Hints 48
3. Grant Response 49
3.1. Request Continuation 51
3.2. Access Tokens 52
3.2.1. Single Access Token 52
3.2.2. Multiple Access Tokens 55
3.3. Interaction Modes 57
3.3.1. Redirection to an Arbitrary URI 58
3.3.2. Launch of an Application URI 58
3.3.3. Display of a Short User Code 59
3.3.4. Display of a Short User Code and URI 59
3.3.5. Interaction Finish 60
3.4. Returning Subject Information 61
3.4.1. Assertion Formats 63
3.5. Returning a Dynamically Bound Client Instance Identifier 63
3.6. Error Response 64
4. Determining Authorization and Consent 65
4.1. Starting Interaction with the End User 68
4.1.1. Interaction at a Redirected URI 69
4.1.2. Interaction at the Static User Code URI 69
4.1.3. Interaction at a Dynamic User Code URI 70
4.1.4. Interaction through an Application URI 71
4.2. Post-Interaction Completion 71
4.2.1. Completing Interaction with a Browser Redirect to the Callback URI 72
4.2.2. Completing Interaction with a Direct HTTP Request Callback 73
4.2.3. Calculating the Interaction Hash 74
5. Continuing a Grant Request 75
5.1. Continuing after a Completed Interaction 77
5.2. Continuing during Pending Interaction (Polling) 79

5.3. Modifying an Existing Request 80
5.4. Revoking a Grant Request 86
6. Token Management 86
6.1. Rotating the Access Token Value 87
6.1.1. Binding a New Key to the Rotated Access Token 88
6.2. Revoking the Access Token 89
7. Securing Requests from the Client Instance 90
7.1. Key Formats 90
7.1.1. Key References 92
7.1.2. Key Protection 92
7.2. Presenting Access Tokens 92
7.3. Proving Possession of a Key with a Request 93
7.3.1. HTTP Message Signatures 96
7.3.2. Mutual TLS 102
7.3.3. Detached JWS 104
7.3.4. Attached JWS 108
8. Resource Access Rights 112
8.1. Requesting Resources by Reference 115
9. Discovery 117
9.1. RS-First Method of AS Discovery 118
9.2. Dynamic Grant Endpoint Discovery 120
10. IANA Considerations 120
10.1. HTTP Authentication Scheme Registration 120
10.2. Media Type Registration 121
10.2.1. application/gnap-binding-jwsd 121
10.2.2. application/gnap-binding-jws 121
10.2.3. application/gnap-binding-rotation-jwsd 122
10.2.4. application/gnap-binding-rotation-jws 123

10.3. GNAP Grant Request Parameters 124
10.3.1. Registration Template 124
10.3.2. Initial Contents 125
10.4. GNAP Access Token Flags 125
10.4.1. Registration Template 125
10.4.2. Initial Contents 126
10.5. GNAP Subject Information Request Fields 126
10.5.1. Registration Template 126
10.5.2. Initial Contents 126
10.6. GNAP Assertion Formats 127
10.6.1. Registration Template 127
10.6.2. Initial Contents 127
10.7. GNAP Client Instance Fields 127
10.7.1. Registration Template 128
10.7.2. Initial Contents 128
10.8. GNAP Client Instance Display Fields 128
10.8.1. Registration Template 128
10.8.2. Initial Contents 129
10.9. GNAP Interaction Start Modes 129
10.9.1. Registration Template 129
10.9.2. Initial Contents 130
10.10. GNAP Interaction Finish Methods 130
10.10.1. Registration Template 130
10.10.2. Initial Contents 131
10.11. GNAP Interaction Hints 131
10.11.1. Registration Template 131
10.11.2. Initial Contents 131
10.12. GNAP Grant Response Parameters 132
10.12.1. Registration Template 132

10.12.2. Initial Contents 132
10.13. GNAP Interaction Mode Responses 133
10.13.1. Registration Template 133
10.13.2. Initial Contents 133
10.14. GNAP Subject Information Response Fields 134
10.14.1. Registration Template 134
10.14.2. Initial Contents 134
10.15. GNAP Error Codes 134
10.15.1. Registration Template 135
10.15.2. Initial Contents 135
10.16. GNAP Key Proofing Methods 136
10.16.1. Registration Template 136
10.16.2. Initial Contents 136
10.17. GNAP Key Formats 137
10.17.1. Registration Template 137
10.17.2. Initial Contents 137
10.18. GNAP Authorization Server Discovery Fields 137
10.18.1. Registration Template 138
10.18.2. Initial Contents 138
11. Security Considerations 138
11.1. TLS Protection in Transit 138
11.2. Signing Requests from the Client Software 139
11.3. MTLS Message Integrity 140
11.4. MTLS Deployment Patterns 141
11.5. Protection of Client Instance Key Material 141
11.6. Protection of Authorization Server 142
11.7. Symmetric and Asymmetric Client Instance Keys 143
11.8. Generation of Access Tokens 144
11.9. Bearer Access Tokens 144
11.10. Key-Bound Access Tokens 145

11.11. Exposure of End-User Credentials to Client Instance 146
11.12. Mixing Up Authorization Servers 146
11.13. Processing of Client-Presented User Information 147
11.14. Client Instance Pre-registration 148
11.15. Client Instance Impersonation 149
11.16. Client-Hosted Logo URI 149
11.17. Interception of Information in the Browser 150
11.18. Callback URI Manipulation 150
11.19. Redirection Status Codes 150
11.20. Interception of Responses from the AS 151
11.21. Key Distribution 151
11.22. Key Rotation Policy 152
11.23. Interaction Finish Modes and Polling 152
11.24. Session Management for Interaction Finish Methods 153
11.25. Calculating Interaction Hash 154
11.26. Storage of Information during Interaction and Continuation 156
11.27. Denial of Service (DoS) through Grant Continuation 156
11.28. Exhaustion of Random Value Space 157
11.29. Front-Channel URIs 157
11.30. Processing Assertions 158
11.31. Stolen Token Replay 159
11.32. Self-Contained Stateless Access Tokens 159
11.33. Network Problems and Token and Grant Management 160
11.34. Server-Side Request Forgery (SSRF) 160
11.35. Multiple Key Formats 161
11.36. Asynchronous Interactions 162
11.37. Compromised RS 163
11.38. AS-Provided Token Keys 163

12. Privacy Considerations 163
12.1. Surveillance 163
12.1.1. Surveillance by the Client 164
12.1.2. Surveillance by the Authorization Server 164
12.2. Stored Data 164
12.3. Intrusion 165
12.4. Correlation 165
12.4.1. Correlation by Clients 165
12.4.2. Correlation by Resource Servers 166
12.4.3. Correlation by Authorization Servers 166
12.5. Disclosure in Shared References 166
13. References 166
13.1. Normative References 166
13.2. Informative References 168
Appendix A. Comparison with OAuth 2.0 170
Appendix B. Example Protocol Flows 172
B.1. Redirect-Based User Interaction 172
B.2. Secondary Device Interaction 176
B.3. No User Involvement 178
B.4. Asynchronous Authorization 179
B.5. Applying OAuth 2.0 Scopes and Client IDs 182
Appendix C. Interoperability Profiles 183
C.1. Web-Based Redirection 184
C.2. Secondary Device 184
Appendix D. Guidance for Extensions 184
Appendix E. JSON Structures and Polymorphism 185
Acknowledgements 186
Authors' Addresses 187

1. Introduction

GNAP allows a piece of software, the client instance, to request delegated authorization to resource servers and subject information. The delegated access to the resource server can be used by the client instance to access resources and APIs on behalf a resource owner, and delegated access to subject information can in turn be used by the client instance to make authentication decisions. This delegation is facilitated by an authorization server, usually on behalf of a resource owner. The end user operating the software can interact with the authorization server to authenticate, provide consent, and authorize the request as a resource owner.

The process by which the delegation happens is known as a grant, and GNAP allows for the negotiation of the grant process over time by multiple parties acting in distinct roles.

This specification focuses on the portions of the delegation process facing the client instance. In particular, this specification defines interoperable methods for a client instance to request, negotiate, and receive access to information facilitated by the authorization server. This specification additionally defines methods for the client instance to access protected resources at a resource server. This specification also discusses discovery mechanisms that enable the client instance to configure itself dynamically. The means for an authorization server and resource server to interoperate are discussed in [GNAP-RS].

The focus of this protocol is to provide interoperability between the different parties acting in each role, not to specify implementation details of each. Where appropriate, GNAP may make recommendations about internal implementation details, but these recommendations are to ensure the security of the overall deployment rather than to be prescriptive in the implementation.

This protocol solves many of the same use cases as OAuth 2.0 [RFC6749], OpenID Connect [OIDC], and the family of protocols that have grown up around that ecosystem. However, GNAP is not an extension of OAuth 2.0 and is not intended to be directly compatible with OAuth 2.0. GNAP seeks to provide functionality and solve use cases that OAuth 2.0 cannot easily or cleanly address. Appendix A further details the protocol rationale compared to OAuth 2.0. GNAP and OAuth 2.0 will likely exist in parallel for many deployments, and considerations have been taken to facilitate the mapping and transition from existing OAuth 2.0 systems to GNAP. Some examples of these can be found in Appendix B.5.

1.1. 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 document contains non-normative examples of partial and complete HTTP messages, JSON structures, URIs, query components, keys, and other elements. Whenever possible, the document uses URI as a generic term, since it aligns with the recommendations in [RFC3986] and better matches the intent that the identifier may be reachable through various/generic means (compared to URLs). Some examples use a single trailing backslash (\) to indicate line wrapping for long values, as per [RFC8792]. The `\` character and leading spaces on wrapped lines are not part of the value.

This document uses the term "mutual TLS" as defined by [RFC8705]. The shortened form "MTLS" is used to mean the same thing.

For brevity, the term "signature" on its own is used in this document to refer to both digital signatures (which use asymmetric cryptography) and keyed Message Authentication Codes (MACs) (which use symmetric cryptography). Similarly, the verb "sign" refers to the generation of either a digital signature or a keyed MAC over a given signature base. The qualified term "digital signature" refers specifically to the output of an asymmetric cryptographic signing operation.

1.2. Roles

The parties in GNAP perform actions under different roles. Roles are defined by the actions taken and the expectations leveraged on the role by the overall protocol.

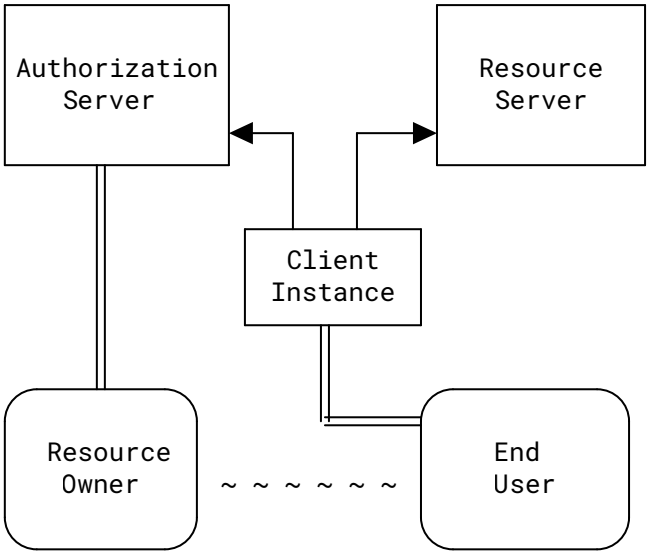

Figure 1: Roles in GNAP

Authorization Server (AS): Server that grants delegated privileges to a particular instance of client software in the form of access tokens or other information (such as subject information). The AS is uniquely defined by the grant endpoint URI, which is the absolute URI where grant requests are started by clients.

Client: Application that consumes resources from one or several resource servers, possibly requiring access privileges from one or several ASes. The client is operated by the end user, or it runs autonomously on behalf of a resource owner.

For example, a client can be a mobile application, a web application, a backend data processor, etc.

Note: This specification differentiates between a specific instance (the client instance, identified by its unique key) and the software running the instance (the client software). For some kinds of client software, there could be many instances of that software, each instance with a different key.

Resource Server (RS): Server that provides an API on protected resources, where operations on the API require a valid access token issued by a trusted AS.

Resource Owner (RO): Subject entity that may grant or deny operations on resources it has authority upon.

Note: The act of granting or denying an operation may be manual (i.e., through an interaction with a physical person) or automatic (i.e., through predefined organizational rules).

End user: Natural person that operates a client instance.

Note: That natural person may or may not be the same entity as the RO.

The design of GNAP does not assume any one deployment architecture but instead attempts to define roles that can be fulfilled in a number of different ways for different use cases. As long as a given role fulfills all of its obligations and behaviors as defined by the protocol, GNAP does not make additional requirements on its structure or setup.

Multiple roles can be fulfilled by the same party, and a given party can switch roles in different instances of the protocol. For example, in many instances, the RO and end user are the same person, where a user authorizes the client instance to act on their own behalf at the RS. In this case, one party fulfills the roles of both RO and end user, but the roles themselves are still defined separately from each other to allow for other use cases where they are fulfilled by different parties.

As another example, in some complex scenarios, an RS receiving requests from one client instance can act as a client instance for a downstream secondary RS in order to fulfill the original request. In this case, one piece of software is both an RS and a client instance from different perspectives, and it fulfills these roles separately as far as the overall protocol is concerned.

A single role need not be deployed as a monolithic service. For example, a client instance could have frontend components that are installed on the end user's device as well as a backend system that the frontend communicates with. If both of these components participate in the delegation protocol, they are both considered part of the client instance. If there are several copies of the client software that run separately but all share the same key material, such as a deployed cluster, then this cluster is considered a single client instance. In these cases, the distinct components of what is considered a GNAP client instance may use any number of different communication mechanisms between them, all of which would be considered an implementation detail of the client instances and out of scope of GNAP.

As another example, an AS could likewise be built out of many constituent components in a distributed architecture. The component that the client instance calls directly could be different from the component that the RO interacts with to drive consent, since API calls and user interaction have different security considerations in many environments. Furthermore, the AS could need to collect identity claims about the RO from one system that deals with user attributes while generating access tokens at another system that deals with security rights. From the perspective of GNAP, all of these are pieces of the AS and together fulfill the role of the AS as defined by the protocol. These pieces may have their own internal communications mechanisms, which are considered out of scope of GNAP.

1.3. Elements

In addition to the roles above, the protocol also involves several elements that are acted upon by the roles throughout the process.

Access Token: A data artifact representing a set of rights and/or attributes.

Note: An access token can be first issued to a client instance (requiring authorization by the RO) and subsequently rotated.

Grant: (verb): To permit an instance of client software to receive some attributes at a specific time and with a specific duration of validity and/or to exercise some set of delegated rights to access a protected resource.

(noun): The act of granting permission to a client instance.

Privilege: Right or attribute associated with a subject.

Note: The RO defines and maintains the rights and attributes associated to the protected resource and might temporarily delegate some set of those privileges to an end user. This process is referred to as "privilege delegation".

Protected Resource: Protected API that is served by an RS and that can be accessed by a client, if and only if a valid and sufficient access token is provided.

Note: To avoid complex sentences, the specification document may simply refer to "resource" instead of "protected resource".

Right: Ability given to a subject to perform a given operation on a resource under the control of an RS.

Subject: Person or organization. The subject decides whether and under which conditions its attributes can be disclosed to other parties.

Subject Information: Set of statements and attributes asserted by an AS about a subject. These statements can be used by the client instance as part of an authentication decision.

1.4. Trust Relationships

GNAP defines its trust objective as follows: the RO trusts the AS to ensure access validation and delegation of protected resources to end users, through third party clients.

This trust objective can be decomposed into trust relationships between software elements and roles, especially the pairs end user/RO, end user/client, client/AS, RS/RO, AS/RO, and AS/RS. Trust of an agent by its pair can exist if the pair is informed that the agent has made a promise to follow the protocol in the past (e.g., pre-registration and uncompromised cryptographic components) or if the pair is able to infer by indirect means that the agent has made such a

promise (e.g., a compliant client request). Each agent defines its own valuation function of promises given or received. Examples of such valuations can be the benefits from interacting with other agents (e.g., safety in client access and interoperability with identity standards), the cost of following the protocol (including its security and privacy requirements and recommendations), a ranking of promise importance (e.g., a policy decision made by the AS), the assessment of one's vulnerability or risk of not being able to defend against threats, etc. Those valuations may depend on the context of the request. For instance, depending on the specific case in which GNAP is used, the AS may decide to either take into account or discard hints provided by the client, or the RS may refuse bearer tokens. Some promises can be affected by previous interactions (e.g., repeated requests).

Below are details of each trust relationship:

end user/RO: This relationship exists only when the end user and the RO are different, in which case the end user needs some out-of-band mechanism of getting the RO consent (see Section 4). GNAP generally assumes that humans can be authenticated, thanks to identity protocols (for instance, through an id_token assertion as described in Section 2.2).

end user/client: The client acts as a user agent. Depending on the technology used (browser, single-page application (SPA), mobile application, Internet of Things (IoT) device, etc.), some interactions may or may not be possible (as described in Section 2.5.1). Client developers implement requirements and generally some recommendations or best practices, so that the end users may confidently use their software. However, end users might also face an attacker's client software or a poorly implemented client without even realizing it.

end user/AS: When the client supports the interaction feature (see Section 3.3), the end user interacts with the AS through an AS-provided interface. In many cases, this happens through a front-channel interaction through the end user's browser. See Section 11.29 for some considerations in trusting these interactions.

client/AS: An honest AS may face an attacker's client (as discussed just above), or the reverse, and GNAP aims to make common attacks impractical. This specification makes access tokens opaque to the client and defines the request/response scheme in detail, therefore avoiding extra trust hypotheses from this critical piece of software. Yet, the AS may further define cryptographic attestations or optional rules to simplify the access of clients it already trusts, due to past behavior or organizational policies (see Section 2.3).

RS/RO: On behalf of the RO, the RS promises to protect its resources from unauthorized access and only accepts valid access tokens issued by a trusted AS. In case tokens are key bound, proper validation of the proofing method is expected from the RS.

AS/RO: The AS is expected to follow the decisions made by the RO, through either interactive consent requests, repeated interactions, or automated rules (as described in Section 1.6). Privacy considerations aim to reduce the risk of an honest but too-curious AS or the consequences of an unexpected user data exposure.

AS/RS: The AS promises to issue valid access tokens to legitimate client requests (i.e., after carrying out appropriate due diligence, as defined in the GNAP). Some optional configurations are covered by [GNAP-RS].

A global assumption made by GNAP is that authorization requests are security and privacy sensitive, and appropriate measures are detailed in Sections 11 and 12, respectively.

A formal trust model is out of scope of this specification, but one could be developed using techniques such as the Promise Theory [promise-theory].

1.5. Protocol Flow

GNAP is fundamentally designed to allow delegated access to APIs and other information, such as subject information, using a multi-stage, stateful process. This process allows different parties to provide information into the system to alter and augment the state of the delegated access and its artifacts.

The underlying requested grant moves through several states as different actions take place during the protocol, as shown in Figure 2.

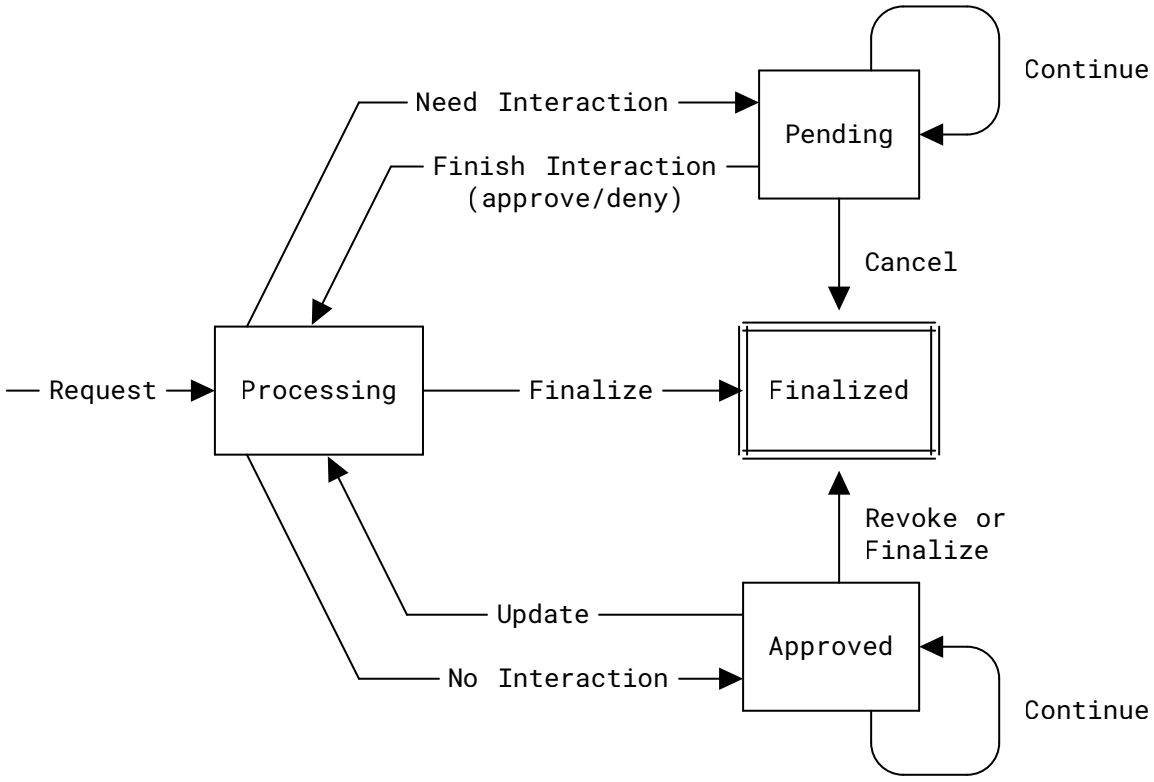

Figure 2: State Diagram of a Grant Request in GNAP

The state of the grant request is defined and managed by the AS, though the client instance also needs to manage its view of the grant request over time. The means by which these roles manage their state are outside the scope of this specification.

Processing: When a request for access (Section 2) is received by the AS, a new grant request is created and placed in the *processing* state by the AS. This state is also entered when an existing grant request is updated by the client instance and when interaction is completed. In this state, the AS processes the context of the grant request to determine whether interaction with the end user or RO is required for approval of the request. The grant request has to exit this state before a response can be returned to the client instance. If approval is required, the request moves to the *pending* state, and the AS returns a continuation response (Section 3.1) along with any appropriate interaction responses (Section 3.3). If no such approval is required, such as when the client instance is acting on its own behalf or the AS can determine that access has been fulfilled, the request moves to the *approved* state where access tokens for API access (Section 3.2) and subject information (Section 3.4) can be issued to the client instance. If the AS determines that no additional processing can occur (such as a timeout or an unrecoverable error), the grant request is moved to the *finalized* state and is terminated.

Pending: When a request needs to be approved by an RO, or interaction with the end user is required, the grant request enters a state of *pending*. In this state, no access tokens can be granted, and no subject information can be released to the client instance. While a grant request is in this state, the AS seeks to gather the required consent and authorization (Section 4) for the requested access. A grant request in this state is always associated with a continuation access token bound to the client instance's key (see Section 3.1 for details of the continuation access token). If no interaction finish method (Section 2.5.2) is associated with this request, the client instance can send a polling continuation request (Section 5.2) to the AS. This returns a continuation response (Section 3.1) while the grant request remains in this state, allowing the client instance to continue to check the state of the pending grant request. If an interaction finish method (Section 2.5.2) is specified in the grant request, the client instance can continue the request after interaction (Section 5.1) to the AS to move this request to the *processing* state to be re-evaluated by the AS. Note that this occurs whether the grant request has been approved or denied by the RO, since the AS needs to take into account the full context of the request before determining the next step for the grant request. When other information is made available in the context of the grant request, such as through the asynchronous actions of the RO, the AS moves this request to the *processing* state to be re-evaluated. If the AS determines that no additional interaction can occur, e.g., all the interaction methods have timed out or a revocation request (Section 5.4) is received from the client instance, the grant request can be moved to the *finalized* state.

Approved: When a request has been approved by an RO and no further interaction with the end user is required, the grant request enters a state of *approved*. In this state, responses to the client instance can include access tokens for API access (Section 3.2) and subject information (Section 3.4). If continuation and updates are allowed for this grant request, the AS can include the continuation response (Section 3.1). In this state, post-interaction continuation requests (Section 5.1) are not allowed and will result in an error, since all interaction is assumed to have been completed. If the client instance sends a polling continuation request

(Section 5.2) while the request is in this state, new access tokens (Section 3.2) can be issued in the response. Note that this always creates a new access token, but any existing access tokens could be rotated and revoked using the token management API (Section 6). The client instance can send an update continuation request (Section 5.3) to modify the requested access, causing the AS to move the request back to the *processing* state for re-evaluation. If the AS determines that no additional tokens can be issued and that no additional updates are to be accepted (e.g., the continuation access tokens have expired), the grant is moved to the *finalized* state.

Finalized: After the access tokens are issued, if the AS does not allow any additional updates on the grant request, the grant request enters the *finalized* state. This state is also entered when an existing grant request is revoked by the client instance (Section 5.4) or otherwise revoked by the AS (such as through out-of-band action by the RO). This state can also be entered if the AS determines that no additional processing is possible, for example, if the RO has denied the requested access or if interaction is required but no compatible interaction methods are available. Once in this state, no new access tokens can be issued, no subject information can be returned, and no interactions can take place. Once in this state, the grant request is dead and cannot be revived. If future access is desired by the client instance, a new grant request can be created, unrelated to this grant request.

While it is possible to deploy an AS in a stateless environment, GNAP is a stateful protocol, and such deployments will need a way to manage the current state of the grant request in a secure and deterministic fashion without relying on other components, such as the client software, to keep track of the current state.

1.6. Sequences

GNAP can be used in a variety of ways to allow the core delegation process to take place. Many portions of this process are conditionally present depending on the context of the deployments, and not every step in this overview will happen in all circumstances.

Note that a connection between roles in this process does not necessarily indicate that a specific protocol message is sent across the wire between the components fulfilling the roles in question or that a particular step is required every time. For example, for a client instance interested in only getting subject information directly and not calling an RS, all steps involving the RS below do not apply.

In some circumstances, the information needed at a given stage is communicated out of band or is pre-configured between the components or entities performing the roles. For example, one entity can fulfill multiple roles, so explicit communication between the roles is not necessary within the protocol flow. Additionally, some components may not be involved in all use cases. For example, a client instance could be calling the AS just to get direct user information and have no need to get an access token to call an RS.

1.6.1. Overall Protocol Sequence

The following diagram provides a general overview of GNAP, including many different optional phases and connections. The diagrams in the following sections provide views of GNAP under more specific circumstances. These additional diagrams use the same conventions as the overall diagram below.

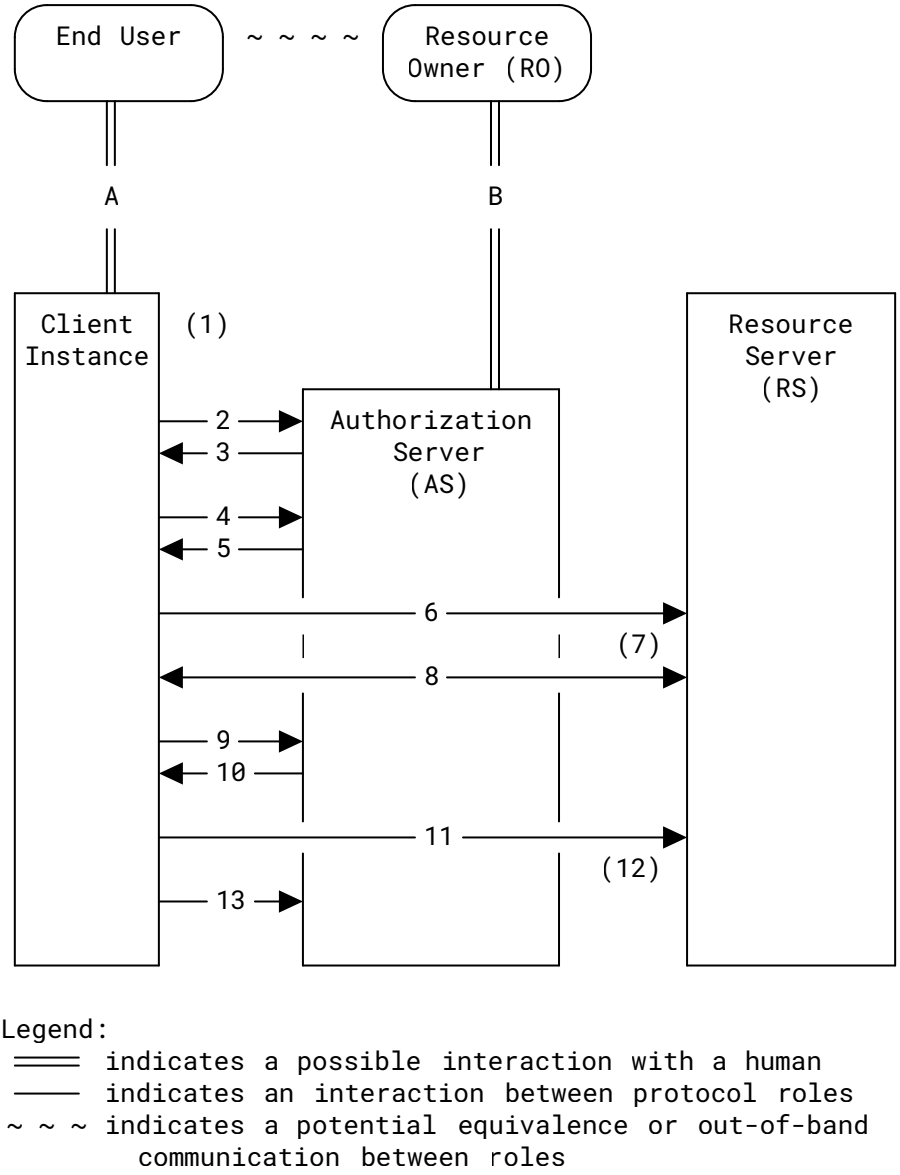

Figure 3: Overall Sequence of GNAP

- (A) The end user interacts with the client instance to indicate a need for resources on behalf of the RO. This could identify the RS that the client instance needs to call, the resources needed, or the RO that is needed to approve the request. Note that the RO and end user are often the same entity in practice, but GNAP makes no general assumption that they are.
- (1) The client instance determines what access is needed and which AS to approach for access. Note that for most situations, the client instance is pre-configured with which AS to talk to and which kinds of access it needs, but some more dynamic processes are discussed in Section 9.1.

- (2) The client instance requests access at the AS (Section 2).
- (3) The AS processes the request and determines what is needed to fulfill the request (see Section 4). The AS sends its response to the client instance (Section 3).
- (B) If interaction is required, the AS interacts with the RO (Section 4) to gather authorization. The interactive component of the AS can function using a variety of possible mechanisms, including web page redirects, applications, challenge/response protocols, or other methods. The RO approves the request for the client instance being operated by the end user. Note that the RO and end user are often the same entity in practice, and many of GNAP's interaction methods allow the client instance to facilitate the end user interacting with the AS in order to fulfill the role of the RO.
- (4) The client instance continues the grant at the AS (Section 5). This action could occur in response to receiving a signal that interaction has finished (Section 4.2) or through a periodic polling mechanism, depending on the interaction capabilities of the client software and the options active in the grant request.
- (5) If the AS determines that access can be granted, it returns a response to the client instance (Section 3), including an access token (Section 3.2) for calling the RS and any directly returned information (Section 3.4) about the RO.
- (6) The client instance uses the access token (Section 7.2) to call the RS.
- (7) The RS determines if the token is sufficient for the request by examining the token. The means of the RS determining this access are out of scope of this specification, but some options are discussed in [GNAP-RS].
- (8) The client instance calls the RS (Section 7.2) using the access token until the RS or client instance determines that the token is no longer valid.
- (9) When the token no longer works, the client instance rotates the access token (Section 6.1).
- (10) The AS issues a new access token (Section 3.2) to the client instance with the same rights as the original access token returned in (5).
- (11) The client instance uses the new access token (Section 7.2) to call the RS.
- (12) The RS determines if the new token is sufficient for the request, as in (7).
- (13) The client instance disposes of the token (Section 6.2) once the client instance has completed its access of the RS and no longer needs the token.

The following sections and Appendix B contain specific guidance on how to use GNAP in different situations and deployments. For example, it is possible for the client instance to never request an access token and never call an RS, just as it is possible to have no end user involved in the delegation process.

1.6.2. Redirect-Based Interaction

In this example flow, the client instance is a web application that wants access to resources on behalf of the current user, who acts as both the end user and the RO. Since the client instance is capable of directing the user to an arbitrary URI and receiving responses from the user's browser, interaction here is handled through front-channel redirects using the user's browser.

The redirection URI used for interaction is a service hosted by the AS in this example. The client instance uses a persistent session with the user to ensure the same user that is starting the interaction is the user that returns from the interaction.

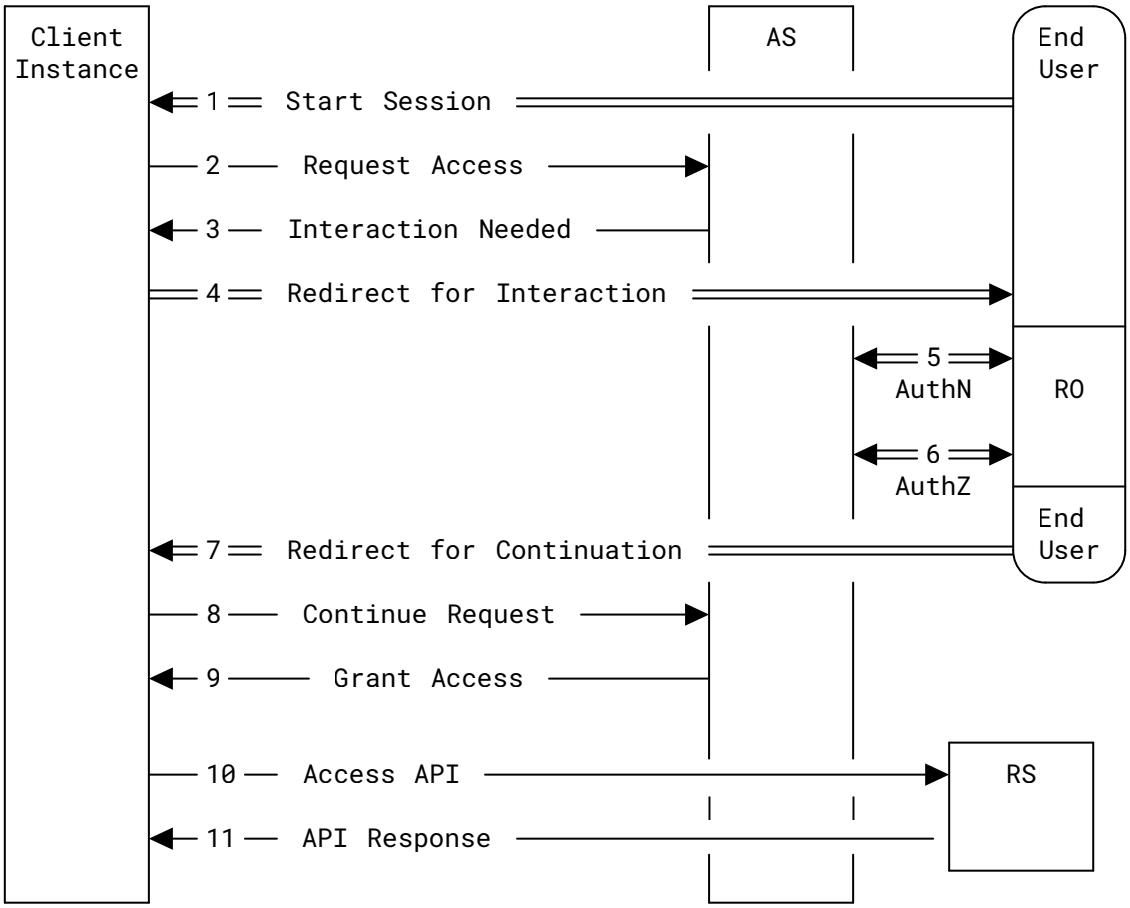

Figure 4: Diagram of a Redirect-Based Interaction

- (1) The client instance establishes a session with the user, in the role of the end user.
- (2) The client instance requests access to the resource (Section 2). The client instance indicates that it can redirect to an arbitrary URI (Section 2.5.1.1) and receive a redirect from the browser (Section 2.5.2.1). The client instance stores verification information for its redirect in the session created in (1).
- (3) The AS determines that interaction is needed and responds (Section 3) with a URI to send the user to (Section 3.3.1) and information needed to verify the redirect (Section 3.3.5) in (7). The AS also includes information the client instance will need to continue the request (Section 3.1) in (8). The AS associates this continuation information with an ongoing request that will be referenced in (4), (6), and (8).

- (4) The client instance stores the verification and continuation information from (3) in the session from (1). The client instance then redirects the user to the URI (Section 4.1.1) given by the AS in (3). The user's browser loads the interaction redirect URI. The AS loads the pending request based on the incoming URI generated in (3).
- (5) The user authenticates at the AS, taking on the role of the RO.
- (6) As the RO, the user authorizes the pending request from the client instance.
- (7) When the AS is done interacting with the user, the AS redirects the user back (Section 4.2.1) to the client instance using the redirect URI provided in (2). The redirect URI is augmented with an interaction reference that the AS associates with the ongoing request created in (2) and referenced in (4). The redirect URI is also augmented with a hash of the security information provided in (2) and (3). The client instance loads the verification information from (2) and (3) from the session created in (1). The client instance calculates a hash (Section 4.2.3) based on this information and continues only if the hash validates. Note that the client instance needs to ensure that the parameters for the incoming request match those that it is expecting from the session created in (1). The client instance also needs to be prepared for the end user never being returned to the client instance and handle timeouts appropriately.
- (8) The client instance loads the continuation information from (3) and sends the interaction reference from (7) in a request to continue the request (Section 5.1). The AS validates the interaction reference, ensuring that the reference is associated with the request being continued.
- (9) If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the client instance.
- (10) The client instance uses the access token (Section 7.2) to call the RS.
- (11) The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix B.1.

1.6.3. User Code Interaction

In this example flow, the client instance is a device that is capable of presenting a short, human-readable code to the user and directing the user to enter that code at a known URI. The user enters the code at a URI that is an interactive service hosted by the AS in this example. The client instance is not capable of presenting an arbitrary URI to the user, nor is it capable of accepting incoming HTTP requests from the user's browser. The client instance polls the AS while it is waiting for the RO to authorize the request. The user's interaction is assumed to occur on a secondary device. In this example, it is assumed that the user is both the end user and RO. Note that since the user is not assumed to be interacting with the client instance through the same web browser used for interaction at the AS, the user is not shown as being connected to the client instance in this diagram.

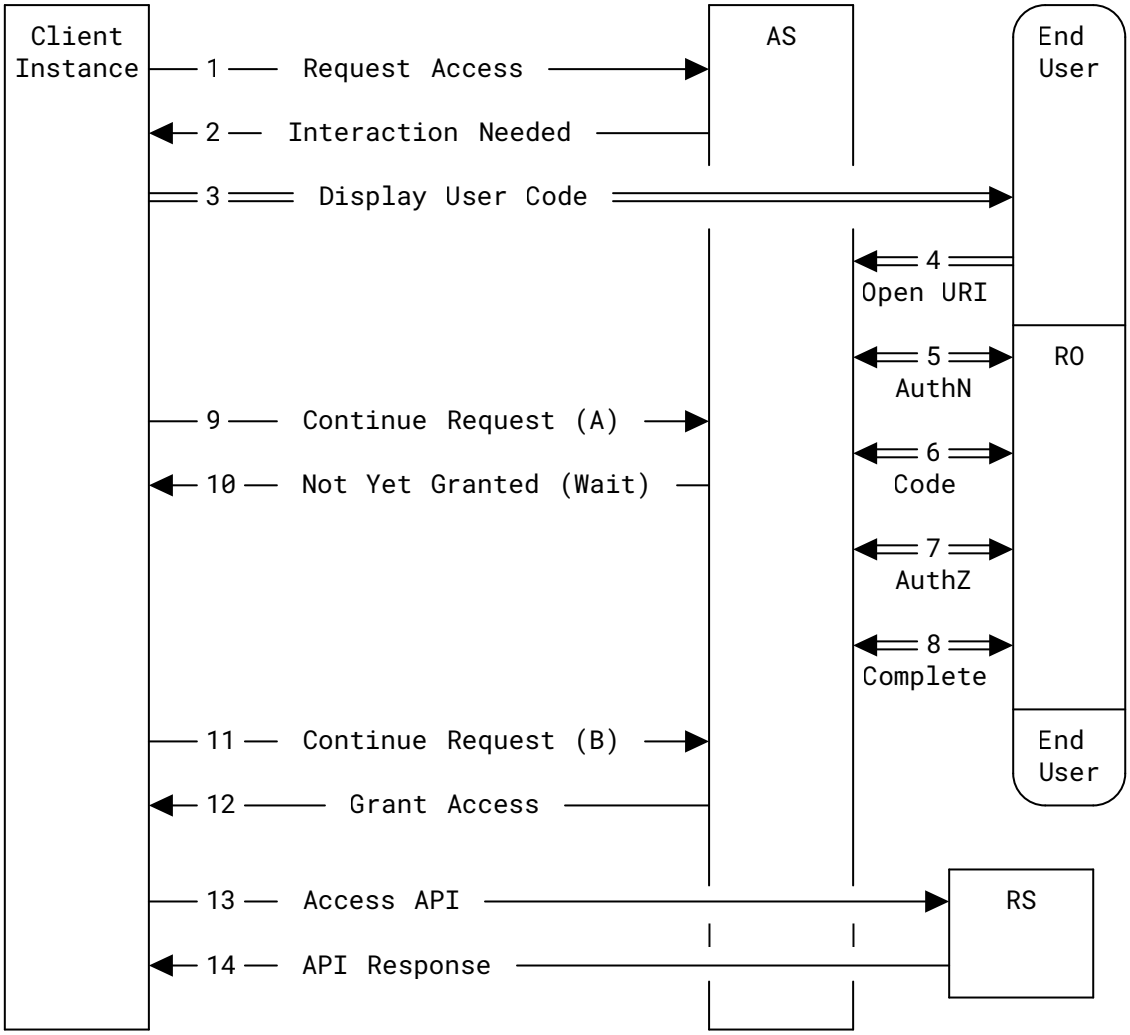

Figure 5: Diagram of a User-Code-Based Interaction

- (1) The client instance requests access to the resource (Section 2). The client instance indicates that it can display a user code (Section 2.5.1.3).
- (2) The AS determines that interaction is needed and responds (Section 3) with a user code to communicate to the user (Section 3.3.3). The AS also includes information the client instance will need to continue the request (Section 3.1) in (8) and (10). The AS associates this continuation information with an ongoing request that will be referenced in (4), (6), (8), and (10).
- (3) The client instance stores the continuation information from (2) for use in (8) and (10). The client instance then communicates the code to the user (Section 4.1.2) given by the AS in (2).

- (4) The user directs their browser to the user code URI. This URI is stable and can be communicated via the client software's documentation, the AS documentation, or the client software itself. Since it is assumed that the RO will interact with the AS through a secondary device, the client instance does not provide a mechanism to launch the RO's browser at this URI.
- (5) The end user authenticates at the AS, taking on the role of the RO.
- (6) The RO enters the code communicated in (3) to the AS. The AS validates this code against a current request in process.
- (7) As the RO, the user authorizes the pending request from the client instance.
- (8) When the AS is done interacting with the user, the AS indicates to the RO that the request has been completed.
- (9) Meanwhile, the client instance loads the continuation information stored at (3) and continues the request (Section 5). The AS determines which ongoing access request is referenced here and checks its state.
- (10) If the access request has not yet been authorized by the RO in (6), the AS responds to the client instance to continue the request (Section 3.1) at a future time through additional polled continuation requests. This response can include updated continuation information as well as information regarding how long the client instance should wait before calling again. The client instance replaces its stored continuation information from the previous response (2). Note that the AS may need to determine that the RO has not approved the request in a sufficient amount of time and return an appropriate error to the client instance.
- (11) The client instance continues to poll the AS (Section 5.2) with the new continuation information in (9).
- (12) If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the client instance.
- (13) The client instance uses the access token (Section 7.2) to call the RS.
- (14) The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix B.2.

1.6.4. Asynchronous Authorization

In this example flow, the end user and RO roles are fulfilled by different parties, and the RO does not interact with the client instance. The AS reaches out asynchronously to the RO during the request process to gather the RO's authorization for the client instance's request. The client instance polls the AS while it is waiting for the RO to authorize the request.

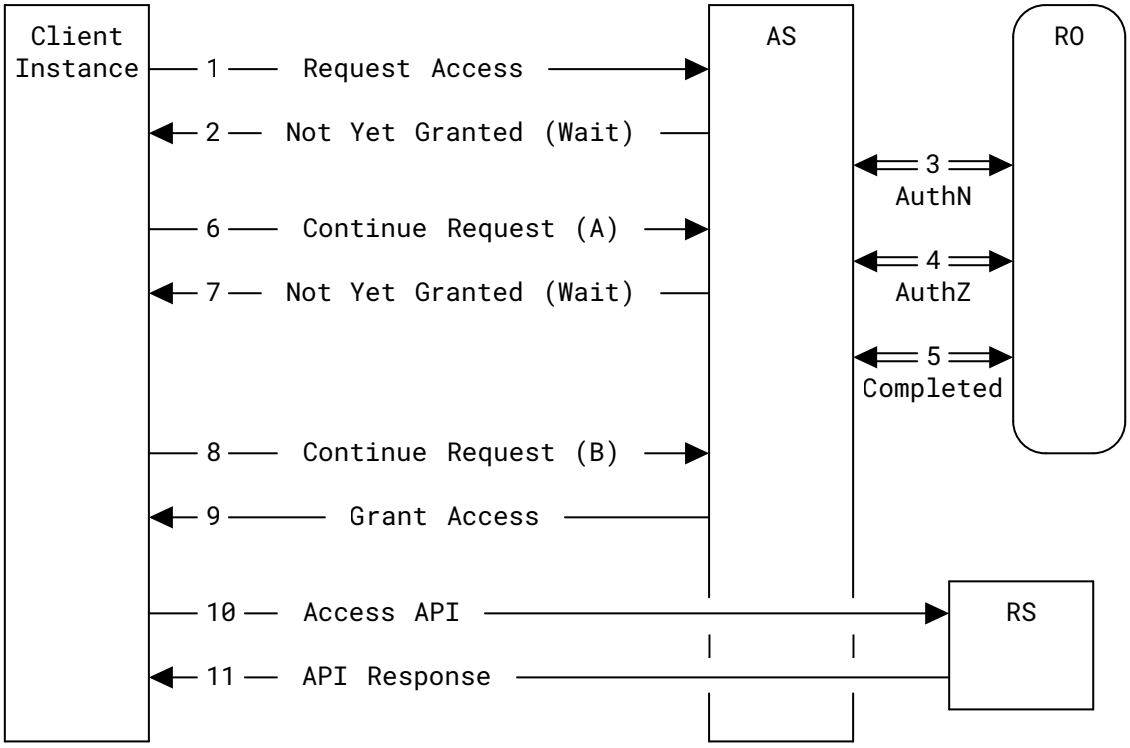

Figure 6: Diagram of an Asynchronous Authorization Process, with No End-User Interaction

- (1) The client instance requests access to the resource (Section 2). The client instance does not send any interaction modes to the server, indicating that it does not expect to interact with the RO. The client instance can also signal which RO it requires authorization from, if known, by using the subject request field (Section 2.2) and user request field (Section 2.4). It's also possible for the AS to determine which RO needs to be contacted by the nature of what access is being requested.
- (2) The AS determines that interaction is needed, but the client instance cannot interact with the RO. The AS responds (Section 3) with the information the client instance will need to continue the request (Section 3.1) in (6) and (8), including a signal that the client instance should wait before checking the status of the request again. The AS associates this continuation information with an ongoing request that will be referenced in (3), (4), (5), (6), and (8).
- (3) The AS determines which RO to contact based on the request in (1), through a combination of the user request (Section 2.4), the subject request (Section 2.2), the access request (Section 2.1), and other policy information. The AS contacts the RO and authenticates them.
- (4) The RO authorizes the pending request from the client instance.
- (5) When the AS is done interacting with the RO, the AS indicates to the RO that the request has been completed.

- (6) Meanwhile, the client instance loads the continuation information stored at (2) and continues the request (Section 5). The AS determines which ongoing access request is referenced here and checks its state.
- (7) If the access request has not yet been authorized by the RO in (6), the AS responds to the client instance to continue the request (Section 3.1) at a future time through additional polling. Note that this response is not an error message, since no error has yet occurred. This response can include refreshed credentials as well as information regarding how long the client instance should wait before calling again. The client instance replaces its stored continuation information from the previous response (2). Note that the AS may need to determine that the RO has not approved the request in a sufficient amount of time and return an appropriate error to the client instance.
- (8) The client instance continues to poll the AS (Section 5.2) with the new continuation information from (7).
- (9) If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the client instance.
- (10) The client instance uses the access token (Section 7.2) to call the RS.
- (11) The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix B.4.

Additional considerations for asynchronous interactions like this are discussed in Section 11.36.

1.6.5. Software-Only Authorization

In this example flow, the AS policy allows the client instance to make a call on its own behalf, without the need for an RO to be involved at runtime to approve the decision. Since there is no explicit RO, the client instance does not interact with an RO.

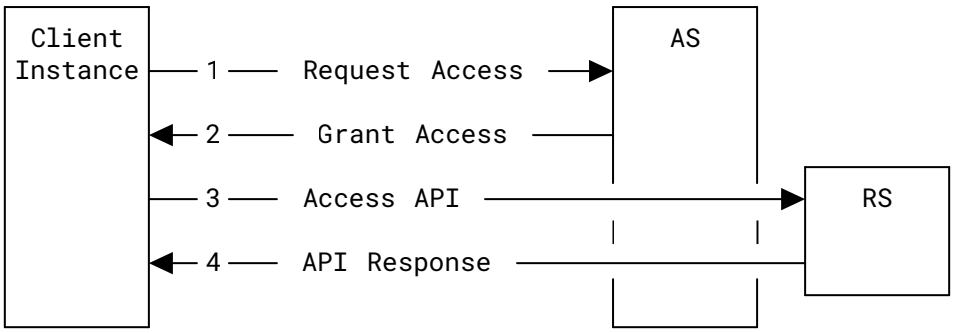

Figure 7: Diagram of a Software-Only Authorization, with No End User or Explicit Resource Owner

- (1) The client instance requests access to the resource (Section 2). The client instance does not send any interaction modes to the server.

- (2) The AS determines that the request has been authorized based on the identity of the client instance making the request and the access requested (Section 2.1). The AS grants access to the resource in the form of access tokens (Section 3.2) to the client instance. Note that direct subject information (Section 3.4) is not generally applicable in this use case, as there is no user involved.
- (3) The client instance uses the access token (Section 7.2) to call the RS.
- (4) The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix B.3.

1.6.6. Refreshing an Expired Access Token

In this example flow, the client instance receives an access token to access an RS through some valid GNAP process. The client instance uses that token at the RS for some time, but eventually the access token expires. The client instance then gets a refreshed access token by rotating the expired access token's value at the AS using the token management API.

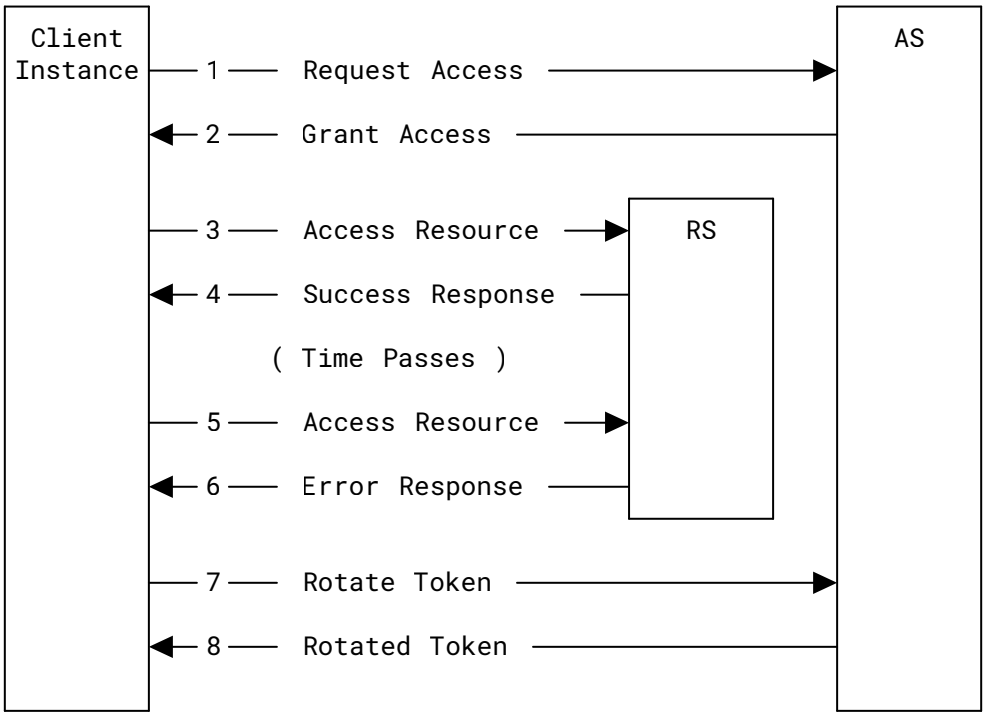

Figure 8: Diagram of the Process of Refreshing an Expired Access Token

- (1) The client instance requests access to the resource (Section 2).
- (2) The AS grants access to the resource (Section 3) with an access token (Section 3.2) usable at the RS. The access token response includes a token management URI.
- (3) The client instance uses the access token (Section 7.2) to call the RS.

- (4) The RS validates the access token and returns an appropriate response for the API.
- (5) Time passes and the client instance uses the access token to call the RS again.
- (6) The RS validates the access token and determines that the access token is expired. The RS responds to the client instance with an error.
- (7) The client instance calls the token management URI returned in (2) to rotate the access token (Section 6.1). The client instance uses the access token (Section 7.2) in this call as well as the appropriate key; see Section 6.1 for details.
- (8) The AS validates the rotation request, including the signature and keys presented in (7), and refreshes the access token (Section 3.2.1). The response includes a new version of the access token and can also include updated token management information, which the client instance will store in place of the values returned in (2).

1.6.7. Requesting Subject Information Only

In this scenario, the client instance does not call an RS and does not request an access token. Instead, the client instance only requests and is returned direct subject information (Section 3.4). Many different interaction modes can be used in this scenario, so these are shown only in the abstract as functions of the AS here.

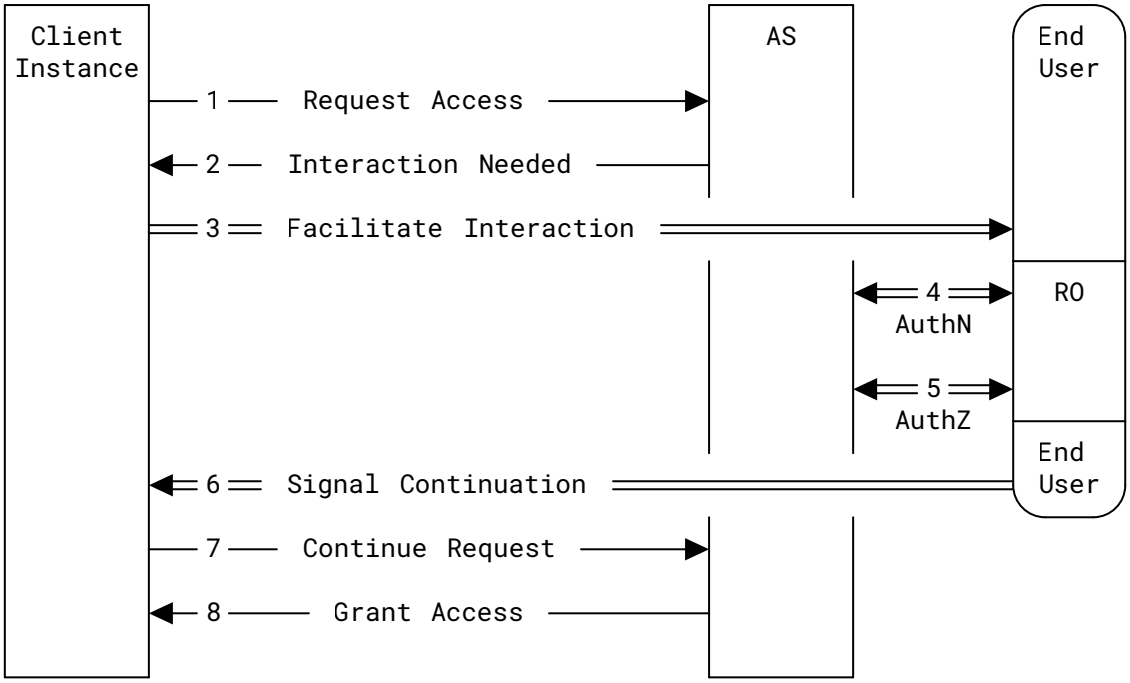

Figure 9: Diagram of the Process of Requesting and Releasing Subject Information apart from Access Tokens

- (1) The client instance requests access to subject information (Section 2).

- (2) The AS determines that interaction is needed and responds (Section 3) with appropriate information for facilitating user interaction (Section 3.3).
- (3) The client instance facilitates the user interacting with the AS (Section 4) as directed in (2).
- (4) The user authenticates at the AS, taking on the role of the RO.
- (5) As the RO, the user authorizes the pending request from the client instance.
- (6) When the AS is done interacting with the user, the AS returns the user to the client instance and signals continuation.
- (7) The client instance loads the continuation information from (2) and calls the AS to continue the request (Section 5).
- (8) If the request has been authorized, the AS grants access to the requested direct subject information (Section 3.4) to the client instance. At this stage, the user is generally considered "logged in" to the client instance based on the identifiers and assertions provided by the AS. Note that the AS can restrict the subject information returned, and it might not match what the client instance requested; see Section 3.4 for details.

1.6.8. Cross-User Authentication

In this scenario, the end user and RO are two different people. Here, the client instance already knows who the end user is, likely through a separate authentication process. The end user, operating the client instance, needs to get subject information about another person in the system, the RO. The RO is given an opportunity to release this information using an asynchronous interaction method with the AS. This scenario would apply, for instance, when the end user is an agent in a call center and the RO is a customer authorizing the call-center agent to access their account on their behalf.

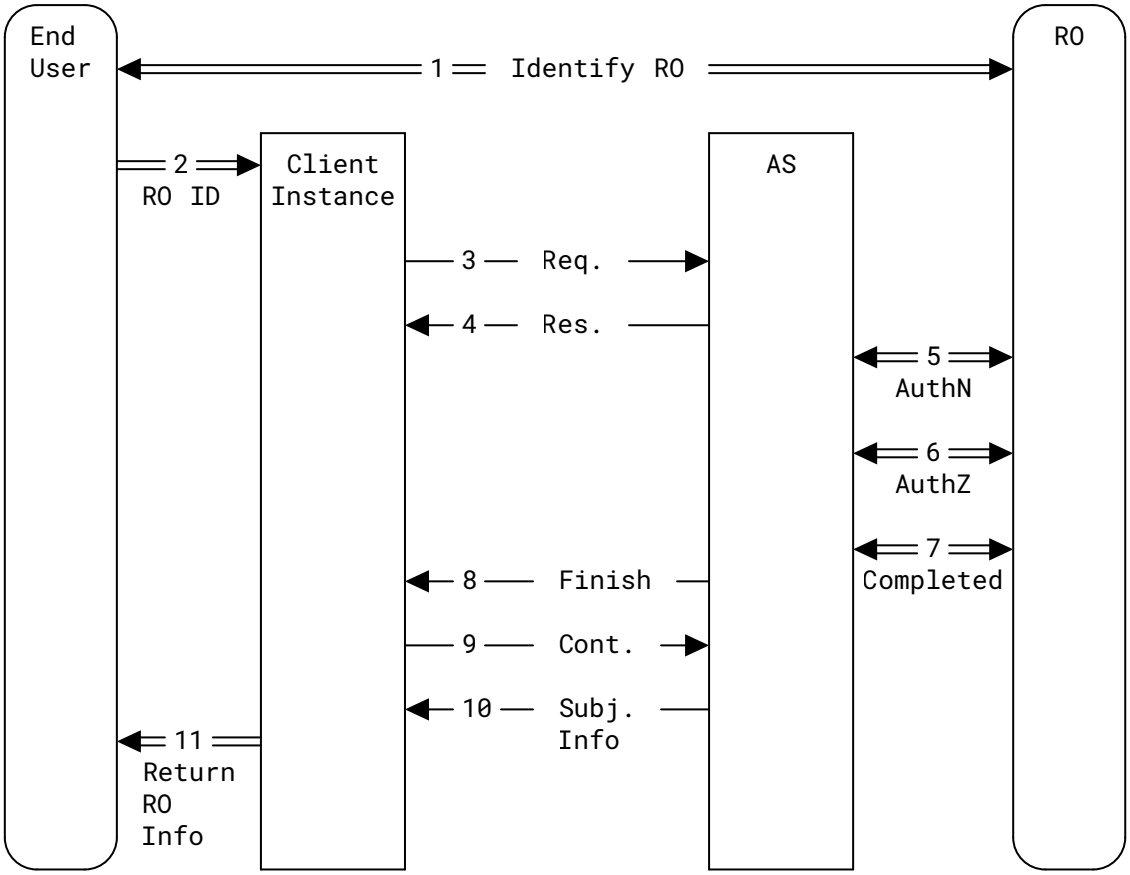

Figure 10: Diagram of Cross-User Authorization, Where the End User and RO Are Different

Precondition: The end user is authenticated to the client instance, and the client instance has an identifier representing the end user that it can present to the AS. This identifier should be unique to the particular session with the client instance and the AS. The client instance is also known to the AS and allowed to access this advanced functionality where the information of someone other than the end user is returned to the client instance.

- (1) The RO communicates a human-readable identifier to the end user, such as an email address or account number. This communication happens out of band from the protocol, such as over the phone between parties. Note that the RO is not interacting with the client instance.
- (2) The end user communicates the identifier to the client instance. The means by which the identifier is communicated to the client instance are out of scope for this specification.
- (3) The client instance requests access to subject information (Section 2). The request includes the RO's identifier in the `sub_ids` field of the subject information request (Section 2.2) and the end user's identifier in the `user` field (Section 2.4). The request includes no

interaction start methods, since the end user is not expected to be the one interacting with the AS. The request does include the push-based interaction finish method (Section 2.5.2.2) to allow the AS to signal to the client instance when the interaction with the RO has concluded.

- (4) The AS sees that the identifiers for the end user and subject being requested are different. The AS determines that it can reach out to the RO asynchronously for approval. While it is doing so, the AS returns a continuation response (Section 3.1) with a `finish` nonce to allow the client instance to continue the grant request after interaction with the RO has concluded.
- (5) The AS contacts the RO and has them authenticate to the system. The means for doing this are outside the scope of this specification, but the identity of the RO is known from the Subject Identifier sent in (3).
- (6) The RO is prompted to authorize the end user's request via the client instance. Since the end user was identified in (3) via the `user` field, the AS can show this information to the RO during the authorization request.
- (7) The RO completes the authorization with the AS. The AS marks the request as *approved*.
- (8) The RO pushes the interaction finish message (Section 4.2.2) to the client instance. Note that in the case the RO cannot be reached or the RO denies the request, the AS still sends the interaction finish message to the client instance, after which the client instance can negotiate next steps if possible.
- (9) The client instance validates the interaction finish message and continues the grant request (Section 5.1).
- (10) The AS returns the RO's subject information (Section 3.4) to the client instance.
- (11) The client instance can display or otherwise utilize the RO's user information in its session with the end user. Note that since the client instance requested different sets of user information in (3), the client instance does not conflate the end user with the RO.

Additional considerations for asynchronous interactions like this are discussed in Section 11.36.

2. Requesting Access

To start a request, the client instance sends an HTTP POST with a JSON [RFC8259] document to the grant endpoint of the AS. The grant endpoint is a URI that uniquely identifies the AS to client instances and serves as the identifier for the AS. The document is a JSON object where each field represents a different aspect of the client instance's request. Each field is described in detail in a subsection below.

`access_token` (object / array of objects): Describes the rights and properties associated with the requested access token. **REQUIRED** if requesting an access token. See Section 2.1.

`subject` (object): Describes the information about the RO that the client instance is requesting to be returned directly in the response from the AS. **REQUIRED** if requesting subject information. See Section 2.2.

`client` (object / string): Describes the client instance that is making this request, including the key that the client instance will use to protect this request, any continuation requests at the AS, and any user-facing information about the client instance used in interactions. **REQUIRED**. See Section 2.3.

`user` (object / string): Identifies the end user to the AS in a manner that the AS can verify, either directly or by interacting with the end user to determine their status as the RO. **OPTIONAL**. See Section 2.4.

`interact` (object): Describes the modes that the client instance supports for allowing the RO to interact with the AS and modes for the client instance to receive updates when interaction is complete. **REQUIRED** if interaction is supported. See Section 2.5.

Additional members of this request object can be defined by extensions using the "GNAP Grant Request Parameters" registry (Section 10.3).

A non-normative example of a grant request is below:

```
{
    "access_token": {
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "dolphin-metadata"
        ]
    },
    "client": {
      "display": {
        "name": "My Client Display Name",
        "uri": "https://example.net/client"
      },
      "key": {
        "proof": "httpsig",
        "jwk": {
          "kty": "RSA",
          "e": "AQAB",
          "kid": "xyz-1",
          "alg": "RS256",
          "n": "kOB5rR4Jv0GMeL...."
        }
```

```
        }
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    },
    "subject": {
        "sub_id_formats": ["iss_sub", "opaque"],
        "assertion_formats": ["id_token"]
    }
}
```

Sending a request to the grant endpoint creates a grant request in the *processing* state. The AS processes this request to determine whether interaction or authorization are necessary (moving to the *pending* state) or if access can be granted immediately (moving to the *approved* state).

The request **MUST** be sent as a JSON object in the content of the HTTP POST request with Content-Type `application/json`. A key proofing mechanism **MAY** define an alternative content type, as long as the content is formed from the JSON object. For example, the attached JSON Web Signature (JWS) key proofing mechanism (see Section 7.3.4) places the JSON object into the payload of a JWS wrapper, which is in turn sent as the message content.

2.1. Requesting Access to Resources

If the client instance is requesting one or more access tokens for the purpose of accessing an API, the client instance **MUST** include an `access_token` field. This field **MUST** be an object (for a single access token (Section 2.1.1)) or an array of these objects (for multiple access tokens (Section 2.1.2)), as described in the following subsections.

2.1.1. Requesting a Single Access Token

To request a single access token, the client instance sends an `access_token` object composed of the following fields.

`access` (array of objects/strings): Describes the rights that the client instance is requesting for the access token to be used at the RS. **REQUIRED**. See Section 8.

`label` (string): A unique name chosen by the client instance to refer to the resulting access token. The value of this field is opaque to the AS and is not intended to be exposed to or used by the end user. If this field is included in the request, the AS **MUST** include the same label in the token response (Section 3.2). **REQUIRED** if used as part of a request for multiple access tokens (Section 2.1.2); **OPTIONAL** otherwise.

`flags` (array of strings): A set of flags that indicate desired attributes or behavior to be attached to the access token by the AS. **OPTIONAL**.

The values of the `flags` field defined by this specification are as follows:

`"bearer"`: If this flag is included, the access token being requested is a bearer token. If this flag is omitted, the access token is bound to the key used by the client instance in this request (or that key's most recent rotation), and the access token **MUST** be presented using the same key and proofing method. Methods for presenting bound and bearer access tokens are described in Section 7.2. See Section 11.9 for additional considerations on the use of bearer tokens.

Flag values **MUST NOT** be included more than once. If the request includes a flag value multiple times, the AS **MUST** return an `invalid_flag` error defined in Section 3.6.

Additional flags can be defined by extensions using the "GNAP Access Token Flags" registry (Section 10.4).

In the following non-normative example, the client instance is requesting access to a complex resource described by a pair of access request object.

```
"access_token": {
    "access": [
        {
            "type": "photo-api",
            "actions": [
                "read",
                "write",
                "delete"
            ],
            "locations": [
                "https://server.example.net/",
                "https://resource.local/other"
            ],
            "datatypes": [
                "metadata",
                "images"
            ]
        },
        {
            "type": "walrus-access",
            "actions": [
                "foo",
                "bar"
            ],
            "locations": [
                "https://resource.other/"
            ],
            "datatypes": [
                "data",
                "pictures",
                "walrus whiskers"
            ]
        }
    ],
    "label": "token1-23"
}
```

If access is approved, the resulting access token is valid for the described resource. Since the `bearer` flag is not provided in this example, the token is bound to the client instance's key (or its most recent rotation). The token is labeled "token1-23". The token response structure is described in Section 3.2.1.

2.1.2. Requesting Multiple Access Tokens

To request that multiple access tokens be returned in a single response, the client instance sends an array of objects as the value of the `access_token` parameter. Each object **MUST** conform to the request format for a single access token request, as specified in Section 2.1.1. Additionally, each object in the array **MUST** include the `label` field, and all values of these fields **MUST** be unique within the request. If the client instance does not include a `label` value for any entry in the array or the values of the `label` field are not unique within the array, the AS **MUST** return an "invalid_request" error (Section 3.6).

The following non-normative example shows a request for two separate access tokens: `token1` and `token2`.

```
"access_token": [
    {
        "label": "token1",
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "dolphin-metadata"
        ]
    },
    {
        "label": "token2",
        "access": [
            {
                "type": "walrus-access",
                "actions": [
                    "foo",
                    "bar"
                ],
                "locations": [
                    "https://resource.other/"
                ],
                "datatypes": [
                    "data",
                    "pictures",
                    "walrus whiskers"
                ]
            }
        ],
        "flags": [ "bearer" ]
    }
]
```

All approved access requests are returned in the response structure for multiple access tokens (Section 3.2.2) using the values of the `label` fields in the request.

2.2. Requesting Subject Information

If the client instance is requesting information about the RO from the AS, it sends a `subject` field as a JSON object. This object **MAY** contain the following fields.

`sub_id_formats` (array of strings): An array of Subject Identifier subject formats requested for the RO, as defined by [RFC9493]. **REQUIRED** if Subject Identifiers are requested.

`assertion_formats` (array of strings): An array of requested assertion formats. Possible values include `id_token` for an OpenID Connect ID Token [OIDC] and `saml2` for a Security Assertion Markup Language (SAML) 2 assertion [SAML2]. Additional assertion formats can be defined in the "GNAP Assertion Formats" registry (Section 10.6). **REQUIRED** if assertions are requested.

`sub_ids` (array of objects): An array of Subject Identifiers representing the subject for which information is being requested. Each object is a Subject Identifier as defined by [RFC9493]. All identifiers in the `sub_ids` array **MUST** identify the same subject. If omitted, the AS **SHOULD** assume that subject information requests are about the current user and **SHOULD** require direct interaction or proof of presence before releasing information. **OPTIONAL**.

Additional fields can be defined in the "GNAP Subject Information Request Fields" registry (Section 10.5).

```
"subject": {
  "sub_id_formats": [ "iss_sub", "opaque" ],
  "assertion_formats": [ "id_token", "saml2" ]
}
```

The AS can determine the RO's identity and permission for releasing this information through interaction with the RO (Section 4), AS policies, or assertions presented by the client instance (Section 2.4). If this is determined positively, the AS **MAY** return the RO's information in its response (Section 3.4) as requested.

Subject Identifier types requested by the client instance serve only to identify the RO in the context of the AS and can't be used as communication channels by the client instance, as discussed in Section 3.4.

2.3. Identifying the Client Instance

When sending a new grant request to the AS, the client instance **MUST** identify itself by including its client information in the `client` field of the request and by signing the request with its unique key as described in Section 7.3. Note that once a grant has been created and is in either the *pending* or the *approved* state, the AS can determine which client is associated with the grant by dereferencing the continuation access token sent in the continuation request (Section 5). As a consequence, the `client` field is not sent or accepted for continuation requests.

Client information is sent by value as an object or by reference as a string (see Section 2.3.1).

When client instance information is sent by value, the `client` field of the request consists of a JSON object with the following fields.

`key` (object / string): The public key of the client instance to be used in this request as described in Section 7.1 or a reference to a key as described in Section 7.1.1. **REQUIRED**.

`class_id` (string): An identifier string that the AS can use to identify the client software comprising this client instance. The contents and format of this field are up to the AS. **OPTIONAL**.

`display` (object): An object containing additional information that the AS **MAY** display to the RO during interaction, authorization, and management. **OPTIONAL**. See Section 2.3.2.

```
"client": {
    "key": {
        "proof": "httpsig",
        "jwk": {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "xyz-1",
            "alg": "RS256",
            "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8..."
        }
    },
    "class_id": "web-server-1234",
    "display": {
        "name": "My Client Display Name",
        "uri": "https://example.net/client"
    }
}
```

Additional fields can be defined in the "GNAP Client Instance Fields" registry (Section 10.7).

Absent additional attestations, profiles, or trust mechanisms, both the `display` and `class_id` fields are self-declarative, presented by the client instance. The AS needs to exercise caution in their interpretation, taking them as a hint but not as absolute truth. The `class_id` field can be used in a variety of ways to help the AS make sense of the particular context in which the client instance is operating. In corporate environments, for example, different levels of trust might apply depending on security policies. This field aims to help the AS adjust its own access decisions for different classes of client software. It is possible to configure a set of values and rules during a pre-registration and then have the client instances provide them later in runtime as a hint to the AS. In other cases, the client runs with a specific AS in mind, so a single hardcoded value would be acceptable (for instance, a set-top box with a `class_id` claiming to be "FooBarTV version 4"). While the client instance may not have contacted the AS yet, the value of this `class_id` field can be evaluated by the AS according to a broader context of dynamic use, alongside other related information available elsewhere (for instance, corresponding fields in a certificate). If the AS is not able to interpret or validate the class_id field, it **MUST** either return an `invalid_client` error (Section 3.6) or interpret the request as if the class_id were not present. See additional discussion of client instance impersonation in Section 11.15.

The client instance **MUST** prove possession of any presented key by the proofing mechanism associated with the key in the request. Key proofing methods are defined in the "GNAP Key Proofing Methods" registry (Section 10.16), and an initial set of methods is described in Section 7.3.

If the same public key is sent by value on different access requests, the AS **MUST** treat these requests as coming from the same client instance for purposes of identification, authentication, and policy application.

If the AS does not know the client instance's public key ahead of time, the AS can choose how to process the unknown key. Common approaches include:

- Allowing the request and requiring RO authorization in a trust-on-first-use model
- Limiting the client's requested access to only certain APIs and information
- Denying the request entirely by returning an `invalid_client` error (Section 3.6)

The client instance **MUST NOT** send a symmetric key by value in the `key` field of the request, as doing so would expose the key directly instead of simply proving possession of it. See considerations on symmetric keys in Section 11.7. To use symmetric keys, the client instance can send the `key` by reference (Section 7.1.1) or send the entire client identity by reference (Section 2.3.1).

The client instance's key can be pre-registered with the AS ahead of time and associated with a set of policies and allowable actions pertaining to that client. If this pre-registration includes other fields that can occur in the `client` request object described in this section, such as `class_id` or `display`, the pre-registered values **MUST** take precedence over any values given at runtime. Additional fields sent during a request but not present in a pre-registered client instance record at the AS **SHOULD NOT** be added to the client's pre-registered record. See additional considerations regarding client instance impersonation in Section 11.15.

A client instance that is capable of talking to multiple ASes **SHOULD** use a different key for each AS to prevent a class of mix-up attacks as described in Section 11.31, unless other mechanisms can be used to assure the identity of the AS for a given request.

2.3.1. Identifying the Client Instance by Reference

If the client instance has an instance identifier that the AS can use to determine appropriate key information, the client instance can send this instance identifier as a direct reference value in lieu of the `client` object. The instance identifier **MAY** be assigned to a client instance at runtime through a grant response (Section 3.5) or **MAY** be obtained in another fashion, such as a static registration process at the AS.

```
"client": "client-541-ab"
```

When the AS receives a request with an instance identifier, the AS **MUST** ensure that the key used to sign the request (Section 7.3) is associated with the instance identifier.

If the AS does not recognize the instance identifier, the request **MUST** be rejected with an `invalid_client` error (Section 3.6).

2.3.2. Providing Displayable Client Instance Information

If the client instance has additional information to display to the RO during any interactions at the AS, it **MAY** send that information in the "display" field. This field is a JSON object that declares information to present to the RO during any interactive sequences.

`name` (string): Display name of the client software. **RECOMMENDED**.

`uri` (string): User-facing information about the client software, such as a web page. This URI **MUST** be an absolute URI. **OPTIONAL**.

`logo_uri` (string): Display image to represent the client software. This URI **MUST** be an absolute URI. The logo **MAY** be passed by value by using a data: URI [RFC2397] referencing an image media type. **OPTIONAL**.

```
"display": {
    "name": "My Client Display Name",
    "uri": "https://example.net/client",
    "logo_uri": "...="
}
```

Additional display fields can be defined in the "GNAP Client Instance Display Fields" registry (Section 10.8).

The AS **SHOULD** use these values during interaction with the RO. The values are for informational purposes only and **MUST NOT** be taken as authentic proof of the client instance's identity or source. The AS **MAY** restrict display values to specific client instances, as identified by their keys in Section 2.3. See additional considerations for displayed client information in Section 11.15 and for the `logo_uri` in particular in Section 11.16.

2.3.3. Authenticating the Client Instance

If the presented key is known to the AS and is associated with a single instance of the client software, the process of presenting a key and proving possession of that key is sufficient to authenticate the client instance to the AS. The AS **MAY** associate policies with the client instance identified by this key, such as limiting which resources can be requested and which interaction methods can be used. For example, only specific client instances with certain known keys might be trusted with access tokens without the AS interacting directly with the RO, as in Appendix B.3.

The presentation of a key allows the AS to strongly associate multiple successive requests from the same client instance with each other. This is true when the AS knows the key ahead of time and can use the key to authenticate the client instance, but it is also true if the key is ephemeral and created just for this series of requests. As such, the AS **MAY** allow for client instances to make requests with unknown keys. This pattern allows for ephemeral client instances (such as single-page applications) and client software with many individual long-lived instances (such as mobile

applications) to generate key pairs per instance and use the keys within the protocol without having to go through a separate registration step. The AS **MAY** limit which capabilities are made available to client instances with unknown keys. For example, the AS could have a policy saying that only previously registered client instances can request particular resources or that all client instances with unknown keys have to be interactively approved by an RO.

2.4. Identifying the User

If the client instance knows the identity of the end user through one or more identifiers or assertions, the client instance **MAY** send that information to the AS in the `user` field. The client instance **MAY** pass this information by value or by reference (see Section 2.4.1).

`sub_ids` (array of objects): An array of Subject Identifiers for the end user, as defined by [RFC9493]. **OPTIONAL.**

`assertions` (array of objects): An array containing assertions as objects, each containing the assertion format and the assertion value as the JSON string serialization of the assertion, as defined in Section 3.4. **OPTIONAL.**

```
"user": {
  "sub_ids": [ {
    "format": "opaque",
    "id": "J2G8G804AZ"
  } ],
  "assertions": [ {
    "format": "id_token",
    "value": "eyj..."
  } ]
}
```

Subject Identifiers are hints to the AS in determining the RO and **MUST NOT** be taken as authoritative statements that a particular RO is present at the client instance and acting as the end user.

Assertions presented by the client instance **SHOULD** be validated by the AS. While the details of such validation are outside the scope of this specification, common validation steps include verifying the signature of the assertion against a trusted signing key, verifying the audience and issuer of the assertion map to expected values, and verifying the time window for the assertion itself. However, note that in many use cases, some of these common steps are relaxed. For example, an AS acting as an identity provider (IdP) could expect that assertions being presented using this mechanism were issued by the AS to the client software. The AS would verify that the AS is the issuer of the assertion, not the audience, and that the client instance is instead the audience of the assertion. Similarly, an AS might accept a recently expired assertion in order to help bootstrap a new session with a specific end user.

If the identified end user does not match the RO present at the AS during an interaction step and the AS is not explicitly allowing a cross-user authorization, the AS **SHOULD** reject the request with an `unknown_user` error (Section 3.6).

If the AS trusts the client instance to present verifiable assertions or known Subject Identifiers, such as an opaque identifier issued by the AS for this specific client instance, the AS **MAY** decide, based on its policy, to skip interaction with the RO, even if the client instance provides one or more interaction modes in its request.

See Section 11.30 for considerations for the AS when accepting and processing assertions from the client instance.

2.4.1. Identifying the User by Reference

The AS can identify the current end user to the client instance with a reference that can be used by the client instance to refer to the end user across multiple requests. If the client instance has a reference for the end user at this AS, the client instance **MAY** pass that reference as a string. The format of this string is opaque to the client instance.

```
"user": "XUT2MFM1XBIKJKSDU8QM"
```

One means of dynamically obtaining such a user reference is from the AS returning an `opaque` Subject Identifier as described in Section 3.4. Other means of configuring a client instance with a user identifier are out of scope of this specification. The lifetime and validity of these user references are determined by the AS, and this lifetime is not exposed to the client instance in GNAP. As such, a client instance using such a user reference is likely to keep using that reference until it stops working.

User reference identifiers are not intended to be human-readable user identifiers or structured assertions. For the client instance to send either of these, the client can use the full user request object (Section 2.4) instead.

If the AS does not recognize the user reference, it **MUST** return an `unknown_user` error (Section 3.6).

2.5. Interacting with the User

Often, the AS will require interaction with the RO (Section 4) in order to approve a requested delegation to the client instance for both access to resources and direct subject information. Many times, the end user using the client instance is the same person as the RO, and the client instance can directly drive interaction with the end user by facilitating the process through means such as redirection to a URI or launching an application. Other times, the client instance can provide information to start the RO's interaction on a secondary device, or the client instance will wait for the RO to approve the request asynchronously. The client instance could also be signaled that interaction has concluded through a callback mechanism.

The client instance declares the parameters for interaction methods that it can support using the `interact` field.

The `interact` field is a JSON object with three keys whose values declare how the client can initiate and complete the request, as well as provide hints to the AS about user preferences such as locale. A client instance **MUST NOT** declare an interaction mode it does not support. The client instance **MAY** send multiple modes in the same request. There is no preference order specified in this request. An AS **MAY** respond to any, all, or none of the presented interaction modes (Section 3.3) in a request, depending on its capabilities and what is allowed to fulfill the request.

`start` (array of objects/strings): Indicates how the client instance can start an interaction. **REQUIRED**. See Section 2.5.1.

`finish` (object): Indicates how the client instance can receive an indication that interaction has finished at the AS. **OPTIONAL**. See Section 2.5.2.

`hints` (object): Provides additional information to inform the interaction process at the AS. **OPTIONAL**. See Section 2.5.3.

In the following non-normative example, the client instance is indicating that it can redirect (Section 2.5.1.1) the end user to an arbitrary URI and can receive a redirect (Section 2.5.2.1) through a browser request. Note that the client instance does not accept a push-style callback. The pattern of using a redirect for both interaction start and finish is common for web-based client software.

```
"interact": {
    "start": ["redirect"],
    "finish": {
        "method": "redirect",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

In the following non-normative example, the client instance is indicating that it can display a user code (Section 2.5.1.3) and direct the end user to an arbitrary URI (Section 2.5.1.1), but it cannot accept a redirect or push-style callback. This pattern is common for devices that have robust display capabilities but expect the use of a secondary device to facilitate end-user interaction with the AS, such as a set-top box capable of displaying an interaction URL as a QR code.

```
"interact": {
    "start": ["redirect", "user_code"]
}
```

In the following non-normative example, the client instance is indicating that it cannot start any interaction with the end user but that the AS can push an interaction finish message (Section 2.5.2.2) when authorization from the RO is received asynchronously. This pattern is common for scenarios where a service needs to be authorized, but the RO is able to be contacted separately from the GNAP transaction itself, such as through a push notification or existing interactive session on a secondary device.

```
"interact": {
    "start": [],
    "finish": {
        "method": "push",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

If all of the following conditions are true, the AS **MUST** return an `invalid_interaction` error (Section 3.6) since the client instance will be unable to complete the request without authorization:

- The client instance does not provide a suitable interaction mechanism.
- The AS cannot contact the RO asynchronously.
- The AS determines that interaction is required.

2.5.1. Start Mode Definitions

If the client instance is capable of starting interaction with the end user, the client instance indicates this by sending an array of start modes under the `start` key. Each interaction start mode has a unique identifying name. Interaction start modes are specified in the array either by a string, which consists of the start mode name on its own, or by a JSON object with the required field `mode`:

`mode`: The interaction start mode. **REQUIRED**.

Interaction start modes defined as objects **MAY** define additional parameters to be required in the object.

The `start` array can contain both string-type and object-type modes.

This specification defines the following interaction start modes:

`"redirect"` (string): Indicates that the client instance can direct the end user to an arbitrary URI for interaction. See Section 2.5.1.1.

`"app"` (string): Indicates that the client instance can launch an application on the end user's device for interaction. See Section 2.5.1.2.

`"user_code"` (string): Indicates that the client instance can communicate a short, human-readable code to the end user for use with a stable URI. See Section 2.5.1.3.

`"user_code_uri"` (string): Indicates that the client instance can communicate a short, human-readable code to the end user for use with a short, dynamic URI. See Section 2.5.1.4.

Additional start modes can be defined in the "GNAP Interaction Start Modes" registry (Section 10.9).

2.5.1.1. Redirect to an Arbitrary URI

If the client instance is capable of directing the end user to a URI defined by the AS at runtime, the client instance indicates this by including `redirect` in the array under the `start` key. The means by which the client instance will activate this URI are out of scope of this specification, but common methods include an HTTP redirect, launching a browser on the end user's device, providing a scannable image encoding, and printing out a URI to an interactive console. While this URI is generally hosted at the AS, the client instance can make no assumptions about its contents, composition, or relationship to the grant endpoint URI.

```
"interact": {
  "start": ["redirect"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns a redirect interaction response (Section 3.3.1). The client instance manages this interaction method as described in Section 4.1.1.

See Section 11.29 for more considerations regarding the use of front-channel communication techniques.

2.5.1.2. Open an Application-Specific URI

If the client instance can open a URI associated with an application on the end user's device, the client instance indicates this by including `app` in the array under the `start` key. The means by which the client instance determines the application to open with this URI are out of scope of this specification.

```
"interact": {
  "start": ["app"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns an app interaction response with an app URI payload (Section 3.3.2). The client instance manages this interaction method as described in Section 4.1.4.

2.5.1.3. Display a Short User Code

If the client instance is capable of displaying or otherwise communicating a short, human-entered code to the RO, the client instance indicates this by including `user_code` in the array under the `start` key. This code is to be entered at a static URI that does not change at runtime. The client instance has no reasonable means to communicate a dynamic URI to the RO, so this URI is usually communicated out of band to the RO through documentation or other messaging outside of GNAP. While this URI is generally hosted at the AS, the client instance can make no assumptions about its contents, composition, or relationship to the grant endpoint URI.

```
"interact": {
    "start": ["user_code"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns a user code as specified in Section 3.3.3. The client instance manages this interaction method as described in Section 4.1.2.

2.5.1.4. Display a Short User Code and URI

If the client instance is capable of displaying or otherwise communicating a short, human-entered code along with a short, human-entered URI to the RO, the client instance indicates this by including `user_code_uri` in the array under the `start` key. This code is to be entered at the dynamic URL given in the response. While this URL is generally hosted at the AS, the client instance can make no assumptions about its contents, composition, or relationship to the grant endpoint URI.

```
"interact": {
    "start": ["user_code_uri"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns a user code and interaction URL as specified in Section 3.3.4. The client instance manages this interaction method as described in Section 4.1.3.

2.5.2. Interaction Finish Methods

If the client instance is capable of receiving a message from the AS indicating that the RO has completed their interaction, the client instance indicates this by sending the following members of an object under the `finish` key.

`method` (string): The callback method that the AS will use to contact the client instance. **REQUIRED**.

`uri` (string): Indicates the URI that the AS will use to signal the client instance that interaction has completed. This URI **MAY** be unique per request and **MUST** be hosted by or accessible to the client instance. This URI **MUST** be an absolute URI and **MUST NOT** contain any fragment

component. If the client instance needs any state information to tie to the front-channel interaction response, it **MUST** use a unique callback URI to link to that ongoing state. The allowable URIs and URI patterns **MAY** be restricted by the AS based on the client instance's presented key information. The callback URI **SHOULD** be presented to the RO during the interaction phase before redirect. **REQUIRED** for `redirect` and `push` methods.

`nonce` (string): Unique ASCII string value to be used in the calculation of the "hash" query parameter sent to the callback URI. It must be sufficiently random to be unguessable by an attacker. It **MUST** be generated by the client instance as a unique value for this request. **REQUIRED**.

`hash_method` (string): An identifier of a hash calculation mechanism to be used for the callback hash in Section 4.2.3, as defined in the IANA "Named Information Hash Algorithm Registry" [HASH-ALG]. If absent, the default value is `sha-256`. **OPTIONAL**.

This specification defines the following values for the `method` parameter; additional values can be defined in the "GNAP Interaction Finish Methods" registry (Section 10.10):

`"redirect"`: Indicates that the client instance can receive a redirect from the end user's device after interaction with the RO has concluded. See Section 2.5.2.1.

`"push"`: Indicates that the client instance can receive an HTTP POST request from the AS after interaction with the RO has concluded. See Section 2.5.2.2.

If interaction finishing is supported for this client instance and request, the AS will return a nonce (Section 3.3.5) used by the client instance to validate the callback. All interaction finish methods **MUST** use this nonce to allow the client to verify the connection between the pending interaction request and the callback. GNAP does this through the use of the interaction hash, defined in Section 4.2.3. All requests to the callback URI **MUST** be processed as described in Section 4.2.

All interaction finish methods **MUST** require presentation of an interaction reference for continuing this grant request. This means that the interaction reference **MUST** be returned by the AS and **MUST** be presented by the client as described in Section 5.1. The means by which the interaction reference is returned to the client instance are specific to the interaction finish method.

2.5.2.1. Receive an HTTP Callback through the Browser

A finish `method` value of `redirect` indicates that the client instance will expect a request from the RO's browser using the HTTP method GET as described in Section 4.2.1.

The client instance's URI **MUST** be protected by HTTPS, be hosted on a server local to the RO's browser ("localhost"), or use an application-specific URI scheme that is loaded on the end user's device.

```
"interact": {
    "finish": {
        "method": "redirect",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

Requests to the callback URI **MUST** be processed by the client instance as described in Section 4.2.1.

Since the incoming request to the callback URI is from the RO's browser, this method is usually used when the RO and end user are the same entity. See Section 11.24 for considerations on ensuring the incoming HTTP message matches the expected context of the request. See Section 11.29 for more considerations regarding the use of front-channel communication techniques.

2.5.2.2. Receive an HTTP Direct Callback

A finish `method` value of `push` indicates that the client instance will expect a request from the AS directly using the HTTP method POST as described in Section 4.2.2.

The client instance's URI **MUST** be protected by HTTPS, be hosted on a server local to the RO's browser ("localhost"), or use an application-specific URI scheme that is loaded on the end user's device.

```
"interact": {
    "finish": {
        "method": "push",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

Requests to the callback URI **MUST** be processed by the client instance as described in Section 4.2.2.

Since the incoming request to the callback URI is from the AS and not from the RO's browser, this request is not expected to have any shared session information from the start method. See Sections 11.24 and 11.23 for more considerations regarding the use of back-channel and polling mechanisms like this.

2.5.3. Hints

The `hints` key is an object describing one or more suggestions from the client instance that the AS can use to help drive user interaction.

This specification defines the following property under the `hints` key:

`ui_locales` (array of strings): Indicates the end user's preferred locales that the AS can use during interaction, particularly before the RO has authenticated. **OPTIONAL**. Section 2.5.3.1

The following subsection details requests for interaction hints. Additional interaction hints can be defined in the "GNAP Interaction Hints" registry (Section 10.11).

2.5.3.1. Indicate Desired Interaction Locales

If the client instance knows the end user's locale and language preferences, the client instance can send this information to the AS using the `ui_locales` field with an array of locale strings as defined by [RFC5646].

```
"interact": {
    "hints": {
        "ui_locales": ["en-US", "fr-CA"]
    }
}
```

If possible, the AS **SHOULD** use one of the locales in the array, with preference to the first item in the array supported by the AS. If none of the given locales are supported, the AS **MAY** use a default locale.

3. Grant Response

In response to a client instance's request, the AS responds with a JSON object as the HTTP content. Each possible field is detailed in the subsections below.

`continue` (object): Indicates that the client instance can continue the request by making one or more continuation requests. **REQUIRED** if continuation calls are allowed for this client instance on this grant request. See Section 3.1.

`access_token` (object / array of objects): A single access token or set of access tokens that the client instance can use to call the RS on behalf of the RO. **REQUIRED** if an access token is included. See Section 3.2.

`interact` (object): Indicates that interaction through some set of defined mechanisms needs to take place. **REQUIRED** if interaction is expected. See Section 3.3.

`subject` (object): Claims about the RO as known and declared by the AS. **REQUIRED** if subject information is included. See Section 3.4.

`instance_id` (string): An identifier this client instance can use to identify itself when making future requests. **OPTIONAL**. See Section 3.5.

`error` (object or string): An error code indicating that something has gone wrong. **REQUIRED** for an error condition. See Section 3.6.

Additional fields can be defined by extensions to GNAP in the "GNAP Grant Response Parameters" registry (Section 10.12).

In the following non-normative example, the AS is returning an interaction URI (Section 3.3.1), a callback nonce (Section 3.3.5), and a continuation response (Section 3.1).

```
NOTE: '\' line wrapping per RFC 8792

{
    "interact": {
        "redirect": "https://server.example.com/interact/4CF492ML\
          VMSW9MKMXKHQ",
        "finish": "MBDOFXG4Y5CVJCX821LH"
    },
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU",
        },
        "uri": "https://server.example.com/tx"
    }
}
```

In the following non-normative example, the AS is returning a bearer access token (Section 3.2.1) with a management URI and a Subject Identifier (Section 3.4) in the form of an opaque identifier.

```
{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "flags": ["bearer"],
        "manage": {
            "uri": "https://server.example.com/token/PRY5NM33O",
            "access_token": {
                "value": "B8CDFONP21-4TB8N6.BW7ONM"
            }
        }
    },
    "subject": {
        "sub_ids": [ {
          "format": "opaque",
          "id": "J2G8G8O4AZ"
        } ]
    }
}
```

In the following non-normative example, the AS is returning set of Subject Identifiers (Section 3.4), simultaneously as an opaque identifier, an email address, and a decentralized identifier (DID), formatted as a set of Subject Identifiers as defined in [RFC9493].

```
{
    "subject": {
        "sub_ids": [ {
          "format": "opaque",
          "id": "J2G8G8O4AZ"
        }, {
          "format": "email",
          "email": "user@example.com"
        }, {
          "format": "did",
          "url": "did:example:123456"
        } ]
    }
}
```

The response **MUST** be sent as a JSON object in the content of the HTTP response with Content-Type `application/json`, unless otherwise specified by the specific response (e.g., an empty response with no Content-Type).

The AS **MUST** include the HTTP Cache-Control response header field [RFC9111] with a value set to "no-store".

3.1. Request Continuation

If the AS determines that the grant request can be continued by the client instance, the AS responds with the `continue` field. This field contains a JSON object with the following properties.

`uri` (string): The URI at which the client instance can make continuation requests. This URI **MAY** vary per request or **MAY** be stable at the AS. This URI **MUST** be an absolute URI. The client instance **MUST** use this value exactly as given when making a continuation request (Section 5). **REQUIRED**.

`wait` (integer): The amount of time in integer seconds the client instance **MUST** wait after receiving this request continuation response and calling the continuation URI. The value **SHOULD NOT** be less than five seconds, and omission of the value **MUST** be interpreted as five seconds. **RECOMMENDED**.

`access_token` (object): A unique access token for continuing the request, called the "continuation access token". The value of this property **MUST** be an object in the format specified in Section 3.2.1. This access token **MUST** be bound to the client instance's key used in the request and **MUST NOT** be a bearer token. As a consequence, the `flags` array of this access token **MUST NOT** contain the string `bearer`, and the `key` field **MUST** be omitted. This access token **MUST NOT** have a `manage` field. The client instance **MUST** present the continuation access token in all requests to the continuation URI as described in Section 7.2. **REQUIRED**.

```
{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 60
    }
}
```

This field is **REQUIRED** if the grant request is in the *pending* state, as the field contains the information needed by the client request to continue the request as described in Section 5. Note that the continuation access token is bound to the client instance's key; therefore, the client instance **MUST** sign all continuation requests with its key as described in Section 7.3 and **MUST** present the continuation access token in its continuation request.

3.2. Access Tokens

If the AS has successfully granted one or more access tokens to the client instance, the AS responds with the `access_token` field. This field contains either a single access token as described in Section 3.2.1 or an array of access tokens as described in Section 3.2.2.

The client instance uses any access tokens in this response to call the RS as described in Section 7.2.

The grant request **MUST** be in the *approved* state to include this field in the response.

3.2.1. Single Access Token

If the client instance has requested a single access token and the AS has granted that access token, the AS responds with the "access_token" field. The value of this field is an object with the following properties.

`value` (string): The value of the access token as a string. The value is opaque to the client instance. The value **MUST** be limited to the `token68` character set defined in Section 11.2 of [HTTP] to facilitate transmission over HTTP headers and within other protocols without requiring additional encoding. **REQUIRED**.

`label` (string): The value of the `label` the client instance provided in the associated token request (Section 2.1), if present. **REQUIRED** for multiple access tokens or if a `label` was included in the single access token request; **OPTIONAL** for a single access token where no `label` was included in the request.

`manage` (object): Access information for the token management API for this access token. If provided, the client instance **MAY** manage its access token as described in Section 6. This management API is a function of the AS and is separate from the RS the client instance is requesting access to. **OPTIONAL**.

`access` (array of objects/strings): A description of the rights associated with this access token, as defined in Section 8. If included, this **MUST** reflect the rights associated with the issued access token. These rights **MAY** vary from what was requested by the client instance. **REQUIRED**.

`expires_in` (integer): The number of seconds in which the access will expire. The client instance **MUST NOT** use the access token past this time. Note that the access token **MAY** be revoked by the AS or RS at any point prior to its expiration. **OPTIONAL**.

`key` (object / string): The key that the token is bound to, if different from the client instance's presented key. The key **MUST** be an object or string in a format described in Section 7.1. The client instance **MUST** be able to dereference or process the key information in order to be able to sign subsequent requests using the access token (Section 7.2). When the key is provided by value from the AS, the token shares some security properties with bearer tokens as discussed in Section 11.38. It is **RECOMMENDED** that keys returned for use with access tokens be key references as described in Section 7.1.1 that the client instance can correlate to its known keys. **OPTIONAL**.

`flags` (array of strings): A set of flags that represent attributes or behaviors of the access token issued by the AS. **OPTIONAL**.

The value of the `manage` field is an object with the following properties:

`uri` (string): The URI of the token management API for this access token. This URI **MUST** be an absolute URI. This URI **MUST NOT** include the value of the access token being managed or the value of the access token used to protect the URI. This URI **SHOULD** be different for each access token issued in a request. **REQUIRED**.

`access_token` (object): A unique access token for continuing the request, called the "token management access token". The value of this property **MUST** be an object in the format specified in Section 3.2.1. This access token **MUST** be bound to the client instance's key used in the request (or its most recent rotation) and **MUST NOT** be a bearer token. As a consequence, the `flags` array of this access token **MUST NOT** contain the string `bearer`, and the `key` field **MUST** be omitted. This access token **MUST NOT** have a `manage` field. This access token **MUST NOT** have the same value as the token it is managing. The client instance **MUST** present the continuation access token in all requests to the continuation URI as described in Section 7.2. **REQUIRED**.

The values of the `flags` field defined by this specification are as follows:

`"bearer"`: Flag indicating whether the token is a bearer token, not bound to a key and proofing mechanism. If the `bearer` flag is present, the access token is a bearer token, and the `key` field in this response **MUST** be omitted. See Section 11.9 for additional considerations on the use of bearer tokens.

`"durable"`: Flag indicating a hint of AS behavior on token rotation. If this flag is present, then the client instance can expect a previously issued access token to continue to work after it has been rotated (Section 6.1) or the underlying grant request has been modified (Section 5.3),

resulting in the issuance of new access tokens. If this flag is omitted, the client instance can anticipate a given access token could stop working after token rotation or grant request modification. Note that a token flagged as `durable` can still expire or be revoked through any normal means.

Flag values **MUST NOT** be included more than once.

Additional flags can be defined by extensions using the "GNAP Access Token Flags" registry (Section 10.4).

If the `bearer` flag and the `key` field in this response are omitted, the token is bound to the key used by the client instance (Section 2.3) in its request for access. If the `bearer` flag is omitted and the `key` field is present, the token is bound to the key and proofing mechanism indicated in the `key` field. The means by which the AS determines how to bind an access token to a key other than that presented by the client instance are out of scope for this specification, but common practices include pre-registering specific keys in a static fashion.

The client software **MUST** reject any access token where the `flags` field contains the `bearer` flag and the `key` field is present with any value.

The following non-normative example shows a single access token bound to the client instance's key used in the initial request. The access token has a management URI and has access to three described resources (one using an object and two described by reference strings).

```
NOTE: '\' line wrapping per RFC 8792

"access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "manage": {
        "uri": "https://server.example.com/token/PRY5NM33O",
        "access_token": {
            "value": "B8CDFONP21-4TB8N6.BW7ONM"
        }
    },
    "access": [
        {
            "type": "photo-api",
            "actions": [
                "read",
                "write",
                "dolphin"
            ],
            "locations": [
                "https://server.example.net/",
                "https://resource.local/other"
            ],
            "datatypes": [
                "metadata",
                "images"
            ]
        },
        "read", "dolphin-metadata"
    ]
}
```

The following non-normative example shows a single bearer access token with access to two described resources.

```
"access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "flags": ["bearer"],
    "access": [
        "finance", "medical"
    ]
}
```

If the client instance requested a single access token (Section 2.1.1), the AS **MUST NOT** respond with the structure for multiple access tokens.

3.2.2. Multiple Access Tokens

If the client instance has requested multiple access tokens and the AS has granted at least one of them, the AS responds with the "access_token" field. The value of this field is a JSON array, the members of which are distinct access tokens as described in Section 3.2.1. Each object **MUST** have a unique `label` field, corresponding to the token labels chosen by the client instance in the request for multiple access tokens (Section 2.1.2).

In the following non-normative example, two tokens are issued under the names `token1` and `token2`, and only the first token has a management URI associated with it.

```
NOTE: '\' line wrapping per RFC 8792

"access_token": [
    {
        "label": "token1",
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": {
            "uri": "https://server.example.com/token/PRY5NM33O",
            "access_token": {
                "value": "B8CDFONP21-4TB8N6.BW7ONM"
            }
        },
        "access": [ "finance" ]
    },
    {
        "label": "token2",
        "value": "UFGLO2FDAFG7VGZZPJ3IZEMN21EVU71FHCARP4J1",
        "access": [ "medical" ]
    }
}
```

Each access token corresponds to one of the objects in the `access_token` array of the client instance's request (Section 2.1.2).

The AS **MAY** refuse to issue one or more of the requested access tokens for any reason. In such cases, the refused token is omitted from the response, and all of the other issued access tokens are included in the response under their respective requested labels. If the client instance requested multiple access tokens (Section 2.1.2), the AS **MUST NOT** respond with a single access token structure, even if only a single access token is granted. In such cases, the AS **MUST** respond with a structure for multiple access tokens containing one access token.

```
"access_token": [
    {
        "label": "token2",
        "value": "8N6BW7OZB8CDFONP219-OS9M2PMHKUR64TBRP1LT0",
        "manage": {
            "uri": "https://server.example.com/token/PRY5NM33O",
            "access_token": {
                "value": "B8CDFONP21-4TB8N6.BW7ONM"
            }
        },
        "access": [ "fruits" ]
    }
]
```

The parameters of each access token are separate. For example, each access token is expected to have a unique value and (if present) label, and each access token likely has different access rights associated with it. Each access token could also be bound to different keys with different proofing mechanisms.

3.3. Interaction Modes

If the client instance has indicated a capability to interact with the RO in its request (Section 2.5) and the AS has determined that interaction is both supported and necessary, the AS responds to the client instance with any of the following values in the `interact` field of the response. There is no preference order for interaction modes in the response, and it is up to the client instance to determine which ones to use. All supported interaction methods are included in the same `interact` object.

`redirect` (string): Redirect to an arbitrary URI. **REQUIRED** if the `redirect` interaction start mode is possible for this request. See Section 3.3.1.

`app` (string): Launch of an application URI. **REQUIRED** if the `app` interaction start mode is possible for this request. See Section 3.3.2.

`user_code` (string): Display a short user code. **REQUIRED** if the `user_code` interaction start mode is possible for this request. See Section 3.3.3.

`user_code_uri` (object): Display a short user code and URI. **REQUIRED** if the `user_code_uri` interaction start mode is possible for this request. Section 3.3.4

`finish` (string): A unique ASCII string value provided by the AS as a nonce. This is used by the client instance to verify the callback after interaction is completed. **REQUIRED** if the interaction finish method requested by the client instance is possible for this request. See Section 3.3.5.

`expires_in` (integer): The number of integer seconds after which this set of interaction responses will expire and no longer be usable by the client instance. If the interaction methods expire, the client **MAY** restart the interaction process for this grant request by sending an update (Section 5.3) with a new interaction request field (Section 2.5). **OPTIONAL**. If omitted, the interaction response modes returned do not expire but **MAY** be invalidated by the AS at any time.

Additional interaction mode responses can be defined in the "GNAP Interaction Mode Responses" registry (Section 10.13).

The AS **MUST NOT** respond with any interaction mode that the client instance did not indicate in its request, and the AS **MUST NOT** respond with any interaction mode that the AS does not support. Since interaction responses include secret or unique information, the AS **SHOULD** respond to each interaction mode only once in an ongoing request, particularly if the client instance modifies its request (Section 5.3).

The grant request **MUST** be in the *pending* state to include this field in the response.

3.3.1. Redirection to an Arbitrary URI

If the client instance indicates that it can redirect to an arbitrary URI (Section 2.5.1.1) and the AS supports this mode for the client instance's request, the AS responds with the "redirect" field, which is a string containing the URI for the end user to visit. This URI **MUST** be unique for the request and **MUST NOT** contain any security-sensitive information such as user identifiers or access tokens.

```
"interact": {
    "redirect": "https://interact.example.com/4CF492MLVMSW9MKMXKHQ"
}
```

The URI returned is a function of the AS, but the URI itself **MAY** be completely distinct from the grant endpoint URI that the client instance uses to request access (Section 2), allowing an AS to separate its user-interaction functionality from its backend security functionality. The AS will need to dereference the specific grant request and its information from the URI alone. If the AS does not directly host the functionality accessed through the redirect URI, then the means for the interaction functionality to communicate with the rest of the AS are out of scope for this specification.

The client instance sends the end user to the URI to interact with the AS. The client instance **MUST NOT** alter the URI in any way. The means for the client instance to send the end user to this URI are out of scope of this specification, but common methods include an HTTP redirect, launching the system browser, displaying a scannable code, or printing out the URI in an interactive console. See details of the interaction in Section 4.1.1.

3.3.2. Launch of an Application URI

If the client instance indicates that it can launch an application URI (Section 2.5.1.2) and the AS supports this mode for the client instance's request, the AS responds with the "app" field, which is a string containing the URI for the client instance to launch. This URI **MUST** be unique for the request and **MUST NOT** contain any security-sensitive information such as user identifiers or access tokens.

```
"interact": {
    "app": "https://app.example.com/launch?tx=4CF492MLV"
}
```

The means for the launched application to communicate with the AS are out of scope for this specification.

The client instance launches the URI as appropriate on its platform; the means for the client instance to launch this URI are out of scope of this specification. The client instance **MUST NOT** alter the URI in any way. The client instance **MAY** attempt to detect if an installed application will service the URI being sent before attempting to launch the application URI. See details of the interaction in Section 4.1.4.

3.3.3. Display of a Short User Code

If the client instance indicates that it can display a short, user-typeable code (Section 2.5.1.3) and the AS supports this mode for the client instance's request, the AS responds with a "user_code" field. This field is string containing a unique short code that the user can type into a web page. To facilitate usability, this string **MUST** consist only of characters that can be easily typed by the end user (such as ASCII letters or numbers) and **MUST** be processed by the AS in a case-insensitive manner (see Section 4.1.2). The string **MUST** be randomly generated so as to be unguessable by an attacker within the time it is accepted. The time in which this code will be accepted **SHOULD** be short lived, such as several minutes. It is **RECOMMENDED** that this code be between six and eight characters in length.

```
"interact": {
    "user_code": "A1BC3DFF"
}
```

The client instance **MUST** communicate the "user_code" value to the end user in some fashion, such as displaying it on a screen or reading it out audibly. This code is used by the interaction component of the AS as a means of identifying the pending grant request and does not function as an authentication factor for the RO.

The URI that the end user is intended to enter the code into **MUST** be stable, since the client instance is expected to have no means of communicating a dynamic URI to the end user at runtime.

As this interaction mode is designed to facilitate interaction via a secondary device, it is not expected that the client instance redirect the end user to the URI where the code is entered. If the client instance is capable of communicating a short arbitrary URI to the end user for use with the user code, the client instance **SHOULD** instead use the "user_code_uri" mode (Section 2.5.1.4). If the client instance is capable of communicating a long arbitrary URI to the end user, such as through a scannable code, the client instance **SHOULD** use the "redirect" mode (Section 2.5.1.1) for this purpose, instead of or in addition to the user code mode.

See details of the interaction in Section 4.1.2.

3.3.4. Display of a Short User Code and URI

If the client instance indicates that it can display a short, user-typeable code (Section 2.5.1.3) and the AS supports this mode for the client instance's request, the AS responds with a "user_code_uri" object that contains the following members.

`code` (string): A unique short code that the end user can type into a provided URI. To facilitate usability, this string **MUST** consist only of characters that can be easily typed by the end user (such as ASCII letters or numbers) and **MUST** be processed by the AS in a case-insensitive manner (see Section 4.1.3). The string **MUST** be randomly generated so as to be unguessable by an attacker within the time it is accepted. The time in which this code will be accepted **SHOULD** be short lived, such as several minutes. It is **RECOMMENDED** that this code be between six and eight characters in length. **REQUIRED**.

`uri` (string): The interaction URI that the client instance will direct the RO to. This URI **MUST** be short enough to be communicated to the end user by the client instance. It is **RECOMMENDED** that this URI be short enough for an end user to type in manually. The URI **MUST NOT** contain the `code` value. This URI **MUST** be an absolute URI. **REQUIRED**.

```
"interact": {
    "user_code_uri": {
        "code": "A1BC3DFF",
        "uri": "https://s.example/device"
    }
}
```

The client instance **MUST** communicate the "code" to the end user in some fashion, such as displaying it on a screen or reading it out audibly. This code is used by the interaction component of the AS as a means of identifying the pending grant request and does not function as an authentication factor for the RO.

The client instance **MUST** also communicate the URI to the end user. Since it is expected that the end user will continue interaction on a secondary device, the URI needs to be short enough to allow the end user to type or copy it to a secondary device without mistakes.

The URI returned is a function of the AS, but the URI itself **MAY** be completely distinct from the grant endpoint URI that the client instance uses to request access (Section 2), allowing an AS to separate its user-interaction functionality from its backend security functionality. If the AS does not directly host the functionality accessed through the given URI, then the means for the interaction functionality to communicate with the rest of the AS are out of scope for this specification.

See details of the interaction in Section 4.1.2.

3.3.5. Interaction Finish

If the client instance indicates that it can receive a post-interaction redirect or push at a URI (Section 2.5.2) and the AS supports this mode for the client instance's request, the AS responds with a `finish` field containing a nonce that the client instance will use in validating the callback as defined in Section 4.2.

```
"interact": {
    "finish": "MBDOFXG4Y5CVJCX821LH"
}
```

When the interaction is completed, the interaction component of the AS **MUST** contact the client instance using the means defined by the finish method as described in Section 4.2.

If the AS returns the `finish` field, the client instance **MUST NOT** continue a grant request before it receives the associated interaction reference on the callback URI. See details in Section 4.2.

3.4. Returning Subject Information

If information about the RO is requested and the AS grants the client instance access to that data, the AS returns the approved information in the "subject" response field. The AS **MUST** return the `subject` field only in cases where the AS is sure that the RO and the end user are the same party. This can be accomplished through some forms of interaction with the RO (Section 4).

This field is an object with the following properties.

`sub_ids` (array of objects): An array of Subject Identifiers for the RO, as defined by [RFC9493]. **REQUIRED** if returning Subject Identifiers.

`assertions` (array of objects): An array containing assertions as objects, each containing the assertion object described below. **REQUIRED** if returning assertions.

`updated_at` (string): Timestamp as a date string as described in [RFC3339], indicating when the identified account was last updated. The client instance **MAY** use this value to determine if it needs to request updated profile information through an identity API. The definition of such an identity API is out of scope for this specification. **RECOMMENDED**.

Assertion objects contain the following fields:

`format` (string): The assertion format. Possible formats are listed in Section 3.4.1. Additional assertion formats can be defined in the "GNAP Assertion Formats" registry (Section 10.6). **REQUIRED**.

`value` (string): The assertion value as the JSON string serialization of the assertion. **REQUIRED**.

The following non-normative example contains an opaque identifier and an OpenID Connect ID Token:

```
"subject": {
  "sub_ids": [ {
    "format": "opaque",
    "id": "XUT2MFM1XBIKJKSDU8QM"
  } ],
  "assertions": [ {
    "format": "id_token",
    "value": "eyj..."
  } ]
}
```

Subject Identifiers returned by the AS **SHOULD** uniquely identify the RO at the AS. Some forms of Subject Identifiers are opaque to the client instance (such as the subject of an issuer and subject pair), while other forms (such as email address and phone number) are intended to allow the client instance to correlate the identifier with other account information at the client instance. The client instance **MUST NOT** request or use any returned Subject Identifiers for communication purposes (see Section 2.2). That is, a Subject Identifier returned in the format of an email address or a phone number only identifies the RO to the AS and does not indicate that the AS has validated that the represented email address or phone number in the identifier is suitable for communication with the current user. To get such information, the client instance **MUST** use an identity protocol to request and receive additional identity claims. The details of an identity protocol and associated schema are outside the scope of this specification.

The AS **MUST** ensure that the returned subject information represents the RO. In most cases, the AS will also ensure that the returned subject information represents the end user authenticated interactively at the AS. The AS **SHOULD NOT** reuse Subject Identifiers for multiple different ROs.

The "sub_ids" and "assertions" response fields are independent of each other. That is, a returned assertion **MAY** use a different Subject Identifier than other assertions and Subject Identifiers in the response. However, all Subject Identifiers and assertions returned **MUST** refer to the same party.

The client instance **MUST** interpret all subject information in the context of the AS from which the subject information is received, as is discussed in Section 6 of [SP80063C]. For example, one AS could return an email identifier of "user@example.com" for one RO, and a different AS could return that same email identifier of "user@example.com" for a completely different RO. A client instance talking to both ASes needs to differentiate between these two accounts by accounting for the AS source of each identifier and not assuming that either has a canonical claim on the identifier without additional configuration and trust agreements. Otherwise, a rogue AS could exploit this to take over a targeted account asserted by a different AS.

Extensions to this specification **MAY** define additional response properties in the "GNAP Subject Information Response Fields" registry (Section 10.14).

The grant request **MUST** be in the *approved* state to return this field in the response.

See Section 11.30 for considerations that the client instance has to make when accepting and processing assertions from the AS.

3.4.1. Assertion Formats

The following assertion formats are defined in this specification:

`id_token`: An OpenID Connect ID Token [OIDC], in JSON Web Token (JWT) compact format as a single string.

`saml2`: A SAML 2.0 assertion [SAML2], encoded as a single base64url string with no padding.

3.5. Returning a Dynamically Bound Client Instance Identifier

Many parts of the client instance's request can be passed as either a value or a reference. The use of a reference in place of a value allows for a client instance to optimize requests to the AS.

Some references, such as for the client instance's identity (Section 2.3.1) or the requested resources (Section 8.1), can be managed statically through an admin console or developer portal provided by the AS or RS. The developer of the client software can include these values in their code for a more efficient and compact request.

If desired, the AS **MAY** also generate and return an instance identifier dynamically to the client instance in the response to facilitate multiple interactions with the same client instance over time. The client instance **SHOULD** use this instance identifier in future requests in lieu of sending the associated data values in the `client` field.

Dynamically generated client instance identifiers are string values that **MUST** be protected by the client instance as secrets. Instance identifier values **MUST** be unguessable and **MUST NOT** contain any information that would compromise any party if revealed. Instance identifier values are opaque to the client instance, and their content is determined by the AS. The instance identifier **MUST** be unique per client instance at the AS.

`instance_id` (string): A string value used to represent the information in the `client` object that the client instance can use in a future request, as described in Section 2.3.1. **OPTIONAL**.

The following non-normative example shows an instance identifier alongside an issued access token.

```
{
    "instance_id": "7C7C4AZ9KHRS6X63AJAO",
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0"
    }
}
```

3.6. Error Response

If the AS determines that the request cannot be completed for any reason, it responds to the client instance with an `error` field in the response message. This field is either an object or a string.

When returned as an object, the object contains the following fields:

`code` (string): A single ASCII error code defining the error. The value **MUST** be defined in the "GNAP Error Codes" registry (Section 10.15). **REQUIRED**.

`description` (string): A human-readable string description of the error intended for the developer of the client. The value is chosen by the implementation. **OPTIONAL**.

This specification defines the following `code` values:

`"invalid_request"`: The request is missing a required parameter, includes an invalid parameter value, or is otherwise malformed.

`"invalid_client"`: The request was made from a client that was not recognized or allowed by the AS, or the client's signature validation failed.

`"invalid_interaction"`: The client instance has provided an interaction reference that is incorrect for this request, or the interaction modes in use have expired.

`"invalid_flag"`: The flag configuration is not valid.

`"invalid_rotation"`: The token rotation request is not valid.

`"key_rotation_not_supported"`: The AS does not allow rotation of this access token's key.

`"invalid_continuation"`: The continuation of the referenced grant could not be processed.

`"user_denied"`: The RO denied the request.

`"request_denied"`: The request was denied for an unspecified reason.

`"unknown_user"`: The user presented in the request is not known to the AS or does not match the user present during interaction.

`"unknown_interaction"`: The interaction integrity could not be established.

`"too_fast"`: The client instance did not respect the timeout in the wait response before the next call.

`"too_many_attempts"`: A limit has been reached in the total number of reasonable attempts. This number is either defined statically or adjusted based on runtime conditions by the AS.

Additional error codes can be defined in the "GNAP Error Codes" registry (Section 10.15).

For example, if the RO denied the request while interacting with the AS, the AS would return the following error when the client instance tries to continue the grant request:

```
{
    "error": {
        "code": "user_denied",
        "description": "The RO denied the request"
    }
}
```

Alternatively, the AS **MAY** choose to only return the error as codes and provide the error as a string. Since the `description` field is not intended to be machine-readable, the following response is considered functionally equivalent to the previous example for the purposes of the client software's understanding:

```
{
    "error": "user_denied"
}
```

If an error state is reached but the grant is in the *pending* state (and therefore the client instance can continue), the AS **MAY** include the `continue` field in the response along with the `error`, as defined in Section 3.1. This allows the client instance to modify its request for access, potentially leading to prompting the RO again. Other fields **MUST NOT** be included in the response.

4. Determining Authorization and Consent

When the client instance makes its initial request (Section 2) to the AS for delegated access, it is capable of asking for several different kinds of information in response:

- the access being requested, in the `access_token` request parameter
- the subject information being requested, in the `subject` request parameter
- any additional requested information defined by extensions of this protocol

When the grant request is in the *processing* state, the AS determines what authorizations and consents are required to fulfill this requested delegation. The details of how the AS makes this determination are out of scope for this document. However, there are several common patterns defined and supported by GNAP for fulfilling these requirements, including information sent by the client instance, information gathered through the interaction process, and information supplied by external parties. An individual AS can define its own policies and processes for deciding when and how to gather the necessary authorizations and consent and how those are applied to the grant request.

To facilitate the AS fulfilling this request, the client instance sends information about the actions the client software can take, including:

- starting interaction with the end user, in the `interact` request parameter
- receiving notification that interaction with the RO has concluded, in the `interact` request parameter
- any additional capabilities defined by extensions of this protocol

The client instance can also supply information directly to the AS in its request. The client instance can send several kinds of things, including:

- the identity of the client instance, known from the keys or identifiers in the `client` request parameter
- the identity of the end user, in the `user` request parameter
- any additional information presented by the client instance in the request defined by extensions of this protocol

The AS will process this presented information in the context of the client instance's request and can only trust the information as much as it trusts the presentation and context of that request. If the AS determines that the information presented in the initial request is sufficient for granting the requested access, the AS **MAY** move the grant request to the *approved* state and return results immediately in its response (Section 3) with access tokens and subject information.

If the AS determines that additional runtime authorization is required, the AS can either deny the request outright (if there is no possible recovery) or move the grant request to the *pending* state and use a number of means at its disposal to gather that authorization from the appropriate ROs, including:

- starting interaction with the end user facilitated by the client software, such as a redirection or user code
- challenging the client instance through a challenge-response mechanism
- requesting that the client instance present specific additional information, such as a user's credential or an assertion
- contacting an RO through an out-of-band mechanism, such as a push notification
- executing an auxiliary software process through an out-of-band mechanism, such as querying a digital wallet

The process of gathering authorization and consent in GNAP is left deliberately flexible to allow for a wide variety of different deployments, interactions, and methodologies. In this process, the AS can gather consent from the RO or apply the RO's policy as necessitated by the access that has been requested. The AS can sometimes determine which RO needs to prompt for consent based on what has been requested by the client instance, such as a specific RS record, an identified subject, or a request requiring specific access such as approval by an administrator. In other cases, the request is applied to whichever RO is present at the time of consent gathering. This

pattern is especially prevalent when the end user is sent to the AS for an interactive session, during which the end user takes on the role of the RO. In these cases, the end user is delegating their own access as RO to the client instance.

The client instance can indicate that it is capable of facilitating interaction with the end user, another party, or another piece of software through its interaction start request (Section 2.5.1). Here, the AS usually needs to interact directly with the end user to determine their identity, determine their status as an RO, and collect their consent. If the AS has determined that authorization is required and the AS can support one or more of the requested interaction start methods, the AS returns the associated interaction start responses (Section 3.3). The client instance **SHOULD** initiate one or more of these interaction methods (Section 4.1) in order to facilitate the granting of the request. If more than one interaction start method is available, the means by which the client chooses which methods to follow are out of scope of this specification.

After starting interaction, the client instance can then make a continuation request (Section 5) either in response to a signal indicating the finish of the interaction (Section 4.2), after a time-based polling, or through some other method defined by an extension of this specification through the "GNAP Interaction Mode Responses" registry (Section 10.13).

If the grant request is not in the *approved* state, the client instance can repeat the interaction process by sending a grant update request (Section 5.3) with new interaction methods (Section 2.5).

The client instance **MUST** use each interaction method once at most if a response can be detected. The AS **MUST** handle any interact request as a one-time-use mechanism and **SHOULD** apply suitable timeouts to any interaction start methods provided, including user codes and redirection URIs. The client instance **SHOULD** apply suitable timeouts to any interaction finish method.

In order to support client software deployed in disadvantaged network conditions, the AS **MAY** allow for processing of the same interaction method multiple times if the AS can determine that the request is from the same party and the results are idempotent. For example, if a client instance launches a redirect to the AS but does not receive a response within a reasonable time, the client software can launch the redirect again, assuming that it never reached the AS in the first place. However, if the AS in question receives both requests, it could mistakenly process them separately, creating an undefined state for the client instance. If the AS can determine that both requests come from the same origin or under the same session, and the requests both came before any additional state change to the grant occurs, the AS can reasonably conclude that the initial response was not received and the same response can be returned to the client instance.

If the AS instead has a means of contacting the RO directly, it could do so without involving the client instance in its consent-gathering process. For example, the AS could push a notification to a known RO and have the RO approve the pending request asynchronously. These interactions can be through an interface of the AS itself (such as a hosted web page), through another application (such as something installed on the RO's device), through a messaging fabric, or any other means.

When interacting with an RO, the AS can use various strategies to determine the authorization of the requested grant, including:

- authenticate the RO through a local account or some other means, such as federated login
- validate the RO through presentation of claims, attributes, or other information
- prompt the RO for consent for the requested delegation
- describe to the RO what information is being released, to whom, and for what purpose
- provide warnings to the RO about potential attacks or negative effects of allowing the information
- allow the RO to modify the client instance's requested access, including limiting or expanding that access
- provide the RO with artifacts such as receipts to facilitate an audit trail of authorizations
- allow the RO to deny the requested delegation

The AS is also allowed to request authorization from more than one RO, if the AS deems fit. For example, a medical record might need to be released by both an attending nurse and a physician, or both owners of a bank account need to sign off on a transfer request. Alternatively, the AS could require N of M possible ROs to approve a given request. In some circumstances, the AS could even determine that the end user present during the interaction is not the appropriate RO for a given request and reach out to the appropriate RO asynchronously.

The RO is also allowed to define an automated policy at the AS to determine which kind of end user can get access to the resource and under which conditions. For instance, such a condition might require the end user to log in and accept the RO's legal provisions. Alternatively, client software could be acting without an end user, and the RO's policy allows issuance of access tokens to specific instances of that client software without human interaction.

While all of these cases are supported by GNAP, the details of their implementation and the methods for determining which ROs or related policies are required for a given request are out of scope for this specification.

4.1. Starting Interaction with the End User

When a grant request is in the *pending* state, the interaction start methods sent by the client instance can be used to facilitate interaction with the end user. To initiate an interaction start method indicated by the interaction start responses (Section 3.3) from the AS, the client instance follows the steps defined by that interaction start mode. The actions of the client instance required for the interaction start modes defined in this specification are described in the following subsections. Interaction start modes defined in extensions to this specification **MUST** define the expected actions of the client software when that interaction start mode is used.

If the client instance does not start an interaction start mode within an AS-determined amount of time, the AS **MUST** reject attempts to use the interaction start modes. If the client instance has already begun one interaction start mode and the interaction has been successfully completed, the AS **MUST** reject attempts to use other interaction start modes. For example, if a user code has

been successfully entered for a grant request, the AS will need to reject requests to an arbitrary redirect URI on the same grant request in order to prevent an attacker from capturing and altering an active authorization process.

4.1.1. Interaction at a Redirected URI

When the end user is directed to an arbitrary URI through the "redirect" mode (Section 3.3.1), the client instance facilitates opening the URI through the end user's web browser. The client instance could launch the URI through the system browser, provide a clickable link, redirect the user through HTTP response codes, or display the URI in a form the end user can use to launch, such as a multidimensional barcode. In all cases, the URI is accessed with an HTTP GET request, and the resulting page is assumed to allow direct interaction with the end user through an HTTP user agent. With this method, it is common (though not required) for the RO to be the same party as the end user, since the client instance has to communicate the redirection URI to the end user.

In many cases, the URI indicates a web page hosted at the AS, allowing the AS to authenticate the end user as the RO and interactively provide consent. The URI value is used to identify the grant request being authorized. If the URI cannot be associated with a currently active request, the AS **MUST** display an error to the RO and **MUST NOT** attempt to redirect the RO back to any client instance, even if a redirect finish method is supplied (Section 2.5.2.1). If the URI is not hosted by the AS directly, the means of communication between the AS and the service provided by this URI are out of scope for this specification.

The client instance **MUST NOT** modify the URI when launching it; in particular, the client instance **MUST NOT** add any parameters to the URI. The URI **MUST** be reachable from the end user's browser, though the URI **MAY** be opened on a separate device from the client instance itself. The URI **MUST** be accessible from an HTTP GET request, and it **MUST** be protected by HTTPS, be hosted on a server local to the RO's browser ("localhost"), or use an application-specific URI scheme that is loaded on the end user's device.

4.1.2. Interaction at the Static User Code URI

When the end user is directed to enter a short code through the "user_code" mode (Section 3.3.3), the client instance communicates the user code to the end user and directs the end user to enter that code at an associated URI. The client instance **MAY** format the user code in such a way as to facilitate memorability and transfer of the code, so long as this formatting does not alter the value as accepted at the user code URI. For example, a client instance receiving the user code "A1BC3DFF" could choose to display this to the user as "A1BC 3DFF", breaking up the long string into two shorter strings.

When processing input codes, the AS **MUST** transform the input string to remove invalid characters. In the above example, the space in between the two parts would be removed upon its entry into the interactive form at the user code URI. Additionally, the AS **MUST** treat user input as case insensitive. For example, if the user inputs the string "a1bc 3DFF", the AS will treat the input the same as "A1BC3DFF". To facilitate this, it is **RECOMMENDED** that the AS use only ASCII letters and numbers as valid characters for the user code.

It is **RECOMMENDED** that the AS choose from character values that are easily copied and typed without ambiguity. For example, some glyphs have multiple Unicode code points for the same visual character, and the end user could potentially type a different character than what the AS has returned. For additional considerations of internationalized character strings, see [RFC8264].

This mode is designed to be used when the client instance is not able to communicate or facilitate launching an arbitrary URI. The associated URI could be statically configured with the client instance or in the client software's documentation. As a consequence, these URIs **SHOULD** be short. The user code URI **MUST** be reachable from the end user's browser, though the URI is usually opened on a separate device from the client instance itself. The URI **MUST** be accessible from an HTTP GET request, and it **MUST** be protected by HTTPS, be hosted on a server local to the RO's browser ("localhost"), or use an application-specific URI scheme that is loaded on the end user's device.

In many cases, the URI indicates a web page hosted at the AS, allowing the AS to authenticate the end user as the RO and interactively provide consent. The value of the user code is used to identify the grant request being authorized. If the user code cannot be associated with a currently active request, the AS **MUST** display an error to the RO and **MUST NOT** attempt to redirect the RO back to any client instance, even if a redirect finish method is supplied (Section 2.5.2.1). If the interaction component at the user code URI is not hosted by the AS directly, the means of communication between the AS and this URI, including communication of the user code itself, are out of scope for this specification.

When the RO enters this code at the user code URI, the AS **MUST** uniquely identify the pending request that the code was associated with. If the AS does not recognize the entered code, the interaction component **MUST** display an error to the user. If the AS detects too many unrecognized code enter attempts, the interaction component **SHOULD** display an error to the user indicating too many attempts and **MAY** take additional actions such as slowing down the input interactions. The user should be warned as such an error state is approached, if possible.

4.1.3. Interaction at a Dynamic User Code URI

When the end user is directed to enter a short code through the "user_code_uri" mode (Section 3.3.4), the client instance communicates the user code and associated URI to the end user and directs the end user to enter that code at the URI. The client instance **MAY** format the user code in such a way as to facilitate memorability and transfer of the code, so long as this formatting does not alter the value as accepted at the user code URI. For example, a client instance receiving the user code "A1BC3DFF" could choose to display this to the user as "A1BC 3DFF", breaking up the long string into two shorter strings.

When processing input codes, the AS **MUST** transform the input string to remove invalid characters. In the above example, the space in between the two parts would be removed upon its entry into the interactive form at the user code URI. Additionally, the AS **MUST** treat user input as case insensitive. For example, if the user inputs the string "a1bc 3DFF", the AS will treat the input the same as "A1BC3DFF". To facilitate this, it is **RECOMMENDED** that the AS use only ASCII letters and numbers as valid characters for the user code.

This mode is used when the client instance is not able to facilitate launching a complex arbitrary URI but can communicate arbitrary values like URIs. As a consequence, these URIs **SHOULD** be short enough to allow the URI to be typed by the end user, such as a total length of 20 characters or fewer. The client instance **MUST NOT** modify the URI when communicating it to the end user; in particular the client instance **MUST NOT** add any parameters to the URI. The user code URI **MUST** be reachable from the end user's browser, though the URI is usually be opened on a separate device from the client instance itself. The URI **MUST** be accessible from an HTTP GET request, and it **MUST** be protected by HTTPS, be hosted on a server local to the RO's browser ("localhost"), or use an application-specific URI scheme that is loaded on the end user's device.

In many cases, the URI indicates a web page hosted at the AS, allowing the AS to authenticate the end user as the RO and interactively provide consent. The value of the user code is used to identify the grant request being authorized. If the user code cannot be associated with a currently active request, the AS **MUST** display an error to the RO and **MUST NOT** attempt to redirect the RO back to any client instance, even if a redirect finish method is supplied (Section 2.5.2.1). If the interaction component at the user code URI is not hosted by the AS directly, the means of communication between the AS and this URI, including communication of the user code itself, are out of scope for this specification.

When the RO enters this code at the given URI, the AS **MUST** uniquely identify the pending request that the code was associated with. If the AS does not recognize the entered code, the interaction component **MUST** display an error to the user. If the AS detects too many unrecognized code enter attempts, the interaction component **SHOULD** display an error to the user indicating too many attempts and **MAY** take additional actions such as slowing down the input interactions. The user should be warned as such an error state is approached, if possible.

4.1.4. Interaction through an Application URI

When the client instance is directed to launch an application through the "app" mode (Section 3.3.2), the client launches the URI as appropriate to the system, such as through a deep link or custom URI scheme registered to a mobile application. The means by which the AS and the launched application communicate with each other and perform any of the required actions are out of scope for this specification.

4.2. Post-Interaction Completion

If an interaction "finish" method (Section 3.3.5) is associated with the current request, the AS **MUST** follow the appropriate method upon completion of interaction in order to signal the client instance to continue, except for some limited error cases discussed below. If a finish method is not available, the AS **SHOULD** instruct the RO to return to the client instance upon completion. In such cases, it is expected that the client instance will poll the continuation endpoint as described in Section 5.2.

The AS **MUST** create an interaction reference and associate that reference with the current interaction and the underlying pending request. The interaction reference value is an ASCII string consisting of only unreserved characters per Section 2.3 of [RFC3986]. The interaction reference value **MUST** be sufficiently random so as not to be guessable by an attacker. The interaction reference **MUST** be one-time-use to prevent interception and replay attacks.

The AS **MUST** calculate a hash value based on the client instance, AS nonces, and the interaction reference, as described in Section 4.2.3. The client instance will use this value to validate the "finish" call.

All interaction finish methods **MUST** define a way to convey the hash and interaction reference back to the client instance. When an interaction finish method is used, the client instance **MUST** present the interaction reference back to the AS as part of its continuation request (Section 5.1).

Note that in many error cases, such as when the RO has denied access, the "finish" method is still enacted by the AS. This pattern allows the client instance to potentially recover from the error state by modifying its request or providing additional information directly to the AS in a continuation request. The AS **MUST NOT** follow the "finish" method in the following circumstances:

- The AS has determined that any URIs involved with the finish method are dangerous or blocked.
- The AS cannot determine which ongoing grant request is being referenced.
- The ongoing grant request has been canceled or otherwise blocked.

4.2.1. Completing Interaction with a Browser Redirect to the Callback URI

When using the `redirect` interaction finish method defined in Sections 2.5.2.1 and 3.3.5, the AS signals to the client instance that interaction is complete and the request can be continued by directing the RO (in their browser) back to the client instance's redirect URI.

The AS secures this redirect by adding the hash and interaction reference as query parameters to the client instance's redirect URI.

`hash`: The interaction hash value as described in Section 4.2.3. **REQUIRED**.

`interact_ref`: The interaction reference generated for this interaction. **REQUIRED**.

The means of directing the RO to this URI are outside the scope of this specification, but common options include redirecting the RO from a web page and launching the system browser with the target URI. See Section 11.19 for considerations on which HTTP status code to use when redirecting a request that potentially contains credentials.

```
NOTE: '\' line wrapping per RFC 8792

https://client.example.net/return/123455\
  ?hash=x-gguKWTj8rQf7d7i3w3UhzvuJ5bpOlKyAlVpLxBffY\
  &interact_ref=4IFWWIKYBC2PQ6U56NL1
```

The client instance **MUST** be able to process a request on the URI. If the URI is HTTP, the request **MUST** be an HTTP GET.

When receiving the request, the client instance **MUST** parse the query parameters to extract the hash and interaction reference values. The client instance **MUST** calculate and validate the hash value as described in Section 4.2.3. If the hash validates, the client instance sends a continuation request to the AS as described in Section 5.1, using the interaction reference value received here. If the hash does not validate, the client instance **MUST NOT** send the interaction reference to the AS.

4.2.2. Completing Interaction with a Direct HTTP Request Callback

When using the `push` interaction finish method defined in Sections 2.5.2.1 and 3.3.5, the AS signals to the client instance that interaction is complete and the request can be continued by sending an HTTP POST request to the client instance's callback URI.

The HTTP message content is a JSON object consisting of the following two fields:

`hash` (string): The interaction hash value as described in Section 4.2.3. **REQUIRED**.

`interact_ref` (string): The interaction reference generated for this interaction. **REQUIRED**.

```
POST /push/554321 HTTP/1.1
Host: client.example.net
Content-Type: application/json

{
  "hash": "pjdHcrti02HLCwGU3qhUZ3wZXt8IjrV_BtE3oUyOuKNk",
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

Since the AS is making an outbound connection to a URI supplied by an outside party (the client instance), the AS **MUST** protect itself against Server-Side Request Forgery (SSRF) attacks when making this call, as discussed in Section 11.34.

When receiving the request, the client instance **MUST** parse the JSON object and validate the hash value as described in Section 4.2.3. If either fails, the client instance **MUST** return an `unknown_interaction` error (Section 3.6). If the hash validates, the client instance sends a continuation request to the AS as described in Section 5.1, using the interaction reference value received here.

4.2.3. Calculating the Interaction Hash

The "hash" parameter in the request to the client instance's callback URI ties the front-channel response to an ongoing request by using values known only to the parties involved. This security mechanism allows the client instance to protect itself against several kinds of session fixation and injection attacks as discussed in Section 11.25. The AS **MUST** always provide this hash, and the client instance **MUST** validate the hash when received.

To calculate the "hash" value, the party doing the calculation creates a hash base string by concatenating the following values in the following order using a single newline (0x0A) character to separate them:

- the "nonce" value sent by the client instance in the interaction `finish` field of the initial request (Section 2.5.2)
- the AS's nonce value from the interaction finish response (Section 3.3.5)
- the "interact_ref" returned from the AS as part of the interaction finish method (Section 4.2)
- the grant endpoint URI the client instance used to make its initial request (Section 2)

There is no padding or whitespace before or after any of the lines and no trailing newline character. The following non-normative example shows a constructed hash base string consisting of these four elements.

```
VJLO6A4CATR0KRO
MBDOFXG4Y5CVJCX821LH
4IFWWIKYB2PQ6U56NL1
https://server.example.com/tx
```

The party then hashes the bytes of the ASCII encoding of this string with the appropriate algorithm based on the "hash_method" parameter under the "finish" key of the interaction finish request (Section 2.5.2). The resulting byte array from the hash function is then encoded using URL-Safe base64 with no padding [RFC4648]. The resulting string is the hash value.

If provided, the "hash_method" value **MUST** be one of the hash name strings defined in the IANA "Named Information Hash Algorithm Registry" [HASH-ALG]. If the "hash_method" value is not present in the client instance's request, the algorithm defaults to "sha-256".

For example, the "sha-256" hash method consists of hashing the input string with the 256-bit SHA2 algorithm. The following is the encoded "sha-256" hash of the hash base string in the example above.

```
x-gguKWTj8rQf7d7i3w3UhzvuJ5bpOlKyAlVpLxBffY
```

As another example, the "sha3-512" hash method consists of hashing the input string with the 512-bit SHA3 algorithm. The following is the encoded "sha3-512" hash of the hash base string in the example above.

```
NOTE: '\' line wrapping per RFC 8792

pyUkVJSmpqSJMaDYsk5G8WCvgY91l-agUPe1wgn-cc5rUtN69gPI2-S_s-Eswed8iB4\
  PJ_a5Hg6DNi7qGgKwSQ
```

5. Continuing a Grant Request

While it is possible for the AS to return an approved grant response (Section 3) with all the client instance's requested information (including access tokens (Section 3.2) and subject information (Section 3.4)) immediately, it's more common that the AS will place the grant request into the *pending* state and require communication with the client instance several times over the lifetime of a grant request. This is often part of facilitating interaction (Section 4), but it could also be used to allow the AS and client instance to continue negotiating the parameters of the original grant request (Section 2) through modification of the request.

The ability to continue an already-started request allows the client instance to perform several important functions, including presenting additional information from interaction, modifying the initial request, and revoking a grant request in progress.

To enable this ongoing negotiation, the AS provides a continuation API to the client software. The AS returns a `continue` field in the response (Section 3.1) that contains information the client instance needs to access this API, including a URI to access as well as a special access token to use during the requests, called the "continuation access token".

All requests to the continuation API are protected by a bound continuation access token. The continuation access token is bound to the same key and method the client instance used to make the initial request (or its most recent rotation). As a consequence, when the client instance makes any calls to the continuation URI, the client instance **MUST** present the continuation access token as described in Section 7.2 and present proof of the client instance's key (or its most recent rotation) by signing the request as described in Section 7.3. The AS **MUST** validate the signature and ensure that it is bound to the appropriate key for the continuation access token.

Access tokens other than the continuation access tokens **MUST NOT** be usable for continuation requests. Conversely, continuation access tokens **MUST NOT** be usable to make authorized requests to RSs, even if co-located within the AS.

In the following non-normative example, the client instance makes a POST request to a unique URI and signs the request with HTTP message signatures:

```
POST /continue/KSKUOMUKM HTTP/1.1
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Host: server.example.com
Content-Length: 0
Signature-Input: sig1=...
Signature: sig1=...
```

The AS **MUST** be able to tell from the client instance's request which specific ongoing request is being accessed, using a combination of the continuation URI and the continuation access token. If the AS cannot determine a single active grant request to map the continuation request to, the AS **MUST** return an `invalid_continuation` error (Section 3.6).

In the following non-normative example, the client instance makes a POST request to a stable continuation endpoint URI with the interaction reference (Section 5.1), includes the access token, and signs with HTTP message signatures:

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

In the following non-normative alternative example, the client instance had been provided a continuation URI unique to this ongoing grant request:

```
POST /tx/rxgIIEVMBV-BQUO7kxbsp HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP eyJhbGciOiJub25lIiwidHlwIjoiYmFkIn0
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

In both cases, the AS determines which grant is being asked for based on the URI and continuation access token provided.

If a `wait` parameter was included in the continuation response (Section 3.1), the client instance **MUST NOT** call the continuation URI prior to waiting the number of seconds indicated. If no `wait` period is indicated, the client instance **MUST NOT** poll immediately and **SHOULD** wait at least 5 seconds. If the client instance does not respect the given wait period, the AS **MUST** return the `too_fast` error (Section 3.6).

The response from the AS is a JSON object of a grant response and **MAY** contain any of the fields described in Section 3, as described in more detail in the subsections below.

If the AS determines that the client instance can make further requests to the continuation API, the AS **MUST** include a new continuation response (Section 3.1). The new continuation response **MUST** include a continuation access token as well, and this token **SHOULD** be a new access token, invalidating the previous access token. If the AS does not return a new continuation response, the client instance **MUST NOT** make an additional continuation request. If a client instance does so, the AS **MUST** return an `invalid_continuation` error (Section 3.6).

For continuation functions that require the client instance to send message content, the content **MUST** be a JSON object.

For all requests to the grant continuation API, the AS **MAY** make use of long polling mechanisms such as those discussed in [RFC6202]. That is to say, instead of returning the current status immediately, the long polling technique allows the AS additional time to process and fulfill the request before returning the HTTP response to the client instance. For example, when the AS receives a continuation request but the grant request is in the *processing* state, the AS could wait until the grant request has moved to the *pending* or *approved* state before returning the response message.

5.1. Continuing after a Completed Interaction

When the AS responds to the client instance's `finish` method as in Section 4.2.1, this response includes an interaction reference. The client instance **MUST** include that value as the field `interact_ref` in a POST request to the continuation URI.

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

Since the interaction reference is a one-time-use value as described in Section 4.2.1, if the client instance needs to make additional continuation calls after this request, the client instance **MUST NOT** include the interaction reference in subsequent calls. If the AS detects a client instance submitting an interaction reference when the request is not in the *pending* state, the AS **MUST** return a `too_many_attempts` error (Section 3.6) and **SHOULD** invalidate the ongoing request by moving it to the *finalized* state.

If the grant request is in the *approved* state, the grant response (Section 3) **MAY** contain any newly created access tokens (Section 3.2) or newly released subject information (Section 3.4). The response **MAY** contain a new continuation response (Section 3.1) as described above. The response **SHOULD NOT** contain any interaction responses (Section 3.3).

If the grant request is in the *pending* state, the grant response (Section 3) **MUST NOT** contain access tokens or subject information and **MAY** contain a new interaction response (Section 3.3) to any interaction methods that have not been exhausted at the AS.

For example, if the request is successful in causing the AS to issue access tokens and release opaque subject claims, the response could look like this:

```
NOTE: '\' line wrapping per RFC 8792

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": {
            "uri": "https://server.example.com/token/PRY5NM33O",
            "access_token": {
                "value": "B8CDFONP21-4TB8N6.BW7ONM"
            }
        }
    },
    "subject": {
        "sub_ids": [ {
          "format": "opaque",
          "id": "J2G8G8O4AZ"
        } ]
    }
}
```

With the above example, the client instance cannot make an additional continuation request because a `continue` field is not included.

In the following non-normative example, the RO has denied the client instance's request, and the AS responds with the following response:

```
{
    "error": "user_denied",
    "continue": {
        "access_token": {
            "value": "33OMUKMKSKU80UPRY5NM"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    }
}
```

In the preceding example, the AS includes the `continue` field in the response. Therefore, the client instance can continue the grant negotiation process, perhaps modifying the request as discussed in Section 5.3.

5.2. Continuing during Pending Interaction (Polling)

When the client instance does not include a `finish` parameter, the client instance will often need to poll the AS until the RO has authorized the request. To do so, the client instance makes a POST request to the continuation URI as in Section 5.1 but does not include message content.

```
POST /continue HTTP/1.1
Host: server.example.com
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
```

If the grant request is in the *approved* state, the grant response (Section 3) **MAY** contain any newly created access tokens (Section 3.2) or newly released subject claims (Section 3.4). The response **MAY** contain a new continuation response (Section 3.1) as described above. If a `continue` field is included, it **SHOULD** include a `wait` field to facilitate a reasonable polling rate by the client instance. The response **SHOULD NOT** contain interaction responses (Section 3.3).

If the grant request is in the *pending* state, the grant response (Section 3) **MUST NOT** contain access tokens or subject information and **MAY** contain a new interaction response (Section 3.3) to any interaction methods that have not been exhausted at the AS.

For example, if the request has not yet been authorized by the RO, the AS could respond by telling the client instance to make another continuation request in the future. In the following non-normative example, a new, unique access token has been issued for the call, which the client instance will use in its next continuation request.

```
{
    "continue": {
        "access_token": {
            "value": "33OMUKMKSKU80UPRY5NM"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    }
}
```

If the request is successful in causing the AS to issue access tokens and release subject information, the response could look like the following non-normative example:

```
NOTE: '\' line wrapping per RFC 8792

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": {
            "uri": "https://server.example.com/token/PRY5NM33O",
            "access_token": {
                "value": "B8CDFONP21-4TB8N6.BW7ONM"
            }
        }
    },
    "subject": {
        "sub_ids": [ {
          "format": "opaque",
          "id": "J2G8G8O4AZ"
        } ]
    }
}
```

See Section 11.23 for considerations on polling for continuation without an interaction `finish` method.

In error conditions, the AS responds to the client instance with an error code as discussed in Section 3.6. For example, if the client instance has polled too many times before the RO has approved the request, the AS would respond with a message like the following:

```
{
    "error": "too_many_attempts"
}
```

Since this response does not include a `continue` field, the client instance cannot continue to poll the AS for additional updates and the grant request is *finalized*. If the client instance still needs access to the resource, it will need to start with a new grant request.

5.3. Modifying an Existing Request

The client instance might need to modify an ongoing request, whether or not tokens have already been issued or subject information has already been released. In such cases, the client instance makes an HTTP PATCH request to the continuation URI and includes any fields it needs to modify. Fields that aren't included in the request are considered unchanged from the original request.

A grant request associated with a modification request **MUST** be in the *approved* or *pending* state. When the AS receives a valid modification request, the AS **MUST** place the grant request into the *processing* state and re-evaluate the authorization in the new context created by the update request, since the extent and context of the request could have changed.

The client instance **MAY** include the `access_token` and `subject` fields as described in Sections 2.1 and 2.2. Inclusion of these fields override any values in the initial request, which **MAY** trigger additional requirements and policies by the AS. For example, if the client instance is asking for more access, the AS could require additional interaction with the RO to gather additional consent. If the client instance is asking for more limited access, the AS could determine that sufficient authorization has been granted to the client instance and return the more limited access rights immediately. If the grant request was previously in the *approved* state, the AS could decide to remember the larger scale of access rights associated with the grant request, allowing the client instance to make subsequent requests of different subsets of granted access. The details of this processing are out of scope for this specification, but a one possible approach is as follows:

1. A client instance requests access to `Foo`, and this is granted by the RO. This results in an access token: `AT1`.
2. The client instance later modifies the grant request to include `Foo` and `Bar` together. Since the client instance was previously granted `Foo` under this grant request, the RO is prompted to allow the client instance access to `Foo` and `Bar` together. This results in a new access token: `AT2`. This access token has access to both `Foo` and `Bar`. The rights of the original access token `AT1` are not modified.
3. The client instance makes another grant modification to ask only for `Bar`. Since the client instance was previously granted `Foo` and `Bar` together under this grant request, the RO is not prompted, and the access to `Bar` is granted in a new access token: `AT3`. This new access token does not allow access to `Foo`.
4. The original access token `AT1` expires, and the client seeks a new access token to replace it. The client instance makes another grant modification to ask only for `Foo`. Since the client instance was previously granted `Foo` and `Bar` together under this grant request, the RO is not prompted, and the access to `Foo` is granted in a new access token: `AT4`. This new access token does not allow access to `Bar`.

All four access tokens are independent of each other and associated with the same underlying grant request. Each of these access tokens could possibly also be rotated using token management, if available. For example, instead of asking for a new token to replace `AT1`, the client instance could ask for a refresh of `AT1` using the rotation method of the token management API. This would result in a refreshed `AT1` with a different token value and expiration from the original `AT1` but with the same access rights of allowing only access to `Foo`.

The client instance **MAY** include the `interact` field as described in Section 2.5. Inclusion of this field indicates that the client instance is capable of driving interaction with the end user, and this field replaces any values from a previous request. The AS **MAY** respond to any of the interaction responses as described in Section 3.3, just like it would to a new request.

The client instance **MAY** include the `user` field as described in Section 2.4 to present new assertions or information about the end user. The AS **SHOULD** check that this presented user information is consistent with any user information previously presented by the client instance or otherwise associated with this grant request.

The client instance **MUST NOT** include the `client` field of the request, since the client instance is assumed not to have changed. Modification of client instance information, including rotation of keys associated with the client instance, is outside the scope of this specification.

The client instance **MUST NOT** include post-interaction responses such as those described in Section 5.1.

Modification requests **MUST NOT** alter previously issued access tokens. Instead, any access tokens issued from a continuation are considered new, separate access tokens. The AS **MAY** revoke previously issued access tokens after a modification has occurred.

If the modified request can be granted immediately by the AS (the grant request is in the *approved* state), the grant response (Section 3) **MAY** contain any newly created access tokens (Section 3.2) or newly released subject claims (Section 3.4). The response **MAY** contain a new continuation response (Section 3.1) as described above. If interaction can occur, the response **SHOULD** contain interaction responses (Section 3.3) as well.

For example, a client instance initially requests a set of resources using references:

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read", "write"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    },
    "client": "987YHGRT56789IOLK"
}
```

Access is granted by the RO, and a token is issued by the AS. In its final response, the AS includes a `continue` field, which includes a separate access token for accessing the continuation API:

```
{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    },
    "access_token": {
        "value": "RP1LT0-OS9M2P_R64TB",
        "access": [
            "read", "write"
        ]
    }
}
```

This `continue` field allows the client instance to make an eventual continuation call. Some time later, the client instance realizes that it no longer needs "write" access and therefore modifies its ongoing request, here asking for just "read" access instead of both "read" and "write" as before.

```
PATCH /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read"
        ]
    }
    ...
}
```

The AS replaces the previous `access` from the first request, allowing the AS to determine if any previously granted consent already applies. In this case, the AS would determine that reducing the breadth of the requested access means that new access tokens can be issued to the client instance without additional interaction or consent. The AS would likely revoke previously issued access tokens that had the greater access rights associated with them, unless they had been issued with the `durable` flag.

```
{
    "continue": {
        "access_token": {
            "value": "M33OMUK80UPRY5NMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    },
    "access_token": {
        "value": "0EVKC7-2ZKwZM_6N760",
        "access": [
            "read"
        ]
    }
}
```

As another example, the client instance initially requests read-only access but later needs to step up its access. The initial request could look like the following HTTP message:

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    },
    "client": "987YHGRT56789IOLK"
}
```

Access is granted by the RO, and a token is issued by the AS. In its final response, the AS includes a `continue` field:

```
{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    },
    "access_token": {
        "value": "RP1LT0-OS9M2P_R64TB",
        "access": [
            "read"
        ]
    }
}
```

This allows the client instance to make an eventual continuation call. The client instance later realizes that it now needs "write" access in addition to the "read" access. Since this is an expansion of what it asked for previously, the client instance also includes a new interaction field in case the AS needs to interact with the RO again to gather additional authorization. Note that the client instance's nonce and callback are different from the initial request. Since the original callback was already used in the initial exchange and the callback is intended for one-time use, a new one needs to be included in order to use the callback again.

```
PATCH /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read", "write"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/654321",
            "nonce": "K82FX4T4LKLTI25DQFZC"
        }
    }
}
```

From here, the AS can determine that the client instance is asking for more than it was previously granted, but since the client instance has also provided a mechanism to interact with the RO, the AS can use that to gather the additional consent. The protocol continues as it would

with a new request. Since the old access tokens are good for a subset of the rights requested here, the AS might decide to not revoke them. However, any access tokens granted after this update process are new access tokens and do not modify the rights of existing access tokens.

5.4. Revoking a Grant Request

If the client instance wishes to cancel an ongoing grant request and place it into the *finalized* state, the client instance makes an HTTP DELETE request to the continuation URI.

```
DELETE /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
```

If the request is successfully revoked, the AS responds with HTTP status code 204 (No Content). The AS **SHOULD** revoke all associated access tokens, if possible. The AS **SHOULD** disable all token rotation and other token management functions on such access tokens, if possible. Once the grant request is in the *finalized* state, it **MUST NOT** be moved to any other state.

If the request is not revoked, the AS responds with an `invalid_continuation` error (Section 3.6).

6. Token Management

If an access token response includes the `manage` field as described in Section 3.2.1, the client instance **MAY** call this URI to manage the access token with the rotate and revoke actions defined in the following subsections. Other actions are undefined by this specification.

```
{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "flags": ["bearer"],
        "manage": {
            "uri": "https://server.example.com/token/PRY5NM33O",
            "access_token": {
                "value": "B8CDFONP21-4TB8N6.BW7ONM"
            }
        }
    }
}
```

The token management access token issued under the `manage` field is used to protect all calls to the token management API. The client instance **MUST** present proof of the key associated with the token along with the value of the token management access token.

The AS **MUST** validate the proof and ensure that it is associated with the token management access token.

The AS **MUST** uniquely identify the token being managed from the token management URI, the token management access token, or a combination of both.

6.1. Rotating the Access Token Value

If the client instance has an access token and that access token expires, the client instance might want to rotate the access token to a new value without expiration. Rotating an access token consists of issuing a new access token in place of an existing access token, with the same rights and properties as the original token, apart from an updated token value and expiration time.

To rotate an access token, the client instance makes an HTTP POST to the token management URI with no message content, sending the access token in the authorization header as described in Section 7.2 and signing the request with the appropriate key.

```
POST /token/PRY5NM33O HTTP/1.1
Host: server.example.com
Authorization: GNAP B8CDFONP21-4TB8N6.BW7ONM
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...
```

The client instance cannot request to alter the access rights associated with the access token during a rotation request. To get an access token with different access rights for this grant request, the client instance has to call the continuation API's update functionality (Section 5.3) to get a new access token. The client instance can also create a new grant request with the required access rights.

The AS validates that the token management access token presented is associated with the management URI, that the AS issued the token to the given client instance, and that the presented key is the correct key for the token management access token. The AS determines which access token is being rotated from the token management URI, the token management access token, or both.

If the token is validated and the key is appropriate for the request, the AS **MUST** invalidate the current access token value associated with this URI, if possible. Note that stateless access tokens can make proactive revocation difficult within a system; see Section 11.32.

For successful rotations, the AS responds with an HTTP status code 200 (OK) with JSON-formatted message content consisting of the rotated access token in the `access_token` field described in Section 3.2.1. The value of the access token **MUST NOT** be the same as the current value of the access token used to access the management API. The response **MUST** include an access token management URI, and the value of this URI **MAY** be different from the URI used by the client instance to make the rotation call. The client instance **MUST** use this new URI to manage the rotated access token.

The access rights in the `access` array for the rotated access token **MUST** be included in the response and **MUST** be the same as the token before rotation.

```
NOTE: '\' line wrapping per RFC 8792

{
    "access_token": {
        "value": "FP6A8H6HY37MH13CK76LBZ6Y1UADG6VEUPEER5H2",
        "manage": {
            "uri": "https://server.example.com/token/PRY5NM33O",
            "access_token": {
                "value": "B8CDFONP21-4TB8N6.BW7ONM"
            }
        },
        "expires_in": 3600,
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "read", "dolphin-metadata"
        ]
    }
}
```

If the AS is unable or unwilling to rotate the value of the access token, the AS responds with an `invalid_rotation` error (Section 3.6). Upon receiving such an error, the client instance **MUST** consider the access token to not have changed its state.

6.1.1. Binding a New Key to the Rotated Access Token

If the client instance wishes to bind a new presentation key to an access token, the client instance **MUST** present both the new key and the proof of previous key material in the access token rotation request. The client instance makes an HTTP POST as a JSON object with the following field:

`key`: The new key value or reference in the format described in Section 7.1. Note that keys passed by value are always public keys. **REQUIRED** when doing key rotation.

The proofing method and parameters for the new key **MUST** be the same as those established for the previous key.

The client instance **MUST** prove possession of both the currently bound key and the newly requested key simultaneously in the rotation request. Specifically, the signature from the previous key **MUST** cover the value or reference of the new key, and the signature of the new key **MUST** cover the signature value of the old key. The means of doing so vary depending on the proofing method in use. For example, the HTTP message signatures proofing method uses multiple signatures in the request as described in Section 7.3.1.1. This is shown in the following example.

```
POST /token/PRY5NM33O HTTP/1.1
Host: server.example.com
Authorization: GNAP B8CDFONP21-4TB8N6.BW7ONM
Signature-Input: \
  sig1=("@method" "@target-uri" "content-digest" \
        "authorization"),\
  sig2=("@method" "@target-uri" "content-digest" \
        "authorization" "signature";key="sig1" \
        "signature-input";key="sig1")
Signature: sig1=..., sig2=...
Content-Digest: sha-256=...

{
    "key": {
        "proof": "httpsig",
        "jwk": {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "xyz-2",
            "alg": "RS256",
            "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8xY..."
        }
    }
}
```

Failure to present the appropriate proof of either the new key or the previous key for the access token, as defined by the proofing method, **MUST** result in an `invalid_rotation` error code from the AS (Section 3.6).

An attempt to change the proofing method or parameters, including an attempt to rotate the key of a bearer token (which has no key), **MUST** result in an `invalid_rotation` error code returned from the AS (Section 3.6).

If the AS does not allow rotation of the access token's key for any reason, including but not limited to lack of permission for this client instance or lack of capability by the AS, the AS **MUST** return a `key_rotation_not_supported` error code (Section 3.6).

6.2. Revoking the Access Token

If the client instance wishes to revoke the access token proactively, such as when a user indicates to the client instance that they no longer wish for it to have access or the client instance application detects that it is being uninstalled, the client instance can use the token management URI to indicate to the AS that the AS **SHOULD** invalidate the access token for all purposes.

The client instance makes an HTTP DELETE request to the token management URI, presenting the access token and signing the request with the appropriate key.

```
DELETE /token/PRY5NM33O HTTP/1.1
Host: server.example.com
Authorization: GNAP B8CDFONP21-4TB8N6.BW7ONM
Signature-Input: sig1=...
Signature: sig1=...
```

If the key presented is associated with the token (or the client instance, in the case of a bearer token), the AS **MUST** invalidate the access token, if possible, and return an HTTP response code 204.

```
204 No Content
```

Though the AS **MAY** revoke an access token at any time for any reason, the token management function is specifically for the client instance's use. If the access token has already expired or has been revoked through other means, the AS **SHOULD** honor the revocation request to the token management URI as valid, since the end result is that the token is still not usable.

7. Securing Requests from the Client Instance

In GNAP, the client instance secures its requests to an AS and RS by presenting an access token, proof of a key that it possesses (aka, a "key proof"), or both an access token and key proof together.

- When an access token is used with a key proof, this is a bound token request. This type of request is used for calls to the RS as well as the AS during grant negotiation.
- When a key proof is used with no access token, this is a non-authorized signed request. This type of request is used for calls to the AS to initiate a grant negotiation.
- When an access token is used with no key proof, this is a bearer token request. This type of request is used only for calls to the RS and only with access tokens that are not bound to any key as described in Section 3.2.1.
- When neither an access token nor key proof are used, this is an unsecured request. This type of request is used optionally for calls to the RS as part of an RS-first discovery process as described in Section 9.1.

7.1. Key Formats

Several different places in GNAP require the presentation of key material by value or by reference. Key material sent by value is sent using a JSON object with several fields described in this section.

All keys are associated with a specific key proofing method. The proofing method associated with the key is indicated using the `proof` field of the key object.

`proof` (string or object): The form of proof that the client instance will use when presenting the key. The valid values of this field and the processing requirements for each are detailed in Section 7.3. **REQUIRED**.

A key presented by value **MUST** be a public key and **MUST** be presented in only one supported format, as discussed in Section 11.35. Note that while most formats present the full value of the public key, some formats present a value cryptographically derived from the public key. See additional discussion of the presentation of public keys in Section 11.7.

`jwk` (object): The public key and its properties represented as a JSON Web Key (JWK) [RFC7517]. A JWK **MUST** contain the `alg` (Algorithm) and `kid` (Key ID) parameters. The `alg` parameter **MUST NOT** be "none". The `x5c` (X.509 Certificate Chain) parameter **MAY** be used to provide the X.509 representation of the provided public key. **OPTIONAL**.

`cert` (string): The Privacy-Enhanced Mail (PEM) serialized value of the certificate used to sign the request, with optional internal whitespace per [RFC7468]. The PEM header and footer are optionally removed. **OPTIONAL**.

`cert#S256` (string): The certificate thumbprint calculated as per MTLS for OAuth [RFC8705] in base64url encoding. Note that this format does not include the full public key. **OPTIONAL**.

Additional key formats can be defined in the "GNAP Key Formats" registry (Section 10.17).

The following non-normative example shows a single key presented in two different formats. The example key is intended to be used with the HTTP message signatures proofing mechanism (Section 7.3.1), as indicated by the `httpsig` value of the `proof` field.

As a JWK:

```
"key": {
    "proof": "httpsig",
    "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "kid": "xyz-1",
        "alg": "RS256",
        "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8xY..."
    }
}
```

As a certificate in PEM format:

```
"key": {
    "proof": "httpsig",
    "cert": "MIIEHDCCAwSgAwIBAgIBATANBgkqhkiG9w0BAQsFA..."
}
```

When the key is presented in GNAP, proof of this key material **MUST** be used to bind the request, the nature of which varies with the location in the protocol where the key is used. For a key used as part of a client instance's initial request in Section 2.3, the key value represents the client instance's public key, and proof of that key **MUST** be presented in that request. For a key used as part of an access token response in Section 3.2.1, the proof of that key **MUST** be used when the client instance later presents the access token to the RS.

7.1.1. Key References

Keys in GNAP can also be passed by reference such that the party receiving the reference will be able to determine the appropriate keying material for use in that part of the protocol. A key reference is a single opaque string.

```
"key": "S-P4XJQ_RYJCRTSU1.63N3E"
```

Keys referenced in this manner **MAY** be shared symmetric keys. See the additional considerations for symmetric keys in Section 11.7. The key reference **MUST NOT** contain any unencrypted private or shared symmetric key information.

Keys referenced in this manner **MUST** be bound to a single proofing mechanism.

The means of dereferencing this reference to a key value and proofing mechanism are out of scope for this specification. Commonly, key references are created by the AS and do not necessarily need to be understood by the client. These types of key references are an internal reference to the AS, such as an identifier of a record in a database. In other applications, it can be useful to use key references that are resolvable by both clients and the AS, which could be accomplished by a client publishing a public key at a URI, for example. For interoperability, this method could later be described as an extension, but doing so is out of scope for this specification.

7.1.2. Key Protection

The security of GNAP relies on the cryptographic security of the keys themselves. When symmetric keys are used in GNAP, a key management system or secure key derivation mechanism **MUST** be used to supply the keys. Symmetric keys **MUST NOT** be a human-memorable password or a value derived from one. Symmetric keys **MUST NOT** be passed by value from the client instance to the AS.

Additional security considerations apply when rotating keys (see Section 11.22).

7.2. Presenting Access Tokens

Access tokens are issued to client instances in GNAP to allow the client instance to make an authorized call to an API. The method the client instance uses to send an access token depends on whether the token is bound to a key and, if so, which proofing method is associated with the key. This information is conveyed by the `key` parameter and the `bearer` flag in the access token response structure (Section 3.2.1).

If the `flags` field does not contain the `bearer` flag and the `key` is absent, the access token **MUST** be sent using the same key and proofing mechanism that the client instance used in its initial request (or its most recent rotation).

If the `flags` field does not contain the `bearer` flag and the `key` value is an object as described in Section 7.1, the access token **MUST** be sent using the key and proofing mechanism defined by the value of the `proof` field within the key object.

The access token **MUST** be sent using the HTTP Authorization request header field and the "GNAP" authorization scheme along with a key proof as described in Section 7.3 for the key bound to the access token. For example, an access token bound using HTTP message signatures would be sent as follows:

```
NOTE: '\' line wrapping per RFC 8792

GET /stuff HTTP/1.1
Host: resource.example.com
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=("@method" "@target-uri" "authorization")\
  ;created=1618884473;keyid="gnap-rsa";nonce="NAOEJF12ER2";tag="gnap"
Signature: sig1=:FQ+EjWqc38uLFByKa5y+c4WyYYwCTGUhidWKfr5L1Cha8FiPEw\
  DxG7nWttpBLS/B6VLfkZJogPbclySs9MDIsAIJwHnzlcJjwXWR2lfvm2z3X7EkJHm\
  Zp4SmyKOS34luAiKR1xwf32NYFolHmZf/SbHZJuWvQuS4U33C+BbsXz8MflFH1Dht\
  H/C1E5i244gSbdLCPxzABc/Q0NHVSLo1qaouYIvnxXB8OT3K7mwWjsLh1GC5vFThb\
  3XQ363r6f0OPRa4qWHhubR/d/J/lNOjbBdjq9AJ69oqNJ+A2XT+ZCrVasEJE0OBvD\
  auQoiywhb8BMB7+PEINsPk5/8UvaNxbw==:
```

If the `flags` field contains the `bearer` flag, the access token is a bearer token that **MUST** be sent using the Authorization request header field method defined in [RFC6750].

```
Authorization: Bearer OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
```

The `Form-Encoded Body Parameter` and `URI Query Parameter` methods of [RFC6750] **MUST NOT** be used for GNAP access tokens.

7.3. Proving Possession of a Key with a Request

Any keys presented by the client instance to the AS or RS **MUST** be validated as part of the request in which they are presented. The type of binding used is indicated by the `proof` parameter of the key object in Section 7.1. Key proofing methods are specified either by a string, which consists of the key proofing method name on its own, or by a JSON object with the required field `method`:

`method`: The name of the key proofing method to be used. **REQUIRED**.

Individual methods defined as objects **MAY** define additional parameters as members in this object.

Values for the `method` defined by this specification are as follows:

`"httpsig"` (string or object): HTTP message signing. See Section 7.3.1.

`"mtls"` (string): MTLS certificate verification. See Section 7.3.2.

`"jwsd"` (string): A detached JWS signature header. See Section 7.3.3.

`"jws"` (string): Attached JWS Payload. See Section 7.3.4.

Additional proofing methods can be defined in the "GNAP Key Proofing Methods" registry (Section 10.16).

Proofing methods **MAY** be defined as both an object and a string. For example, the `httpsig` method can be specified as an object with its parameters explicitly declared, such as:

```
{
    "proof": {
        "method": "httpsig",
        "alg": "ecdsa-p384-sha384",
        "content-digest-alg": "sha-256"
    }
}
```

The `httpsig` method also defines default behavior when it is passed as a string form, using the signature algorithm specified by the associated key material and the content digest is calculated using sha-256. This configuration can be selected using the following shortened form:

```
{
    "proof": "httpsig"
}
```

All key binding methods used by this specification **MUST** cover all relevant portions of the request, including anything that would change the nature of the request, to allow for secure validation of the request. Relevant aspects include the URI being called, the HTTP method being used, any relevant HTTP headers and values, and the HTTP message content itself. The verifier of the signed message **MUST** validate all components of the signed message to ensure that nothing has been tampered with or substituted in a way that would change the nature of the request. Definitions of key binding methods **MUST** enumerate how these requirements are fulfilled.

When a key proofing mechanism is bound to an access token, the key being presented **MUST** be the key associated with the access token, and the access token **MUST** be covered by the signature method of the proofing mechanism.

The key binding methods in this section **MAY** be used by other components making calls as part of GNAP, such as the extensions allowing the RS to make calls to the AS defined in [GNAP-RS]. To facilitate this extended use, "signer" and "verifier" are used as generic terms in the subsections

below. In the core functions of GNAP specified in this document, the "signer" is the client instance, and the "verifier" is the AS (for grant requests) or RS (for resource requests), as appropriate.

When used for delegation in GNAP, these key binding mechanisms allow the AS to ensure that the keys presented by the client instance in the initial request are in control of the party calling any follow-up or continuation requests. To facilitate this requirement, the continuation response (Section 3.1) includes an access token bound to the client instance's key (Section 2.3), and that key (or its most recent rotation) **MUST** be proved in all continuation requests (Section 5). Token management requests (Section 6) are similarly bound to either the access token's own key or, in the case of bearer tokens, the client instance's key.

In the following subsections, unless otherwise noted, the `RS256` JSON Object Signing and Encryption (JOSE) signature algorithm (defined in Section 3.3 of [RFC7518]) is applied using the following RSA key (presented here in JWK format):

```
NOTE: '\' line wrapping per RFC 8792

{
    "kid": "gnap-rsa",
    "p": "xS4-YbQ0SgrsmcA7xDzZKuVNxJe3pCYwdAe6efSy4hdDgF9-vhC5gjaRk\
        i1wWuERSMW4Tv44l5HNrL-Bbj_nCJxr_HAOaesDiPn2PnywwEfg3Nv95Nn-\
        eilhqXRaW-tJKEMjDHu_fmJBeemHNZI412gBnXdGzDVo22dvYoxd6GM",
    "kty": "RSA",
    "q": "rVdcT_uy-CD0GKVLGpEGRR7k4JO6Tktc8MEHkC6NIFXihk_6vAIOCzCD6\
        LMovMinOYttpRndKoGTNdJfWlDFDScAs8C5n2y1STCQPRximBY-bw39-aZq\
        JXMxOLyPjzuVgiTOCBIvLD6-8-mvFjXZk_eefD0at6mQ5qV3U1jZt88",
    "d": "FHlhdTF0ozTliDxMBffT6aJVKZKmbbFJOVNten9c3lXKB3ux3NAb_D2dB\
        7inp9EV23oWrDspFtvCvD9dZrXgRKMHofkEpo_SSvBZfgtH-OTkbY_TqtPF\
        FLPKAw0JX5cFPnn4Q2xE4n-dQ7tpRCKl59vZLHBrHShr90zqzFp0AKXU5fj\
        b1gC9LPwsFA2Fd7KXmI1drQQEVq9R-o18Pnn4BGQNQNjO_VkcJTiBmEIVT_\
        KJRPdpVJAmbgnYWafL_hAfeb_dK8p85yurEVF8nCK5oO3EPrqB7IL4UqaEn\
        5Sl3u0j8x5or-xrrAoNz-gdOv7ONfZY6NFoa-3f8q9wBAHUuQ",
    "e": "AQAB",
    "qi": "ogpNEkDKg22Rj9cDV_-PJBZaXMk66Fp557RT1tafIuqJRHEufSOYnsto\
        bWPJ0gHxv1gVJw3gm-zYvV-wTMNgr2wVsBSezSJjPSjxWZtmT2z68W1DuvK\
        kZy15vz7Jd85hmDlriGcXNCoFEUsGLWkpHH9RwPIzguUHWmTt8y0oXyI",
    "dp": "dvCKGI2G7RLh3WyjoJ_Dr6hZ3LhXweB3YcY3qdD9BnxZ71mrLiMQg4c_\
        EBnwqCETN_5sStn2cRc2JXnvLP3G8t7IFKHTT_i_TSTacJ7uT04MSa053Y3\
        RfwbvLjRNPR0UKAE3ZxROUoIaVNuU_6-QMf8-2ilUv2GIOrCN87gP_Vk",
    "alg": "RS256",
    "dq": "iMZmELaKgT9_W_MRT-UfDWtTLeFjIGRW8aFeVmZk9R7Pnyt8rNzyN-IQ\
        M40ql8u8J6vc2GmQGfokLlPQ6XLSCY68_xkTXrhoU1f-eDntkhP7L6XawSK\
        Onv5F2H7wyBQ75HUmHTg8AK2B_vRlMyFKjXbVlzKf4kvqChSGEz4IjQ",
    "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8BfYdHsFzAt\
        YKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZGYX\
        jHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZx\
        e0jRETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0\
        bunS0K3bA_3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kO\
        zywzwPTuq-cVQDyEN7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
}
```

Key proofing methods **SHOULD** define a mechanism to allow the rotation of keys discussed in Section 6.1.1. Key rotation mechanisms **MUST** define a way for presenting proof of two keys simultaneously with the following attributes:

- The value of or reference to the new key material **MUST** be signed by the existing key. Generally speaking, this amounts to using the existing key to sign the content of the message that contains the new key.
- The signature of the old key **MUST** be signed by the new key. Generally speaking, this means including the signature value of the old key under the coverage of the new key.

7.3.1. HTTP Message Signatures

This method is indicated by the method value `httpsig` and can be declared in either object form or string form.

When the proofing method is specified in object form, the following parameters are defined:

`alg`: The HTTP signature algorithm, from the "HTTP Signature Algorithms" registry. **REQUIRED**.

`content-digest-alg`: The algorithm used for the Content-Digest field, used to protect the content when present in the message. **REQUIRED**.

This example uses the Elliptic Curve Digital Signature Algorithm (ECDSA) signing algorithm over the P384 curve and the SHA-512 hashing algorithm for the content digest.

```
{
    "proof": {
        "method": "httpsig",
        "alg": "ecdsa-p384-sha384",
        "content-digest-alg": "sha-512"
    }
}
```

When the proofing method is specified in string form, the signing algorithm **MUST** be derived from the key material (such as using the JWS algorithm in a JWK formatted key), and the content digest algorithm **MUST** be `sha-256`.

```
{
    "proof": "httpsig"
}
```

When using this method, the signer creates an HTTP message signature as described in [RFC9421]. The covered components of the signature **MUST** include the following:

`"@method"`: The method used in the HTTP request.

`"@target-uri"`: The full request URI of the HTTP request.

When the message contains request content, the covered components **MUST** also include the following:

`"content-digest"`: The Content-Digest header as defined in [RFC9530]. When the request message has content, the signer **MUST** calculate this field value and include the field in the request. The verifier **MUST** validate this field value. **REQUIRED** when the message request contains message content.

When the request is bound to an access token, the covered components **MUST** also include the following:

`"authorization"`: The Authorization header used to present the access token as discussed in Section 7.2.

Other message components **MAY** also be included.

The signer **MUST** include the `tag` signature parameter with the value `gnap`, and the verifier **MUST** verify that the parameter exists with this value. The signer **MUST** include the `created` signature parameter with a timestamp of when the signature was created, and the verifier **MUST** ensure that the creation timestamp is sufficiently close to the current time given expected network delay and clock skew. The signer **SHOULD** include the `nonce` parameter with a unique and unguessable value. When included, the verifier **MUST** determine that the nonce value is unique within a reasonably short time period such as several minutes.

If the signer's key presented is a JWK, the `keyid` parameter of the signature **MUST** be set to the `kid` value of the JWK, and the signing algorithm used **MUST** be the JWS algorithm denoted by the key's `alg` field of the JWK.

The explicit `alg` signature parameter **MUST NOT** be included in the signature, since the algorithm will be derived from either the key material or the `proof` value.

In the following non-normative example, the message content is a JSON object:

```
NOTE: '\' line wrapping per RFC 8792

{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "httpsig",
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "PS512",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    }
}
```

This content is hashed for the Content-Digest header using `sha-256` into the following encoded value:

```
sha-256=:q2XBmzRDCREcS2nWo/6LYwYyjrlN1bRfv+HKLbeGAGg=:
```

The HTTP message signature input string is calculated to be the following:

```
NOTE: '\' line wrapping per RFC 8792

"@method": POST
"@target-uri": https://server.example.com/gnap
"content-digest": \
  sha-256=:q2XBmzRDCREcS2nWo/6LYwYyjrlN1bRfv+HKLbeGAGg=:
"content-length": 988
"content-type": application/json
"@signature-params": ("@method" "@target-uri" "content-digest" \
  "content-length" "content-type");created=1618884473\
  ;keyid="gnap-rsa";nonce="NAOEJF12ER2";tag="gnap"
```

This leads to the following full HTTP message request:

```
NOTE: '\' line wrapping per RFC 8792

POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/json
Content-Length: 988
Content-Digest: sha-256=:q2XBmzRDCREcS2nWo/6LYwYyjrlN1bRfv+HKLbeGAG\
  g=:
Signature-Input: sig1=("@method" "@target-uri" "content-digest" \
  "content-length" "content-type");created=1618884473\
  ;keyid="gnap-rsa";nonce="NAOEJF12ER2";tag="gnap"
Signature: sig1=:c2uwTa6ok3iHZsaRKl1ediKlgd5cCAYztbym68XgX8gSOgK0Bt\
  +zLJ19oGjSAHDjJxX2gXP2iR6lh9bLMTfPzbFVn4Eh+5UlceP+0Z5mES7v0R1+eHe\
  OqBl0YlYKaSQ11YT7n+cwPnCSdv/6+62m5zwXEEftnBeA1ECorfTuPtau/yrTYEvD\
  9A/JqR2h9VzAE17kSlSSsDHYA6ohsFqcRJavX29duPZDfYgkZa76u7hJ23yVxoUpu\
  2J+7VUdedN/72N3u3/z2dC8vQXbzCPTOiLru12lb6vnBZoDbUGsRR/zHPauxhj9T+\
  218o5+tgwYXw17othJSxIIOZ9PkIgz4g==:

{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "httpsig",
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "PS512",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
```

```
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    }
}
```

The verifier **MUST** ensure that the signature covers all required message components. If the HTTP message includes content, the verifier **MUST** calculate and verify the value of the `Content-Digest` header. The verifier **MUST** validate the signature against the expected key of the signer.

A received message **MAY** include multiple signatures, each with its own label. The verifier **MUST** examine all included signatures until it finds (at least) one that is acceptable according to its policy and meets the requirements in this section.

7.3.1.1. Key Rotation Using HTTP Message Signatures

When rotating a key using HTTP message signatures, the message, which includes the new public key value or reference, is first signed with the old key following all of the requirements in Section 7.3.1. The message is then signed again with the new key by following all of the requirements in Section 7.3.1 again, with the following additional requirements:

- The covered components **MUST** include the Signature and Signature-Input values from the signature generated with the old key.
- The tag value **MUST** be `gnap-rotate`.

For example, the following request to the token management endpoint for rotating a token value contains the new key in the request. The message is first signed using the old key, and the resulting signature is placed in "old-key":

```
NOTE: '\' line wrapping per RFC 8792

POST /token/PRY5NM33 HTTP/1.1
Host: server.example.com
Authorization: GNAP 4398.34-12-asvDa.a
Content-Digest: sha-512=:Fb/A5vnawhuuJ5xk2RjGrbbxr6cvinZqd4+JPY85u/\
  JNyTlmRmCOtyVhZ1Oz/cSS4tsYen6fzpCwizy6UQxNBQ==:
Signature-Input: old-key=("@method" "@target-uri" "content-digest" \
  "authorization");created=1618884475;keyid="test-key-ecc-p256"\
  ;tag="gnap"
Signature: old-key=:vN4IKYsJl2RLFe+tYEm4dHM4R4BToqx5D2FfH4ge5WOkgxo\
  dI2QRrjB8rysvoSEGvAfiVJOWsGcPD1lU639Amw==:

{
    "key": {
        "proof": "httpsig",
        "jwk": {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "xyz-2",
            "alg": "RS256",
            "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8xY..."
        }
    }
}
```

The signer then creates a new signature using the new key, adding the signature input and value to the signature base.

```
NOTE: '\' line wrapping per RFC 8792

"@method": POST
"@target-uri": https://server.example.com/token/PRY5NM33
"content-digest": sha-512=:Fb/A5vnawhuuJ5xk2RjGrbbxr6cvinZqd4+JPY85\
  u/JNyTlmRmCOtyVhZ1Oz/cSS4tsYen6fzpCwizy6UQxNBQ==:
"authorization": GNAP 4398.34-12-asvDa.a
"signature";key="old-key": :YdDJjDn2Sq8FR82e5IcOLWmmf6wILoswlnRcz+n\
  M+e8xjFDpWS2YmiMYDqUdri2UiJsZx63T1z7As9Kl6HTGkQ==:
"signature-input";key="old-key": ("@method" "@target-uri" \
  "content-digest" "authorization");created=1618884475\
  ;keyid="test-key-ecc-p256";tag="gnap"
"@signature-params": ("@method" "@target-uri" "content-digest" \
  "authorization" "signature";key="old-key" "signature-input"\
  ;key="old-key");created=1618884480;keyid="xyz-2"
  ;tag="gnap-rotate"
```

This signature is then added to the message:

```
NOTE: '\' line wrapping per RFC 8792

POST /token/PRY5NM33 HTTP/1.1
Host: server.example.com
Authorization: GNAP 4398.34-12-asvDa.a
Content-Digest: sha-512=:Fb/A5vnawhuuJ5xk2RjGrbbxr6cvinZqd4+JPY85u/\
  JNyTlmRmCOtyVhZ1Oz/cSS4tsYen6fzpCwizy6UQxNBQ==:
Signature-Input: old-key=("@method" "@target-uri" "content-digest" \
    "authorization");created=1618884475;keyid="test-key-ecc-p256"\
    ;tag="gnap", \
  new-key=("@method" "@target-uri" "content-digest" \
    "authorization" "signature";key="old-key" "signature-input"\
    ;key="old-key");created=1618884480;keyid="xyz-2"
    ;tag="gnap-rotate"
Signature: old-key=:vN4IKYsJl2RLFe+tYEm4dHM4R4BToqx5D2FfH4ge5WOkgxo\
    dI2QRrjB8rysvoSEGvAfiVJOWsGcPD1lU639Amw==:, \
  new-key=:VWUExXQ0geWeTUKhCfDT7WJyT++OHSVbfPA1ukW0o7mmstdbvIz9iOuH\
    DRFzRBm0MQPFVMpLDFXQdE3vi2SL3ZjzcX2qLwzAtyRB9+RsV2caAA80A5ZGMoo\
    gUsKPk4FFDN7KRUZ0vT9Mo9ycx9Dq/996TOWtAmq5z0YUYEwwn+T6+NcW8rFtms\
    s1ZfXG0EoAfV6ve25p+x40Y1rvDHsfkakTRB4J8jWVDybSe39tjIKQBo3uicDVw\
    twewBMNidIa+66iF3pWj8w9RSb0cncEgvbkHgASqaZeXmxxG4gM8p1HH9v/OqQT\
    Oggm5gTWmCQs4oxEmWsfTOxefunfh3X+Qw==:

{
    "key": {
        "proof": "httpsig",
        "jwk": {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "xyz-2",
            "alg": "RS256",
            "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8xY..."
        }
    }
}
```

The verifier **MUST** validate both signatures before processing the request for key rotation.

7.3.2. Mutual TLS

This method is indicated by the method value `mtls` in string form.

```
{
    "proof": "mtls"
}
```

The signer presents its TLS client certificate during TLS negotiation with the verifier.

In the following non-normative example, the certificate is communicated to the application through the Client-Cert header field from a TLS reverse proxy as per [RFC9440], leading to the following full HTTP request message:

```
POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/jose
Content-Length: 1567
Client-Cert: \
  :MIIC6jCCAdKgAwIBAgIGAXjw74xPMA0GCSqGSIb3DQEBCwUAMDYxNDAyBgNVBAMM\
  K05JWU15QmpzRGp5QkM5UDUzN0Q2SVR6a3BEOE50UmppOXlhcEV6QzY2bVEwHhcN\
  MjEwNDIwMjAxODU0WhcNMjIwMjE0MjAxODU0WjA2MTQwMgYDVQQDDCtOSVlNeUJq\
  c0RqeUJDOVA1MzdENklUemtwRDhOdFJqaTl5YXBFekM2Nm1RMIIBIjANBgkqhkiG\
  9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhYOJ+XOKISdMMShn/G4W9m20mT0VWtQBsmBB\
  kI2cmRt4Ai8BfYdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8I\
  kZ8NMwSrcUIBZGYXjHpwjzvfGvXH/5KJlnR3/uRUp4Z4Ujk2bCaKegDn11V2vxE4\
  1hqaPUnhRZxe0jRETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo+\
  uv4BC0bunS0K3bA/3UgVp7zBlQFoFnLTO2uWp/muLEWGl67gBq9MO3brKXfGhi3k\
  OzywzwPTuq+cVQDyEN7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQIDAQABMA0GCSqG\
  SIb3DQEBCwUAA4IBAQBnYFK0eYHy+hVf2D58usj39lhL5znb/q9G35GBd/XsWfCE\
  wHuLOSZSUmG71bZtrOcx0ptle9bp2kKl4HlSTTfbtpuG5onSa3swRNhtKtUy5NH9\
  W/FLViKWfoPS3kwoEpC1XqKY6l7evoTCtS+kTQRSrCe4vbNprCAZRxz6z1nEeCgu\
  NMk38yTRvx8ihZpVOuU+Ih+dOtVe/ex5IAPYxlQsvtfhsUZqc7IyCcy72WHnRHlU\
  fn3pJm0S5270+Yls3Iv6h3oBAP19i906UjiUTNH3g0xMW+V4uLxgyckt4wD4Mlyv\
  jnaQ7Z3sR6EsXMocAbXHIAJhwKdtU/fLgdwL5vtx:


{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "mtls",
        "cert": "MIIC6jCCAdKgAwIBAgIGAXjw74xPMA0GCSqGSIb3DQEBCwUAMD\
  YxNDAyBgNVBAMMK05JWU15QmpzRGp5QkM5UDUzN0Q2SVR6a3BEOE50UmppOXlhcEV\
  6QzY2bVEwHhcNMjEwNDIwMjAxODU0WhcNMjIwMjE0MjAxODU0WjA2MTQwMgYDVQQD\
  DCtOSVlNeUJqc0RqeUJDOVA1MzdENklUemtwRDhOdFJqaTl5YXBFekM2Nm1RMIIBI\
  jANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhYOJ+XOKISdMMShn/G4W9m20mT\
  0VWtQBsmBBkI2cmRt4Ai8BfYdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8\
  KowlyVy8IkZ8NMwSrcUIBZGYXjHpwjzvfGvXH/5KJlnR3/uRUp4Z4Ujk2bCaKegDn\
  11V2vxE41hqaPUnhRZxe0jRETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDad\
  z8BkPo+uv4BC0bunS0K3bA/3UgVp7zBlQFoFnLTO2uWp/muLEWGl67gBq9MO3brKX\
  fGhi3kOzywzwPTuq+cVQDyEN7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQIDAQABMA0\
  GCSqGSIb3DQEBCwUAA4IBAQBnYFK0eYHy+hVf2D58usj39lhL5znb/q9G35GBd/Xs\
  WfCEwHuLOSZSUmG71bZtrOcx0ptle9bp2kKl4HlSTTfbtpuG5onSa3swRNhtKtUy5\
  NH9W/FLViKWfoPS3kwoEpC1XqKY6l7evoTCtS+kTQRSrCe4vbNprCAZRxz6z1nEeC\
  guNMk38yTRvx8ihZpVOuU+Ih+dOtVe/ex5IAPYxlQsvtfhsUZqc7IyCcy72WHnRHl\
  Ufn3pJm0S5270+Yls3Iv6h3oBAP19i906UjiUTNH3g0xMW+V4uLxgyckt4wD4Mlyv\
  jnaQ7Z3sR6EsXMocAbXHIAJhwKdtU/fLgdwL5vtx"
      }
```

```
        "display": {
          "name": "My Client Display Name",
          "uri": "https://client.foo/"
        },
      },
      "subject": {
          "formats": ["iss_sub", "opaque"]
      }
}
```

The verifier compares the TLS client certificate presented during MTLS negotiation to the expected key of the signer. Since the TLS connection covers the entire message, there are no additional requirements to check.

Note that in many instances, the verifier will not do a full certificate chain validation of the presented TLS client certificate, as the means of trust for this certificate could be in something other than a PKI system, such as a static registration or trust-on-first-use. See Sections 11.3 and 11.4 for some additional considerations for this key proofing method.

7.3.2.1. Key Rotation Using MTLS

Since it is not possible to present two client authenticated certificates to a MTLS connection simultaneously, dynamic key rotation for this proofing method is not defined. Instead, key rotation for MTLS-based client instances is expected to be managed through deployment practices, as discussed in Section 11.4.

7.3.3. Detached JWS

This method is indicated by the method value `jwsd` in string form.

```
{
    "proof": "jwsd"
}
```

The signer creates a JSON Web Signature (JWS) [RFC7515] object as follows.

To protect the request, the JOSE header of the signature contains the following claims:

`kid` (string): The key identifier. **REQUIRED** if the key is presented in JWK format. This **MUST** be the value of the `kid` field of the key.

`alg` (string): The algorithm used to sign the request. The algorithm **MUST** be appropriate to the key presented. If the key is presented as a JWK, this **MUST** be equal to the `alg` parameter of the key. The algorithm **MUST NOT** be `none`. **REQUIRED**.

`typ` (string): The type header, value "gnap-binding-jwsd". **REQUIRED**.

`htm` (string): The HTTP method used to make this request, as a case-sensitive ASCII string. Note that most public HTTP methods are in uppercase ASCII by convention. **REQUIRED**.

`uri` (string): The HTTP URI used for this request. This value **MUST** be an absolute URI, including all path and query components and no fragment components. **REQUIRED**.

`created` (integer): A timestamp of when the signature was created, in integer seconds since UNIX Epoch. **REQUIRED**.

When the request is bound to an access token, the JOSE header **MUST** also include the following:

`ath` (string): The hash of the access token. The value **MUST** be the result of base64url encoding (with no padding) the SHA-256 digest of the ASCII encoding of the associated access token's value. **REQUIRED**.

If the HTTP request has content (such as an HTTP POST or PUT method), the payload of the JWS object is the base64url encoding (without padding) of the SHA-256 digest of the bytes of the content. If the request being made does not have content (such as an HTTP GET, OPTIONS, or DELETE method), the JWS signature is calculated over an empty payload.

The signer presents the signed object in compact form [RFC7515] in the Detached-JWS header field.

In the following non-normative example, the JOSE header contains the following parameters:

```
{
    "alg": "RS256",
    "kid": "gnap-rsa",
    "uri": "https://server.example.com/gnap",
    "htm": "POST",
    "typ": "gnap-binding-jwsd",
    "created": 1618884475
}
```

The request content is the following JSON object:

```
NOTE: '\' line wrapping per RFC 8792

{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "jwsd",
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    }
}
```

This is hashed to the following base64-encoded value:

```
PGiVuOZUcN1tRtUS6tx2b4cBgw9mPgXG3IPB3wY7ctc
```

This leads to the following full HTTP request message:

```
NOTE: '\' line wrapping per RFC 8792

POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/json
Content-Length: 983
Detached-JWS: eyJhbGciOiJSUzI1NiIsImNyZWF0ZWQiOjE2MTg4ODQ0NzUsImh0b\
  SI6IlBPU1QiLCJraWQiOiJnbmFwLXJzYSIsInR5cCI6ImduYXAtYmluZGluZytqd3\
```

```
  NkIiwidXJpIjoiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20vZ25hcCJ9.PGiVuO\
  ZUcN1tRtUS6tx2b4cBgw9mPgXG3IPB3wY7ctc.fUq-SV-A1iFN2MwCRW_yolVtT2_\
  TZA2h5YeXUoi5F2Q2iToC0Tc4drYFOSHIX68knd68RUA7yHqCVP-ZQEd6aL32H69e\
  9zuMiw6O_s4TBKB3vDOvwrhYtDH6fX2hP70cQoO-47OwbqP-ifkrvI3hVgMX9TfjV\
  eKNwnhoNnw3vbu7SNKeqJEbbwZfpESaGepS52xNBlDNMYBQQXxM9OqKJaXffzLFEl\
  -Xe0UnfolVtBraz3aPrPy1C6a4uT7wLda3PaTOVtgysxzii3oJWpuz0WP5kRujzDF\
  wX_EOzW0jsjCSkL-PXaKSpZgEjNjKDMg9irSxUISt1C1T6q3SzRgfuQ


{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "jwsd",
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    }
}
```

When the verifier receives the Detached-JWS header, it **MUST** parse and validate the JWS object. The signature **MUST** be validated against the expected key of the signer. If the HTTP message request contains content, the verifier **MUST** calculate the hash of the content just as the signer does, with no normalization or transformation of the request. All required fields **MUST** be present, and their values **MUST** be valid. All fields **MUST** match the corresponding portions of the HTTP message. For example, the `htm` field of the JWS header has to be the same as the HTTP verb used in the request.

Note that this proofing method depends on a specific cryptographic algorithm, SHA-256, in two ways: 1) the `ath` hash algorithm is hardcoded and 2) the payload of the detached/attached signature is computed using a hardcoded hash. A future version of this document may address crypto-agility for both these uses by replacing ath with a new header that upgrades the algorithm and possibly defining a new JWS header that indicates the HTTP content's hash method.

7.3.3.1. Key Rotation Using Detached JWS

When rotating a key using detached JWS, the message, which includes the new public key value or reference, is first signed with the old key as described above using a JWS object with `typ` header value "gnap-binding-rotation-jwsd". The value of the JWS object is then taken as the payload of a new JWS object, to be signed by the new key using the parameters above.

The value of the new JWS object is sent in the Detached-JWS header.

7.3.4. Attached JWS

This method is indicated by the method value `jws` in string form.

```
{
    "proof": "jws"
}
```

The signer creates a JWS [RFC7515] object as follows.

To protect the request, the JWS header contains the following claims:

`kid` (string): The key identifier. **REQUIRED** if the key is presented in JWK format. This **MUST** be the value of the `kid` field of the key.

`alg` (string): The algorithm used to sign the request. **MUST** be appropriate to the key presented. If the key is presented as a JWK, this **MUST** be equal to the `alg` parameter of the key. **MUST NOT** be `none`. **REQUIRED**.

`typ` (string): The type header, value "gnap-binding-jws". **REQUIRED**.

`htm` (string): The HTTP method used to make this request, as a case-sensitive ASCII string. (Note that most public HTTP methods are in uppercase.) **REQUIRED**.

`uri` (string): The HTTP URI used for this request, including all path and query components and no fragment components. **REQUIRED**.

`created` (integer): A timestamp of when the signature was created, in integer seconds since UNIX Epoch. **REQUIRED**.

When the request is bound to an access token, the JOSE header **MUST** also include the following:

`ath` (string):

The hash of the access token. The value **MUST** be the result of base64url encoding (with no padding) the SHA-256 digest of the ASCII encoding of the associated access token's value. **REQUIRED**.

If the HTTP request has content (such as an HTTP POST or PUT method), the payload of the JWS object is the JSON serialized content of the request, and the object is signed according to JWS and serialized into compact form [RFC7515]. The signer presents the JWS as the content of the request along with a content type of `application/jose`. The verifier **MUST** extract the payload of the JWS and treat it as the request content for further processing.

If the request being made does not have content (such as an HTTP GET, OPTIONS, or DELETE method), the JWS signature is calculated over an empty payload and passed in the `Detached-JWS` header as described in Section 7.3.3.

In the following non-normative example, the JOSE header contains the following parameters:

```
{
    "alg": "RS256",
    "kid": "gnap-rsa",
    "uri": "https://server.example.com/gnap",
    "htm": "POST",
    "typ": "gnap-binding-jws",
    "created": 1618884475
}
```

The request content, used as the JWS Payload, is the following JSON object:

```
NOTE: '\' line wrapping per RFC 8792

{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "jws",
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    },
    "subject": {
        "formats": ["iss_sub", "opaque"]
    }
}
```

This leads to the following full HTTP request message:

```
NOTE: '\' line wrapping per RFC 8792

POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/jose
Content-Length: 1047

eyJhbGciOiJSUzI1NiIsImNyZWF0ZWQiOjE2MTg4ODQ0NzUsImh0bSI6IlBPU1QiLCJ\
raWQiOiJnbmFwLXJzYSIsInR5cCI6ImduYXAtYmluZGluZytqd3NkIiwidXJpIjoiaH\
R0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20vZ25hcCJ9.CnsKICAgICJhY2Nlc3NfdG9r\
ZW4iOiB7CiAgICAgICAgImFjY2VzcyI6IFsKICAgICAgICAgICAgImRvbHBoaW4tbWV\
0YWRhdGEiCiAgICAgICAgXQogICAgfSwKICAgICJpbnRlcmFjdCI6IHsKICAgICAgIC\
Aic3RhcnQiOiBbInJlZGlyZWN0Il0sCiAgICAgICAgImZpbmlzaCI6IHsKICAgICAgI\
CAgICAgIm1ldGhvZCI6ICJyZWRpcmVjdCIsCiAgICAgICAgICAgICJ1cmkiOiAiaHR0\
cHM6Ly9jbGllbnQuZm9vL2NhbGxiYWNrIiwKICAgICAgICAgICAgIm5vbmNlIjogIlZ\
KTE82QTRDQVlMQlhIVFIwS1JPIgogICAgICAgIH0KICAgICAgICAgIH0sCiAgICAiY2xpZW50Ij\
ogewogICAgICAicHJvb2YiOiAiandzIiwKICAgICAgImtleSI6IHsKICAgICAgICAia\
ndrIjogewogICAgICAgICAgICAia2lkIjogImduYXAtcnNhIiwKICAgICAgICAgICAg\
Imt0eSI6ICJSU0EiLAogICAgICAgICAgICAiZSI6ICJBUUFCIiwKICAgICAgICAgICA\
gImFsZyI6ICJSUzI1NiIsCiAgICAgICAgICAgICJuIjogImhZT0otWE9LSVNkTU1TaG\
5fRzRXOW0yMG1UMFZXdFFCc21CQmtJMmNtUnQ0QWk4QmZZZEhzRnpBdFlLT2pwQlIxU\
nBLcEptVkt4SUdOeTBnNlozYWQyWFlzaDhLb3dseVZ5OElrWjhOTXdTcmNVSUJaR1lY\
akhwd2p6dmZHdlhIXzVLSmxuUjNfdVJVcDRaNFVqazJiQ2FLZWdEbjExVjJ2eEU0MWh\
xYVBVbmhSWnhlMGpSRVRkZHpzRTNtdTFTSzhkVENST2p3VWwxNG1VTm84aVRyVG00bj\
BxRGFkejhCa1BvLXV2NEJDMGJ1blMwSzNiQV8zVWdWcDd6QmxRRm9GbkxUTzJ1V3Bfb\
XVMRVdHbDY3Z0JxOU1PM2JyS1hmR2hpM2tPenl3endQVHVxLWNWUUR5RU43YUwwU3hD\
YjNIYzRJZHFEYU1nOHFIVXlPYnBQaXREUSIKICAgICAgICB9CiAgICAgICAgIH0KICAgICA\
gImRpc3BsYXkiOiB7CiAgICAgICAgIm5hbWUiOiAiTXkgQ2xpZW50IERpc3BsYXkgTm\
FtZSIsCiAgICAgICAgInVyaSI6ICJodHRwczovL2NsaWVudC5mb28vIgogICAgICB9L\
AogICAgfSwKICAgICJzdWJqZWN0IjogewogICAgICAgICJmb3JtYXRzIjogWyJpc3Nf\
c3ViIiwgIm9wYXF1ZSJdCiAgICAgICB9Cn0K.MwNoVMQp5hVxI0mCs9LlOUdFtkDXaA1_eT\
vOXq7DOGrtDKH7q4vP2xUq3fH2jRAZqnobo0WdPP3eM3NH5QUjW8pa6_QpwdIWkK7r-\
u_52puE0lPBp7J4U2w4l9gIbg8iknsmWmXeY5F6wiGT8ptfuEYGgmloAJd9LIeNvD3U\
LW2h2dz1Pn2eDnbyvgB0Ugae0BoZB4f69fKWj8Z9wvTIjk1LZJN1PcL7_zT8Lrlic9a\
PyzT7Q9ovkd1s-4whE7TrnGUzFc5mgWUn_gsOpsP5mIIljoEEv-FqOW2RyNYulOZl0Q\
8EnnDHV_vPzrHlUarbGg4YffgtwkQhdK72-JOxYQ
```

When the verifier receives an attached JWS request, it **MUST** parse and validate the JWS object. The signature **MUST** be validated against the expected key of the signer. All required fields **MUST** be present, and their values **MUST** be valid. All fields **MUST** match the corresponding portions of the HTTP message. For example, the `htm` field of the JWS header has to be the same as the HTTP verb used in the request.

Note that this proofing method depends on a specific cryptographic algorithm, SHA-256, in two ways: the `ath` hash algorithm is hardcoded, and computing the payload of the detached/attached signature also uses a hardcoded hash. A future version of this document may address crypto-agility for both these uses by replacing ath with a new header that upgrades the algorithm and possibly defining a new header that indicates the HTTP content's hash method.

7.3.4.1. Key Rotation Using Attached JWS

When rotating a key using attached JWS, the message, which includes the new public key value or reference, is first signed with the old key using a JWS object with `typ` header value "gnap-binding-rotation-jws". The value of the JWS object is then taken as the payload of a new JWS object, to be signed by the new key.

8. Resource Access Rights

GNAP provides a rich structure for describing the protected resources hosted by RSs and accessed by client software. This structure is used when the client instance requests an access token (Section 2.1) and when an access token is returned (Section 3.2). GNAP's structure is designed to be analogous to the OAuth 2.0 Rich Authorization Requests data structure defined in [RFC9396].

The root of this structure is a JSON array. The elements of the JSON array represent rights of access that are associated with the access token. Individual rights of access can be defined by the RS as either an object or a string. The resulting access is the union of all elements within the array.

The access associated with the access token is described using objects that each contain multiple dimensions of access. Each object contains a **REQUIRED** `type` property that determines the type of API that the token is used for and the structure of the rest of the object. There is no expected interoperability between different `type` definitions.

`type` (string): The type of resource request as a string. This field **MAY** define which other fields are allowed in the request object. **REQUIRED**.

The value of the `type` field is under the control of the AS. This field **MUST** be compared using an exact byte match of the string value against known types by the AS. The AS **MUST** ensure that there is no collision between different authorization data types that it supports. The AS **MUST NOT** do any collation or normalization of data types during comparison. It is **RECOMMENDED** that designers of general-purpose APIs use a URI for this field to avoid collisions between multiple API types protected by a single AS.

While it is expected that many APIs will have their own properties, this specification defines a set of common data fields that are designed to be usable across different types of APIs. This specification does not require the use of these common fields by an API definition but, instead, provides them as reusable generic components for API designers to make use of. The allowable values of all fields are determined by the API being protected, as defined by a particular `type` value.

`actions` (array of strings): The types of actions the client instance will take at the RS as an array of strings (for example, a client instance asking for a combination of "read" and "write" access).

`locations` (array of strings): The location of the RS as an array of strings. These strings are typically URIs identifying the location of the RS.

`datatypes` (array of strings): The kinds of data available to the client instance at the RS's API as an array of strings (for example, a client instance asking for access to raw "image" data and "metadata" at a photograph API).

`identifier` (string): A string identifier indicating a specific resource at the RS (for example, a patient identifier for a medical API or a bank account number for a financial API).

`privileges` (array of strings): The types or levels of privilege being requested at the resource (for example, a client instance asking for administrative-level access or access when the RO is no longer online).

The following non-normative example describes three kinds of access (read, write, and delete) to each of two different locations and two different data types (metadata and images) for a single access token using the fictitious `photo-api` type definition.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read",
            "write",
            "delete"
        ],
        "locations": [
            "https://server.example.net/",
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata",
            "images"
        ]
    }
]
```

While the exact semantics of interpreting the fields of an access request object are subject to the definition of the `type`, it is expected that the access requested for each object in the array is the cross-product of all fields of the object. That is to say, the object represents a request for all `actions` listed to be used at all `locations` listed for all possible `datatypes` listed within the object. Assuming the request above was granted, the client instance could assume that it would be able to do a `read` action against the `images` on the first server as well as a `delete` action on the `metadata` of the second server, or any other combination of these fields, using the same access token.

To request a different combination of access, such as requesting one of the possible `actions` against one of the possible `locations` and a different choice of possible `actions` against a different one of the possible `locations`, the client instance can include multiple separate objects in the `resources` array. The total access rights for the resulting access token are the union of all

objects. The following non-normative example uses the same fictitious `photo-api` type definition to request a single access token with more specifically targeted access rights by using two discrete objects within the request.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read"
        ],
        "locations": [
            "https://server.example.net/"
        ],
        "datatypes": [
            "images"
        ]
    },
    {
        "type": "photo-api",
        "actions": [
            "write",
            "delete"
        ],
        "locations": [
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata"
        ]
    }
]
```

The access requested here is for `read` access to `images` on one server as well as `write` and `delete` access for `metadata` on a different server (importantly, without requesting `write` or `delete` access to `images` on the first server).

It is anticipated that API designers will use a combination of common fields defined in this specification as well as fields specific to the API itself. The following non-normative example shows the use of both common and API-specific fields as part of two different fictitious API `type` values. The first access request includes the `actions`, `locations`, and `datatypes` fields specified here as well as the API-specific `geolocation` field. The second access request includes the `actions` and `identifier` fields specified here as well as the API-specific `currency` field.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read",
            "write"
        ],
        "locations": [
            "https://server.example.net/",
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata",
            "images"
        ],
        "geolocation": [
            { lat: -32.364, lng: 153.207 },
            { lat: -35.364, lng: 158.207 }
        ]
    },
    {
        "type": "financial-transaction",
        "actions": [
            "withdraw"
        ],
        "identifier": "account-14-32-32-3",
        "currency": "USD"
    }
]
```

If this request is approved, the resulting access token's access rights will be the union of the requested types of access for each of the two APIs, just as above.

8.1. Requesting Resources by Reference

Instead of sending an object describing the requested resource (Section 8), access rights **MAY** be communicated as a string known to the AS representing the access being requested. Just like access rights communicated as an object, access rights communicated as reference strings indicate a specific access at a protected resource. In the following non-normative example, three distinct resource access rights are being requested.

```
"access": [
    "read", "dolphin-metadata", "some other thing"
]
```

This value is opaque to the client instance and **MAY** be any valid JSON string; therefore, it could include spaces, Unicode characters, and properly escaped string sequences. However, in some situations, the value is intended to be seen and understood by the client software's developer. In such cases, the API designer choosing any such human-readable strings **SHOULD** take steps to ensure the string values are not easily confused by a developer, such as by limiting the strings to easily disambiguated characters.

This functionality is similar in practice to OAuth 2.0's `scope` parameter [RFC6749], where a single string represents the set of access rights requested by the client instance. As such, the reference string could contain any valid OAuth 2.0 scope value, as in Appendix B.5. Note that the reference string here is not bound to the same character restrictions as OAuth 2.0's `scope` definition.

A single `access` array **MAY** include both object-type and string-type resource items. In this non-normative example, the client instance is requesting access to a `photo-api` and `financial-transaction` API type as well as the reference values of `read`, `dolphin-metadata`, and `some other thing`.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read",
            "write",
            "delete"
        ],
        "locations": [
            "https://server.example.net/",
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata",
            "images"
        ]
    },
    "read",
    "dolphin-metadata",
    {
        "type": "financial-transaction",
        "actions": [
            "withdraw"
        ],
        "identifier": "account-14-32-32-3",
        "currency": "USD"
    },
    "some other thing"
]
```

The requested access is the union of all elements of the array, including both objects and reference strings.

In order to facilitate the use of both object and reference strings to access the same kind of APIs, the API designer can define a clear mapping between these forms. One possible approach for choosing reference string values is to use the same value as the `type` parameter from the fully specified object, with the API defining a set of default behaviors in this case. For example, an API definition could declare the following string:

```
"access": [
    "photo-api"
]
```

As being equivalent to the following fully defined object:

```
"access": [
    {
        "type": "photo-api",
        "actions": [ "read", "write", "delete" ],
        "datatypes": [ "metadata", "image" ]
    }
]
```

The exact mechanisms for relating reference strings is up to the API designer. These are enforced by the AS, and the details are out of scope for this specification.

9. Discovery

By design, GNAP minimizes the need for any pre-flight discovery. To begin a request, the client instance only needs to know the grant endpoint of the AS (a single URI) and which keys it will use to sign the request. Everything else can be negotiated dynamically in the course of the protocol.

However, the AS can have limits on its allowed functionality. If the client instance wants to optimize its calls to the AS before making a request, it **MAY** send an HTTP OPTIONS request to the grant request endpoint to retrieve the server's discovery information. The AS **MUST** respond with a JSON document with Content-Type `application/json` containing a single object with the following fields:

`grant_request_endpoint` (string): The location of the AS's grant request endpoint. The location **MUST** be an absolute URL [RFC3986] with a scheme component (which **MUST** be "https"), a host component, and optionally port, path, and query components and no fragment components. This URL **MUST** match the URL the client instance used to make the discovery request. **REQUIRED**.

`interaction_start_modes_supported` (array of strings): A list of the AS's interaction start methods. The values of this list correspond to the possible values for the interaction start field of the request (Section 2.5.1) and **MUST** be values from the "GNAP Interaction Start Modes" registry (Section 10.9). **OPTIONAL**.

`interaction_finish_methods_supported` (array of strings): A list of the AS's interaction finish methods. The values of this list correspond to the possible values for the method element of the interaction finish field of the request (Section 2.5.2) and **MUST** be values from the "GNAP Interaction Finish Methods" registry (Section 10.10). **OPTIONAL**.

`key_proofs_supported` (array of strings): A list of the AS's supported key proofing mechanisms. The values of this list correspond to possible values of the `proof` field of the key section of the request (Section 7.1) and **MUST** be values from the "GNAP Key Proofing Methods" registry (Section 10.16). **OPTIONAL**.

`sub_id_formats_supported` (array of strings): A list of the AS's supported Subject Identifier formats. The values of this list correspond to possible values of the Subject Identifier field of the request (Section 2.2) and **MUST** be values from the "Subject Identifier Formats" registry [Subj-ID-Formats]. **OPTIONAL**.

`assertion_formats_supported` (array of strings): A list of the AS's supported assertion formats. The values of this list correspond to possible values of the subject assertion field of the request (Section 2.2) and **MUST** be values from the "GNAP Assertion Formats" registry (Section 10.6). **OPTIONAL**.

`key_rotation_supported` (boolean): The boolean "true" indicates that rotation of access token bound keys by the client (Section 6.1.1) is supported by the AS. The absence of this field or a boolean "false" value indicates that this feature is not supported.

The information returned from this method is for optimization purposes only. The AS **MAY** deny any request, or any portion of a request, even if it lists a capability as supported. For example, if a given client instance can be registered with the `mtls` key proofing mechanism but the AS also returns other proofing methods from the discovery document, then the AS will still deny a request from that client instance using a different proofing mechanism. Similarly, an AS with `key_rotation_supported` set to "true" can still deny any request for rotating any access token's key for a variety of reasons.

Additional fields can be defined in the "GNAP Authorization Server Discovery Fields" registry (Section 10.18).

9.1. RS-First Method of AS Discovery

If the client instance calls an RS without an access token or with an invalid access token, the RS **SHOULD** be explicit about the fact that GNAP needs to be used to access the resource by responding with the WWW-Authenticate header field and a GNAP challenge.

In some situations, the client instance might want to know with which specific AS it needs to negotiate for access to that RS. The RS **MAY** additionally return the following **OPTIONAL** parameters:

`as_uri`: The URI of the grant endpoint of the GNAP AS. Used by the client instance to call the AS to request an access token.

`referrer`: The URI of the GNAP RS. Sent by the client instance in the Referer header as part of the grant request.

`access`: An opaque access reference as defined in Section 8.1. **MUST** be sufficient for at least the action the client instance was attempting to take at the RS and **MAY** allow additional access rights as well. Sent by the client as an access right in the grant request.

The client instance **SHOULD** then use both the `referrer` and `access` parameters in its access token request. The client instance **MUST** check that the `referrer` parameter is equal to the URI of the RS using the simple string comparison method in Section 6.2.1 of [RFC3986].

The means for the RS to determine the value for the `access` reference are out of scope of this specification, but some dynamic methods are discussed in [GNAP-RS].

When receiving the following response from the RS:

```
NOTE: '\' line wrapping per RFC 8792

WWW-Authenticate: \
  GNAP as_uri=https://as.example/tx\
  ;access=FWWIKYBQ6U56NL1\
  ;referrer=https://rs.example
```

The client instance then makes a request to the `as_uri` as described in Section 2, with the value of `referrer` passed as an HTTP Referer header field and the `access` reference passed unchanged into the `access` array in the `access_token` portion of the request. The client instance **MAY** request additional resources and other information.

In the following non-normative example, the client instance is requesting a single access token using the opaque access reference `FWWIKYBQ6U56NL1` received from the RS in addition to the `dolphin-metadata` that the client instance has been configured with out of band.

```
POST /tx HTTP/1.1
Host: as.example
Referer: https://rs.example/resource
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "FWWIKYBQ6U56NL1",
            "dolphin-metadata"
        ]
    },
    "client": "KHRS6X63AJ7C7C4AZ9AO"
}
```

The client instance includes the Referer header field as a way for the AS to know that the process is initiated through a discovery process at the RS.

If issued, the resulting access token would contain sufficient access to be used at both referenced resources.

Security considerations, especially related to the potential of a compromised RS (Section 11.37) redirecting the requests of an otherwise properly authenticated client, need to be carefully considered when allowing such a discovery process. This risk can be mitigated by an alternative pre-registration process so that the client knows which AS protects which RS. There are also privacy considerations related to revealing which AS is protecting a given resource; these are discussed in Section 12.4.1.

9.2. Dynamic Grant Endpoint Discovery

Additional methods of discovering the appropriate grant endpoint for a given application are outside the scope of this specification. This limitation is intentional, as many applications rely on static configuration between the client instance and AS, as is common in OAuth 2.0. However, the dynamic nature of GNAP makes it a prime candidate for other extensions defining methods for discovery of the appropriate AS grant endpoint at runtime. Advanced use cases could define contextual methods for securely providing this endpoint to the client instance. Furthermore, GNAP's design intentionally requires the client instance to only know the grant endpoint and not additional parameters, since other functions and values can be disclosed and negotiated during the grant process.

10. IANA Considerations

IANA has added values to existing registries as well as created 16 registries for GNAP [GNAP-REG] and populated those registries with initial values as described in this section.

All use of value typing is based on data types in [RFC8259] and **MUST** be one of the following: number, object, string, boolean, or array. When the type is array, the contents of the array **MUST** be specified, as in "array of objects" when one subtype is allowed or "array of strings/objects" when multiple simultaneous subtypes are allowed. When the type is object, the structure of the object **MUST** be specified in the definition. If a parameter is available in different types, each type **SHOULD** be registered separately.

General guidance for extension parameters is found in Appendix D.

10.1. HTTP Authentication Scheme Registration

IANA has registered of the following scheme in the "HTTP Authentication Schemes" registry [Auth-Schemes] defined in Section 18.5 of [HTTP]:

Authentication Scheme Name: GNAP

Reference: Section 7.2 of RFC 9635

10.2. Media Type Registration

Per this section, IANA has registered the following media types [RFC2046] in the "Media Types" registry [MediaTypes] as described in [RFC6838].

10.2.1. application/gnap-binding-jwsd

This media type indicates that the content is a GNAP message to be bound with a detached JWS mechanism.

Type name: application

Subtype name: gnap-binding-jwsd

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See Section 11 of RFC 9635.

Interoperability considerations: N/A

Published specification: RFC 9635

Applications that use this media type: GNAP

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A
Magic number(s): N/A
File extension(s): N/A
Macintosh file type code(s): N/A

Person & email address to contact for further information: IETF GNAP Working Group (txauth@ietf.org)

Intended usage: COMMON

Restrictions on usage: none

Author: IETF GNAP Working Group (txauth@ietf.org)

Change Controller: IETF

10.2.2. application/gnap-binding-jws

This media type indicates that the content is a GNAP message to be bound with an attached JWS mechanism.

Type name: application

Subtype name: gnap-binding-jws

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See Section 11 of RFC 9635.

Interoperability considerations: N/A

Published specification: RFC 9635

Applications that use this media type: GNAP

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A
Magic number(s): N/A
File extension(s): N/A
Macintosh file type code(s): N/A

Person & email address to contact for further information: IETF GNAP Working Group (txauth@ietf.org)

Intended usage: COMMON

Restrictions on usage: none

Author: IETF GNAP Working Group (txauth@ietf.org)

Change Controller: IETF

10.2.3. application/gnap-binding-rotation-jwsd

This media type indicates that the content is a GNAP token rotation message to be bound with a detached JWS mechanism.

Type name: application

Subtype name: gnap-binding-rotation-jwsd

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See Section 11 of RFC 9635.

Interoperability considerations: N/A

Published specification: RFC 9635

Applications that use this media type: GNAP

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A
Magic number(s): N/A
File extension(s): N/A
Macintosh file type code(s): N/A

Person & email address to contact for further information: IETF GNAP Working Group (txauth@ietf.org)

Intended usage: COMMON

Restrictions on usage: none

Author: IETF GNAP Working Group (txauth@ietf.org)

Change Controller: IETF

10.2.4. application/gnap-binding-rotation-jws

This media type indicates that the content is a GNAP token rotation message to be bound with an attached JWS mechanism.

Type name: application

Subtype name: gnap-binding-rotation-jws

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See Section 11 of RFC 9635.

Interoperability considerations: N/A

Published specification: RFC 9635

Applications that use this media type: GNAP

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): N/A
File extension(s): N/A
Macintosh file type code(s): N/A

Person & email address to contact for further information: IETF GNAP Working Group (txauth@ietf.org)

Intended usage: COMMON

Restrictions on usage: none

Author: IETF GNAP Working Group (txauth@ietf.org)

Change Controller: IETF

10.3. GNAP Grant Request Parameters

This document defines a GNAP grant request, for which IANA has created and maintains a new registry titled "GNAP Grant Request Parameters". Initial values for this registry are given in Section 10.3.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The designated expert (DE) is expected to ensure the following:

- All registrations follow the template presented in Section 10.3.1.
- The request parameter's definition is sufficiently orthogonal to existing functionality provided by existing parameters.
- Registrations for the same name with different types are sufficiently close in functionality so as not to cause confusion for developers.
- The request parameter's definition specifies the expected behavior of the AS in response to the request parameter for each potential state of the grant request.

10.3.1. Registration Template

Name:
An identifier for the parameter.

Type:
The JSON type allowed for the value.

Reference:
Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.3.2. Initial Contents

Name	Type	Reference
access_token	object	Section 2.1.1 of RFC 9635
access_token	array of objects	Section 2.1.2 of RFC 9635
subject	object	Section 2.2 of RFC 9635
client	object	Section 2.3 of RFC 9635
client	string	Section 2.3.1 of RFC 9635
user	object	Section 2.4 of RFC 9635
user	string	Section 2.4.1 of RFC 9635
interact	object	Section 2.5 of RFC 9635
interact_ref	string	Section 5.1 of RFC 9635

Table 1

10.4. GNAP Access Token Flags

This document defines GNAP access token flags, for which IANA has created and maintains a new registry titled "GNAP Access Token Flags". Initial values for this registry are given in Section 10.4.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.4.1.
- The flag specifies whether it applies to requests for tokens to the AS, responses with tokens from the AS, or both.

10.4.1. Registration Template

Name:
: An identifier for the parameter.

Allowed Use:
: Where the flag is allowed to occur. Possible values are "Request", "Response", and "Request, Response".

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.4.2. Initial Contents

Name	Allowed Use	Reference
bearer	Request, Response	Sections 2.1.1 and 3.2.1 of RFC 9635
durable	Response	Section 3.2.1 of RFC 9635

Table 2

10.5. GNAP Subject Information Request Fields

This document defines a means to request subject information from the AS to the client instance, for which IANA has created and maintains a new registry titled "GNAP Subject Information Request Fields". Initial values for this registry are given in Section 10.5.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.5.1.
- Registrations for the same name with different types are sufficiently close in functionality so as not to cause confusion for developers.

10.5.1. Registration Template

Name:
: An identifier for the parameter.

Type:
: The JSON type allowed for the value.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.5.2. Initial Contents

Name	Type	Reference
sub_id_formats	array of strings	Section 2.2 of RFC 9635
assertion_formats	array of strings	Section 2.2 of RFC 9635

Name	Type	Reference
sub_ids	array of objects	Section 2.2 of RFC 9635

Table 3

10.6. GNAP Assertion Formats

This document defines a means to pass identity assertions between the AS and client instance, for which IANA has created and maintains a new registry titled "GNAP Assertion Formats". Initial values for this registry are given in Section 10.6.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.6.1.
- The definition specifies the serialization format of the assertion value as used within GNAP.

10.6.1. Registration Template

Name:
: An identifier for the assertion format.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.6.2. Initial Contents

Name	Reference
id_token	Section 3.4.1 of RFC 9635
saml2	Section 3.4.1 of RFC 9635

Table 4

10.7. GNAP Client Instance Fields

This document defines a means to send information about the client instance, for which IANA has created and maintains a new registry titled "GNAP Client Instance Fields". Initial values for this registry are given in Section 10.7.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.7.1.

- Registrations for the same name with different types are sufficiently close in functionality so as not to cause confusion for developers.

10.7.1. Registration Template

Name:
: An identifier for the parameter.

Type:
: The JSON type allowed for the value.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.7.2. Initial Contents

Name	Type	Reference
key	object	Section 7.1 of RFC 9635
key	string	Section 7.1.1 of RFC 9635
class_id	string	Section 2.3 of RFC 9635
display	object	Section 2.3.2 of RFC 9635

Table 5

10.8. GNAP Client Instance Display Fields

This document defines a means to send end-user-facing displayable information about the client instance, for which IANA has created and maintains a new registry titled "GNAP Client Instance Display Fields". Initial values for this registry are given in Section 10.8.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.8.1.
- Registrations for the same name with different types are sufficiently close in functionality so as not to cause confusion for developers.

10.8.1. Registration Template

Name:
: An identifier for the parameter.

Type:
: The JSON type allowed for the value.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.8.2. Initial Contents

Name	Type	Reference
name	string	Section 2.3.2 of RFC 9635
uri	string	Section 2.3.2 of RFC 9635
logo_uri	string	Section 2.3.2 of RFC 9635

Table 6

10.9. GNAP Interaction Start Modes

This document defines a means for the client instance to begin interaction between the end user and the AS, for which IANA has created and maintains a new registry titled "GNAP Interaction Start Modes". Initial values for this registry are given in Section 10.9.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.9.1.
- Registrations for the same name with different types are sufficiently close in functionality so as not to cause confusion for developers.
- Any registration using an "object" type declares all additional parameters, their optionality, and their purpose.
- The start mode clearly defines what actions the client is expected to take to begin interaction, what the expected user experience is, and any security considerations for this communication from either party.
- The start mode documents incompatibilities with other start modes or finish methods, if applicable.
- The start mode provides enough information to uniquely identify the grant request during the interaction. For example, in the `redirect` and `app` modes, this is done using a unique URI (including its parameters). In the `user_code` and `user_code_uri` modes, this is done using the value of the user code.

10.9.1. Registration Template

Mode:
: An identifier for the interaction start mode.

Type:
: The JSON type for the value, either "string" or "object", as described in Section 2.5.1.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.9.2. Initial Contents

Mode	Type	Reference
redirect	string	Section 2.5.1.1 of RFC 9635
app	string	Section 2.5.1.2 of RFC 9635
user_code	string	Section 2.5.1.3 of RFC 9635
user_code_uri	string	Section 2.5.1.4 of RFC 9635

Table 7

10.10. GNAP Interaction Finish Methods

This document defines a means for the client instance to be notified of the end of interaction between the end user and the AS, for which IANA has created and maintains a new registry titled "GNAP Interaction Finish Methods". Initial values for this registry are given in Section 10.10.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.10.1.
- All finish methods clearly define what actions the AS is expected to take, what listening methods the client instance needs to enable, and any security considerations for this communication from either party.
- All finish methods document incompatibilities with any start modes, if applicable.

10.10.1. Registration Template

Method:
: An identifier for the interaction finish method.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.10.2. Initial Contents

Method	Reference
redirect	Section 2.5.2.1 of RFC 9635
push	Section 2.5.2.2 of RFC 9635

Table 8

10.11. GNAP Interaction Hints

This document defines a set of hints that a client instance can provide to the AS to facilitate interaction with the end user, for which IANA has created and maintains a new registry titled "GNAP Interaction Hints". Initial values for this registry are given in Section 10.11.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.11.1.
- All interaction hints clearly document the expected behaviors of the AS in response to the hint, and an AS not processing the hint does not impede the operation of the AS or client instance.

10.11.1. Registration Template

Name:
: An identifier for the parameter.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.11.2. Initial Contents

Name	Reference
ui_locales	Section 2.5.3 of RFC 9635

Table 9

10.12. GNAP Grant Response Parameters

This document defines a GNAP grant response, for which IANA has created and maintains a new registry titled "GNAP Grant Response Parameters". Initial values for this registry are given in Section 10.12.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.12.1.
- The response parameter's definition is sufficiently orthogonal to existing functionality provided by existing parameters.
- Registrations for the same name with different types are sufficiently close in functionality so as not to cause confusion for developers.
- The response parameter's definition specifies grant states for which the client instance can expect this parameter to appear in a response message.

10.12.1. Registration Template

Name:
: An identifier for the parameter.

Type:
: The JSON type allowed for the value.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.12.2. Initial Contents

Name	Type	Reference
continue	object	Section 3.1 of RFC 9635
acces_token	object	Section 3.2.1 of RFC 9635
acces_token	array of objects	Section 3.2.2 of RFC 9635
interact	object	Section 3.3 of RFC 9635
subject	object	Section 3.4 of RFC 9635
instance_id	string	Section 3.5 of RFC 9635

Name	Type	Reference
error	object	Section 3.6 of RFC 9635

Table 10

10.13. GNAP Interaction Mode Responses

This document defines a means for the AS to provide the client instance with information that is required to complete a particular interaction mode, for which IANA has created and maintains a new registry titled "GNAP Interaction Mode Responses". Initial values for this registry are given in Section 10.13.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.13.1.
- If the name of the registration matches the name of an interaction start mode, the response parameter is unambiguously associated with the interaction start mode of the same name.

10.13.1. Registration Template

Name:
: An identifier for the parameter.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.13.2. Initial Contents

Name	Reference
redirect	Section 3.3 of RFC 9635
app	Section 3.3 of RFC 9635
user_code	Section 3.3 of RFC 9635
user_code_uri	Section 3.3 of RFC 9635
finish	Section 3.3 of RFC 9635
expires_in	Section 3.3 of RFC 9635

Table 11

10.14. GNAP Subject Information Response Fields

This document defines a means to return subject information from the AS to the client instance, for which IANA has created and maintains a new registry titled "GNAP Subject Information Response Fields". Initial values for this registry are given in Section 10.14.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.14.1.
- Registrations for the same name with different types are sufficiently close in functionality so as not to cause confusion for developers.

10.14.1. Registration Template

Name:
: An identifier for the parameter.

Type:
: The JSON type allowed for the value.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.14.2. Initial Contents

Name	Type	Reference
sub_ids	array of objects	Section 3.4 of RFC 9635
assertions	array of objects	Section 3.4 of RFC 9635
updated_at	string	Section 3.4 of RFC 9635

Table 12

10.15. GNAP Error Codes

This document defines a set of errors that the AS can return to the client instance, for which IANA has created and maintains a new registry titled "GNAP Error Codes". Initial values for this registry are given in Section 10.15.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.15.1.
- The error response is sufficiently unique from other errors to provide actionable information to the client instance.
- The definition of the error response specifies all conditions in which the error response is returned and the client instance's expected action.

10.15.1. Registration Template

Error:
: A unique string code for the error.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.15.2. Initial Contents

Error	Reference
invalid_request	Section 3.6 of RFC 9635
invalid_client	Section 3.6 of RFC 9635
invalid_interaction	Section 3.6 of RFC 9635
invalid_flag	Section 3.6 of RFC 9635
invalid_rotation	Section 3.6 of RFC 9635
key_rotation_not_supported	Section 3.6 of RFC 9635
invalid_continuation	Section 3.6 of RFC 9635
user_denied	Section 3.6 of RFC 9635
request_denied	Section 3.6 of RFC 9635
unknown_user	Section 3.6 of RFC 9635
unknown_interaction	Section 3.6 of RFC 9635
too_fast	Section 3.6 of RFC 9635
too_many_attempts	Section 3.6 of RFC 9635

Table 13

10.16. GNAP Key Proofing Methods

This document defines methods that the client instance can use to prove possession of a key, for which IANA has created and maintains a new registry titled "GNAP Key Proofing Methods". Initial values for this registry are given in Section 10.16.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.16.1.
- Registrations for the same name with different types are sufficiently close in functionality so as not to cause confusion for developers.
- The proofing method provides sufficient coverage of and binding to the protocol messages to which it is applied.
- The proofing method definition clearly enumerates how all requirements in Section 7.3 are fulfilled by the definition.

10.16.1. Registration Template

Method:
: A unique string code for the key proofing method.

Type:
: The JSON type allowed for the value.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.16.2. Initial Contents

Method	Type	Reference
httpsig	string	Section 7.3.1 of RFC 9635
httpsig	object	Section 7.3.1 of RFC 9635
mtls	string	Section 7.3.2 of RFC 9635
jwsd	string	Section 7.3.3 of RFC 9635
jws	string	Section 7.3.4 of RFC 9635

Table 14

10.17. GNAP Key Formats

This document defines formats for a public key value, for which IANA has created and maintains a new registry titled "GNAP Key Formats". Initial values for this registry are given in Section 10.17.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.17.1.
- The key format specifies the structure and serialization of the key material.

10.17.1. Registration Template

Format:
: A unique string code for the key format.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.17.2. Initial Contents

Format	Reference
jwk	Section 7.1 of RFC 9635
cert	Section 7.1 of RFC 9635
cert#S256	Section 7.1 of RFC 9635

Table 15

10.18. GNAP Authorization Server Discovery Fields

This document defines a discovery document for an AS, for which IANA has created and maintains a new registry titled "GNAP Authorization Server Discovery Fields". Initial values for this registry are given in Section 10.18.2. Future assignments and modifications to existing assignments are to be made through the Specification Required registration policy [RFC8126].

The DE is expected to ensure the following:

- All registrations follow the template presented in Section 10.18.1.
- Registrations for the same name with different types are sufficiently close in functionality so as not to cause confusion for developers.

- The values in the discovery document are sufficient to provide optimization and hints to the client instance, but knowledge of the discovered value is not required for starting a transaction with the AS.

10.18.1. Registration Template

Name:
: An identifier for the parameter.

Type:
: The JSON type allowed for the value.

Reference:
: Reference to one or more documents that specify the value, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

10.18.2. Initial Contents

Name	Type	Reference
grant_request_endpoint	string	Section 9 of RFC 9635
interaction_start_modes_supported	array of strings	Section 9 of RFC 9635
interaction_finish_methods_supported	array of strings	Section 9 of RFC 9635
key_proofs_supported	array of strings	Section 9 of RFC 9635
sub_id_formats_supported	array of strings	Section 9 of RFC 9635
assertion_formats_supported	array of strings	Section 9 of RFC 9635
key_rotation_supported	boolean	Section 9 of RFC 9635

Table 16

11. Security Considerations

In addition to the normative requirements in this document, implementors are strongly encouraged to consider these additional security considerations in implementations and deployments of GNAP.

11.1. TLS Protection in Transit

All requests in GNAP made over untrusted network connections have to be made over TLS as outlined in [BCP195] to protect the contents of the request and response from manipulation and interception by an attacker. This includes all requests from a client instance to the AS, all requests from the client instance to an RS, and any requests back to a client instance such as the

push-based interaction finish method. Additionally, all requests between a browser and other components, such as during redirect-based interaction, need to be made over TLS or use equivalent protection such as a network connection local to the browser ("localhost").

Even though requests from the client instance to the AS are signed, the signature method alone does not protect the request from interception by an attacker. TLS protects the response as well as the request, preventing an attacker from intercepting requested information as it is returned. This is particularly important in this specification for security artifacts such as nonces and for personal information such as subject information.

The use of key-bound access tokens does not negate the requirement for protecting calls to the RS with TLS. The keys and signatures associated with a bound access token will prevent an attacker from using a stolen token; however, without TLS, an attacker would be able to watch the data being sent to the RS and returned from the RS during legitimate use of the client instance under attack. Additionally, without TLS, an attacker would be able to profile the calls made between the client instance and RS, possibly gaining information about the functioning of the API between the client software and RS software that would otherwise be unknown to the attacker.

Note that connections from the end user and RO's browser also need to be protected with TLS. This applies during initial redirects to an AS's components during interaction, during any interaction with the RO, and during any redirect back to the client instance. Without TLS protection on these portions of the process, an attacker could wait for a valid request to start and then take over the RO's interaction session.

11.2. Signing Requests from the Client Software

Even though all requests in GNAP need to be transmitted over TLS or its equivalent, the use of TLS alone is not sufficient to protect all parts of a multi-party and multi-stage protocol like GNAP, and TLS is not targeted at tying multiple requests to each other over time. To account for this, GNAP makes use of message-level protection and key presentation mechanisms that strongly associate a request with a key held by the client instance (see Section 7).

During the initial request from a client instance to the AS, the client instance has to identify and prove possession of a cryptographic key. If the key is known to the AS, e.g., previously registered or dereferenceable to a trusted source, the AS can associate a set of policies to the client instance identified by the key. Without the requirement that the client instance prove that it holds that key, the AS could not trust that the connection came from any particular client and could not apply any associated policies.

Even more importantly, the client instance proving possession of a key on the first request allows the AS to associate future requests with each other by binding all future requests in that transaction to the same key. The access token used for grant continuation is bound to the same key and proofing mechanism used by the client instance in its initial request; this means that the client instance needs to prove possession of that same key in future requests, which allows the AS to be sure that the same client instance is executing the follow-ups for a given ongoing grant request. Therefore, the AS has to ensure that all subsequent requests for a grant are associated with the same key that started the grant or with the most recent rotation of that key. This need

holds true even if the initial key is previously unknown to the AS, such as would be the case when a client instance creates an ephemeral key for its request. Without this ongoing association, an attacker would be able to impersonate a client instance in the midst of a grant request, potentially stealing access tokens and subject information with impunity.

Additionally, all access tokens in GNAP default to be associated with the key that was presented during the grant request that created the access token. This association allows an RS to know that the presenter of the access token is the same party that the token was issued to, as identified by their keys. While non-bound bearer tokens are an option in GNAP, these types of tokens have their own trade-offs, which are discussed in Section 11.9.

TLS functions at the transport layer, ensuring that only the parties on either end of that connection can read the information passed along that connection. Each time a new connection is made, such as for a new HTTP request, a new trust that is mostly unrelated to previous connections is re-established. While modern TLS does make use of session resumption, this still needs to be augmented with authentication methods to determine the identity of parties on the connections. In other words, it is not possible with TLS alone to know that the same party is making a set of calls over time, since each time a new TLS connection is established, both the client and the server (or the server only when using MTLS (Section 7.3.2)) have to validate the other party's identity. Such a verification can be achieved via methods described in [RFC9525], but these are not enough to establish the identity of the client instance in many cases.

To counter this, GNAP defines a set of key binding methods in Section 7.3 that allows authentication and proof of possession by the caller, which is usually the client instance. These methods are intended to be used in addition to TLS on all connections.

11.3. MTLS Message Integrity

The MTLS key proofing mechanism (Section 7.3.2) provides a means for a client instance to present a key using a certificate at the TLS layer. Since TLS protects the entire HTTP message in transit, verification of the TLS client certificate presented with the message provides a sufficient binding between the two. However, since TLS is functioning at a separate layer from HTTP, there is no direct connection between the TLS key presentation and the message itself, other than the fact that the message was presented over the TLS channel. That is to say, any HTTP message can be presented over the TLS channel in question with the same level of trust. The verifier is responsible for ensuring the key in the TLS client certificate is the one expected for a particular request. For example, if the request is a grant request (Section 2), the AS needs to compare the TLS client certificate presented at the TLS layer to the key identified in the request content itself (either by value or through a referenced identifier).

Furthermore, the prevalence of the TLS terminating reverse proxy (TTRP) pattern in deployments adds a wrinkle to the situation. In this common pattern, the TTRP validates the TLS connection and then forwards the HTTP message contents onward to an internal system for processing. The system processing the HTTP message no longer has access to the original TLS connection's information and context. To compensate for this, the TTRP could inject the TLS client certificate into the forwarded request using the HTTP Client-Cert header field [RFC9111], giving the downstream system access to the certificate information. The TTRP has to be trusted to

provide accurate certificate information, and the connection between the TTRP and the downstream system also has to be protected. The TTRP could provide some additional assurance, for example, by adding its own signature to the Client-Cert header field using HTTP message signatures [RFC9421]. This signature would be effectively ignored by GNAP (since it would not use GNAP's `tag` parameter value) but would be understood by the downstream service as part of its deployment.

Additional considerations for different types of deployment patterns and key distribution mechanisms for MTLS are found in Section 11.4.

11.4. MTLS Deployment Patterns

GNAP does not specify how a client instance's keys could be made known to the AS ahead of time. The Public Key Infrastructure (PKI) can be used to manage the keys used by client instances when calling the AS, allowing the AS to trust a root key from a trusted authority. This method is particularly relevant to the MTLS key proofing method, where the client instance presents its certificate to the AS as part of the TLS connection. An AS using PKI to validate the MTLS connection would need to ensure that the presented certificate was issued by a trusted certificate authority before allowing the connection to continue. PKI-based certificates would allow a key to be revoked and rotated through management at the certificate authority without requiring additional registration or management at the AS. The PKI required to manage mutually authenticated TLS has historically been difficult to deploy, especially at scale, but it remains an appropriate solution for systems where the required management overhead is not an impediment.

MTLS in GNAP need not use a PKI backing, as self-signed certificates and certificates from untrusted authorities can still be presented as part of a TLS connection. In this case, the verifier would validate the connection but accept whatever certificate was presented by the client software. This specific certificate can then be bound to all future connections from that client software by being bound to the resulting access tokens, in a trust-on-first-use pattern. See Section 11.3 for more considerations on MTLS as a key proofing mechanism.

11.5. Protection of Client Instance Key Material

Client instances are identified by their unique keys, and anyone with access to a client instance's key material will be able to impersonate that client instance to all parties. This is true for both calls to the AS as well as calls to an RS using an access token bound to the client instance's unique key. As a consequence, it is of utmost importance for a client instance to protect its private key material.

Different types of client software have different methods for creating, managing, and registering keys. GNAP explicitly allows for ephemeral clients such as single-page applications (SPAs) and single-user clients (such as mobile applications) to create and present their own keys during the initial grant request without any explicit pre-registration step. The client software can securely generate a key pair on the device and present the public key, along with proof of holding the associated private key, to the AS as part of the initial request. To facilitate trust in these ephemeral keys, GNAP further allows for an extensible set of client information to be passed

with the request. This information can include device posture and third-party attestations of the client software's provenance and authenticity, depending on the needs and capabilities of the client software and its deployment.

From GNAP's perspective, each distinct key is a different client instance. However, multiple client instances can be grouped together by an AS policy and treated similarly to each other. For instance, if an AS knows of several different keys for different servers within a cluster, the AS can decide that authorization of one of these servers applies to all other servers within the cluster. An AS that chooses to do this needs to be careful with how it groups different client keys together in its policy, since the breach of one instance would have direct effects on the others in the cluster.

Additionally, if an end user controls multiple instances of a single type of client software, such as having an application installed on multiple devices, each of these instances is expected to have a separate key and be issued separate access tokens. However, if the AS is able to group these separate instances together as described above, it can streamline the authorization process for new instances of the same client software. For example, if two client instances can present proof of a valid installation of a piece of client software, the AS would be able to associate the approval of the first instance of this software to all related instances. The AS could then choose to bypass an explicit prompt of the RO for approval during authorization, since such approval has already been given. An AS doing such a process would need to take assurance measures that the different instances are in fact correlated and authentic, as well as ensure that the expected RO is in control of the client instance.

Finally, if multiple instances of client software each have the same key, then from GNAP's perspective, these are functionally the same client instance as GNAP has no reasonable way to differentiate between them. This situation could happen if multiple instances within a cluster can securely share secret information among themselves. Even though there are multiple copies of the software, the shared key makes these copies all present as a single instance. It is considered bad practice to share keys between copies of software unless they are very tightly integrated with each other and can be closely managed. It is particularly bad practice to allow an end user to copy keys between client instances and to willingly use the same key in multiple instances.

11.6. Protection of Authorization Server

The AS performs critical functions in GNAP, including authenticating client software, managing interactions with end users to gather consent and provide notice, and issuing access tokens for client instances to present to RSs. As such, protecting the AS is central to any GNAP deployment.

If an attacker is able to gain control over an AS, they would be able to create fraudulent tokens and manipulate registration information to allow for malicious clients. These tokens and clients would be trusted by other components in the ecosystem under the protection of the AS.

If the AS uses signed access tokens, an attacker in control of the AS's signing keys would be able to manufacture fraudulent tokens for use at RSs under the protection of the AS.

If an attacker is able to impersonate an AS, they would be able to trick legitimate client instances into making signed requests for information that could potentially be proxied to a real AS. To combat this, all communications to the AS need to be made over TLS or its equivalent, and the software making the connection has to validate the certificate chain of the host it is connecting to.

Consequently, protecting, monitoring, and auditing the AS is paramount to preserving the security of a GNAP-protected ecosystem. The AS presents attackers with a valuable target for attack. Fortunately, the core focus and function of the AS is to provide security for the ecosystem, unlike the RS whose focus is to provide an API or the client software whose focus is to access the API.

11.7. Symmetric and Asymmetric Client Instance Keys

Many of the cryptographic methods used by GNAP for key proofing can support both asymmetric and symmetric cryptography, and they can be extended to use a wide variety of mechanisms. Implementors will find the available guidelines on cryptographic key management provided in [RFC4107] useful. While symmetric cryptographic systems have some benefits in speed and simplicity, they have a distinct drawback -- both parties need access to the same key in order to do both signing and verification of the message. When more than two parties share the same symmetric key, data origin authentication is not provided. Any party that knows the symmetric key can compute a valid MAC; therefore, the contents could originate from any one of the parties.

Use of symmetric cryptography means that when the client instance calls the AS to request a token, the AS needs to know the exact value of the client instance's key (or be able to derive it) in order to validate the key proof signature. With asymmetric keys, the client needs to only send its public key to the AS to allow for verification that the client holds the associated private key, regardless of whether or not that key was pre-registered with the AS.

Symmetric keys also have the expected advantage of providing better protection against quantum threats in the future. Also, these types of keys (and their secure derivations) are widely supported among many cloud-based key management systems.

When used to bind to an access token, a key value must be known by the RS in order to validate the proof signature on the request. Common methods for communicating these proofing keys include putting information in a structured access token and allowing the RS to look up the associated key material against the value of the access token. With symmetric cryptography, both of these methods would expose the signing key to the RS and, in the case of a structured access token, potentially to any party that can see the access token itself unless the token's payload has been encrypted. Any of these parties would then be able to make calls using the access token by creating a valid signature using the shared key. With asymmetric cryptography, the RS needs to only know the public key associated with the token in order to validate the request; therefore, the RS cannot create any new signed calls.

While both signing approaches are allowed, GNAP treats these two classes of keys somewhat differently. Only the public portion of asymmetric keys are allowed to be sent by value in requests to the AS when establishing a connection. Since sending a symmetric key (or the private

portion of an asymmetric key) would expose the signing material to any parties on the request path, including any attackers, sending these kinds of keys by value is prohibited. Symmetric keys can still be used by client instances, but only if the client instance can send a reference to the key and not its value. This approach allows the AS to use pre-registered symmetric keys as well as key derivation schemes to take advantage of symmetric cryptography without requiring key distribution at runtime, which would expose the keys in transit.

Both the AS and client software can use systems such as hardware security modules to strengthen their key security storage and generation for both asymmetric and symmetric keys (see also Section 7.1.2).

11.8. Generation of Access Tokens

The contents of access tokens need to be such that only the generating AS would be able to create them, and the contents cannot be manipulated by an attacker to gain different or additional access rights.

One method for accomplishing this is to use a cryptographically random value for the access token, generated by the AS using a secure randomization function with sufficiently high entropy. The odds of an attacker guessing the output of the randomization function to collide with a valid access token are exceedingly small, and even then, the attacker would not have any control over what the access token would represent since that information would be held close by the AS.

Another method for accomplishing this is to use a structured token that is cryptographically signed. In this case, the payload of the access token declares to the RS what the token is good for, but the signature applied by the AS during token generation covers this payload. Only the AS can create such a signature; therefore, only the AS can create such a signed token. The odds of an attacker being able to guess a signature value with a useful payload are exceedingly small. This technique only works if all targeted RSs check the signature of the access token. Any RS that does not validate the signature of all presented tokens would be susceptible to injection of a modified or falsified token. Furthermore, an AS has to carefully protect the keys used to sign access tokens, since anyone with access to these signing keys would be able to create seemingly valid access tokens using them.

11.9. Bearer Access Tokens

Bearer access tokens can be used by any party that has access to the token itself, without any additional information. As a natural consequence, any RS that a bearer token is presented to has the technical capability of presenting that bearer token to another RS, as long as the token is valid. It also means that any party that is able to capture the token value in storage or in transit is able to use the access token. While bearer tokens are inherently simpler, this simplicity has been misapplied and abused in making needlessly insecure systems. The downsides of bearer tokens have become more pertinent lately as stronger authentication systems have caused some attacks to shift to target tokens and APIs.

In GNAP, key-bound access tokens are the default due to their higher security properties. While bearer tokens can be used in GNAP, their use should be limited to cases where the simplicity benefits outweigh the significant security downsides. One common deployment pattern is to use a gateway that takes in key-bound tokens on the outside and verifies the signatures on the incoming requests but translates the requests to a bearer token for use by trusted internal systems. The bearer tokens are never issued or available outside of the internal systems, greatly limiting the exposure of the less-secure tokens but allowing the internal deployment to benefit from the advantages of bearer tokens.

11.10. Key-Bound Access Tokens

Key-bound access tokens, as the name suggests, are bound to a specific key and must be presented along with proof of that key during use. The key itself is not presented at the same time as the token, so even if a token value is captured, it cannot be used to make a new request. This is particularly true for an RS, which will see the token value but will not see the keys used to make the request (assuming asymmetric cryptography is in use, see Section 11.7).

Key-bound access tokens provide this additional layer of protection only when the RS checks the signature of the message presented with the token. Acceptance of an invalid presentation signature, or failure to check the signature entirely, would allow an attacker to make calls with a captured access token without having access to the related signing key material.

In addition to validating the signature of the presentation message itself, the RS also needs to ensure that the signing key used is appropriate for the presented token. If an RS does not ensure that the right keys were used to sign a message with a specific token, an attacker would be able to capture an access token and sign the request with their own keys, thereby negating the benefits of using key-bound access tokens.

The RS also needs to ensure that sufficient portions of the message are covered by the signature. Any items outside the signature could still affect the API's processing decisions, but these items would not be strongly bound to the token presentation. As such, an attacker could capture a valid request and then manipulate portions of the request outside of the signature envelope in order to cause unwanted actions at the protected API.

Some key-bound tokens are susceptible to replay attacks, depending on the details of the signing method used. Therefore, key proofing mechanisms used with access tokens need to use replay-protection mechanisms covered under the signature such as a per-message nonce, a reasonably short time validity window, or other uniqueness constraints. The details of using these will vary depending on the key proofing mechanism in use. For example, HTTP message signatures have both a `created` and `nonce` signature parameter as well as the ability to cover significant portions of the HTTP message. All of these can be used to limit the attack surface.

11.11. Exposure of End-User Credentials to Client Instance

As a delegation protocol, one of the main goals of GNAP is to prevent the client software from being exposed to any credentials or information about the end user or RO as a requirement of the delegation process. By using the variety of interaction mechanisms, the RO can interact with the AS without ever authenticating to the client software and without the client software having to impersonate the RO through replay of their credentials.

Consequently, no interaction methods defined in this specification require the end user to enter their credentials, but it is technologically possible for an extension to be defined to carry such values. Such an extension would be dangerous as it would allow rogue client software to directly collect, store, and replay the end user's credentials outside of any legitimate use within a GNAP request.

The concerns of such an extension could be mitigated through use of a challenge and response unlocked by the end user's credentials. For example, the AS presents a challenge as part of an interaction start method, and the client instance signs that challenge using a key derived from a password presented by the end user. It would be possible for the client software to collect this password in a secure software enclave without exposing the password to the rest of the client software or putting it across the wire to the AS. The AS can validate this challenge response against a known password for the identified end user. While an approach such as this does not remove all of the concerns surrounding such a password-based scheme, it is at least possible to implement in a more secure fashion than simply collecting and replaying the password. Even so, such schemes should only ever be used by trusted clients due to the ease of abusing them.

11.12. Mixing Up Authorization Servers

If a client instance is able to work with multiple ASes simultaneously, it is possible for an attacker to add a compromised AS to the client instance's configuration and cause the client software to start a request at the compromised AS. This AS could then proxy the client's request to a valid AS in order to attempt to get the RO to approve access for the legitimate client instance.

A client instance needs to always be aware of which AS it is talking to throughout a grant process and ensure that any callback for one AS does not get conflated with the callback to different AS. The interaction finish hash calculation in Section 4.2.3 allows a client instance to protect against this kind of substitution, but only if the client instance validates the hash. If the client instance does not use an interaction finish method or does not check the interaction finish hash value, the compromised AS can be granted a valid access token on behalf of the RO. See Sections 4.5.5 and 5.5 of [AXELAND2021] for details of one such attack, which has been addressed in this document by including the grant endpoint in the interaction hash calculation. Note that the client instance still needs to validate the hash for the attack to be prevented.

11.13. Processing of Client-Presented User Information

GNAP allows the client instance to present assertions and identifiers of the current user to the AS as part of the initial request. This information should only ever be taken by the AS as a hint, since the AS has no way to tell if the represented person is present at the client software without using an interaction mechanism. This information does not guarantee the given user is there, but it does constitute a statement by the client software that the AS can take into account.

For example, if a specific user is claimed to be present prior to interaction, but a different user is shown to be present during interaction, the AS can either determine this to be an error or signal to the client instance through returned subject information that the current user has changed from what the client instance thought. This user information can also be used by the AS to streamline the interaction process when the user is present. For example, instead of having the user type in their account identifier during interaction at a redirected URI, the AS can immediately challenge the user for their account credentials. Alternatively, if an existing session is detected, the AS can determine that it matches the identifier provided by the client and subsequently skip an explicit authentication event by the RO.

In cases where the AS trusts the client software more completely, due to policy or previous approval of a given client instance, the AS can take this user information as a statement that the user is present and could issue access tokens and release subject information without interaction. The AS should only take such action in very limited circumstances, as a client instance could assert whatever it likes for the user's identifiers in its request. The AS can limit the possibility of this by issuing randomized opaque identifiers to client instances to represent different end-user accounts after an initial login.

When a client instance presents an assertion to the AS, the AS needs to evaluate that assertion. Since the AS is unlikely to be the intended audience of an assertion held by the client software, the AS will need to evaluate the assertion in a different context. Even in this case, the AS can still evaluate that the assertion was generated by a trusted party, was appropriately signed, and is within any time validity windows stated by the assertion. If the client instance's audience identifier is known to the AS and can be associated with the client instance's presented key, the AS can also evaluate that the appropriate client instance is presenting the claimed assertion. All of this will prevent an attacker from presenting a manufactured assertion or one captured from an untrusted system. However, without validating the audience of the assertion, a captured assertion could be presented by the client instance to impersonate a given end user. In such cases, the assertion offers little more protection than a simple identifier would.

A special case exists where the AS is the generator of the assertion being presented by the client instance. In these cases, the AS can validate that it did issue the assertion and it is associated with the client instance presenting the assertion.

11.14. Client Instance Pre-registration

Each client instance is identified by its own unique key, and for some kinds of client software such as a web server or backend system, this identification can be facilitated by registering a single key for a piece of client software ahead of time. This registration can be associated with a set of display attributes to be used during the authorization process to identify the client software to the user. In these cases, it can be assumed that only one instance of client software will exist, likely to serve many different users.

A client's registration record needs to include its identifying key. Furthermore, it is the case that any clients using symmetric cryptography for key proofing mechanisms need to have their keys pre-registered. The registration should also include any information that would aid in the authorization process, such as a display name and logo. The registration record can also limit a given client to ask for certain kinds of information or use specific interaction mechanisms at runtime.

It also is sensible to pre-register client instances when the software is acting autonomously, without the need for a runtime approval by a RO or any interaction with an end user. In these cases, an AS needs to rely on the trust decisions that have been determined prior to runtime to determine what rights and tokens to grant to a given client instance.

However, it does not make sense to pre-register many types of clients. Single-page applications (SPAs) and mobile/desktop applications in particular present problems with pre-registration. For SPAs, the instances are ephemeral in nature, and long-term registration of a single instance leads to significant storage and management overhead at the AS. For mobile applications, each installation of the client software is a separate instance, and sharing a key among all instances would be detrimental to security as the compromise of any single installation would compromise all copies for all users.

An AS can treat these classes of client software differently from each other, perhaps by allowing access to certain high-value APIs only to pre-registered known clients or by requiring an active end-user delegation of authority to any client software not pre-registered.

An AS can also provide warnings and caveats to ROs during the authorization process, allowing the user to make an informed decision regarding the software they are authorizing. For example, if the AS has vetted the client software and this specific instance, it can present a different authorization screen compared to a client instance that is presenting all of its information at runtime.

Finally, an AS can use platform attestations and other signals from the client instance at runtime to determine whether or not the software making the request is legitimate. The details of such attestations are outside the scope of this specification, but the `client` portion of a grant request provides a natural extension point to such information through the "GNAP Client Instance Fields" registry (Section 10.7).

11.15. Client Instance Impersonation

If client instances are allowed to set their own user-facing display information, such as a display name and website URL, a malicious client instance could impersonate legitimate client software for the purposes of tricking users into authorizing the malicious client.

Requiring clients to pre-register does not fully mitigate this problem since many pre-registration systems have self-service portals for management of client registration, allowing authenticated developers to enter self-asserted information into the management portal.

An AS can mitigate this by actively filtering all self-asserted values presented by client software, both dynamically as part of GNAP and through a registration portal, to limit the kinds of impersonation that could be done.

An AS can also warn the RO about the provenance of the information it is displaying, allowing the RO to make a more informed delegation decision. For example, an AS can visually differentiate between a client instance that can be traced back to a specific developer's registration and an instance that has self-asserted its own display information.

11.16. Client-Hosted Logo URI

The `logo_uri` client display field defined in Section 2.3.2 allows the client instance to specify a URI from which an image can be fetched for display during authorization decisions. When the URI points to an externally hosted resource (as opposed to a data: URI), the `logo_uri` field presents challenges in addition to the considerations in Section 11.15.

When a `logo_uri` is externally hosted, the client software (or the host of the asset) can change the contents of the logo without informing the AS. Since the logo is considered an aspect of the client software's identity, this flexibility allows for a more dynamically managed client instance that makes use of the distributed systems.

However, this same flexibility allows the host of the asset to change the hosted file in a malicious way, such as replacing the image content with malicious software for download or imitating a different piece of client software. Additionally, the act of fetching the URI could accidentally leak information to the image host in the HTTP Referer header field, if one is sent. Even though GNAP intentionally does not include security parameters in front-channel URIs wherever possible, the AS still should take steps to ensure that this information does not leak accidentally, such as setting a referrer policy on image links or displaying images only from pages served from a URI with no sensitive security or identity information.

To avoid these issues, the AS can insist on the use of data: URIs, though that might not be practical for all types of client software. Alternatively, the AS could pre-fetch the content of the URI and present its own copy to the RO instead. This practice opens the AS to potential SSRF attacks, as discussed in Section 11.34.

11.17. Interception of Information in the Browser

Most information passed through the web browser is susceptible to interception and possible manipulation by elements within the browser such as scripts loaded within pages. Information in the URI is exposed through browser and server logs, and it can also leak to other parties through HTTP `Referer` headers.

GNAP's design limits the information passed directly through the browser, allowing for opaque URIs in most circumstances. For the redirect-based interaction finish mechanism, named query parameters are used to carry unguessable opaque values. For these, GNAP requires creation and validation of a cryptographic hash to protect the query parameters added to the URI and associate them with an ongoing grant process and values not passed in the URI. The client instance has to properly validate this hash to prevent an attacker from injecting an interaction reference intended for a different AS or client instance.

Several interaction start mechanisms use URIs created by the AS and passed to the client instance. While these URIs are opaque to the client instance, it's possible for the AS to include parameters, paths, and other pieces of information that could leak security data or be manipulated by a party in the middle of the transaction. An AS implementation can avoid this problem by creating URIs using unguessable values that are randomized for each new grant request.

11.18. Callback URI Manipulation

The callback URI used in interaction finish mechanisms is defined by the client instance. This URI is opaque to the AS but can contain information relevant to the client instance's operations. In particular, the client instance can include state information to allow the callback request to be associated with an ongoing grant request.

Since this URI is exposed to the end user's browser, it is susceptible to both logging and manipulation in transit before the request is made to the client software. As such, a client instance should never put security-critical or private information into the callback URI in a cleartext form. For example, if the client software includes a post-redirect target URI in its callback URI to the AS, this target URI could be manipulated by an attacker, creating an open redirector at the client. Instead, a client instance can use an unguessable identifier in the URI that can then be used by the client software to look up the details of the pending request. Since this approach requires some form of statefulness by the client software during the redirection process, clients that are not capable of holding state through a redirect should not use redirect-based interaction mechanisms.

11.19. Redirection Status Codes

As described in [OAUTH-SEC-TOPICS], a server should never use HTTP status code 307 (Temporary Redirect) to redirect a request that potentially contains user credentials. If an HTTP redirect is used for such a request, HTTP status code 303 (See Other) should be used instead.

Status code 307 (Temporary Redirect), as defined in the HTTP standard [HTTP], requires the user agent to preserve the method and content of a request, thus submitting the content of the POST request to the redirect target. In the HTTP standard [HTTP], only status code 303 (See Other) unambiguously enforces rewriting the HTTP POST request to an HTTP GET request, which eliminates the POST content from the redirected request. For all other status codes, including status code 302 (Found), user agents are allowed to keep a redirected POST request as a POST and thus can resubmit the content.

The use of status code 307 (Temporary Redirect) results in a vulnerability when using the `redirect` interaction finish method (Section 3.3.5). With this method, the AS potentially prompts the RO to enter their credentials in a form that is then submitted back to the AS (using an HTTP POST request). The AS checks the credentials and, if successful, may immediately redirect the RO to the client instance's redirect URI. Due to the use of status code 307 (Temporary Redirect), the RO's user agent now transmits the RO's credentials to the client instance. A malicious client instance can then use the obtained credentials to impersonate the RO at the AS.

Redirection away from the initial URI in an interaction session could also leak information found in that initial URI through the HTTP Referer header field, which would be sent by the user agent to the redirect target. To avoid such leakage, a server can first redirect to an internal interstitial page without any identifying or sensitive information on the URI before processing the request. When the user agent is ultimately redirected from this page, no part of the original interaction URI will be found in the Referer header.

11.20. Interception of Responses from the AS

Responses from the AS contain information vital to both the security and privacy operations of GNAP. This information includes nonces used in cryptographic calculations, Subject Identifiers, assertions, public keys, and information about what client software is requesting and was granted.

In addition, if bearer tokens are used or keys are issued alongside a bound access token, the response from the AS contains all information necessary for use of the contained access token. Any party that is capable of viewing such a response, such as an intermediary proxy, would be able to exfiltrate and use this token. If the access token is instead bound to the client instance's presented key, intermediaries no longer have sufficient information to use the token. They can still, however, gain information about the end user as well as the actions of the client software.

11.21. Key Distribution

GNAP does not define ways for the client instances keys to be provided to the client instances, particularly in light of how those keys are made known to the AS. These keys could be generated dynamically on the client software or pre-registered at the AS in a static developer portal. The keys for client instances could also be distributed as part of the deployment process of instances of the client software. For example, an application installation framework could generate a key pair for each copy of client software and then both install it into the client software upon installation and register that instance with the AS.

Alternatively, it's possible for the AS to generate keys to be used with access tokens that are separate from the keys used by the client instance to request tokens. In this method, the AS would generate the asymmetric key pair or symmetric key and return the public key or key reference to the client instance alongside the access token itself. The means for the AS to return generated key values to the client instance are out of scope, since GNAP does not allow the transmission of private or shared key information within the protocol itself.

Additionally, if the token is bound to a key other than the client instance's presented key, this opens a possible attack surface for an attacker's AS to request an access token and then substitute their own key material in the response to the client instance. The attacker's AS would need to be able to use the same key as the client instance, but this setup would allow an attacker's AS to make use of a compromised key within a system. This attack can be prevented by only binding access tokens to the client instance's presented keys and by having client instances have a strong association between which keys they expect to use and the AS they expect to use them on. This attack is also only able to be propagated on client instances that talk to more than one AS at runtime, which can be limited by the client software.

11.22. Key Rotation Policy

When keys are rotated, there could be a delay in the propagation of that rotation to various components in the AS's ecosystem. The AS can define its own policy regarding the timeout of the previously bound key, either making it immediately obsolete or allowing for a limited grace period during which both the previously bound key and the current key can be used for signing requests. Such a grace period can be useful when there are multiple running copies of the client that are coordinated with each other. For example, the client software could be deployed as a cloud service with multiple orchestrated nodes. Each of these copies is deployed using the same key; therefore, all the nodes represent the same client instance to the AS. In such cases, it can be difficult, or even impossible, to update the keys on all these copies in the same instant.

The need to accommodate such known delays in the system needs to be balanced with the risk of allowing an old key to still be used. Narrowly restricting the exposure opportunities for exploit at the AS in terms of time, place, and method makes exploit significantly more difficult, especially if the exception happens only once. For example, the AS can reject requests from the previously bound key (or any previous one before it) to cause rotation to a new key or at least ensure that the rotation happens in an idempotent way to the same new key.

See also the related considerations for token values in Section 11.33.

11.23. Interaction Finish Modes and Polling

During the interaction process, the client instance usually hands control of the user experience over to another component, be it the system browser, another application, or some action the RO is instructed to take on another device. By using an interaction finish method, the client instance can be securely notified by the AS when the interaction is completed and the next phase of the protocol should occur. This process includes information that the client instance can use to validate the finish call from the AS and prevent some injection, session hijacking, and phishing attacks.

Some types of client deployment are unable to receive an interaction finish message. Without an interaction finish method to notify it, the client instance will need to poll the grant continuation API while waiting for the RO to approve or deny the request. An attacker could take advantage of this situation by capturing the interaction start parameters and phishing a legitimate user into authorizing the attacker's waiting client instance, which would in turn have no way of associating the completed interaction from the targeted user with the start of the request from the attacker.

However, it is important to note that this pattern is practically indistinguishable from some legitimate use cases. For example, a smart device emits a code for the RO to enter on a separate device. The smart device has to poll because the expected behavior is that the interaction will take place on the separate device, without a way to return information to the original device's context.

As such, developers need to weigh the risks of forgoing an interaction finish method against the deployment capabilities of the client software and its environment. Due to the increased security, an interaction finish method should be employed whenever possible.

11.24. Session Management for Interaction Finish Methods

When using an interaction finish method such as `redirect` or `push`, the client instance receives an unsolicited inbound request from an unknown party over HTTPS. The client instance needs to be able to successfully associate this incoming request with a specific pending grant request being managed by the client instance. If the client instance is not careful and precise about this, an attacker could associate their own session at the client instance with a stolen interaction response. The means of preventing this vary by the type of client software and interaction methods in use. Some common patterns are enumerated here.

If the end user interacts with the client instance through a web browser and the `redirect` interaction finish method is used, the client instance can ensure that the incoming HTTP request from the finish method is presented in the same browser session that the grant request was started in. This technique is particularly useful when the `redirect` interaction start mode is used as well, since in many cases, the end user will follow the redirection with the same browser that they are using to interact with the client instance. The client instance can then store the relevant pending grant information in the session, either in the browser storage directly (such as with a single-page application) or in an associated session store on a backend server. In both cases, when the incoming request reaches the client instance, the session information can be used to ensure that the same party that started the request is present as the request finishes.

Ensuring that the same party that started a request is present when that request finishes can prevent phishing attacks, where an attacker starts a request at an honest client instance and tricks an honest RO into authorizing it. For example, if an honest end user (that also acts as the RO) wants to start a request through a client instance controlled by the attacker, the attacker can start a request at an honest client instance and then redirect the honest end user to the interaction URI from the attackers session with the honest client instance. If the honest end user then fails to realize that they are not authorizing the attacker-controlled client instance (with which it started its request) but instead the honest client instance when interacting with the AS,

the attacker's session with the honest client instance would be authorized. This would give the attacker access to the honest end user's resources that the honest client instance is authorized to access. However, if after the interaction, the AS redirects the honest end user back to the client instance whose grant request the end user just authorized, the honest end user is redirected to the honest client instance. The honest client instance can then detect that the end user is not the party that started the request, since the request at the honest client instance was started by the attacker. This detection can prevent the attack. This is related to the discussion in Section 11.15, because again the attack can be prevented by the AS informing the user as much as possible about the client instance that is to be authorized.

If the end user does not interact with the client instance through a web browser or the interaction start method does not use the same browser or device that the end user is interacting through (such as the launch of a second device through a scannable code or presentation of a user code), the client instance will not be able to strongly associate an incoming HTTP request with an established session with the end user. This is also true when the `push` interaction finish method is used, since the HTTP request comes directly from the interaction component of the AS. In these circumstances, the client instance can at least ensure that the incoming HTTP request can be uniquely associated with an ongoing grant request by making the interaction finish callback URI unique for the grant when making the interaction request (Section 2.5.2). Mobile applications and other client instances that generally serve only a single end user at a time can use this unique incoming URL to differentiate between a legitimate incoming request and an attacker's stolen request.

11.25. Calculating Interaction Hash

While the use of GNAP's signing mechanisms and token-protected grant API provides significant security protections to the protocol, the interaction reference mechanism is susceptible to monitoring, capture, and injection by an attacker. To combat this, GNAP requires the calculation and verification of an interaction hash. A client instance might be tempted to skip this step, but doing so leaves the client instance open to injection and manipulation by an attacker that could lead to additional issues.

The calculation of the interaction hash value provides defense in depth, allowing a client instance to protect itself from spurious injection of interaction references when using an interaction finish method. The AS is protected during this attack through the continuation access token being bound to the expected interaction reference, but without hash calculation, the attacker could cause the client to make an HTTP request on command, which could itself be manipulated -- for example, by including a malicious value in the interaction reference designed to attack the AS. With both of these in place, an attacker attempting to substitute the interaction reference is stopped in several places.

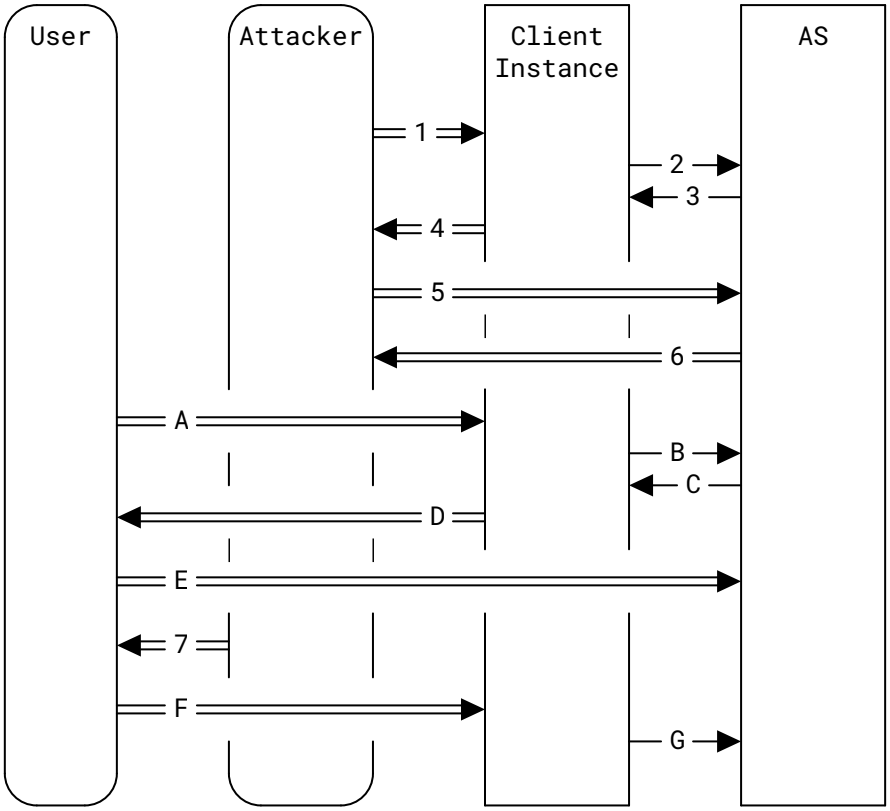

Figure 11: Interaction Hash Attack

Prerequisites: The client instance can allow multiple end users to access the same AS. The attacker is attempting to associate their rights with the target user's session.

- (1) The attacker starts a session at the client instance.
- (2) The client instance creates a grant request with nonce CN1.
- (3) The AS responds to the grant request with a need to interact, nonce SN1, and a continuation token, CT1.
- (4) The client instructs the attacker to interact at the AS.
- (5) The attacker interacts at the AS.
- (6) The AS completes the interact finish with interact reference IR1 and interact hash IH1 calculated from (CN1 + SN1 + IR1 + AS). The attacker prevents IR1 from returning to the client instance.
- (A) The target user starts a session at the client instance.
- (B) The client instance creates a grant request with nonce CN2.
- (C) The AS responds to the grant request with a need to interact, nonce SN2, and a continuation token, CT2.
- (D) The client instance instructs the user to interact at the AS.

- (E) The target user interacts at the AS.
- (7) Before the target user can complete their interaction, the attacker delivers their own interact reference IR1 into the user's session. The attacker cannot calculate the appropriate hash because the attacker does not have access to CN2 and SN2.
- (F) The target user triggers the interaction finish in their own session with the attacker's IR1.
- (G) If the client instance is checking the interaction hash, the attack stops here because the hash calculation of (CN2 + SN2 + IR1 + AS) will fail. If the client instance does not check the interaction hash, the client instance will be tricked into submitting the interaction reference to the AS. Here, the AS will reject the interaction request because it is presented against CT2 and not CT1 as expected. However, an attacker who has potentially injected CT1 as the value of CT2 would be able to continue the attack.

Even with additional checks in place, client instances using interaction finish mechanisms are responsible for checking the interaction hash to provide security to the overall system.

11.26. Storage of Information during Interaction and Continuation

When starting an interactive grant request, a client application has a number of protocol elements that it needs to manage, including nonces, references, keys, access tokens, and other elements. During the interaction process, the client instance usually hands control of the user experience over to another component, be it the system browser, another application, or some action the RO is instructed to take on another device. In order for the client instance to make its continuation call, it will need to recall all of these protocol elements at a future time. Usually, this means the client instance will need to store these protocol elements in some retrievable fashion.

If the security protocol elements are stored on the end user's device, such as in browser storage or in local application data stores, capture and exfiltration of this information could allow an attacker to continue a pending transaction instead of the client instance. Client software can make use of secure storage mechanisms, including hardware-based key and data storage, to prevent such exfiltration.

Note that in GNAP, the client instance has to choose its interaction finish URI prior to making the first call to the AS. As such, the interaction finish URI will often have a unique identifier for the ongoing request, allowing the client instance to access the correct portion of its storage. Since this URI is passed to other parties and often used through a browser, this URI should not contain any security-sensitive information that would be valuable to an attacker, such as any token identifier, nonce, or user information. Instead, a cryptographically random value is suggested, and that value should be used to index into a secure session or storage mechanism.

11.27. Denial of Service (DoS) through Grant Continuation

When a client instance starts off an interactive process, it will eventually need to continue the grant request in a subsequent message to the AS. It's possible for a naive client implementation to continuously send continuation requests to the AS while waiting for approval, especially if no interaction finish method is used. Such constant requests could overwhelm the AS's ability to respond to both these and other requests.

To mitigate this for well-behaved client software, the continuation response contains a `wait` parameter that is intended to tell the client instance how long it should wait until making its next request. This value can be used to back off client software that is checking too quickly by returning increasing wait times for a single client instance.

If client software ignores the `wait` value and makes its continuation calls too quickly or if the client software assumes the absence of the `wait` values means it should poll immediately, the AS can choose to return errors to the offending client instance, including possibly canceling the ongoing grant request. With well-meaning client software, these errors can indicate a need to change the client software's programmed behavior.

11.28. Exhaustion of Random Value Space

Several parts of the GNAP process make use of unguessable randomized values, such as nonces, tokens, user codes, and randomized URIs. Since these values are intended to be unique, a sufficiently powerful attacker could make a large number of requests to trigger generation of randomized values in an attempt to exhaust the random number generation space. While this attack is particularly applicable to the AS, client software could likewise be targeted by an attacker triggering new grant requests against an AS.

To mitigate this, software can ensure that its random values are chosen from a significantly large pool so that exhaustion of that pool is prohibitive for an attacker. Additionally, the random values can be time-boxed in such a way that their validity windows are reasonably short. Since many of the random values used within GNAP are used within limited portions of the protocol, it is reasonable for a particular random value to be valid for only a small amount of time. For example, the nonces used for interaction finish hash calculation need only to be valid while the client instance is waiting for the finish callback and can be functionally expired when the interaction has completed. Similarly, artifacts like access tokens and the interaction reference can be limited to have lifetimes tied to their functional utility. Finally, each different category of artifact (nonce, token, reference, identifier, etc.) can be generated from a separate random pool of values instead of a single global value space.

11.29. Front-Channel URIs

Some interaction methods in GNAP make use of URIs accessed through the end user's browser, known collectively as front-channel communication. These URIs are most notably present in the `redirect` interaction `start` method and the `redirect` interaction `finish` mode. Since these URIs are intended to be given to the end user, the end user and their browser will be subjected to anything hosted at that URI including viruses, malware, and phishing scams. This kind of risk is inherent to all redirection-based protocols, including GNAP, when used in this way.

When talking to a new or unknown AS, a client instance might want to check the URI from the interaction `start` against a blocklist and warn the end user before redirecting them. Many client instances will provide an interstitial message prior to redirection in order to prepare the user for control of the user experience being handed to the domain of the AS, and such a method could be

used to warn the user of potential threats (for instance, a rogue AS impersonating a well-known service provider). Client software can also prevent this by managing an allowlist of known and trusted ASes.

Alternatively, an attacker could start a GNAP request with a known and trusted AS but include their own attack site URI as the callback for the redirect `finish` method. The attacker would then send the interaction `start` URI to the victim and get them to click on it. Since the URI is at the known AS, the victim is inclined to do so. The victim will then be prompted to approve the attacker's application, and in most circumstances, the victim will then be redirected to the attacker's site whether or not the user approved the request. The AS could mitigate this partially by using a blocklist and allowlist of interaction `finish` URIs during the client instance's initial request, but this approach can be especially difficult if the URI has any dynamic portion chosen by the client software. The AS can couple these checks with policies associated with the client instance that has been authenticated in the request. If the AS has any doubt about the interaction finish URI, the AS can provide an interstitial warning to the end user before processing the redirect.

Ultimately, all protocols that use redirect-based communication through the user's browser are susceptible to having an attacker try to co-opt one or more of those URIs in order to harm the user. It is the responsibility of the AS and the client software to provide appropriate warnings, education, and mitigation to protect end users.

11.30. Processing Assertions

Identity assertions can be used in GNAP to convey subject information, both from the AS to the client instance in a response (Section 3.4) and from the client instance to the AS in a request (Section 2.2). In both of these circumstances, when an assertion is passed in GNAP, the receiver of the assertion needs to parse and process the assertion. As assertions are complex artifacts with their own syntax and security, special care needs to be taken to prevent the assertion values from being used as an attack vector.

All assertion processing needs to account for the security aspects of the assertion format in use. In particular, the processor needs to parse the assertion from a JSON string object and apply the appropriate cryptographic processes to ensure the integrity of the assertion.

For example, when SAML 2.0 assertions are used, the receiver has to parse an XML document. There are many well-known security vulnerabilities in XML parsers, and the XML standard itself can be attacked through the use of processing instructions and entity expansions to cause problems with the processor. Therefore, any system capable of processing SAML 2.0 assertions also needs to have a secure and correct XML parser. In addition to this, the SAML 2.0 specification uses XML Signatures, which have their own implementation problems that need to be accounted for. Similar requirements exist for OpenID Connect ID Token, which is based on the JWT format and the related JOSE cryptography suite.

11.31. Stolen Token Replay

If a client instance can request tokens at multiple ASes and the client instance uses the same keys to make its requests across those different ASes, then it is possible for an attacker to replay a stolen token issued by an honest AS from a compromised AS, thereby binding the stolen token to the client instance's key in a different context. The attacker can manipulate the client instance into using the stolen token at an RS, particularly at an RS that is expecting a token from the honest AS. Since the honest AS issued the token and the client instance presents the token with its expected bound key, the attack succeeds.

This attack has several preconditions. In this attack, the attacker does not need access to the client instance's key and cannot use the stolen token directly at the RS, but the attacker is able to get the access token value in some fashion. The client instance also needs to be configured to talk to multiple ASes, including the attacker's controlled AS. Finally, the client instance needs to be able to be manipulated by the attacker to call the RS while using a token issued from the stolen AS. The RS does not need to be compromised or made to trust the attacker's AS.

To protect against this attack, the client instance can use a different key for each AS that it talks to. Since the replayed token will be bound to the key used at the honest AS, the uncompromised RS will reject the call since the client instance will be using the key used at the attacker's AS instead with the same token. When the MTLS key proofing method is used, a client instance can use self-signed certificates to use a different key for each AS that it talks to, as discussed in Section 11.4.

Additionally, the client instance can keep a strong association between the RS and a specific AS that it trusts to issue tokens for that RS. This strong binding also helps against some forms of AS mix-up attacks (Section 11.12). Managing this binding is outside the scope of this specification, but it can be managed either as a configuration element for the client instance or dynamically through discovering the AS from the RS (Section 9.1).

The details of this attack, with additional discussion and considerations, are available in Section 3.2 of [HELMSCHMIDT2022].

11.32. Self-Contained Stateless Access Tokens

The contents and format of the access token are at the discretion of the AS and are opaque to the client instance within GNAP. As discussed in [GNAP-RS], the AS and RS can make use of stateless access tokens with an internal structure and format. These access tokens allow an RS to validate the token without having to make any external calls at runtime, allowing for benefits in some deployments, the discussion of which is outside the scope of this specification.

However, the use of such self-contained access tokens has an effect on the ability of the AS to provide certain functionality defined within this specification. Specifically, since the access token is self-contained, it is difficult or impossible for an AS to signal to all RSs within an ecosystem when a specific access token has been revoked. Therefore, an AS in such an ecosystem should probably not offer token revocation functionality to client instances, since the client instance's

calls to such an endpoint are effectively meaningless. However, a client instance calling the token revocation function will also throw out its copy of the token, so such a placebo endpoint might not be completely meaningless. Token rotation is similarly difficult because the AS has to revoke the old access token after a rotation call has been made. If the access tokens are completely self-contained and non-revocable, this means that there will be a period of time during which both the old and new access tokens are valid and usable, which is an increased security risk for the environment.

These problems can be mitigated by keeping the validity time windows of self-contained access tokens reasonably short, limiting the time after a revocation event that a revoked token could be used. Additionally, the AS could proactively signal to RSs under its control identifiers for revoked tokens that have yet to expire. This type of information push would be expected to be relatively small and infrequent, and its implementation is outside the scope of this specification.

11.33. Network Problems and Token and Grant Management

If a client instance makes a call to rotate an access token but the network connection is dropped before the client instance receives the response with the new access token, the system as a whole can end up in an inconsistent state, where the AS has already rotated the old access token and invalidated it, but the client instance only has access to the invalidated access token and not the newly rotated token value. If the client instance retries the rotation request, it would fail because the client is no longer presenting a valid and current access token. A similar situation can occur during grant continuation, where the same client instance calls to continue or update a grant request without successfully receiving the results of the update.

To combat this, both grant management (Section 5) and token management (Section 6) can be designed to be idempotent, where subsequent calls to the same function with the same credentials are meant to produce the same results. For example, multiple calls to rotate the same access token need to result in the same rotated token value, within a reasonable time window.

In practice, an AS can hold onto an old token value for such limited purposes. For example, to support rotating access tokens over unreliable networks, the AS receives the initial request to rotate an access token and creates a new token value and returns it. The AS also marks the old token value as having been used to create the newly rotated token value. If the AS sees the old token value within a small enough time window, such as a few seconds since the first rotation attempt, the AS can return the same rotated access token value. Furthermore, once the system has seen the newly rotated token in use, the original token can be discarded because the client instance has proved that it did receive the token. The result of this is a system that is eventually self-consistent without placing an undue complexity burden on the client instance to manage problematic networks.

11.34. Server-Side Request Forgery (SSRF)

There are several places within GNAP where a URI can be given to a party, causing it to fetch that URI during normal operation of the protocol. If an attacker is able to control the value of one of these URIs within the protocol, the attacker could cause the target system to execute a request on a URI that is within reach of the target system but normally unavailable to the attacker.

Examples include an attacker sending a URL of `http://localhost/admin` to cause the server to access an internal function on itself or `https://192.168.0.14/` to call a service behind a firewall. Even if the attacker does not gain access to the results of the call, the side effects of such requests coming from a trusted host can be problematic to the security and sanctity of such otherwise unexposed endpoints. This can be particularly problematic if such a URI is used to call non-HTTP endpoints, such as remote code execution services local to the AS.

The most vulnerable place in this specification is the push-based post-interaction finish method (Section 4.2.2), as the client instance is less trusted than the AS and can use this method to make the AS call an arbitrary URI. While it is not required by the protocol, the AS can fetch other URIs provided by the client instance, such as the logo image or home page, for verification or privacy-preserving purposes before displaying them to the RO as part of a consent screen. Even if the AS does not fetch these URIs, their use in GNAP's normal operation could cause an attack against the end user's browser as it fetches these same attack URIs. Furthermore, extensions to GNAP that allow or require URI fetch could also be similarly susceptible, such as a system for having the AS fetch a client instance's keys from a presented URI instead of the client instance presenting the key by value. Such extensions are outside the scope of this specification, but any system deploying such an extension would need to be aware of this issue.

To help mitigate this problem, similar approaches that protect parties against malicious redirects (Section 11.29) can be used. For example, all URIs that can result in a direct request being made by a party in the protocol can be filtered through an allowlist or blocklist. For example, an AS that supports the `push`-based interaction `finish` method can compare the callback URI in the interaction request to a known URI for a pre-registered client instance, or it can ensure that the URI is not on a blocklist of sensitive URLs such as internal network addresses. However, note that because these types of calls happen outside of the view of human interaction, it is not usually feasible to provide notification and warning to someone before the request needs to be executed, as is the case with redirection URLs. As such, SSRF is somewhat more difficult to manage at runtime, and systems should generally refuse to fetch a URI if unsure.

11.35. Multiple Key Formats

All keys presented by value are only allowed to be in a single format. While it would seem beneficial to allow keys to be sent in multiple formats in case the receiver doesn't understand one or more of the formats used, there are security issues with such a feature. If multiple keys formats are allowed, receivers of these key definitions would need to be able to make sure that it's the same key represented in each field and not simply use one of the key formats without checking for equivalence. If equivalence is not carefully checked, it is possible for an attacker to insert their own key into one of the formats without needing to have control over the other formats. This could potentially lead to a situation where one key is used by part of the system (such as identifying the client instance) and a different key in a different format in the same message is used for other things (such as calculating signature validity). However, in such cases, it is impossible for the receiver to ensure that all formats contain the same key information since it is assumed that the receiver cannot understand all of the formats.

To combat this, all keys presented by value have to be in exactly one supported format known by the receiver as discussed in Section 7.1. In most cases, a client instance is going to be configured with its keys in a single format, and it will simply present that format as is to the AS in its request. A client instance capable of multiple formats can use AS discovery (Section 9) to determine which formats are supported, if desired. An AS should be generous in supporting many different key formats to allow different types of client software and client instance deployments. An AS implementation should try to support multiple formats to allow a variety of client software to connect.

11.36. Asynchronous Interactions

GNAP allows the RO to be contacted by the AS asynchronously, outside the regular flow of the protocol. This allows for some advanced use cases, such as cross-user authentication or information release, but such advanced use cases have some distinct issues that implementors need to be fully aware of before using these features.

First, in many applications, the return of subject information to the client instance could indicate to the client instance that the end user is the party represented by that information, functionally allowing the end user to authenticate to the client application. While the details of a fully functional authentication protocol are outside the scope of GNAP, it is a common exercise for a client instance to request information about the end user. This is facilitated by several interaction methods (Section 4.1) defined in GNAP that allow the end user to begin interaction directly with the AS. However, when the subject of the information is intentionally not the end user, the client application will need some way to differentiate between requests for authentication of the end user and requests for information about a different user. Confusing these states could lead to an attacker having their account associated with a privileged user. Client instances can mitigate this by having distinct code paths for primary end-user authentication and for requesting subject information about secondary users, such as in a call center. In such use cases, the client software used by the RO (the caller) and the end user (the agent) are generally distinct, allowing the AS to differentiate between the agent's corporate device making the request and the caller's personal device approving the request.

Second, ROs that interact asynchronously do not usually have the same context as an end user in an application attempting to perform the task needing authorization. As such, the asynchronous requests for authorization coming to the RO from the AS might have very little to do with what the RO is doing at the time. This situation can consequently lead to authorization fatigue on the part of the RO, where any incoming authorization request is quickly approved and dispatched without the RO making a proper verification of the request. An attacker can exploit this fatigue and get the RO to authorize the attacker's system for access. To mitigate this, AS systems deploying asynchronous authorization should only prompt the RO when the RO is expecting such a request, and significant user experience engineering efforts need to be employed to ensure that the RO can clearly make the appropriate security decision. Furthermore, audit capability and the ability to undo access decisions that may be ongoing are particularly important in the asynchronous case.

11.37. Compromised RS

An attacker may aim to gain access to confidential or sensitive resources. The measures for hardening and monitoring RS systems (beyond protection with access tokens) are out of the scope of this document, but the use of GNAP to protect a system does not absolve the RS of following best practices. GNAP generally considers that a breach can occur and therefore advises to prefer key-bound tokens whenever possible, which at least limits the impact of access token leakage by a compromised or malicious RS.

11.38. AS-Provided Token Keys

While the most common token-issuance pattern is to bind the access token to the client instance's presented key, it is possible for the AS to provide a binding key along with an access token, as shown by the `key` field of the token response in Section 3.2.1. This practice allows for an AS to generate and manage the keys associated with tokens independently of the keys known to client instances.

If the key material is returned by value from the AS, then the client instance will simply use this key value when presenting the token. This can be exploited by an attacker to issue a compromised token to an unsuspecting client, assuming that the client instance trusts the attacker's AS to issue tokens for the target RS. In this attack, the attacker first gets a token bound to a key under the attacker's control. This token is likely bound to an authorization or account controlled by the attacker. The attacker then reissues that same token to the client instance, this time acting as an AS. The attacker can return their own key to the client instance, tricking the client instance into using the attacker's token. Such an attack is also possible when the key is returned by reference, if the attacker is able to provide a reference meaningful to the client instance that references a key under the attacker's control. This substitution attack is similar to some of the main issues found with bearer tokens as discussed in Section 11.9.

Returning a key with an access token should be limited to circumstances where both the client and AS can be verified to be honest and when the trade-off of not using a client instance's own keys is worth the additional risk.

12. Privacy Considerations

The privacy considerations in this section are modeled after the list of privacy threats in "Privacy Considerations for Internet Protocols" [RFC6973] and either explain how these threats are mitigated or advise how the threats relate to GNAP.

12.1. Surveillance

Surveillance is the observation or monitoring of an individual's communications or activities. Surveillance can be conducted by observers or eavesdroppers at any point along the communications path.

GNAP assumes the TLS protection used throughout the spec is intact. Without the protection of TLS, there are many points throughout the use of GNAP that could lead to possible surveillance. Even with the proper use of TLS, surveillance could occur by several parties outside of the TLS-protected channels, as discussed in the subsections below.

12.1.1. Surveillance by the Client

The purpose of GNAP is to authorize clients to be able to access information on behalf of a user. So while it is expected that the client may be aware of the user's identity as well as data being fetched for that user, in some cases, the extent of the client may be beyond what the user is aware of. For example, a client may be implemented as multiple distinct pieces of software, such as a logging service or a mobile application that reports usage data to an external backend service. Each of these pieces could gain information about the user without the user being aware of this action.

When the client software uses a hosted asset for its components, such as its logo image, the fetch of these assets can reveal user actions to the host. If the AS presents the logo URI to the RO in a browser page, the browser will fetch the logo URL from the authorization screen. This fetch will tell the host of the logo image that someone is accessing an instance of the client software and requesting access for it. This is particularly problematic when the host of the asset is not the client software itself, such as when a content delivery network is used.

12.1.2. Surveillance by the Authorization Server

The role of the AS is to manage the authorization of client instances to protect access to the user's data. In this role, the AS is by definition aware of each authorization of a client instance by a user. When the AS shares user information with the client instance, it needs to make sure that it has the permission from that user to do so.

Additionally, as part of the authorization grant process, the AS may be aware of which RSs the client intends to use an access token at. However, it is possible to design a system using GNAP in which this knowledge is not made available to the AS, such as by avoiding the use of the `locations` object in the authorization request.

If the AS's implementation of access tokens is such that it requires an RS callback to the AS to validate them, then the AS will be aware of which RSs are actively in use and by which users and clients. To avoid this possibility, the AS would need to structure access tokens in such a way that they can be validated by the RS without notifying the AS that the token is being validated.

12.2. Stored Data

Several parties in the GNAP process are expected to persist data at least temporarily, if not semi-permanently, for the normal functioning of the system. If compromised, this could lead to exposure of sensitive information. This section documents the potentially sensitive information each party in GNAP is expected to store for normal operation. Naturally, it is possible for any party to store information related to protocol mechanics (such as audit logs, etc.) for longer than is technically necessary.

The AS is expected to store Subject Identifiers for users indefinitely, in order to be able to include them in the responses to clients. The AS is also expected to store client key identifiers associated with display information about the client, such as its name and logo.

The client is expected to store its client instance key indefinitely, in order to authenticate to the AS for the normal functioning of the GNAP flows. Additionally, the client will be temporarily storing artifacts issued by the AS during a flow, and these artifacts ought to be discarded by the client when the transaction is complete.

The RS is not required to store any state for its normal operation, as far as its part in implementing GNAP. Depending on the implementation of access tokens, the RS may need to cache public keys from the AS in order to validate access tokens.

12.3. Intrusion

Intrusion refers to the ability of various parties to send unsolicited messages or cause denial of service for unrelated parties.

If the RO is different from the end user, there is an opportunity for the end user to cause unsolicited messages to be sent to the RO if the system prompts the RO for consent when an end user attempts to access their data.

The format and contents of Subject Identifiers are intentionally not defined by GNAP. If the AS uses values for Subject Identifiers that are also identifiers for communication channels (e.g., an email address or phone number), this opens up the possibility for a client to learn this information when it was not otherwise authorized to access this kind of data about the user.

12.4. Correlation

The threat of correlation is the combination of various pieces of information related to an individual in a way that defies their expectations of what others know about them.

12.4.1. Correlation by Clients

The biggest risk of correlation in GNAP is when an AS returns stable, consistent user identifiers to multiple different applications. In this case, applications created by different parties would be able to correlate these user identifiers out of band in order to know which users they have in common.

The most common example of this in practice is tracking for advertising purposes, such that a client shares their list of user IDs with an ad platform that is then able to retarget ads to applications created by other parties. In contrast, a positive example of correlation is a corporate acquisition where two previously unrelated clients now do need to be able to identify the same user between the two clients, such as when software systems are intentionally connected by the end user.

Another means of correlation comes from the use of RS-first discovery (Section 9.1). A client instance that knows nothing other than an RS's URL could make an unauthenticated call to the RS and learn which AS protects the resources there. If the client instance knows something about

the AS, such as it being a single-user AS or belonging to a specific organization, the client instance could, through association, learn things about the resource without ever gaining access to the resource itself.

12.4.2. Correlation by Resource Servers

Unrelated RSs also have an opportunity to correlate users if the AS includes stable user identifiers in access tokens or in access token introspection responses.

In some cases, an RS may not actually need to be able to identify users (such as an RS providing access to a company cafeteria menu, which only needs to validate whether the user is a current employee), so ASes should be thoughtful of when user identifiers are actually necessary to communicate to RSs for the functioning of the system.

However, note that the lack of inclusion of a user identifier in an access token may be a risk if there is a concern that two users may voluntarily share access tokens between them in order to access protected resources. For example, if a website wants to limit access to only people over 18, and such does not need to know any user identifiers, an access token may be issued by an AS contains only the claim "over 18". If the user is aware that this access token doesn't reference them individually, they may be willing to share the access token with a user who is under 18 in order to let them get access to the website. (Note that the binding of an access token to a non-extractable client instance key also prevents the access token from being voluntarily shared.)

12.4.3. Correlation by Authorization Servers

Clients are expected to be identified by their client instance key. If a particular client instance key is used at more than one AS, this could open up the possibility for multiple unrelated ASes to correlate client instances. This is especially a problem in the common case where a client instance is used by a single individual, as it would allow the ASes to correlate that individual between them. If this is a concern of a client, the client should use distinct keys with each AS.

12.5. Disclosure in Shared References

Throughout many parts of GNAP, the parties pass shared references between each other, sometimes in place of the values themselves (for example, the `interact_ref` value used throughout the flow). These references are intended to be random strings and should not contain any private or sensitive data that could potentially leak information between parties.

13. References

13.1. Normative References

[BCP195] Best Current Practice 195, <https://www.rfc-editor.org/info/bcp195>.
At the time of writing, this BCP comprises the following:

Moriarty, K. and S. Farrell, "Deprecating TLS 1.0 and TLS 1.1", BCP 195, RFC 8996, DOI 10.17487/RFC8996, March 2021, <https://www.rfc-editor.org/info/rfc8996>.

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/info/rfc9325>.

[HASH-ALG] IANA, "Named Information Hash Algorithm Registry", <https://www.iana.org/assignments/named-information/>.

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

[OIDC] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 2", December 2023, <https://openid.net/specs/openid-connect-core-1_0.html>.

[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>.

[RFC2397] Masinter, L., "The "data" URL scheme", RFC 2397, DOI 10.17487/RFC2397, August 1998, <https://www.rfc-editor.org/info/rfc2397>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <https://www.rfc-editor.org/info/rfc5646>.

[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>.

[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/info/rfc6750>.

[RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <https://www.rfc-editor.org/info/rfc7468>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

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

[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>.

[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>.

[RFC9111] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/info/rfc9111>.

[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/info/rfc9421>.

[RFC9493] Backman, A., Ed., Scurtescu, M., and P. Jain, "Subject Identifiers for Security Event Tokens", RFC 9493, DOI 10.17487/RFC9493, December 2023, <https://www.rfc-editor.org/info/rfc9493>.

[RFC9530] Polli, R. and L. Pardue, "Digest Fields", RFC 9530, DOI 10.17487/RFC9530, February 2024, <https://www.rfc-editor.org/info/rfc9530>.

[SAML2] Cantor, S., Ed., Kemp, J., Ed., Philpott, R., Ed., and E. Maler, Ed., "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard, March 2005, <https://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf>.

13.2. Informative References

[Auth-Schemes] IANA, "HTTP Authentication Schemes", <https://www.iana.org/assignments/http-authschemes>.

[AXELAND2021] Axeland, Å. and O. Oueidat, "Security Analysis of Attack Surfaces on the Grant Negotiation and Authorization Protocol", Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, 2021, <https://hdl.handle.net/20.500.12380/304105>.

[GNAP-REG] IANA, "Grant Negotiation and Authorization Protocol (GNAP)", <https://www.iana.org/assignments/gnap>.

[GNAP-RS] Richer, J., Ed. and F. Imbault, "Grant Negotiation and Authorization Protocol Resource Server Connections", Work in Progress, Internet-Draft, draft-ietf-gnap-resource-servers-08, 9 August 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-gnap-resource-servers-08>.

[HELMSCHMIDT2022] Helmschmidt, F., "Security Analysis of the Grant Negotiation and Authorization Protocol", Master's thesis, Institute of Information Security, University of Stuggart, DOI 10.18419/opus-12203, 2022, <http://dx.doi.org/10.18419/opus-12203>.

[MediaTypes] IANA, "Media Types", <https://www.iana.org/assignments/media-types>.

[OAUTH-SEC-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-29, 3 June 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-29>.

[promise-theory] Bergstra, J. and M. Burgess, "Promise Theory: Principles and Applications", Second Edition, XtAxis Press, 2019, <http://markburgess.org/promises.html>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[RFC4107] Bellovin, S. and R. Housley, "Guidelines for Cryptographic Key Management", BCP 107, RFC 4107, DOI 10.17487/RFC4107, June 2005, <https://www.rfc-editor.org/info/rfc4107>.

[RFC6202] Loreto, S., Saint-Andre, P., Salsano, S., and G. Wilkins, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP", RFC 6202, DOI 10.17487/RFC6202, April 2011, <https://www.rfc-editor.org/info/rfc6202>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/info/rfc6973>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8264] Saint-Andre, P. and M. Blanchet, "PRECIS Framework: Preparation, Enforcement, and Comparison of Internationalized Strings in Application Protocols", RFC 8264, DOI 10.17487/RFC8264, October 2017, <https://www.rfc-editor.org/info/rfc8264>.

[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>.

[RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/info/rfc8792>.

[RFC9396] Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", RFC 9396, DOI 10.17487/RFC9396, May 2023, <https://www.rfc-editor.org/info/rfc9396>.

[RFC9440] Campbell, B. and M. Bishop, "Client-Cert HTTP Header Field", RFC 9440, DOI 10.17487/RFC9440, July 2023, <https://www.rfc-editor.org/info/rfc9440>.

[RFC9525] Saint-Andre, P. and R. Salz, "Service Identity in TLS", RFC 9525, DOI 10.17487/RFC9525, November 2023, <https://www.rfc-editor.org/info/rfc9525>.

[SP80063C] Grassi, P., Richer, J., Squire, S., Fenton, J., Nadeau, E., Lefkovitz, N., Danker, J., Choong, Y., Greene, K., and M. Theofanos, "Digital Identity Guidelines: Federation and Assertions", NIST SP 800-63C, DOI 10.6028/NIST.SP.800-63c, June 2017, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63c.pdf>.

[Subj-ID-Formats] IANA, "Subject Identifier Formats", <https://www.iana.org/assignments/secevent>.

Appendix A. Comparison with OAuth 2.0

GNAP's protocol design differs from OAuth 2.0's in several fundamental ways:

1. **Consent and authorization flexibility:**

   OAuth 2.0 generally assumes the user has access to a web browser. The type of interaction available is fixed by the grant type, and the most common interactive grant types start in the browser. OAuth 2.0 assumes that the user using the client software is the same user that will interact with the AS to approve access.

   GNAP allows various patterns to manage authorizations and consents required to fulfill this requested delegation, including information sent by the client instance, information supplied by external parties, and information gathered through the interaction process. GNAP allows a client instance to list different ways that it can start and finish an interaction, and these can be mixed together as needed for different use cases. GNAP interactions can use a browser, but they don't have to. Methods can use inter-application messaging protocols, out-of-band data transfer, or anything else. GNAP allows extensions to define new ways to start and

finish an interaction, as new methods and platforms are expected to become available over time. GNAP is designed to allow the end user and the RO to be two different people, but it still works in the optimized case of them being the same party.

2. **Intent registration and inline negotiation:**

   OAuth 2.0 uses different "grant types" that start at different endpoints for different purposes. Many of these require discovery of several interrelated parameters.

   GNAP requests all start with the same type of request to the same endpoint at the AS. Next steps are negotiated between the client instance and AS based on software capabilities, policies surrounding requested access, and the overall context of the ongoing request. GNAP defines a continuation API that allows the client instance and AS to request and send additional information from each other over multiple steps. This continuation API uses the same access token protection that other GNAP-protected APIs use. GNAP allows discovery to optimize the requests, but it isn't required thanks to the negotiation capabilities.

   GNAP is able to handle the life cycle of an authorization request and therefore simplifies the mental model surrounding OAuth2. For instance, there's no need for refresh tokens when the API enables proper rotation of access tokens.

3. **Client instances:**

   OAuth 2.0 requires all clients to be registered at the AS and to use a client_id known to the AS as part of the protocol. This client_id is generally assumed to be assigned by a trusted authority during a registration process, and OAuth places a lot of trust on the client_id as a result. Dynamic registration allows different classes of clients to get a client_id at runtime, even if they only ever use it for one request.

   GNAP allows the client instance to present an unknown key to the AS and use that key to protect the ongoing request. GNAP's client instance identifier mechanism allows for pre-registered clients and dynamically registered clients to exist as an optimized case without requiring the identifier as part of the protocol at all times.

4. **Expanded delegation:**

   OAuth 2.0 defines the "scope" parameter for controlling access to APIs. This parameter has been coopted to mean a number of different things in different protocols, including flags for turning special behavior on and off and the return of data apart from the access token. The "resource" indicator (defined in [RFC8707]) and Rich Authorization Request (RAR) extensions (as defined in [RFC9396]) expand on the "scope" concept in similar but different ways.

   GNAP defines a rich structure for requesting access (analogous to RAR), with string references as an optimization (analogous to scopes). GNAP defines methods for requesting directly returned user information, separate from API access. This information includes identifiers for the current user and structured assertions. GNAP makes no assumptions or demands on the format or contents of the access token, but the RS extension allows a negotiation of token formats between the AS and RS.

5. **Cryptography-based security:**

OAuth 2.0 uses shared bearer secrets, including the client_secret and access token, and advanced authentication and sender constraints have been built on after the fact in inconsistent ways.

In GNAP, all communication between the client instance and AS is bound to a key held by the client instance. GNAP uses the same cryptographic mechanisms for both authenticating the client (to the AS) and binding the access token (to the RS and the AS). GNAP allows extensions to define new cryptographic protection mechanisms, as new methods are expected to become available over time. GNAP does not have the notion of "public clients" because key information can always be sent and used dynamically.

6. **Privacy and usable security:**

   OAuth 2.0's deployment model assumes a strong binding between the AS and the RS.

   GNAP is designed to be interoperable with decentralized identity standards and to provide a human-centric authorization layer. In addition to this specification, GNAP supports various patterns of communication between RSs and ASes through extensions. GNAP tries to limit the odds of a consolidation to just a handful of popular AS services.

Appendix B. Example Protocol Flows

The protocol defined in this specification provides a number of features that can be combined to solve many different kinds of authentication scenarios. This section seeks to show examples of how the protocol could be applied for different situations.

Some longer fields, particularly cryptographic information, have been truncated for display purposes in these examples.

B.1. Redirect-Based User Interaction

In this scenario, the user is the RO and has access to a web browser, and the client instance can take front-channel callbacks on the same device as the user. This combination is analogous to the OAuth 2.0 Authorization Code grant type.

The client instance initiates the request to the AS. Here, the client instance identifies itself using its public key.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            {
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            }
        ],
    },
    "client": {
      "key": {
        "proof": "httpsig",
        "jwk": {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "xyz-1",
            "alg": "RS256",
            "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8..."
        }
      }
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    }
}
```

The AS processes the request and determines that the RO needs to interact. The AS returns the following response that gives the client instance the information it needs to connect. The AS has also indicated to the client instance that it can use the given instance identifier to identify itself in future requests (Section 2.3.1).

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "interact": {
      "redirect":
        "https://server.example.com/interact/4CF492MLVMSW9MKM",
      "finish": "MBDOFXG4Y5CVJCX821LH"
    }
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue"
    },
    "instance_id": "7C7C4AZ9KHRS6X63AJAO"
}
```

The client instance saves the response and redirects the user to the interaction start mode's "redirect" URI by sending the following HTTP message to the user's browser.

```
HTTP 303 Found
Location: https://server.example.com/interact/4CF492MLVMSW9MKM
```

The user's browser fetches the AS's interaction URI. The user logs in, is identified as the RO for the resource being requested, and approves the request. Since the AS has a callback parameter that was sent in the initial request's interaction finish method, the AS generates the interaction reference, calculates the hash, and redirects the user back to the client instance with these additional values added as query parameters.

```
NOTE: '\' line wrapping per RFC 8792

HTTP 302 Found
Location: https://client.example.net/return/123455\
  ?hash=x-gguKWTj8rQf7d7i3w3UhzvuJ5bpOlKyAlVpLxBffY\
  &interact_ref=4IFWWIKYBC2PQ6U56NL1
```

The client instance receives this request from the user's browser. The client instance ensures that this is the same user that was sent out by validating session information and retrieves the stored pending request. The client instance uses the values in this to validate the hash parameter. The client instance then calls the continuation URI using the associated continuation access token and presents the interaction reference in the request content. The client instance signs the request as above.

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

The AS retrieves the pending request by looking up the pending grant request associated with the presented continuation access token. Seeing that the grant is approved, the AS issues an access token and returns this to the client instance.

```
NOTE: '\' line wrapping per RFC 8792

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [{
            "actions": [
                "read",
                "write",
                "dolphin"
            ],
            "locations": [
                "https://server.example.net/",
                "https://resource.local/other"
            ],
            "datatypes": [
                "metadata",
                "images"
            ]
        }]
    },
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue"
    }
}
```

B.2. Secondary Device Interaction

In this scenario, the user does not have access to a web browser on the device and must use a secondary device to interact with the AS. The client instance can display a user code or a printable QR code. The client instance is not able to accept callbacks from the AS and needs to poll for updates while waiting for the user to authorize the request.

The client instance initiates the request to the AS.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "dolphin-metadata", "some other thing"
        ],
    },
    "client": "7C7C4AZ9KHRS6X63AJA0",
    "interact": {
        "start": ["redirect", "user_code"]
    }
}
```

The AS processes this and determines that the RO needs to interact. The AS supports both redirect URIs and user codes for interaction, so it includes both. Since there is no interaction finish mode, the AS does not include a nonce but does include a "wait" parameter on the continuation section because it expects the client instance to poll for results.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "interact": {
        "redirect": "https://srv.ex/MXKHQ",
        "user_code": {
            "code": "A1BC3DFF"
        }
    },
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue/VGJKPTKC50",
        "wait": 60
    }
}
```

The client instance saves the response and displays the user code visually on its screen along with the static device URI. The client instance also displays the short interaction URI as a QR code to be scanned.

If the user scans the code, they are taken to the interaction endpoint, and the AS looks up the current pending request based on the incoming URI. If the user instead goes to the static page and enters the code manually, the AS looks up the current pending request based on the value of the user code. In both cases, the user logs in, is identified as the RO for the resource being requested, and approves the request. Once the request has been approved, the AS displays to the user a message to return to their device.

Meanwhile, the client instance polls the AS every 60 seconds at the continuation URI. The client instance signs the request using the same key and method that it did in the first request.

```
POST /continue/VGJKPTKC50 HTTP/1.1
Host: server.example.com
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...
```

The AS retrieves the pending request based on the pending grant request associated with the continuation access token and determines that it has not yet been authorized. The AS indicates to the client instance that no access token has yet been issued but it can continue to call after another 60-second timeout.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "continue": {
        "access_token": {
            "value": "G7YQT4KQQ5TZY9SLSS5E"
        },
        "uri": "https://server.example.com/continue/ATWHO4Q1WV",
        "wait": 60
    }
}
```

Note that the continuation URI and access token have been rotated since they were used by the client instance to make this call. The client instance polls the continuation URI after a 60-second timeout using this new information.

```
POST /continue/ATWHO4Q1WV HTTP/1.1
Host: server.example.com
Authorization: GNAP G7YQT4KQQ5TZY9SLSS5E
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...
```

The AS retrieves the pending request based on the URI and access token, determines that it has been approved, and issues an access token for the client to use at the RS.

```
NOTE: '\' line wrapping per RFC 8792

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [
            "dolphin-metadata", "some other thing"
        ]
    }
}
```

B.3. No User Involvement

In this scenario, the client instance is requesting access on its own behalf, with no user to interact with.

The client instance creates a request to the AS, identifying itself with its public key and using MTLS to make the request.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json

{
    "access_token": {
        "access": [
            "backend service", "nightly-routine-3"
        ],
    },
    "client": {
      "key": {
        "proof": "mtls",
        "cert#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
      }
    }
}
```

The AS processes this, determines that the client instance can ask for the requested resources, and issues an access token.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token",
        "access": [
            "backend service", "nightly-routine-3"
        ]
    }
}
```

B.4. Asynchronous Authorization

In this scenario, the client instance is requesting on behalf of a specific RO but has no way to interact with the user. The AS can asynchronously reach out to the RO for approval in this scenario.

The client instance starts the request at the AS by requesting a set of resources. The client instance also identifies a particular user.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "read", "dolphin-metadata",
            {
                "type": "financial-transaction",
                "actions": [
                    "withdraw"
                ],
                "identifier": "account-14-32-32-3",
                "currency": "USD"
            },
            "some other thing"
        ],
    },
    "client": "7C7C4AZ9KHRS6X63AJAO",
    "user": {
        "sub_ids": [ {
            "format": "opaque",
            "id": "J2G8G8O4AZ"
        } ]
  }
}
```

The AS processes this and determines that the RO needs to interact. The AS determines that it can reach the identified user asynchronously and that the identified user does have the ability to approve this request. The AS indicates to the client instance that it can poll for continuation.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 60
    }
}
```

The AS reaches out to the RO and prompts them for consent. In this example scenario, the AS has an application that it can push notifications to for the specified account.

Meanwhile, the client instance periodically polls the AS every 60 seconds at the continuation URI.

```
POST /continue HTTP/1.1
Host: server.example.com
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
```

The AS retrieves the pending request based on the continuation access token and determines that it has not yet been authorized. The AS indicates to the client instance that no access token has yet been issued but it can continue to call after another 60-second timeout.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "continue": {
        "access_token": {
            "value": "BI9QNW6V9W3XFJK4R02D"
        },
        "uri": "https://server.example.com/continue",
        "wait": 60
    }
}
```

Note that the continuation access token value has been rotated since it was used by the client instance to make this call. The client instance polls the continuation URI after a 60-second timeout using the new token.

```
POST /continue HTTP/1.1
Host: server.example.com
Authorization: GNAP BI9QNW6V9W3XFJK4R02D
Signature-Input: sig1=...
Signature: sig1=...
```

The AS retrieves the pending request based on the handle, determines that it has been approved, and issues an access token.

```
NOTE: '\' line wrapping per RFC 8792

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [
            "dolphin-metadata", "some other thing"
        ]
    }
}
```

B.5. Applying OAuth 2.0 Scopes and Client IDs

While GNAP is not designed to be directly compatible with OAuth 2.0 [RFC6749], considerations have been made to enable the use of OAuth 2.0 concepts and constructs more smoothly within GNAP.

In this scenario, the client developer has a `client_id` and set of `scope` values from their OAuth 2.0 system and wants to apply them to the new protocol. In OAuth 2.0, the client developer would put their `client_id` and `scope` values as parameters into a redirect request to the authorization endpoint.

```
NOTE: '\' line wrapping per RFC 8792

HTTP 302 Found
Location: https://server.example.com/authorize\
  ?client_id=7C7C4AZ9KHRS6X63AJAO\
  &scope=read%20write%20dolphin\
  &redirect_uri=https://client.example.net/return\
  &response_type=code\
  &state=123455
```

Now the developer wants to make an analogous request to the AS using GNAP. To do so, the client instance makes an HTTP POST and places the OAuth 2.0 values in the appropriate places.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read", "write", "dolphin"
        ],
        "flags": [ "bearer" ]
    },
    "client": "7C7C4AZ9KHRS6X63AJA0",
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return?state=123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    }
}
```

The `client_id` can be used to identify the client instance's keys that it uses for authentication, the scopes represent resources that the client instance is requesting, and the `redirect_uri` and `state` value are pre-combined into a `finish` URI that can be unique per request. The client instance additionally creates a nonce to protect the callback, separate from the state parameter that it has added to its return URI.

From here, the protocol continues as above.

Appendix C. Interoperability Profiles

The GNAP specification has many different modes, options, and mechanisms, allowing it to solve a wide variety of problems in a wide variety of deployments. The wide applicability of GNAP makes it difficult, if not impossible, to define a set of mandatory-to-implement features, since one environment's required feature would be impossible to do in another environment. While this is a large problem in many systems, GNAP's back-and-forth negotiation process allows parties to declare at runtime everything that they support and then have the other party select from that the subset of items that they also support, leading to functional compatibility in many parts of the protocol even in an open world scenario.

In addition, GNAP defines a set of interoperability profiles that gather together core requirements to fix options into common configurations that are likely to be useful to large populations of similar applications.

Conformant AS implementations of these profiles **MUST** implement at least the features as specified in the profile and **MAY** implement additional features or profiles. Conformant client implementations of these profiles **MUST** implement at least the features as specified, except where a subset of the features allows the protocol to function (such as using polling instead of a push finish method for the Secondary Device profile).

C.1. Web-Based Redirection

Implementations conformant to the web-based redirection profile of GNAP **MUST** implement all of the following features:

- Interaction Start Methods: `redirect`
- Interaction Finish Methods: `redirect`
- Interaction Hash Algorithms: `sha-256`
- Key Proofing Methods: `httpsig` with no additional parameters
- Key Formats: `jwks` with signature algorithm included in the key's `alg` parameter
- JOSE Signature Algorithm: PS256
- Subject Identifier Formats: `opaque`
- Assertion Formats: `id_token`

C.2. Secondary Device

Implementations conformant to the Secondary Device profile of GNAP **MUST** implement all of the following features:

- Interaction Start Methods: `user_code` and `user_code_uri`
- Interaction Finish Methods: `push`
- Interaction Hash Algorithms: `sha-256`
- Key Proofing Methods: `httpsig` with no additional parameters
- Key Formats: `jwks` with signature algorithm included in the key's `alg` parameter
- JOSE Signature Algorithm: PS256
- Subject Identifier Formats: `opaque`
- Assertion Formats: `id_token`

Appendix D. Guidance for Extensions

Extensions to this specification have a variety of places to alter the protocol, including many fields and objects that can have additional values in a registry (Section 10) established by this specification. For interoperability and to preserve the security of the protocol, extensions should register new values with IANA by following the specified mechanism. While it may technically be possible to extend the protocol by adding elements to JSON objects that are not governed by an IANA registry, a recipient may ignore such values but is also allowed to reject them.

Most object fields in GNAP are specified with types, and those types can allow different but related behavior. For example, the `access` array can include either strings or objects, as discussed in Section 8. The use of JSON polymorphism (Appendix E) within GNAP allows extensions to define new fields by not only choosing a new name but also by using an existing name with a new type. However, the extension's definition of a new type for a field needs to fit the same kind of item being extended. For example, a hypothetical extension could define a string value for the `access_token` request field, with a URL to download a hosted access token request. Such an extension would be appropriate as the `access_token` field still defines the access tokens being requested. However, if an extension were to define a string value for the `access_token` request field, with the value instead being something unrelated to the access token request such as a value or key format, this would not be an appropriate means of extension. (Note that this specific extension example would create another form of SSRF attack surface as discussed in Section 11.34.)

As another example, both interaction start modes (Section 2.5.1) and key proofing methods (Section 7.3) can be defined as either strings or objects. An extension could take a method defined as a string, such as `app`, and define an object-based version with additional parameters. This extension should still define a method to launch an application on the end user's device, just like `app` does when specified as a string.

Additionally, the ability to deal with different types for a field is not expected to be equal between an AS and client software, with the client software being assumed to be both more varied and more simplified than the AS. Furthermore, the nature of the negotiation process in GNAP allows the AS more chance of recovery from unknown situations and parameters. As such, any extensions that change the type of any field returned to a client instance should only do so when the client instance has indicated specific support for that extension through some kind of request parameter.

Appendix E. JSON Structures and Polymorphism

GNAP makes use of polymorphism within the JSON [RFC8259] structures used for the protocol. Each portion of this protocol is defined in terms of the JSON data type that its values can take, whether it's a string, object, array, boolean, or number. For some fields, different data types offer different descriptive capabilities and are used in different situations for the same field. Each data type provides a different syntax to express the same underlying semantic protocol element, which allows for optimization and simplification in many common cases.

Even though JSON is often used to describe strongly typed structures, JSON on its own is naturally polymorphic. In JSON, the named members of an object have no type associated with them, and any data type can be used as the value for any member. In practice, each member has a semantic type that needs to make sense to the parties creating and consuming the object. Within this protocol, each object member is defined in terms of its semantic content, and this semantic content might have expressions in different concrete data types for different specific purposes. Since each object member has exactly one value in JSON, each data type for an object member field is naturally mutually exclusive with other data types within a single JSON object.

For example, a resource request for a single access token is composed of an object of resource request descriptions, while a request for multiple access tokens is composed of an array whose member values are all objects. Both of these represent requests for access, but the difference in syntax allows the client instance and AS to differentiate between the two request types in the same request.

Another form of polymorphism in JSON comes from the fact that the values within JSON arrays need not all be of the same JSON data type. However, within this protocol, each element within the array needs to be of the same kind of semantic element for the collection to make sense, even when the data types are different from each other.

For example, each aspect of a resource request can be described using an object with multiple dimensional components, or the aspect can be requested using a string. In both cases, the resource request is being described in a way that the AS needs to interpret, but with different levels of specificity and complexity for the client instance to deal with. An API designer can provide a set of common access scopes as simple strings but still allow client software developers to specify custom access when needed for more complex APIs.

Extensions to this specification can use different data types for defined fields, but each extension needs to not only declare what the data type means but also provide justification for the data type representing the same basic kind of thing it extends. For example, an extension declaring an "array" representation for a field would need to explain how the array represents something akin to the non-array element that it is replacing. See additional discussion in Appendix D.

Acknowledgements

The authors would like to thank the following individuals for their reviews, implementations, and contributions: Åke Axeland, Aaron Parecki, Adam Omar Oueidat, Andrii Deinega, Annabelle Backman, Dick Hardt, Dmitri Zagidulin, Dmitry Barinov, Florian Helmschmidt, Francis Pouatcha, George Fletcher, Haardik Haardik, Hamid Massaoud, Jacky Yuan, Joseph Heenan, Kathleen Moriarty, Leif Johansson, Mike Jones, Mike Varley, Nat Sakimura, Takahiko Kawasaki, Takahiro Tsuchiya, and Yaron Sheffer.

The authors would also like to thank the GNAP Working Group design team (Kathleen Moriarty, Dick Hardt, Mike Jones, and the authors), who incorporated elements from the XAuth and XYZ proposals to create the first draft version of this document.

In addition, the authors would like to thank Aaron Parecki and Mike Jones for insights into how to integrate identity and authentication systems into the core protocol. Both Justin Richer and Dick Hardt developed the use cases, diagrams, and insights provided in the XYZ and XAuth proposals that have been incorporated here. The authors would like to especially thank Mike Varley and the team at SecureKey for feedback and development of early versions of the XYZ protocol that fed into this standards work.

Finally, the authors want to acknowledge the immense contributions of Aaron Parecki to the content of this document. We thank him for his insight, input, and hard work, without which GNAP would not have grown to what it is.

Authors' Addresses

Justin Richer (EDITOR)
Bespoke Engineering
Email: ietf@justin.richer.org
URI: https://bspk.io/

Fabien Imbault
acert.io
Email: fabien.imbault@acert.io
URI: https://acert.io/