Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: 4 July 2026

A. Goswami, Ed.
31 December 2025

Secure Intent Protocol: JWT Compatible Agentic Identity and Workflow Management
draft-goswami-agentic-jwt-00

Abstract

This document specifies Agentic JSON Web Token (Agentic JWT), as an extension to OAuth 2.0 that addresses authorization challenges unique to autonomous agentic AI systems. This protocol solves the problem of Zero-Trust drift due to non-deterministic Agentic AI clients causing separation between user's (resource owner's) intent and client application's actions.

Traditional OAuth 2.0 assumes that client applications faithfully represent user intent when requesting authorization. This assumption breaks down when autonomous AI agents dynamically generate workflows, create sub-agents, and make authorization decisions without continuous human oversight. We term this the "intent-execution separation problem."

Agentic JWT introduces three mechanisms to address this problem: (1) cryptographic agent identity through agent checksums (based on agent's system prompts, tools and configurations), (2) workflow-aware token binding that links user intent to agent execution, and (3) a new OAuth 2.0 grant type (agent_checksum) for secure token issuance, (4) a flavor of PoP (Proof-of-Possession) at the level of an agentic identity to prevent token replays by other agents in the same multi-agent process.

This specification enables Zero-Trust security principles in multi-agent systems while maintaining backward compatibility with existing OAuth 2.0 infrastructure. The security analysis and experimental validation are described in the companion research paper.

Status of This Memo

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

This Internet-Draft will expire on 4 July 2026.

Copyright Notice

Table of Contents

1. Introduction

OAuth 2.0 is designed for deterministic client applications that execute some workflow on behalf of a resource owner [RFC6749]. The principle is that an authorization server (or Identity Provider, IDP) validates the client application using an authorization grant appoved by the resource owner and issues an access token as the proof of authorization, which is then used by some fixed workflow on the client to invoke an API hosted on the resource server. This mechanism breaks down for client applications using agentic AI with LLM reasoning because the decision to call resource server APIs could be taken by non-deterministic LLM reasoning that creates a separation between user's intent and actual agentic execution.

Consider a multi-agent vulnerability patching system where a supervisor agent coordinates planner, classifier, and patcher sub-agents. Traditional OAuth 2.0 cannot detect when a low-privilege classifier agent tricks a high-privilege patcher into modifying critical system files, as the bearer token provides no cryptographic binding between user intent and agent execution. While next-generation protocols like GNAP [I-D.ietf-txauth-gnap] address some OAuth 2.0 limitations through JSON-based grant negotiation, they do not provide the agent-specific identity verification and workflow binding mechanisms required for autonomous AI systems.

This document addresses the following critical gaps in existing authorization frameworks for autonomous agents:

* Agent Identity: How to cryptographically verify agent identity beyond client_id credentials

* Intent Binding: How to bind user-authorized intent to specific agent workflows

* Delegation Control: How to prevent unauthorized cross-agent privilege escalation

* Agent level Proof-of-Posession: How to prevent token replaying by another agent running within the same client process.

This specification does NOT address:

* Model non-determinism or LLM safety

* Agent framework-specific implementation details

* Performance optimization techniques

1.1. Motivation

Agentic AI based applications are rapidly solving more use cases and have already become targets for massive investments in enterprises across industries. Agentic clients would likely proliferate much faster than the currently available Zero-Trust implementations, including OAuth 2.0. The fundamental Zero-Trust principles [NIST-SP-800-207] outlined in the NIST 800.63 publication [NIST-SP-800-63] and [NIST-SP-800-63C] are very much applicable and sound but current implementations need to catch up. This protocol enhancement attempts to achieve permanent protection from the possibility of Zero-Trust drift in API invocations done by agentic client applications. This specification outlines this novel protocol as fully backward compatible with the current OAuth 2.0 primitives while expanding the token issuance and resource server validation processes to recognzie agents as separate identities, their currently running version, the currently executing workflow and entire delegation chain in case of multi-agent applications. It also defines a way to make agentic clients auditable on the authorization server side.

1.2. Requirements Language

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.

1.3. Companion Research

A comprehensive threat modelling using the STRIDE methodology [SHOSTACK], including experimental evaluation, and performance analysis is described in the research paper, [PAPER-REF]. This document focuses on protocol specification for implementers.

The system architecture and mechanisms described in this specification are the subject of U.S. Patent Application 19/315,486 [PATENT-REF].

2. Terminology

This document uses the following terms in addition to those defined in [RFC6749] and [RFC7519]

Agentic Client:
An autonomous AI agent that acts as an OAuth 2.0 client. Unlike traditional clients, agentic clients can dynamically generate workflows, create sub-agents, and make authorization decisions without continuous human interaction.

Agent Checksum:
A SHA-256 cryptographic hash computed over the agent's configuration, including system prompt, available tools, and model parameters. The agent checksum uniquely identifies the agent's behavioral specification and serves as its cryptographic identity.

Intent Token:
A JWT (JSON Web token, [RFC7519]) issued by the authorization server that encapsulates user-authorized intent. The intent token includes new claims, and semantics such the workflow definition, requested scopes, agent checksum binding, workflow delegation chain, and user approval status. It is fully compatible with plain JWT tokens and other JWT primitives like JWS (JSON Web Signature, [RFC7515]) and JWE (JSON Web Encryption, [RFC7516]).

Workflow Binding:
The cryptographic association between a user-approved workflow and the access token issued for its execution. Implemented through the workflow_id and workflow_hash claims.

Delegation Assertion:
A cryptographic proof that an agent is authorized to execute a specific workflow on behalf of a user. Combines agent identity verification, intent binding, and scope restrictions. The mechanism is similar in spirit to OAuth 2.0 Token Exchange [RFC8693] but uses agent-specific cryptographic binding instead of token exchange grants.

Agent Checksum Grant or agent_checksum:
A new OAuth 2.0 grant type (defined in Section 7) that allows agents to exchange intent tokens for access tokens using agent checksum verification instead of client credentials.

Supervisor Agent:
In multi-agent systems, the agent that coordinates sub-agent workflows and maintains the delegation chain. Referenced in examples throughout this document.

Intent-Execution Separation:
The security gap that occurs when an agent executes actions that diverge from user-authorized intent. Traditional OAuth 2.0 cannot detect this separation because bearer tokens do not bind authorization to specific execution contexts.

Human in the Loop
An agentic AI architectural pattern that involves some steps of an agentic workflow to be done or approved by human. The agentic workflow executes to a specific point and then pauses by notifying for human approval, it then waits until an approval is received before continuiing with the workflow.

Authorization Server or Identity Provider or IDP
This is the "Authorization Server" as defined in [RFC6749]

Client Shim Library
Designed to be used as a dependency in any agentic client application to make the protocol flow of this intent based agentic jwt protocol easy to implement. It reduces the work on these client applications to a single line of code and hides all the complexity involved in performing the tasks necessary for seamless minting of intent tokens, verifying agent registrations from IDP (or Authorization Server), and tracking runtime agent workflow so that workflow state can be used while requesting an intent token.

2.1. Agent Classifications

This specification recognizes three agent types based on autonomy level:

Fully Supervised Agents: Require human approval for each action

Semi-Autonomous Agents: Execute pre-approved workflows with human oversight, such as when using Human in the Loop pattern.

Fully Autonomous Agents: Execute complete workflows without human intervention

The security requirements vary by agent type, with fully autonomous agents requiring the strongest guarantees.

3. Architecture Overview

3.1. Architectural Components

This agentic JWT architecture introduces the following novel (but backward compatible) protocol features:

* Enhancements in JWT Claims: The protocol introduces some new claims required for supporting the verification of agentic AI clients.

* A new Authorization Grant called agent_checksum: The protcol introduces a novel authorization grant [RFC6749] called agent_checksum that allows the agentic client applications to act on behalf of the resource owner and enables the authorization servers to issue tokens without explicit credentials, by simply using an agent's runtime identity. An agent's runtime identity is typically composed of its system prompt, its tool specification (or a hash of actual tool function code), and LLM configuration (if applicable).

* Authorization Server token issuing protocol: The protocol adds unique verification capabilities on the authorization server side used during token issuance process.

* Resource Server verification protocol: The protocol adds additional verification capabilities on the resource server side.

* Intent Token: This is the new JWT compatible token that contains the additional claims plus the agent specific identity information. (see Intent Token (Section 2)).

All of the above mentioned architectural choices are fully backward compatible with OAuth 2.0.

The architecture components that make these features possible can be implemented using general purpose programming languages like python, nodejs, java etc. The exact logical component design is outlined below:

* Client Shim Library: Implements the client side logic that performs the tasks necessary for seamless minting of intent tokens, verifying agent registrations from IDP (or Authorization Server), and tracking runtime agent workflow so that workflow state can be used while requesting an intent token.

* Authorization Server Library: Implements the Authorization Server side validation logic for authenticating incoming client requests using agent_checksum authorization grant.

* Resource Server Library: Implements the Resource Server side logic for token validation for intent tokens including per agent Proof-of-Posession verification.

3.2. Abstract Protocol Flow

The Agentic JWT protocol flow is conceptually similar to that of OAuth 2.0 [RFC6749] at high level. However, the internal structure and mechanism is different and encapsulates agentic aware protocol features.

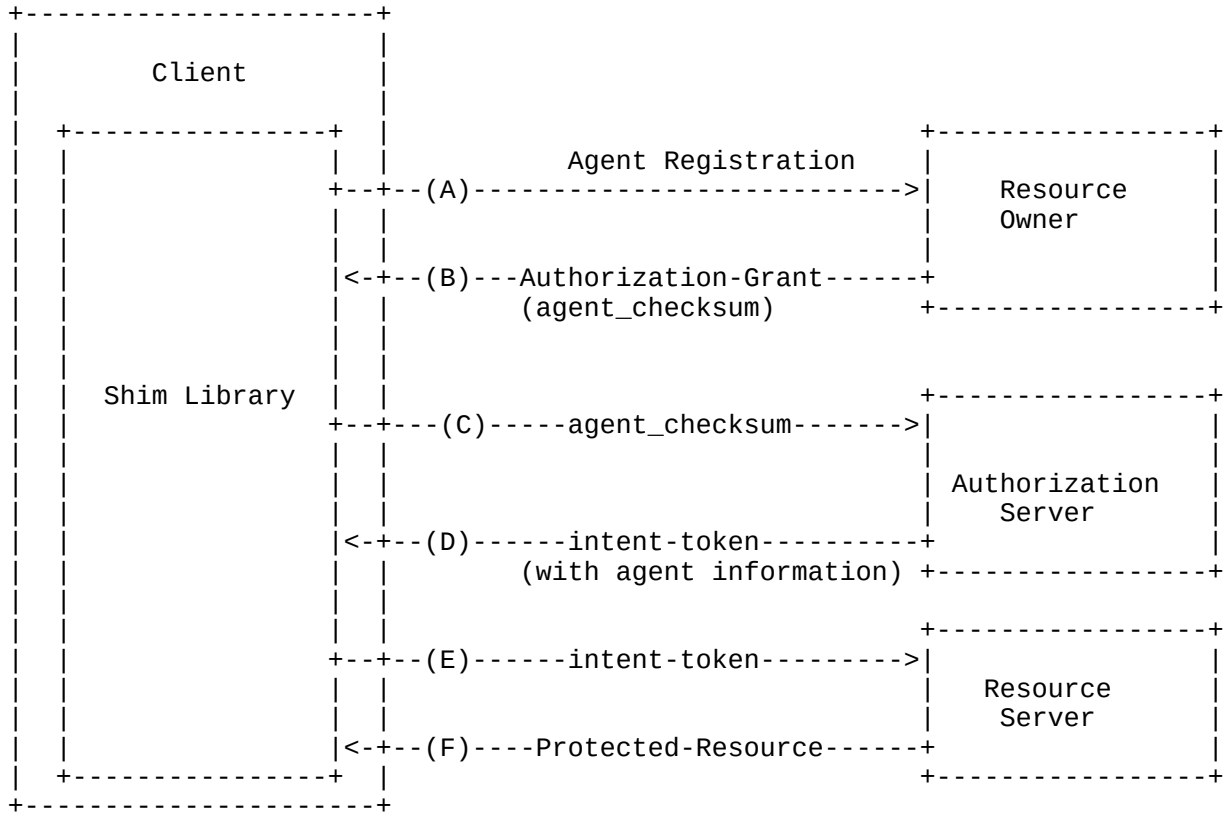


Figure 1: Agentic JWT Abstract Protocol Flow

The protocol flow described in Figure 1 outlines the interactions between the client (via Shim Library), the Resource Owner, the Authorization Server, and the Resource Server. This flow is fully compatible with the original OAuth 2.0 protocol flow. The complexities of verifying registered agents on the Client, capturing runtime identity of an agent for minting intent tokens, tracking

runtime workflow and sending these to Authorization Server with the intent token request, are hidden inside the Shim Library. A Client application just needs to include the Shim library as a dependency to make this flow work. The protocol flow steps are described below:

(A) The client makes an authorization grant request to the resource owner either directly or preferably via the authorization server. Having authorization grant means that each agent running within the client process has been registered as identity in the authorization server with an Agent Checksum (see Section 2) of its system prompt, tool specifications, and LLM configuration.

(B) The client receives an authorization grant that can be used to request intent tokens from the authorization server. This is new authorization grant type called agent_checksum. The agent_checksum authorization grant is based on an agent's inherent signature used as its identity. An agent's signature comprises of its system prompt, tools specifications (or tool function code), and LLM configuration. This new grant type is described in Section 4

(C) The shim library running within the client process requests an intent token by presenting the authorization grant of 'agent_checksum' which uses the currently running agent's signature as part of the input. The shim library computes the runtime Agent Checksum (see Section 2 of the running agent and includes in the intent token request.

(D) The authorization server validates the request against the registered agents, by comparing agent checksum in the input with regisrered checksums, and if valid, issues the intent token.

(E) The shim library, running within the client process, requests the protected resource from the resource server by presenting the intent token for authentication and authorization information.

(F) The resource server validates the intent token,and if valid, serves the request.

4. Agent Checksum Grant Type

4.1. Overview

This specification defines a new OAuth 2.0 grant type for autonomous AI agents: "agent_checksum". This grant type enables agents to obtain access tokens (called intent tokens in context of this specification) by proving their identity through cryptographic checksum verification rather than traditional client credentials.

The grant type is designed specifically for agentic clients that:

* Have registered their agent checksum with the authorization server

* Optionally execute pre-approved workflows with defined steps

* Maintain delegation chains across multi-agent systems

* Require proof-of-possession token binding

Grant Type Identifier:

urn:ietf:params:oauth:grant-type:agent_checksum

Note: For brevity in examples, this document uses the short form "agent_checksum" instead of the full URN.

4.2. Token Request

4.2.1. Request Format

The client shim library (see Section 2), on behalf of an agent, makes a request to the token endpoint by sending the following parameters using the "application/json" format with a character encoding of UTF-8:

grant_type: REQUIRED. Value MUST be "agent_checksum" (or the full URN).

agent_id: REQUIRED. The unique identifier of the registered agent.

computed_checksum: REQUIRED. The SHA-256 checksum computed over the agent's current configuration. Format: "sha256:<hex>".

workflow_id: REQUIRED when workflow_enabled is true. The identifier of the registered workflow being executed.

workflow_step: REQUIRED when workflow_enabled is true. The identifier of the specific workflow step being executed.

workflow_enabled: OPTIONAL. Boolean indicating whether workflow validation should be enforced. Default: false.

requested_scopes: REQUIRED. Array of OAuth 2.0 scopes requested for this token.

audience: REQUIRED. The intended audience (resource server) for the token. May be a single string or array of strings.

delegation_context: OPTIONAL. Object containing delegation chain information for multi-agent workflows. Contains "chain" (array of parent agent_ids) and "completed_steps" (array of completed workflow step identifiers).

4.2.2. Request Example

Example HTTP request (with line breaks for readability):

```
POST /intent/token HTTP/1.1
Host: idp.example.com
Content-Type: application/json
Authorization: Bearer <admin_token>

{
  "grant_type": "agent_checksum",
  "agent_id": "vulnerability-patcher-v1",
  "computed_checksum": "sha256:a3c7f2e8d9b4f1e2c8a7d6f3e9b2c4f1...",
  "workflow_id": "auto-patch-workflow-v1",
  "workflow_step": "step_3_patch_application",
  "workflow_enabled": true,
  "requested_scopes": [
    "repo:write",
    "vulnerability:read"
  ],
  "audience": "https://api.github.com",
  "delegation_context": {
    "chain": [
      "supervisor-agent",
      "planner-agent",
      "vulnerability-patcher-v1"
    ],
    "completed_steps": [
      "step_1_analyze_manifest",
      "step_2_create_patch_plan"
    ]
  }
}
```

4.2.3. Request Authentication

The token endpoint MUST be protected. In the reference implementation, requests require an Authorization header with a bearer token that has the "generate:intent-token" scope and audience "idp.localhost". This token may well be the usual JWT client level access token that uses one of the traditional OAuth 2.0 Authorization Grants [RFC6749].

Implementations MAY use different authentication mechanisms (e.g., client credentials, mutual TLS) but MUST ensure that only authorized entities can request tokens on behalf of agents.

4.3. Authorization Server Processing

4.3.1. Validation Sequence

Upon receiving a token request, the authorization server MUST perform the following validation steps IN ORDER:

1. *Grant Type Validation:* Verify that grant_type equals "agent_checksum". If invalid → Return error "unsupported_grant_type" (Section 4.5.2)

2. *Agent Existence Check:* Verify that agent_id exists in the agent registry. If not found → Return error "unknown_agent" (Section 4.5.3)

3. *Checksum Verification:* Retrieve the registered checksum for agent_id. Compare computed_checksum with the stored checksum. If mismatch → Return error "agent_checksum_mismatch" (Section 4.5.4)

4. *Workflow Step Authorization (if workflow_enabled):* Validate that the agent is authorized to execute the specified workflow_step within workflow_id. Verify prerequisite steps are completed (from delegation_context). Check approval requirements if step requires approval. If unauthorized → Return error "workflow_step_unauthorized" (Section 4.5.5

5. *Token Creation:* If all validations pass, create and sign the intent token (Section 2).

4.3.2. Checksum Verification Details

For the checksum verification procedure please see Section 5.5.2

4.3.3. Workflow Step Validation

When workflow_enabled is true, the authorization server MUST validate the workflow execution state:

1. Verify that workflow_id exists in the workflow registry
2. Verify that workflow_step exists within the workflow definition
3. Check that all required prerequisite steps have been completed:
    * Retrieve the step definition for workflow_step
    * Identify all steps marked as "required" that appear before workflow_step in the workflow sequence
    * Verify all required steps are present in delegation_context.completed_steps
4. If the step has requires_approval=true:
    * Locate the most recent approval gate step before workflow_step
    * Verify that approval gate step is in completed_steps
    * If no approval gate found or not completed → reject request

This validation ensures that agents cannot skip required steps or bypass approval gates in multi-step workflows.

4.3.4. Delegation Chain Validation

If delegation_context is provided, the authorization server SHOULD validate the delegation chain:

* Verify that all agents in the chain are registered
* Verify that the chain represents a valid delegation path (parent agents authorized the delegation)
* Verify that the requesting agent is the last element in the chain

The delegation_context.chain array represents the delegation hierarchy from the original initiating agent to the current agent. For example: ["supervisor", "planner", "patcher"] indicates that the supervisor delegated to the planner, which delegated to the patcher.

4.4. Successful Response

4.4.1. Response Format

If the request is valid and all verification checks pass, the authorization server issues an intent token and returns it in the response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6...",
  "token_type": "Bearer",
  "expires_in": 300,
  "scope": "repo:write vulnerability:read"
}
```

Response fields:

access_token: REQUIRED. The intent token encoded as a JWT.

token_type: REQUIRED. MUST be "Bearer".

expires_in: RECOMMENDED. Lifetime in seconds. For intent tokens, this SHOULD be short (e.g., 300 seconds / 5 minutes).

scope: OPTIONAL. Space-delimited list of scopes granted. MAY differ from requested_scopes if authorization server applies restrictions.

4.4.2. Intent Token Structure

The intent token is a JWT [RFC7519] with the following claims:

Standard JWT Claims:

iss (Issuer): REQUIRED. Authorization server identifier.

aud (Audience): REQUIRED. Intended recipient(s). Value from request.audience.

sub (Subject): REQUIRED. Agent identifier (request.agent_id).

exp (Expiration): REQUIRED. Token expiration time (Unix timestamp). SHOULD be iat + 300 seconds for intent tokens.

iat (Issued At): REQUIRED. Token issuance time (Unix timestamp).

jti (JWT ID): REQUIRED. Unique token identifier for replay prevention.

scope: REQUIRED. Space-delimited OAuth 2.0 scopes.

Proof-of-Possession Claim:

cnf (Confirmation): REQUIRED. Proof-of-possession confirmation as defined in RFC 7800 [RFC7800]. Contains "jwk" field with the agent's public key in JWK format.

Intent-Specific Claims (nested under "intent" object):

workflow_id: REQUIRED when workflow_enabled. Workflow identifier.

workflow_step: REQUIRED when workflow_enabled. Current step identifier.

executed_by: REQUIRED. Agent executing this step (same as sub).

delegation_chain: REQUIRED. Cryptographic hash of the delegation sequence. Computed as SHA-256 over the pipe-delimited chain of agent_ids, truncated to 16 hex characters. See Section 4.4.4

step_sequence_hash: REQUIRED. Cryptographic hash of completed workflow steps. Computed as SHA-256 over pipe-delimited completed step IDs, truncated to 16 hex characters. See Section 4.4.4

Agent Proof Claims (nested under "agent_proof" object):

agent_checksum: REQUIRED. The verified agent checksum (from request.computed_checksum). See Section 5 for detailed agent identity and checksum analysis.

registration_id: REQUIRED. The registration identifier from the agent registry, enabling token revocation at the registration level.

4.4.3. Token Example

Example decoded intent token payload:

```
{
  "iss": "https://idp.example.com",
  "aud": "https://api.github.com",
  "sub": "vulnerability-patcher-v1",
  "exp": 1735690800,
  "iat": 1735690500,
  "jti": "token_a3f7e89c",
  "scope": "repo:write vulnerability:read",
  "cnf": {
    "jwk": {
      "kty": "RSA",
      "use": "sig",
      "alg": "RS256",
      "n": "xGOr-H7A-PWi...",
      "e": "AQAB"
    }
  },
  "intent": {
    "workflow_id": "auto-patch-workflow-v1",
    "workflow_step": "step_3_patch_application",
    "executed_by": "vulnerability-patcher-v1",
    "delegation_chain": "f3e8d9c7b2a41a3c",
    "step_sequence_hash": "a7c9e2f4b8d61c3f"
  },
  "agent_proof": {
    "agent_checksum": "sha256:a3c7f2e8d9b4f1e2c8a7d6f3e9b2c4f1...",
    "registration_id": "reg_vulnerability-patcher-v1_1735680000"
  }
}
```

4.4.4. Sequence Hash Computation

The delegation_chain and step_sequence_hash provide cryptographic integrity over the workflow execution path:

Delegation Chain Hash:

```
# Input: delegation_context.chain + current agent_id
chain = ["supervisor", "planner", "patcher"]
chain_string = "supervisor|planner|patcher"
hash = SHA256(chain_string.encode('utf-8'))
delegation_chain = hash.hexdigest()[:16]
# Result: "f3e8d9c7b2a41a3c"
```

Step Sequence Hash:

```
# Input: delegation_context.completed_steps + current step
steps = ["step_1_analyze", "step_2_plan", "step_3_patch"]
steps_string = "step_1_analyze|step_2_plan|step_3_patch"
hash = SHA256(steps_string.encode('utf-8'))
step_sequence_hash = hash.hexdigest()[:16]
# Result: "a7c9e2f4b8d61c3f"
```

If an agent has skipped steps or altered the delegation chain, the IDP side library will be able to detect during intent token request. However, including these hashes in the intent token additionally enables resource servers to detect if an agent has skipped steps or altered the delegation chain.

4.5. Error Responses

4.5.1. Error Response Format

Error responses follow the OAuth 2.0 error response format defined in Section 5.2 of RFC 6749 [RFC6749]:

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

{
  "error": "agent_checksum_mismatch",
  "error_description": "Agent checksum mismatch - code integrity violation",
  "error_uri": "https://idp.example.com/docs/errors#checksum_mismatch"
}
```

4.5.2. unsupported_grant_type

Error Code: unsupported_grant_type

HTTP Status: 400 Bad Request

Description: The grant_type parameter does not equal "agent_checksum".

Example:

```
{
  "error": "unsupported_grant_type",
  "error_description": "Grant type must be 'agent_checksum'"
}
```

4.5.3. unknown_agent

Error Code: unknown_agent

HTTP Status: 401 Unauthorized

Description: The agent_id is not registered with the authorization server.

Recommended Action: Agent must complete registration (Section 6.1) before requesting tokens.

Example:

```
{
  "error": "unknown_agent",
  "error_description": "Agent ID not found in registry. Register first."
}
```

4.5.4. agent_checksum_mismatch

Error Code: agent_checksum_mismatch

HTTP Status: 401 Unauthorized

Description: The computed_checksum does not match the registered checksum for the agent_id. This indicates that the agent's configuration (prompt, tools, or model parameters) has been modified since registration.

Security Implication: This is a critical security error that may indicate:

* Unauthorized modification of the agent
* Attempted impersonation attack
* Code integrity violation

Recommended Action: Agent must re-register with updated checksum before requesting tokens. Authorization server SHOULD log this event for security monitoring.

Example:

```
{
  "error": "agent_checksum_mismatch",
  "error_description": "Agent checksum mismatch - code integrity violation",
  "error_uri": "https://idp.example.com/docs/errors#checksum_mismatch"
}
```

4.5.5. workflow_step_unauthorized

Error Code: workflow_step_unauthorized

HTTP Status: 403 Forbidden

Description: The agent is not authorized to execute the specified workflow_step. This may occur when:

* Prerequisite steps have not been completed
* Required approval gate has not been passed
* Step is not defined in the workflow
* Agent is not assigned to this workflow step

Recommended Action: Review workflow definition and ensure all required prerequisites are met.

Example:

```
{
  "error": "workflow_step_unauthorized",
  "error_description": "Agent not authorized for workflow step.
                        Required prerequisite steps not completed.",
  "missing_steps": ["step_1_analyze_manifest", "step_2_approval_gate"]
}
```

4.5.6. invalid_request

Error Code: invalid_request

HTTP Status: 400 Bad Request

Description: The request is malformed or missing required parameters. Common causes:

* Missing required fields (agent_id, computed_checksum, etc.)
* Invalid JSON format

* workflow_enabled=true but workflow_id/workflow_step missing

* Invalid checksum format (must be "sha256:<hex>")

Example:

```
{
  "error": "invalid_request",
  "error_description": "Missing required parameter: computed_checksum"
}
```

4.6. Auth Grant Security Considerations

For grant type specific security considerations, see:

* Checksum verification: Section 9.1.1

* Token lifetime: Section Section 9.2.1

* Workflow validation: Section 9.3

For comprehensive security analysis, see Section 9.

4.7. Implementation Notes

4.7.1. Agent Registry

The authorization server maintains an agent registry mapping agent_id to registration records. Each record contains:

* agent_id - Unique identifier

* checksum - SHA-256 agent checksum

* registration_id - Unique registration instance ID

* public_key - Agent’s public key (for cnf claim)

* registered_at - Registration timestamp

* version - Registration version (for checksum updates)

Multiple registrations may exist for the same agent_id when the agent’s configuration changes. The authorization server uses the LATEST registration for verification.

4.7.2. Workflow Registry

The workflow registry stores workflow definitions including:

* workflow_id - Unique workflow identifier
* steps - Ordered array of workflow step definitions

Each step contains:

* step_id - Unique step identifier
* required - Boolean indicating if step is required
* requires_approval - Boolean for approval requirement
* approval_gate - Boolean marking this step as approval gate
* agent_id - Agent authorized to execute this step (optional)

4.7.3. Performance Considerations

The checksum verification and workflow validation add overhead to token issuance. Implementations SHOULD:

* Cache agent registry lookups to reduce database queries
* Use indexed database queries on agent_id for fast lookup
* Pre-compute workflow step dependencies to speed validation
* Implement connection pooling for registry data stores

The reference implementation shows ~4.3% overhead compared to standard OAuth 2.0 token issuance, which is acceptable for most production deployments.

5. Agent Identity

5.1. Overview

Agent identity in this specification is based on cryptographic checksums computed over an agent's configuration. Unlike traditional client credentials that can be shared or stolen, the agent checksum provides a unique cryptographic fingerprint of the agent's actual implementation.

The agent checksum serves three critical functions:

* *Identity Verification:* Proves that an agent's configuration matches what was registered with the authorization server

* *Code Integrity:* Detects unauthorized modifications to the agent's system prompt, tools, or configuration

* *Non-Transferability:* Prevents credentials from being reused by different agent implementations

This section defines how agent identity is established, verified, and maintained throughout the agent lifecycle.

5.2. Agent Configuration Structure

5.2.1. Agent Specification

An agent's identity is derived from its specification, which includes all components that define the agent's behavior:

agent_id:
: REQUIRED. Unique identifier for the agent. MUST be unique within the application scope. Format: alphanumeric with hyphens, e.g., "vulnerability-patcher-v1".

prompt:
: REQUIRED. The complete system prompt that defines the agent's role, capabilities, and constraints. This is the full text provided to the language model that shapes agent behavior.

tools:
: REQUIRED. Array of tool definitions available to the agent. Each tool includes name, description, and parameter schema. The order of tools MUST be deterministic.

configuration:
: OPTIONAL. Additional configuration parameters including:

  * model_name - Language model identifier

  * temperature - Sampling temperature (0.0-1.0)

  * max_tokens - Maximum generation length

  * top_p - Nucleus sampling parameter

5.2.2. Example Agent Specification

Example agent specification for a vulnerability patching agent:

```
{
  "agent_id": "vulnerability-patcher-v1",
  "prompt": "You are a security agent responsible for patching
            vulnerabilities in software dependencies. Analyze
            manifests, identify vulnerable packages, and generate
            patches. Always verify changes before committing.",
  "tools": [
    {
      "name": "read_manifest",
      "description": "Read package manifest file",
      "parameters": {
        "type": "object",
        "properties": {
          "file_path": {"type": "string"}
        },
        "required": ["file_path"]
      }
    },
    {
      "name": "check_vulnerability",
      "description": "Check package for known vulnerabilities",
      "parameters": {
        "type": "object",
        "properties": {
          "package": {"type": "string"},
          "version": {"type": "string"}
        },
        "required": ["package", "version"]
      }
    },
    {
      "name": "create_patch",
      "description": "Create patch for vulnerable package",
      "parameters": {
        "type": "object",
        "properties": {
          "package": {"type": "string"},
          "current_version": {"type": "string"},
          "target_version": {"type": "string"}
        },
        "required": ["package", "current_version", "target_version"]
      }
    }
  ],
```

```
        "configuration": {
          "model_name": "claude-3-5-sonnet-20241022",
          "temperature": 0.0,
          "max_tokens": 4096
        }
      }
```

5.2.3. Tool Checksum Levels

Tools may be included in the checksum computation at two levels:

Shallow (Default): Only the tool signature (name, description, parameters) is included in the checksum. The actual function implementation is not hashed. This allows tool implementation to evolve without requiring re-registration.

Deep: The complete tool implementation, including normalized function source code, is included in the checksum. This provides stronger integrity guarantees but requires re-registration for any code changes.

Implementations MUST support shallow checksums. Deep checksums are OPTIONAL and MAY be indicated through tool metadata.

In the reference implementation, tools are marked for deep checksumming using a decorator:

```
@secure_tool(name="critical_operation", checksum_level=ChecksumLevel.deep)
def critical_operation(params):
    # Tool implementation included in checksum
    pass
```

When including source code in deep checksums, implementations MUST normalize the source code to ensure formatting changes do not affect checksums while logic changes are always detected.

5.2.4. Tool Signature Normalization

Tool function signatures MUST be normalized to remove framework-specific wrapper parameters that do not affect agent behavior. This ensures that checksums remain stable across different agent frameworks.

Wrapper Parameters to Remove:

* config - Framework configuration object

* callbacks - Callback handlers

* run_manager - Execution manager
* tags - Execution tags
* metadata - Execution metadata
* run_id - Execution identifier
* **kwargs - Catchall for framework parameters

Reference implementation for signature normalization:

```
import inspect

def get_core_signature(func) -> str:
    """
    Get function signature with wrapper parameters removed.
    """
    # Unwrap decorators to get original function
    current = func
    try:
        current = inspect.unwrap(func)
    except (ValueError, AttributeError):
        pass

    sig = inspect.signature(current)

    # Known wrapper parameters to filter out
    WRAPPER_PARAMS = {
        'config', 'callbacks', 'run_manager', 'tags', 'metadata',
        'run_id', 'parent_run_id', 'configurable', 'recursion_limit'
    }

    # Build cleaned parameter list
    cleaned_params = []

    for name, param in sig.parameters.items():
        # Skip known wrapper params
        if name in WRAPPER_PARAMS:
            continue

        # Skip VAR_KEYWORD (**kwargs) if it looks like wrapper catchall
        if param.kind == inspect.Parameter.VAR_KEYWORD:
            continue

        # Skip VAR_POSITIONAL (*args) used by wrappers
        if param.kind == inspect.Parameter.VAR_POSITIONAL and name == 'args':
            continue

        cleaned_params.append(param)

    # Create new signature
    new_sig = inspect.Signature(
        cleaned_params,
        return_annotation=sig.return_annotation
    )

    return str(new_sig)
```

This normalization is CRITICAL for cross-framework compatibility. Without it, the same logical tool would have different checksums when used in LangChain [LANGCHAIN] vs CrewAI [CREWAI] vs other frameworks. [LANGGRAPH]

5.2.5. Source Code Normalization

For deep checksum tools that include source code, implementations MUST normalize the source code using Abstract Syntax Tree (AST) parsing. This ensures that formatting changes do not affect checksums while any logic changes are always detected.

Normalization process:

1. Parse source code to AST (Abstract Syntax Tree)

2. Remove docstrings (captured separately in tool description)

3. Remove comments (not part of execution logic)

4. Unparse AST to canonical form

Reference implementation:

```
    import ast
    import textwrap
    import inspect

    def remove_docstrings(tree: ast.AST) -> None:
        """
        Remove docstring nodes from AST in-place.

        Docstrings are captured separately in tool description,
        so we remove them from implementation to avoid duplication.
        """
        for node in ast.walk(tree):
            if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef,
                                 ast.ClassDef, ast.Module)):
                if (node.body and
                    isinstance(node.body[0], ast.Expr) and
                    isinstance(node.body[0].value, ast.Constant) and
                    isinstance(node.body[0].value.value, str)):
                    # This is a docstring - remove it
                    node.body = node.body[1:] if len(node.body) > 1 else [ast.Pass()]

    def normalize_source(source_code: str) -> str:
        """
        Normalize Python source code using AST.
```

```
            This ensures that formatting changes don't affect checksums
            while logic changes are always detected.

            Strips:
                - Comments (not part of execution logic)
                - Formatting differences (whitespace, indentation)
                - Docstrings (captured separately in checksum)

            Preserves:
                - All execution logic
                - Variable names (semantically significant)
                - Control flow structures
                - Function calls and their arguments
            """
            # Remove leading indentation
            source = textwrap.dedent(source_code)

            try:
                # Parse to AST
                tree = ast.parse(source)

                # Remove docstrings
                remove_docstrings(tree)

                # Unparse to canonical form
                normalized = ast.unparse(tree)

                return normalized

            except SyntaxError:
                # If code has syntax errors, return as-is
                # This will cause checksum mismatch (correct behavior)
                return source

        def get_tool_source_code(func: callable) -> str | None:
            """
            Get normalized source code for a tool function.
            Returns None if source cannot be retrieved.
            """
            try:
                source = inspect.getsource(func)
                normalized = normalize_source(source)
                return normalized
            except (OSError, TypeError):
                return None
```

This normalization means:

* *SAME checksum:* Reformatting code, adding comments, changing indentation

* *DIFFERENT checksum:* Changing logic, adding/removing statements, modifying control flow

Example demonstrating normalization:

```
# These produce IDENTICAL checksums after normalization:

def example1(x):
    # This is a comment
    return x + 1

def example1(x): return x+1  # inline

def example1(x):
    """Docstring here"""
    return x + 1


# This produces DIFFERENT checksum:

def example1(x):
    return x + 2  # Logic changed
```

5.3. Checksum Computation

5.3.1. Computation Algorithm

The agent checksum MUST be computed as follows:

1. *Construct Agent Components Object:* Create a JSON object containing the agent’s configuration:

```
{
  "agent_id": "string",
  "prompt_template": "string",
  "tools": [
    {
      "name": "string",
      "description": "string",
      "parameters": { /* JSON schema */ }
    }
  ],
  "configuration": {
    "model_name": "string",
    "temperature": number,
    "max_tokens": number
  }
}
```

2. *Canonicalize JSON:* Serialize the object using RFC 8785 [RFC8785] JSON Canonicalization Scheme (JCS). This ensures deterministic ordering of keys and consistent formatting.

   Canonical serialization is REQUIRED because:

   * JSON objects have no inherent key ordering

   * Whitespace and formatting vary across implementations

   * Floating point precision may differ

   * Unicode escaping can be inconsistent

3. *Compute SHA-256 Hash:* Hash the canonical JSON bytes using SHA-256 as defined in FIPS 180-4 [FIPS-180-4].

4. *Format Checksum:* Encode the hash as lowercase hexadecimal (64 characters):

   ```
   checksum = hex(hash_bytes).lowercase()
   # Result: "a3c7f2e8d9b4f1e2c8a7d6f3e9b2c4f1a8e7d3c2b5f4e9..."
   ```

   Note: The reference implementation uses bare hex encoding. Implementations MAY prefix with "sha256:" for explicit algorithm identification.

5.3.2. Implementation Guidance

Implementations MUST ensure checksum computation is deterministic across different platforms, languages, and runtime environments.

Reference Implementation (Python):

```
import hashlib
import json

def normalize_prompt(prompt: str) -> str:
    """
    Normalize prompt string for consistent checksum computation.
    Apply this in BOTH client and server-side computation.
    """
    if not prompt:
        return ""

    # 1. Strip leading/trailing whitespace
    normalized = prompt.strip()

    # 2. Normalize line endings (Windows \r\n vs Unix \n)
    normalized = normalized.replace('\r\n', '\n')

    # 3. Collapse multiple consecutive newlines into single newline
    import re
    normalized = re.sub(r'\n\s*\n', '\n', normalized)

    # 4. Strip whitespace from each line
    lines = [line.strip() for line in normalized.split('\n')]
    normalized = '\n'.join(lines)

    # 5. Remove empty lines
    lines = [line for line in lines if line]
    normalized = '\n'.join(lines)

    return normalized

def compute_agent_checksum(agent_components) -> str:
    """
    Compute deterministic SHA-256 checksum for agent configuration.

    Args:
        agent_components: Object with fields:
            - agent_id: str
            - prompt_template: str
            - tools: list of Tool objects
            - configuration: dict

    Returns:
        64-character lowercase hex string (SHA-256 hash)
    """
    # Construct canonical components object
```

```
            components = {
                "id": agent_components.agent_id,
                "prompt": normalize_prompt(agent_components.prompt_template),
                "tools": sorted([
                    {
                        "name": tool.name,
                        "signature": tool.signature,
                        "description": tool.description
                        # "source_code" included only for deep checksum tools
                    }
                    for tool in agent_components.tools
                ], key=lambda x: x["name"]),  # Sort tools by name
                "config": agent_components.configuration
            }

            # Serialize with sorted keys for determinism
            content = json.dumps(components, sort_keys=True)

            # Compute SHA-256 hash
            return hashlib.sha256(content.encode('utf-8')).hexdigest()
```

Critical Requirements:

* Use json.dumps() with sort_keys=True for deterministic JSON serialization

* Normalize prompts using the exact normalization function above (whitespace, line endings, empty lines)

* Sort tools by name before serialization

* Encode JSON string as UTF-8 before hashing

* Output lowercase hexadecimal (64 characters)

* Tool signatures MUST use get_core_signature() to remove wrapper parameters (see Section 4.2.4)

Common Implementation Pitfalls:

* Not normalizing prompts identically on client and server (most common cause of checksum mismatch)

* Including wrapper function parameters in tool signatures (config, callbacks, run_manager, etc.)

* Platform-specific line endings (\r\n vs \n) in prompts

* Non-deterministic tool ordering (MUST sort by name)

* Using standard JSON.stringify() without sort_keys

* Uppercase hexadecimal encoding (MUST be lowercase)

5.4. Agent Registration

5.4.1. Registration Overview

Before an agent can request tokens, it MUST register its checksum with the authorization server. Registration creates a binding between the agent_id and its cryptographic identity.

The registration process:

1. Agent computes its checksum client-side

2. Agent sends registration request to IDP

3. IDP validates agent configuration

4. IDP recomputes checksum for verification

5. IDP stores agent_id → checksum mapping

6. IDP returns registration confirmation

5.4.2. IDP Processing

The authorization server MUST perform the following steps:

1. *Validate Request Structure:* Verify all required fields are present and properly formatted

2. *Compute Checksum:* Recompute agent checksum from agent_components using the algorithm defined in Section 4.2.1

3. *Check for Duplicates:* Verify that the computed checksum does not already exist in the registry. If it does, reject with error "duplicate_agent". This prevents agent impersonation.

4. *Generate Registration ID:* Create unique registration identifier:

   registration_id = "reg_" + agent_id + "_" + timestamp

5. *Store Registration:* Persist mapping:

```
          {
            "agent_id": "vulnerability-patcher-v1",
            "registration_id": "reg_vulnerability-patcher-v1_1735680000",
            "checksum": "a3c7f2e8d9b4f1e2c8a7d6f3e9b2c4f1a8e7d3c2b5f4e9a7c3d8f2b6e1a9c4f7
",
            "prompt": "You are a security agent...",
            "tools": [ /* tool definitions */ ],
            "public_key": "-----BEGIN PUBLIC KEY-------...",
            "registered_at": 1735680000000,
            "app_id": "vulnerability-patcher-app",
            "version": 1
          }
```

6. *Return Confirmation:* Return registration details to client

5.4.3. Registration Response

Success response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "agent_id": "vulnerability-patcher-v1",
  "registration_id": "reg_vulnerability-patcher-v1_1735680000",
  "checksum": "a3c7f2e8d9b4f1e2c8a7d6f3e9b2c4f1a8e7d3c2b5f4e9a7c3d8f2b6e1a9c4f7"
}
```

Error response (duplicate checksum):

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": "duplicate_agent",
  "error_description": "Agent with identical checksum already exists",
  "existing_agent_id": "other-agent-v1"
}
```

5.4.4. Agent Updates

When an agent's configuration changes (e.g., tool updates, prompt modifications), a new checksum is computed and the agent MUST re-register.

NOTE: This registration process can be automated by introdcuing additional steps into CI / CD or by Model Context Protocol (MCP, [MCP]) integration to the IDP server, or by any other agent resource management process. Regardless of the methodology used the underlying conceptual process remains the same.

Update procedure:

1. Compute new checksum from updated configuration

2. Submit new registration request (same agent_id)

3. IDP validates new checksum is different from previous

4. IDP creates new registration record with incremented version

5. IDP MAY maintain previous registrations for audit trail

6. IDP uses LATEST registration for token validation

Example: Agent version progression

```
// Version 1 (original)
{
  "agent_id": "patcher-v1",
  "checksum": "aaa1234567890abcdef1234567890abcdef1234567890abcdef1234567890abc",
  "version": 1,
  "registered_at": 1735680000000
}

// Version 2 (tool updated)
{
  "agent_id": "patcher-v1",
  "checksum": "bbb9876543210fedcba9876543210fedcba9876543210fedcba9876543210fed",
  "version": 2,
  "registered_at": 1735690000000
}
```

The IDP MUST use the latest version (highest registered_at) for checksum verification during token requests.

5.5. Checksum Verification

5.5.1. Client-Side Verification

Before requesting a token, the agent MUST compute its current checksum and include it in the token request. This proves that the requesting agent’s configuration matches the registered identity.

Client-side process:

```
# 1. Detect current agent context
agent_spec = get_current_agent_spec()

# 2. Compute checksum from current configuration
current_checksum = compute_agent_checksum(agent_spec)

# 3. Include in token request
token_request = {
    "grant_type": "agent_checksum",
    "agent_id": agent_spec.agent_id,
    "computed_checksum": current_checksum,
    ...
}
```

5.5.2. Server-Side Verification

Upon receiving a token request, the authorization server MUST verify the agent checksum:

```
# From intent.py implementation

# 1. Retrieve registered checksum
registered_checksums = registry.get_agent(agent_id)
stored_checksum = registered_checksums[-1].checksum  # Latest version

# 2. Compare with submitted checksum (constant-time)
if computed_checksum != stored_checksum:
    raise AgentChecksumMismatch(
        "Agent checksum mismatch - code integrity violation"
    )
```

The comparison MUST use constant-time string comparison to prevent timing attacks that could reveal checksum information.

NOTE: The authorization server retrieves the LATEST registration for the agent_id, as agents may re-register with updated checksums when their configuration changes.

5.6. Identity Lifecycle

5.6.1. Lifecycle Stages

Agent identity progresses through several stages:

1. *UNREGISTERED:* Agent exists but has not registered with IDP. Cannot request tokens.

2. *REGISTERED:* Agent checksum stored in IDP registry. Can request tokens if checksum matches.

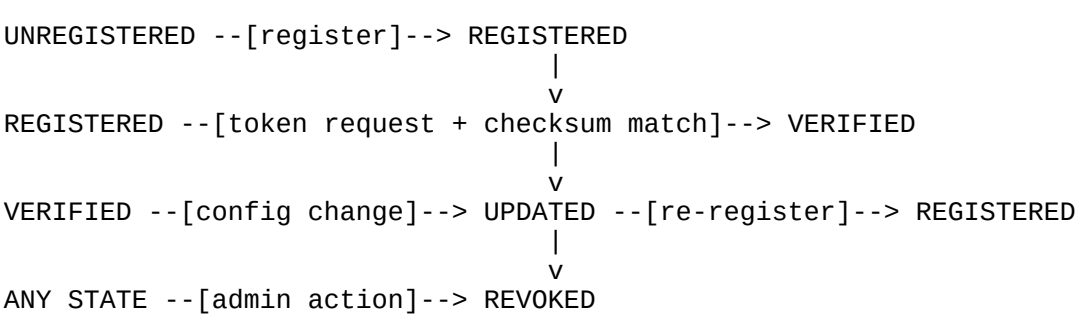
3. *VERIFIED:* Agent successfully authenticated and obtained token. Identity confirmed through checksum match.

4. *UPDATED:* Agent configuration changed, new checksum computed, re-registration required. Previous tokens may be invalidated.

5. *REVOKED:* Agent identity revoked by administrator. All tokens invalidated. Cannot request new tokens.

5.6.2. State Transitions

Valid state transitions:



5.7. Identity Security Considerations

Agent identity security considerations:

* Checksum strength: See Section 9.1.2

* Registration security: See Section 9.1.3

* Agent Registry Security: See Section 9.1.4

For comprehensive security analysis, see Section 9.

6. Protocol Flows

6.1. Overview

This section describes the detailed protocol flows for agent registration, token issuance, and authorization. Each flow includes step-by-step procedures, HTTP request/response examples, and validation requirements.

The flows in this section build upon the abstract protocol flow described in Section 3.2, providing complete implementation guidance for:

* Agent registration and checksum verification
* Workflow registration and definition
* Intent token issuance with user approval
* Access token requests using agent_checksum grant
* API authorization and token validation
* Multi-agent delegation chains

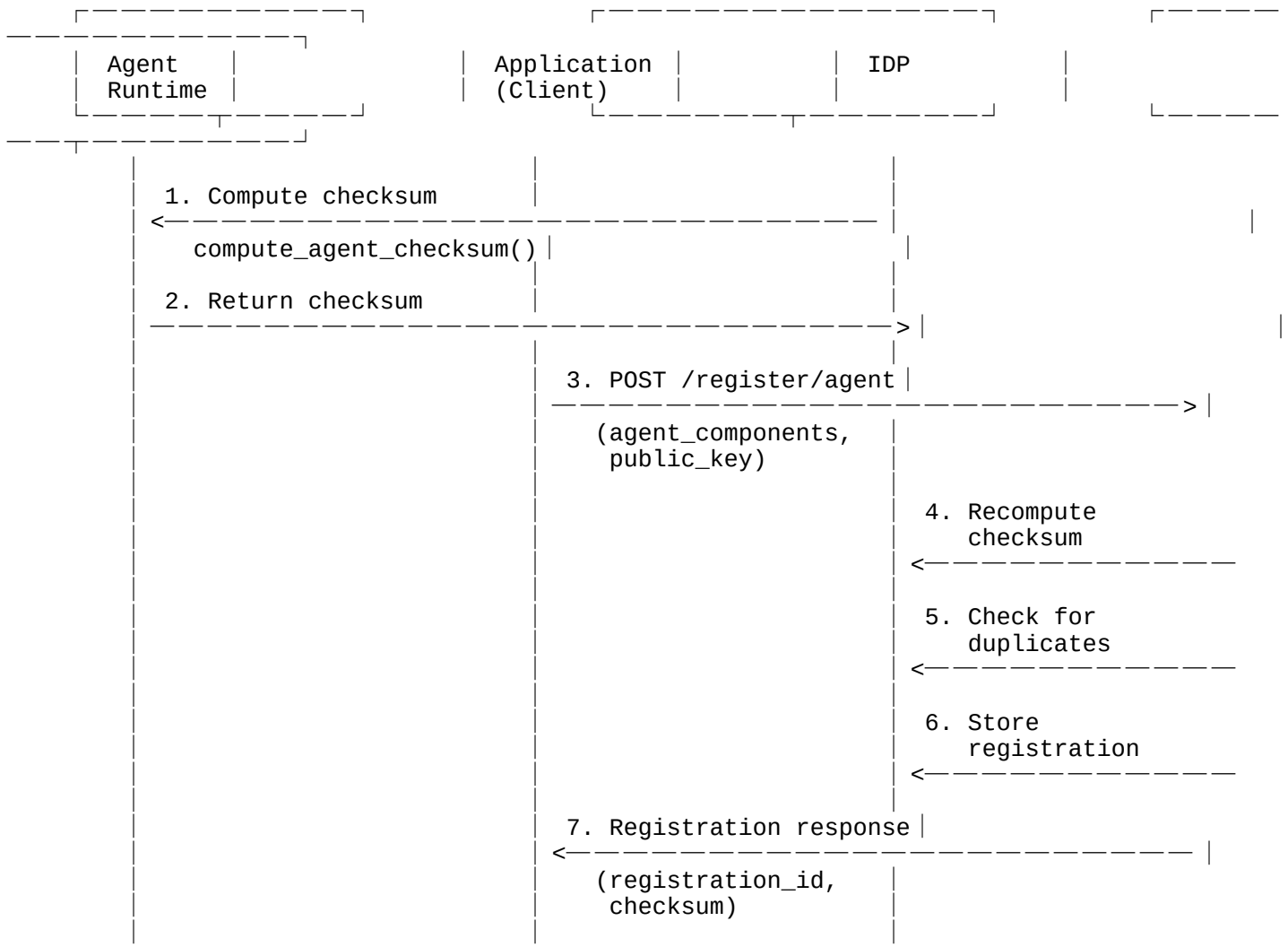
6.2. Agent Registration Flow

6.2.1. Overview

Before an agent can request tokens, it MUST register its identity with the authorization server. This flow establishes the cryptographic binding between the agent_id and its configuration checksum.

6.2.2. Sequence Diagram

Agent registration sequence:



6.2.3. Step-by-Step Procedure

1. *Agent Checksum Computation (Client-Side):* The agent runtime computes its configuration checksum using the algorithm defined in Section 5.3. This includes the agent's prompt, tools, and configuration parameters.

2. *Public Key Generation (Optional):* If proof-of-possession [RFC7800] is required, the agent generates an RSA key pair and prepares the public key in PEM format.

3. *Registration Request:* The application sends a registration request to the IDP's agent registration endpoint, including the complete agent configuration and optional public key.

4. *Server-Side Checksum Verification:* The IDP recomputes the agent checksum from the submitted configuration and verifies it matches the claimed checksum.

5. *Duplicate Detection:* The IDP checks if the computed checksum already exists in the registry. If found, the request is rejected with error "duplicate_agent" to prevent agent impersonation.

6. *Registration Storage:* The IDP creates a unique registration_id, stores the mapping (agent_id → checksum → registration_id), and persists the registration record.

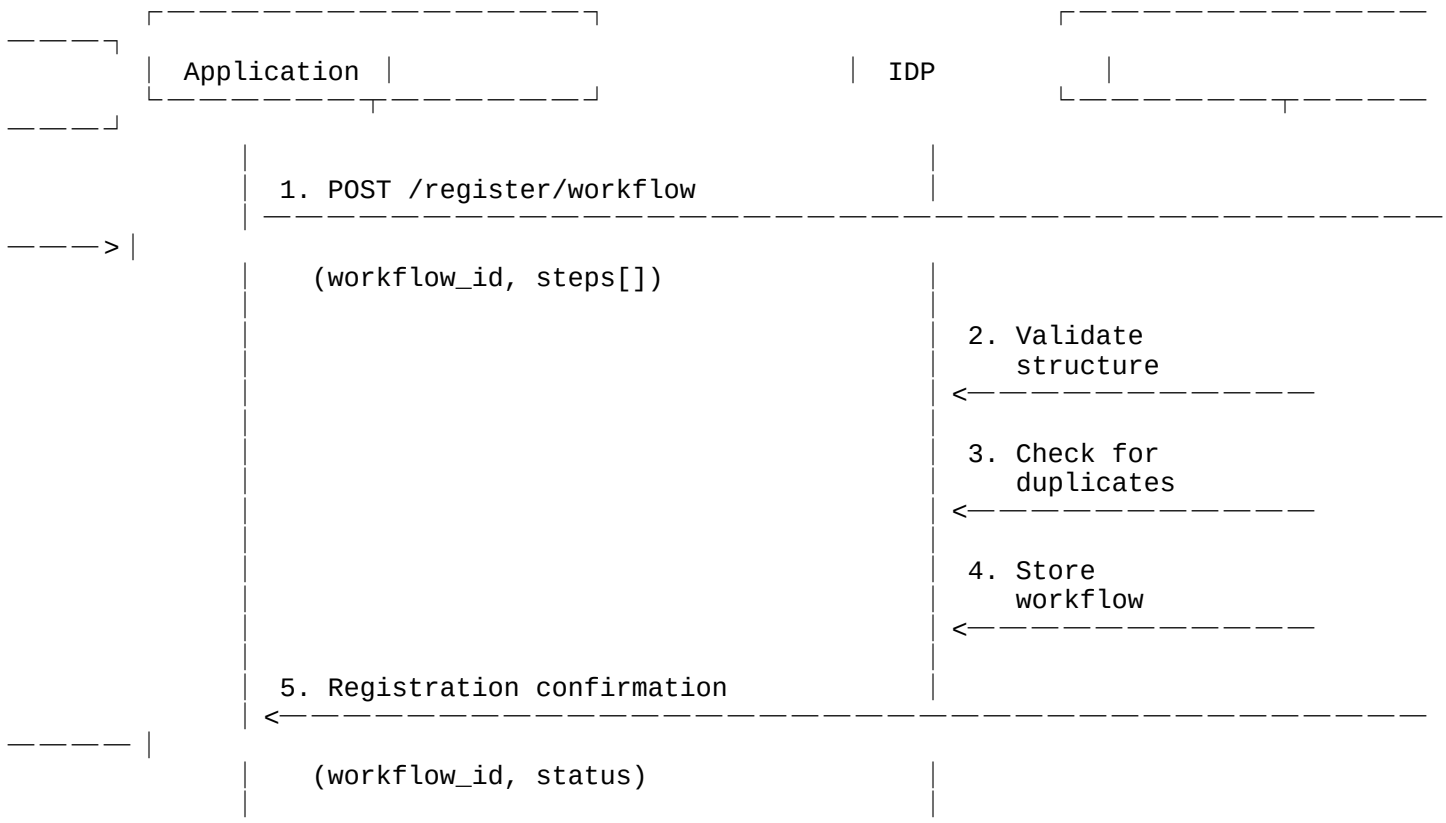
7. *Response:* The IDP returns the registration_id and computed checksum to the client for verification.

6.3. Workflow Registration Flow

6.3.1. Overview

Workflows define the sequence of steps that agents execute and the authorization requirements for each step. Workflow registration establishes the workflow definition with the IDP before execution.

6.3.2. Sequence Diagram



6.3.3. HTTP Example

Workflow registration request:

```
POST /intent/register/workflow HTTP/1.1
Host: idp.example.com
Content-Type: application/json
Authorization: Bearer <admin_token>

{
  "workflow_id": "auto-patch-workflow-v1",
  "steps": {
    "step_1_analyze_manifest": {
      "required": true,
      "requires_approval": false,
      "agent_id": "vulnerability-analyzer"
    },
    "step_2_create_patch_plan": {
      "required": true,
      "requires_approval": false,
      "agent_id": "patch-planner"
    },
    "step_3_approval_gate": {
      "required": true,
      "requires_approval": false,
      "approval_gate": true
    },
    "step_4_apply_patch": {
      "required": true,
      "requires_approval": true,
      "agent_id": "vulnerability-patcher-v1"
    },
    "step_5_verify_patch": {
      "required": true,
      "requires_approval": false,
      "agent_id": "patch-verifier"
    }
  }
}
```

Success response:

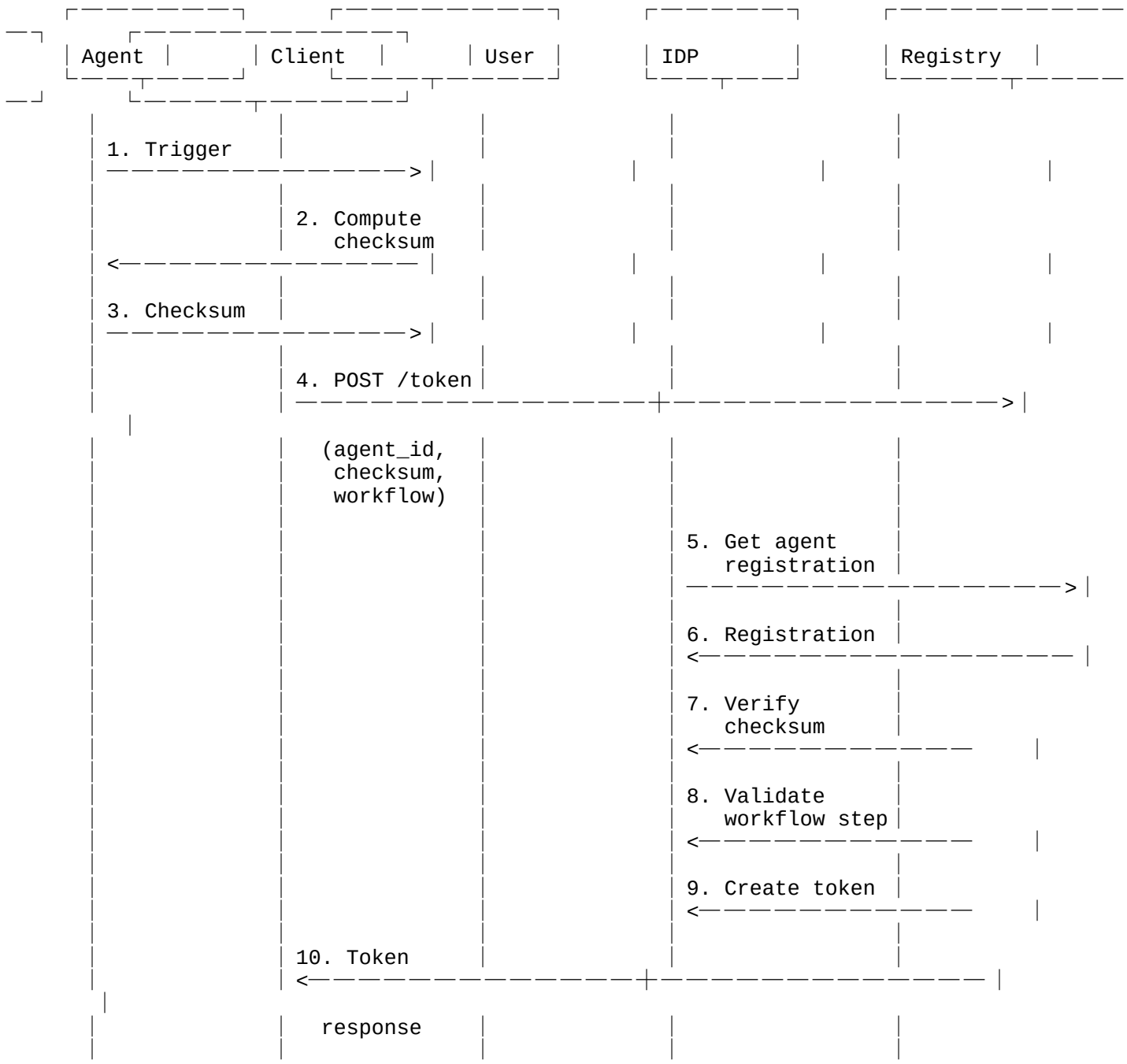
```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "registered",
  "workflow_id": "auto-patch-workflow-v1"
}
```

6.4. Token Request Flow

6.4.1. Overview

This flow describes how an agent requests an intent token using the agent_checksum grant type. The flow includes agent identity verification, workflow step authorization, and token issuance.

6.4.2. Sequence Diagram



6.4.3. Step-by-Step Procedure

1. *Execution Trigger:* Agent execution is triggered by application logic or user request.

2. *Checksum Computation:* Client computes current agent checksum using algorithm from Section 5.3.

3. *Checksum Return:* Agent runtime returns computed checksum to client library.

4. *Token Request:* Client sends token request to IDP with agent_checksum grant type, including agent_id, computed checksum, workflow details, and requested scopes.

5. *Registry Lookup:* IDP retrieves agent registration from registry based on agent_id.

6. *Registration Return:* Registry returns stored registration including registered checksum.

7. *Checksum Verification:* IDP compares computed_checksum (from request) with stored checksum. If mismatch, return error "agent_checksum_mismatch".

8. *Workflow Step Validation:* If workflow_enabled=true, IDP validates:

   * Workflow exists in registry

   * Step exists in workflow definition

   * Required prerequisite steps are completed

   * Approval requirements are satisfied

   If validation fails, return error "workflow_step_unauthorized".

9. *Token Creation:* IDP creates intent token with:

   * Standard JWT claims (iss, aud, sub, exp, iat, jti, scope)

   * cnf claim with agent’s public key

   * intent object with workflow details

   * agent_proof object with checksum and registration_id

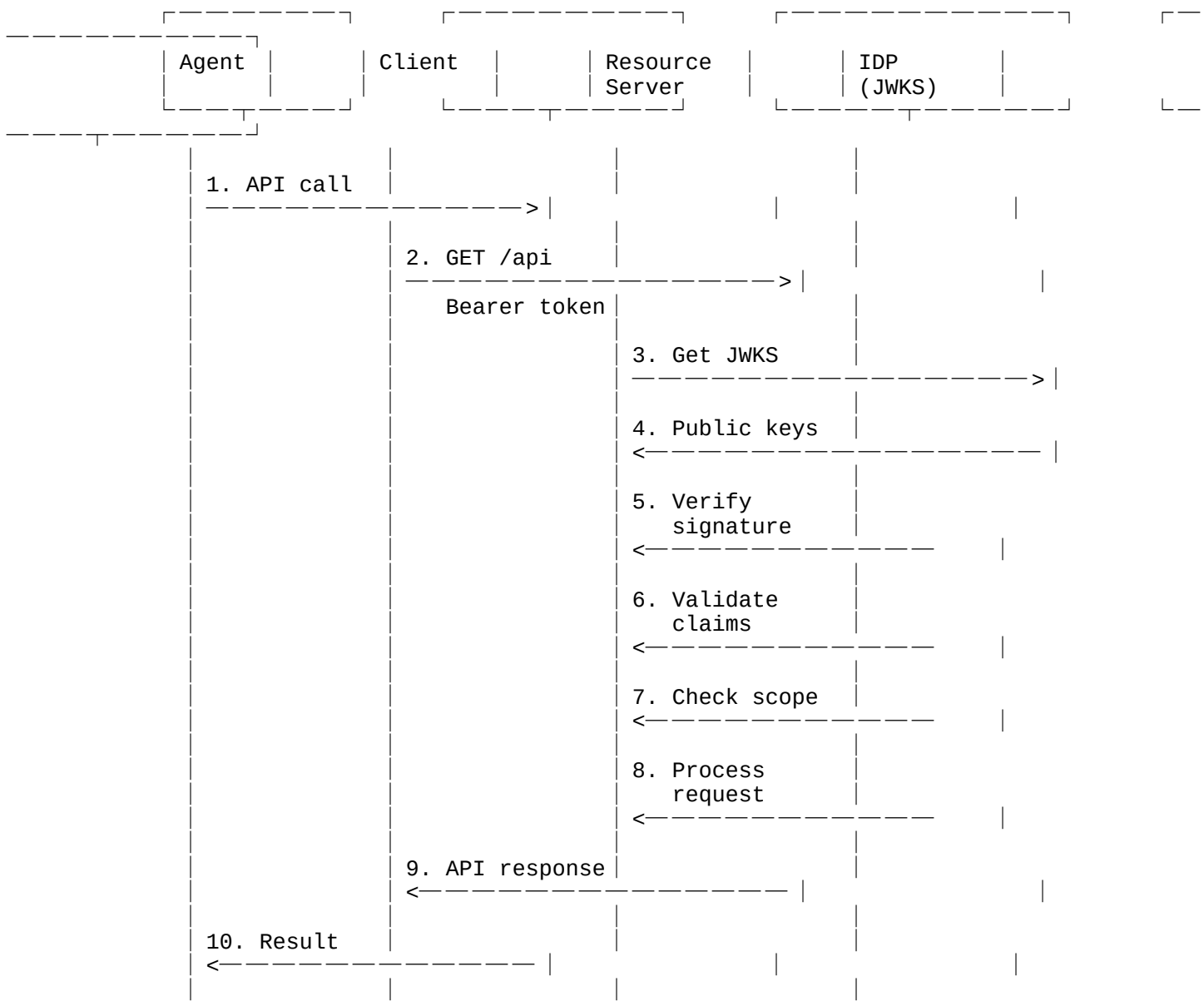
10. *Token Response:* IDP returns token response with access_token, token_type, expires_in, and granted scopes.

6.5. API Authorization Flow

6.5.1. Overview

This flow describes how a resource server validates an intent token and authorizes an API request from an agent. The validation includes token signature verification, checksum validation, workflow binding checks, and scope verification.

6.5.2. Sequence Diagram



6.5.3. Validation Steps

1. *Extract Token:* Resource server extracts Bearer token from Authorization header.
2. *Retrieve JWKS:* Resource server fetches IDP’s JSON Web Key Set for signature verification.
3. *Verify Signature:* Resource server verifies JWT signature using IDP’s public key identified by "kid" header.
4. *Validate Standard Claims:*
    * Check token not expired (exp > current time)
    * Validate issuer (iss matches expected IDP)
    * Validate audience (aud matches resource server)
    * Check token issued time (iat not in future)
5. *Validate Agent Proof:*
    * Verify agent_proof.agent_checksum is present
    * Verify agent_proof.registration_id is present
    * Optionally verify checksum against known agents
6. *Validate Workflow Binding (if applicable):*
    * Check intent.workflow_id matches expected workflow
    * Verify intent.workflow_step is authorized for this API
    * Validate delegation_chain if delegation is restricted
7. *Verify Scopes:* Check that requested operation is permitted by token scopes.
8. *Proof-of-Possession (Optional):* If required, verify PoP signature using public key in cnf claim.
9. *Process Request:* If all validations pass, execute the API operation.

6.6. Multi-Agent Delegation Flow

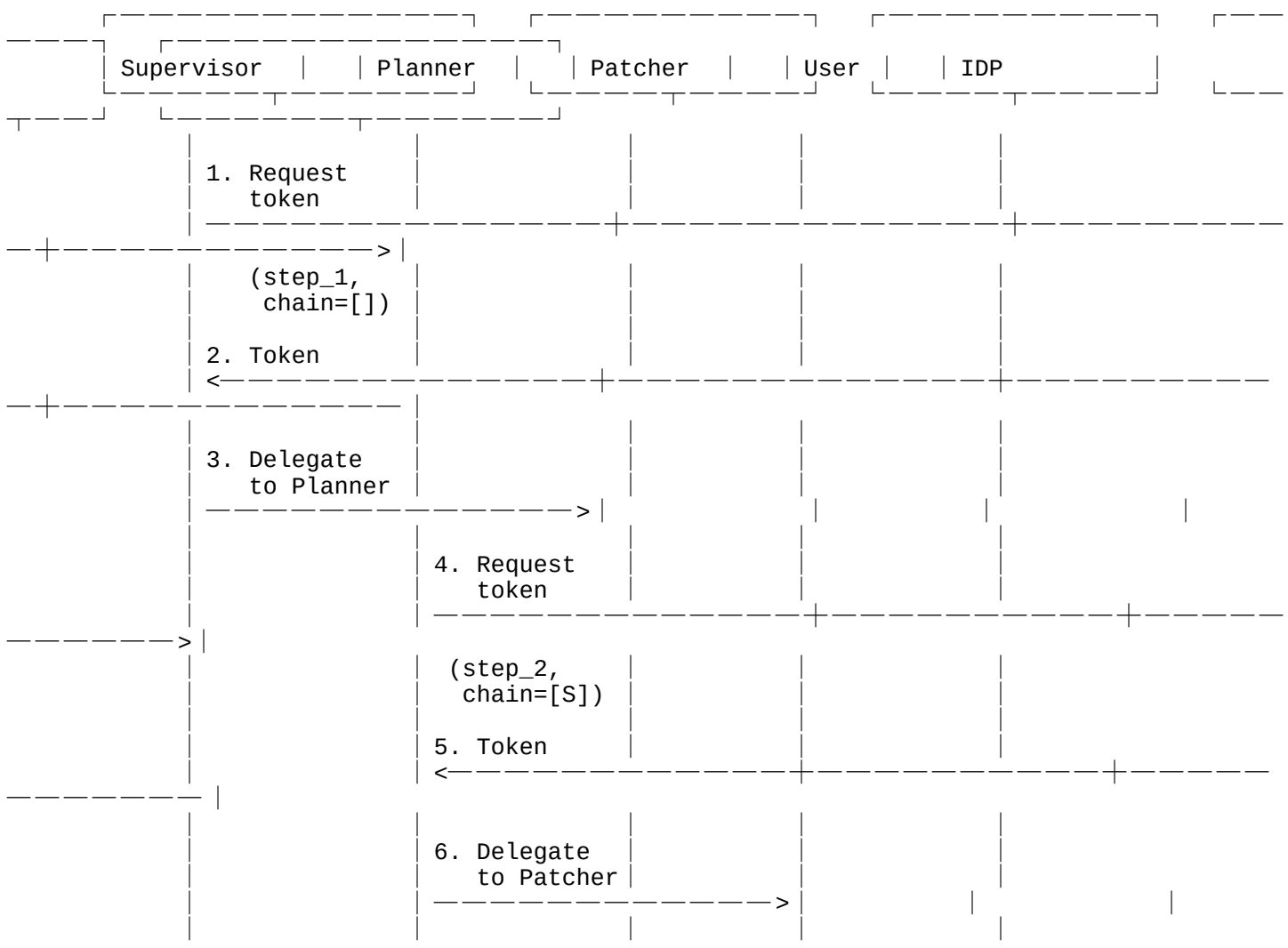
6.6.1. Overview

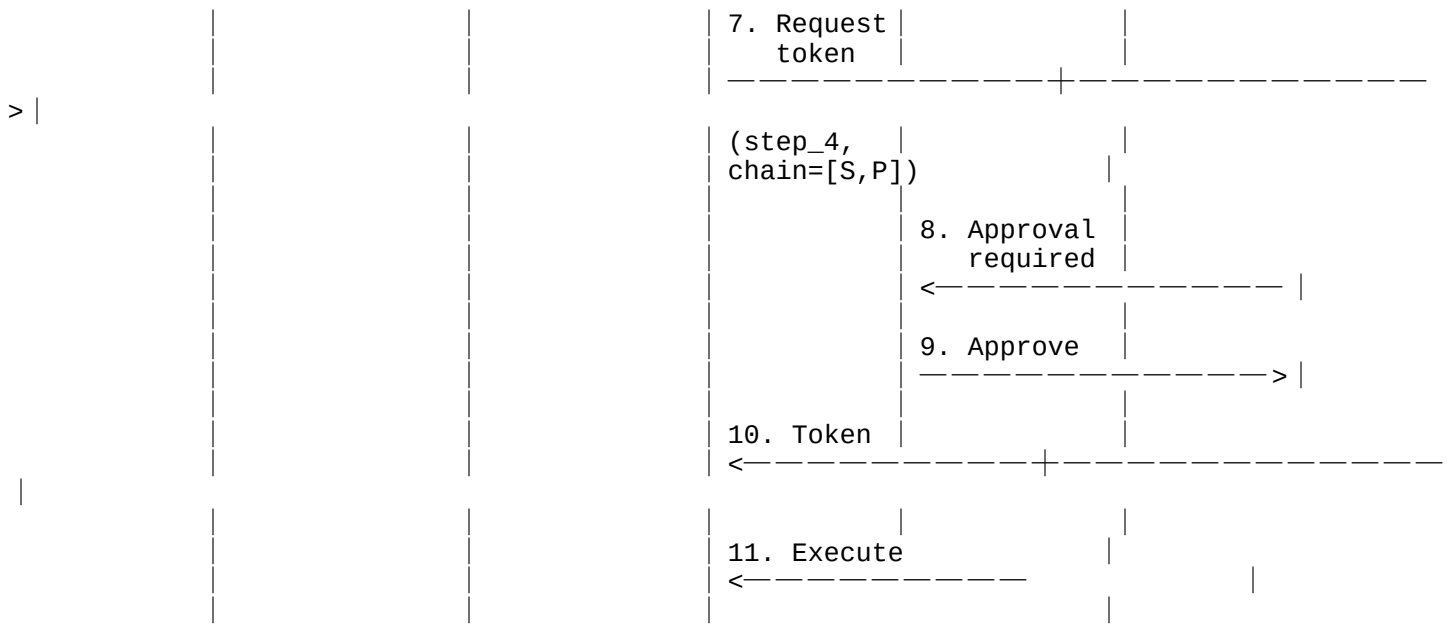
Multi-agent workflows involve delegation chains where a supervisor agent delegates tasks to subordinate agents. Each agent in the chain requests its own token, building upon the delegation context from parent agents.

This flow illustrates a three-agent delegation chain for automated vulnerability patching:

* Supervisor Agent - Orchestrates the workflow
* Planner Agent - Creates patch strategy
* Patcher Agent - Applies the actual patch

6.6.2. Sequence Diagram





Legend: S = Supervisor, P = Planner

6.6.3. Delegation Steps

1. *Supervisor Token Request:* Supervisor agent requests token for step_1 with empty delegation chain.

```
{
  "agent_id": "supervisor-agent",
  "workflow_step": "step_1_analyze_manifest",
  "delegation_context": {
    "chain": [],
    "completed_steps": []
  }
}
```

2. *Supervisor Executes Step 1:* Supervisor analyzes the manifest and determines patching is needed.

3. *Delegation to Planner:* Supervisor delegates to Planner agent, passing delegation context.

4. *Planner Token Request:* Planner requests token for step_2, including Supervisor in chain.

```
{
  "agent_id": "patch-planner",
  "workflow_step": "step_2_create_patch_plan",
  "delegation_context": {
    "chain": ["supervisor-agent"],
    "completed_steps": ["step_1_analyze_manifest"]
  }
}
```

5. *Planner Executes Step 2:* Planner creates patch strategy and identifies target packages.

6. *Delegation to Patcher:* Planner delegates to Patcher agent, passing updated delegation context.

7. *Patcher Token Request:* Patcher requests token for step_4, including full delegation chain.

```
{
  "agent_id": "vulnerability-patcher-v1",
  "workflow_step": "step_4_apply_patch",
  "delegation_context": {
    "chain": ["supervisor-agent", "patch-planner"],
    "completed_steps": [
      "step_1_analyze_manifest",
      "step_2_create_patch_plan",
      "step_3_approval_gate"
    ]
  }
}
```

8. *Approval Check:* IDP detects step_4 requires approval (step_3 is approval gate). Step_3 must be in completed_steps or request is rejected.

9. *User Approval:* User has already approved at step_3 approval gate.

10. *Token Issuance:* IDP issues token to Patcher with:

    * delegation_chain: SHA-256("supervisor-agent|patch-planner|vulnerability-patcher-v1")

    * step_sequence_hash: SHA-256("step_1|step_2|step_3|step_4")

11. *Patcher Execution:* Patcher applies the security patch using the intent token.

6.6.4. Delegation Validation

At each delegation step, the IDP validates:

1. *Chain Continuity:* Current agent is appended to delegation_context.chain, not inserted or replaced.

2. *Step Sequence:* completed_steps contains all required prerequisite steps.

3. *Approval Requirements:* If step requires approval, the most recent approval gate must be in completed_steps.

4. *Agent Authorization:* Current agent is authorized to execute the requested workflow step.

Resource servers MAY additionally validate:

* Delegation depth is within acceptable limits

* All agents in delegation chain are registered and not revoked

* Delegation chain matches expected workflow pattern

7. Future Work

Future versions or extensions of this specification may align with the Grant Negotiation and Authorization Protocol (GNAP) [I-D.ietf-txauth-gnap], which provides a JSON-based foundation for dynamic authorization scenarios. A GNAP binding for Agentic JWT would enable the agent-specific security mechanisms defined in this specification to work within the GNAP framework.

Additional areas for future development include:

* Integration with Model Context Protocol (MCP) standard [MCP]

* Support for additional hash algorithms beyond SHA-256

* Cross-domain agent identity federation

* Support for gRPC protocol. [GRPC]

* Support for prorietary protocols used for database connections, and asynchrnous messaging.

8. IANA Considerations

8.1. Overview

This specification requires IANA to register a new OAuth 2.0 authorization grant type, JWT claims, OAuth parameters, and error codes in the appropriate registries.

8.2. OAuth Authorization Grant Type Registration

This section registers the following grant type value in the IANA "OAuth URI" registry [IANA.OAuth.URI] established by "An IETF URN Sub-Namespace for Auth" [RFC6755]:

Grant Type	URN	Common Name	Change Controller	Reference
agent_checksum	urn:ietf:params:oauth:grant-type:agent_checksum	Agent Checksum Grant type for OAuth 2.0	IETF	[this document], Section 4

Table 1: OAuth Grant Type Registration

8.3. JWT Claims Registration

This section registers the following claims in the IANA "JSON Web Token Claims" registry [IANA.JWT.Claims] established by [RFC7519]:

Claim Name	Claim Description	Change Controller	Specification Document
workflow_id	Identifier of the workflow being executed	IETF	[this document], Section 4.4.2
workflow_step	Identifier of the current workflow step	IETF	[this document], Section 4.4.2
executed_by	Agent identifier executing the workflow step	IETF	[this document], Section 4.4.2
delegation_chain	SHA-256 hash of agent delegation sequence	IETF	[this document], Section 4.4.2
step_sequence_hash	SHA-256 hash of completed workflow steps	IETF	[this document], Section 4.4.2
agent_checksum	SHA-256 checksum of agent configuration	IETF	[this document], Section 4.4.2
registration_id	Unique agent registration instance identifier	IETF	[this document], Section 4.4.2

Table 2: JWT Claims Registrations

8.4. OAuth Parameters Registration

This section registers the following parameters in the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters] established by [RFC6749]:

Parameter Name	Parameter Usage Location	Change Controller	Specification Document
agent_id	token request	IETF	[this document], Section 4.2.1
computed_checksum	token request	IETF	[this document], Section 4.2.1
workflow_id	token request	IETF	[this document], Section 4.2.1
workflow_step	token request	IETF	[this document], Section 4.2.1
workflow_enabled	token request	IETF	[this document], Section 4.2.1
delegation_context	token request	IETF	[this document], Section 4.2.1
requested_scopes	token request	IETF	[this document], Section 4.2.1

Table 3: OAuth Parameters Registrations

8.5. OAuth Extensions Error Registration

This section registers the following error codes in the "OAuth Extensions Error Registry" [IANA.Extensions.Error] established by [RFC6749]:

Error Name	Error Usage Location	Related Protocol Extension	Change Controller	Specification
unknown_agent	token error response	Agent Checksum Grant	IETF	[this document], Section 4.5.3
agent_checksum_mismatch	token error response	Agent Checksum Grant	IETF	[this document], Section 4.5.4
workflow_step_unauthorized	token error response	Agent Checksum Grant	IETF	[this document], Section 4.5.5

Table 4: OAuth Error Code Registrations

8.6. Media Type Registration

This specification does not define any new media types. Agent identity verification and token issuance use existing media types:

* application/json (RFC 7159) - For token requests and responses

* application/jwt (RFC 7519) - For intent tokens

8.7. Registration Summary

Summary of IANA registrations required by this specification:

Registry	Number of Registrations
OAuth URI	1
JSON Web Token Claims	7
OAuth Parameters	7
OAuth Extensions Error Registry	3
TOTAL	*18*

Table 5: IANA Registration Summary

9. Security Considerations

This section describes Threat Model and Security Considerations similar to OAuth 2.0 [RFC6819].

9.1. Checksum Security

9.1.1. Checksum Verification Security

The agent checksum is the primary security mechanism in this grant type. Implementations MUST:

* Use constant-time comparison when verifying checksums to prevent timing attacks
* Log all checksum mismatch events for security monitoring
* Consider implementing rate limiting on checksum verification failures to prevent brute-force attacks
* Reject checksums that don’t match the expected format (sha256:<64 hex chars>)

9.1.2. Checksum Strength

SHA-256 provides 256-bit security against collision and preimage attacks. This is sufficient for agent identity purposes given:

* Checksums are not used for long-term secrets (only identity binding)
* Collision resistance prevents two different agents from having same checksum
* Preimage resistance prevents deriving agent config from checksum

Future versions of this specification MAY support alternative hash algorithms (e.g., SHA-3) through algorithm prefix versioning.

9.1.3. Registration Security

Agent registration endpoints MUST be protected with authentication and authorization to prevent:

* Unauthorized agents registering under legitimate agent_ids
* Denial of service through excessive registrations

* Checksum enumeration attacks

The reference implementation requires admin-level OAuth tokens with "register:intent" scope for registration endpoints.

9.1.4. Registry Storage Security

The agent registry contains sensitive identity information and MUST be protected:

* Store checksums with access controls (only IDP can read)

* Encrypt registry data at rest

* Log all registry access for audit trails

* Implement rate limiting on registry queries

* Regularly backup registry with integrity verification

9.2. Token Security

9.2.1. Token Lifetime

Intent tokens SHOULD have short lifetimes (5-10 minutes) because:

* They represent user intent for a specific workflow step

* Workflow state may change rapidly in multi-agent systems

* Short lifetime limits the window for token theft/replay

The reference implementation uses 5 minutes (300 seconds). But this value should be fully configurable.

9.2.2. Proof-of-Possession

The cnf (confirmation) claim with JWK binding provides proof-of-possession, preventing token theft. Resource servers SHOULD:

* Verify that API requests include proof of possession of the private key corresponding to the JWK in the cnf claim

* Reject tokens presented without valid proof-of-possession

* Use mechanisms like DPoP [I-D.ietf-oauth-dpop] for proof-of-possession validation

9.3. Workflow Security

Authorization servers MUST implement access controls on workflow definition endpoints. Only authenticated administrators with appropriate privileges SHOULD be able to create or modify workflow definitions.

Implementations SHOULD validate workflow definitions for security properties such as: (1) no cycles in agent transitions, (2) appropriate scope restrictions at each step, (3) required approval gates for high-privilege operations.

9.3.1. Workflow Validation

Workflow step validation is critical for preventing privilege escalation. But is applicable only if enabled. Authorization servers MUST:

* Enforce prerequisite step completion before issuing tokens
* Verify approval gates have been passed for steps requiring approval
* Validate that the delegation_chain is consistent with the workflow definition
* Prevent agents from skipping required steps

9.3.2. Delegation Chain Integrity

The delegation_chain hash provides integrity over the agent delegation path. However, implementations SHOULD additionally:

* Maintain a server-side record of delegation chains
* Validate that each delegation was authorized
* Limit delegation depth to prevent infinite delegation chains
* Revoke all tokens in a delegation chain if a parent agent is compromised

9.4. Implementation Considerations

9.4.1. Cryptographic Requirements

Implementations MUST use cryptographically secure hash functions for agent checksum computation. SHA-256 or stronger algorithms are REQUIRED. Weaker algorithms such as MD5 or SHA-1 MUST NOT be used.

For proof-of-possession key binding, implementations MUST support EdDSA with Ed25519 curves [ED25519] or ECDSA with P-256 curves. RSA keys with minimum 2048-bit length MAY be supported for backward compatibility.

Intent hash computation MUST use SHA-256 or stronger. The hash MUST be computed over the canonical JSON representation of the intent token to ensure consistent results across implementations.

9.4.2. Privacy Considerations

Authorization servers and resource servers MUST implement appropriate data retention and deletion policies to minimize privacy risks.

Token logging and audit trails SHOULD exclude sensitive user data when possible. If user data must be logged, implementations SHOULD use tokenization or anonymization techniques to protect privacy.

Cross-border data transfers of tokens containing personal information MUST comply with applicable data protection regulations such as GDPR, CCPA, and similar frameworks.

9.4.3. Denial of Service Considerations

The additional cryptographic operations required by this protocol (checksum computation, signature verification, hash computation) introduce potential denial of service attack vectors. Implementations SHOULD implement rate limiting on token endpoints.

Authorization servers SHOULD monitor for patterns indicating DoS attacks, such as: (1) excessive token requests from single clients, (2) repeated failed checksum validations, (3) malformed token requests designed to trigger expensive validation operations.

Resource servers MAY implement caching of validation results (with appropriate cache invalidation) to reduce computational overhead for repeated token validations.

9.5. Threat Analysis

The following 12 distinct security threats across six STRIDE categories [SHOSTACK] enumerated with their categorization and mitigation status in Table 6, have been used to demonstrate the need for this Agentic JWT protocol.

ID	Threat Name	STRIDE	OWASP	Mitigation
T1	Agent Identity Spoofing	Spoofing	A01:2021	A1, A2
T2	Token Replay Attacks	Spoofing	A02:2021	A6
T3	Shim Library Impersonation	Spoofing	A08:2021	A1, A2
T4	Runtime Code Modification	Tampering	A03:2021	A1, A12
T5	Prompt Injection Attacks	Tampering	LLM01:2025	A12
T6	Workflow Definition Tampering	Tampering	A04:2021	A8, A11
T7	Cross-Agent Privilege Escalation	Priv. Elev.	A01:2021	A3, A7, A8
T8	Workflow Step Bypass	Priv. Elev.	A01:2021	A8, A10
T9	Scope Inflation	Priv. Elev.	LLM06:2025	A7, A8
T10	Intent Origin Forgery	Repudiation	A09:2021	A9, A10
T11	Delegation Chain Manipulation	Repudiation	A02:2021	A6, A9
T12	Agent Configuration Exposure	Info. Disc.	A01:2021	A1, A2

Table 6: Threat Enumeration and Mitigation Status

OWASP categories referenced [OWASP-TOP10-2021] A01:2021 (Broken Access Control), A02:2021 (Cryptographic Failures), A03:2021 (Injection), A04:2021 (Insecure Design), A08:2021 (Software and Data

Integrity Failures), A09:2021 (Security Logging and Monitoring Failures), LLM01:2025 (Prompt Injection) [OWASP-LLM01], LLM06:2025 (Excessive Agency) [OWASP-LLM06].

Mitigation anchors (detailed in Table 8): A1 (Agent Checksum Verification), A2 (Shim Library Integrity), A3 (Scope Binding), A6 (PoP Key Binding), A7 (Delegation Context), A8 (Workflow State Tracking), A9 (Intent Hash Binding), A10 (Step Authorization), A11 (IDP Access Control), A12 (Input Validation).

9.6. Threat Descriptions and Attack Vectors

This section provides detailed descriptions of each identified threat, including attack vectors and real-world precedents. Table 7 presents this information in tabular form.

ID	Description	Attack Vector
T1	A malicious agent impersonates a legitimate agent by replicating its identifier, source code structure, prompts, and tool configurations to create a malicious agent with identical checksum signatures. [OWASP-LLM07]	Attacker gains access to agent source code (e.g., through repository compromise) and creates a malicious agent with identical checksum signatures.
T2	Intercepted intent tokens are replayed by unauthorized agents to gain access to protected resources.	Network interception or memory dumps expose valid intent tokens that are replayed before expiration.
T3	Malicious replacement of legitimate shim library with compromised version that bypasses security controls.	Supply chain compromise or local privilege escalation to replace shim library files.
T4	Agent prompts, tools, or configurations are modified at runtime after successful checksum registration.	Memory injection, debugger attachment, or reflection-based modification of agent properties.
T5	Malicious inputs cause LLM agents	Crafted user inputs

	to generate unintended instructions that bypass security policies. [OWASP-LLM01], [PEREZ-PROMPT-INJECTION].	or external data sources containing prompt injection payloads that manipulate agent reasoning.
T6	Unauthorized modification of workflow definitions in the authorization server to permit unauthorized agent transitions.	Compromised administrative credentials or authorization server vulnerabilities allowing workflow redefinition.
T7	Lower-privilege agent manipulates higher-privilege agent to perform unauthorized operations beyond the original user intent.	Agent A with read-only permissions crafts requests that cause Agent B to execute destructive operations.
T8	Agents skip required approval steps or execute workflow steps out of sequence to gain unauthorized access.	Direct API calls bypassing workflow engine or manipulation of workflow state tracking.
T9	Agents request or utilize broader scopes than originally intended for the specific workflow step. [OWASP-LLM06]	Token minting requests with inflated scopes or misuse of broad scopes for unintended operations.
T10	Unable to cryptographically prove which user intent led to specific agent actions, enabling plausible deniability.	Lack of cryptographic binding between user intent and downstream agent actions.
T11	Modification or forgery of delegation chains to hide true origin of agent actions.	Manipulation of delegation assertion claims or replay of valid delegation

			chains in unauthorized contexts.
T12	Unauthorized access to agent prompts, tools, and configurations leading to system knowledge disclosure.		API endpoints exposing agent metadata or memory dumps revealing agent configurations.

Table 7: Threat Descriptions and Attack Vectors

9.7. Security Anchors and Mitigation Mechanisms

The Agentic JWT protocol employs twelve security anchors that collectively address all identified threats. Table 8 describes each mitigation mechanism and the threats it addresses.

Anchor	Mechanism	Threats
A1	Agent Checksum Verification: Runtime checksum computation and verification against registered values	T1, T3, T4, T12
A2	Shim Library Integrity: Cryptographic validation of shim library authenticity	T1, T3, T12
A3	Scope Binding: Cryptographic binding of authorized scopes to specific workflow steps	T7, T9
A6	Proof-of-Possession Key Binding: Per-agent cryptographic keys bound to tokens	T2, T11
A7	Delegation Context Validation: Verification of delegation chain authenticity	T7, T9
A8	Workflow State Tracking: Authorization server maintains workflow execution state	T6, T7, T8
A9	Intent Hash Binding: Cryptographic binding of user intent to token lifecycle	T10, T11
A10	Step Authorization: Per-step authorization checks at resource servers	T8, T10
A11	Authorization Server Access Control: Protection of workflow definitions	T6
A12	Input Validation and Sanitization: Validation of agent inputs and LLM outputs	T4, T5

Table 8: Security Anchors and Mitigation Mechanisms

10. References

10.1. Normative References

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

[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>.

[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>.

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

[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>.

[RFC7516] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/info/rfc7516>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

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

[FIPS-180-4] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, August 2015.

[RFC7800] Jones, M., Campbell, J., and J. Bradley, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

10.2. Informative References

[RFC6819] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <https://www.rfc-editor.org/info/rfc6819>.

[RFC8693] Campbell, B., Bradley, J., Sakimura, N., Jones, M., and W. Denniss, "OAuth 2.0 Token Exchange", RFC 8693, DOI 10.17487/RFC8693, January 2020, <https://www.rfc-editor.org/info/rfc8693>.

[I-D.ietf-oauth-dpop] Fett, D., Denniss, W., and M. Ansari, "OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)", Work in Progress, Internet-Draft, draft-ietf-oauth-dpop-06, April 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop-06>.

[I-D.ietf-txauth-gnap] Richer, J. and K. Bezemer, "Grant Negotiation and Authorization Protocol", Work in Progress, Internet-Draft, draft-ietf-txauth-gnap-17, March 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-txauth-gnap-17>.

[NIST-SP-800-63] National Institute of Standards and Technology, "Digital Identity Guidelines", NIST Special Publication 800-63-3, DOI 10.6028/NIST.SP.800-63-3, June 2017, <https://doi.org/10.6028/NIST.SP.800-63-3>.

[NIST-SP-800-63C] National Institute of Standards and Technology, "Digital Identity Guidelines: Federation and Assertions", NIST Special Publication 800-63C, DOI 10.6028/NIST.SP.800-63c, June 2017, <https://doi.org/10.6028/NIST.SP.800-63c>.

[NIST-SP-800-207] Rose, S., Borchert, O., Mitchell, S., and S. Connelly, "Zero Trust Architecture", NIST Special Publication 800-207, DOI 10.6028/NIST.SP.800-207, September 2020, <https://doi.org/10.6028/NIST.SP.800-207>.

[OWASP-TOP10-2021]
OWASP Foundation, "OWASP Top 10 - 2021: The Ten Most Critical Web Application Security Risks", Version 2021, 2021.

[OWASP-LLM01]
OWASP Foundation, "LLM01:2025 Prompt Injection", OWASP Top 10 for LLM Applications v1.1, 2025.

[OWASP-LLM06]
OWASP Foundation, "LLM06:2025 Excessive Agency", OWASP Top 10 for LLM Applications v1.1, 2025.

[OWASP-LLM07]
OWASP Foundation, "LLM07:2025 System Prompt Leakage", OWASP Top 10 for LLM Applications v1.1, 2025.

[SHOSTACK] Shostack, A., "Threat Modeling: Designing for Security", Publisher Wiley, ISBN 978-1118809990, 2014.

[LANGCHAIN]
Chase, H., "LangChain: Framework for Developing LLM-Powered Applications", 2022.

[LANGGRAPH]
LangChain, "LangGraph: Library for Building Stateful Multi-Agent Applications", 2024.

[CREWAI] CrewAI, "CrewAI: Framework for Orchestrating Role-Playing Autonomous AI Agents", 2023.

[MCP] Anthropic PBC, "Model Context Protocol (MCP): A Protocol for Connecting AI Agents to Data Sources and Tools", 2024.

[GRPC] Google, "gRPC: A High-Performance, Open Source Universal RPC Framework", 2015.

[ED25519] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-Speed High-Security Signatures", Cryptology ePrint Archive Report 2011/368, 2012.

[PEREZ-PROMPT-INJECTION]
Perez, F. and I. Ribeiro, "Ignore Previous Prompt: Attack Techniques For Language Models", arXiv arXiv:2211.09527, DOI 10.48550/ARXIV.2211.09527, 2022, <https://doi.org/10.48550/ARXIV.2211.09527>.

[PATENT-REF]
Goswami, A., "Cryptographic Agent Authentication and Intent Delegation System with Checksum Based Identity Verification and Workflow Aware Token Binding", U.S. Patent Application 19/315,486, 30 August 2025. pending

[PAPER-REF]
Goswami, A., "Agentic JWT: Securing Autonomous AI Agents Through Cryptographic Intent Binding", arXiv arXiv:2509.13597, 2025. To be updated with actual arXiv identifier after publication

[IANA.OAuth.URI]
IANA, "OAuth URI", <https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml#uri>.

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

[IANA.JWT.Claims]
IANA, "JSON Web Token Claims", <https://www.iana.org/assignments/jwt/jwt.xhtml#claims>.

[IANA.Extensions.Error]
IANA, "JSON Web Token Claims", <https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml#extensions-error>.

Acknowledgements

This work builds upon the foundational contributions of the OAuth working group and the broader internet security community.

Contributors

Abhishek Goswami
Email: abhishek@abhishekgoswami.io

Author’s Address

Abhishek Goswami (editor)
Email: abhishek@abhishekgoswami.io