How AgentAuth Works: Identity Validation and PoA Token Issuance

This document provides a step-by-step walkthrough of how ATB's AgentAuth service validates agent identity and issues Proof-of-Authorization (PoA) tokens.

Overview

AgentAuth is the identity and authorization gateway for AI agents. Before any agent can perform an action through ATB, it must:

1. Prove its identity (via SPIFFE/SPIRE)
2. Request authorization for a specific action
3. Obtain human approval (if required)
4. Receive a short-lived PoA token
5. Present the token to the Broker for action execution

```
                    AgentAuth Authorization Flow



  Agent       AgentAuth       Approver            Broker


      1. Request
         Challenge


      2. Challenge ID


                        3. Human Approval


      4. Request PoA


      5. PoA Token


      6. API Request + PoA Token


      7. Response
```

Step 1: Agent Identity via SPIFFE

Before requesting any authorization, an agent must have a cryptographic identity issued by SPIFFE/SPIRE.

What is SPIFFE?

SPIFFE (Secure Production Identity Framework for Everyone) provides: - **Automatic identity issuance** - No secrets to manage - **Short-lived certificates** - Default 10-minute validity - **Workload attestation** - Identity based on where code runs, not secrets

SPIFFE ID Format

Every agent has a SPIFFE ID that uniquely identifies it:

```
spiffe://<trust-domain>/<workload-path>

Examples:
- spiffe://prod.company.com/agents/crm-assistant
- spiffe://prod.company.com/ns/ai-workloads/sa/claude-agent
```

How the Agent Gets Its Identity

```
     AI Agent                            SPIRE Agent
     Workload                            (per node)


           1. Request SVID via Workload API


                SPIRE attests workload:
                - Kubernetes namespace/SA
                - Process UID/GID
                - Docker labels

           2. X.509-SVID (short-lived cert)


              Certificate contains:
              - SPIFFE ID as SAN URI
              - 10-min validity
              - Auto-rotated by SPIRE
```

AgentAuth Validates SPIFFE ID

When an agent connects to AgentAuth, its SPIFFE ID is validated:

```go
// From agentauth/main.go - SPIFFE ID validation
var validSPIFFEIDRegex = regexp.MustCompile(
    `^spiffe://[a-zA-Z0-9]([a-zA-Z0-9.-]*[a-zA-Z0-9])?(/[a-zA-Z0-9._-]+)+$`
)

func validateSPIFFEID(id string) error {
    if len(id) == 0 {
        return errors.New("SPIFFE ID is empty")
    }
    if len(id) > 2048 {
        return errors.New("SPIFFE ID too long (max 2048)")
    }
    if strings.Contains(id, "..") {
        return errors.New("SPIFFE ID contains path traversal")
    }
    if !validSPIFFEIDRegex.MatchString(id) {
        return errors.New("SPIFFE ID format invalid")
    }
    return nil
}
```

Step 2: Request a Challenge

The agent requests a challenge for a specific action. This creates an authorization request that must be approved before a PoA token is issued.

Challenge Request

```
POST /v1/challenge
Content-Type: application/json

{
    "agent_spiffe_id": "spiffe://prod.company.com/agents/crm-assistant",
    "act": "crm.contact.update",
    "con": {
        "max_records": 10,
        "allowed_fields": ["email", "phone"]
    },
    "leg": {
        "basis": "contract",
        "ref": "MSA-2026-001",
        "jurisdiction": "US",
        "accountable_party": {
            "type": "human",
            "id": "user@company.com"
        }
    }
}
```

What AgentAuth Validates

Field	Validation
`agent_spiffe_id`	Must match SPIFFE URI format, max 512 chars, no path traversal
`act`	Action name, max 256 chars, no null bytes
`con`	Constraints object, max 10 levels deep, no null bytes in keys/values
`leg`	Legal basis object, must include `accountable_party`

Rate Limiting

AgentAuth enforces rate limits to prevent abuse:

```
// Per-IP rate limiting (default: 100/min)
if !ipRateLimiter.Allow(clientIP) {
    return 429 Too Many Requests
}

// Per-agent rate limiting (default: 20/min)
if !agentRateLimiter.Allow(req.AgentSPIFFEID) {
    return 429 Too Many Requests
}
```

Challenge Response

```
{
    "challenge_id": "chal_x7k9m2...",
    "expires_at": "2026-01-15T10:05:00Z",
    "requires_dual_control": false,
    "approvers_needed": 1,
    "approval_hint": "POST /v1/approve with challenge_id and approver identity"
}
```

Step 3: Risk Assessment and Dual Control

AgentAuth determines if the action requires enhanced approval based on risk tier.

High-Risk Actions (Dual Control Required)

Certain actions automatically require two independent approvers:

```
// Default high-risk actions
highRiskActions := []string{
    "sap.vendor.change",
    "iam.privilege.escalate",
    "payments.transfer.execute",
    "ot.system.manual_override",
}
```

This can be configured via environment variable:

```
DUAL_CONTROL_ACTIONS="sap.vendor.change,payments.transfer.execute,..."
```

Explicit Dual Control

Requests can also explicitly require dual control:

```
{
    "leg": {
        "dual_control": {
            "required": true
        }
    }
}
```

Audit Log Entry

Every challenge creation is logged:

```
{
    "timestamp": "2026-01-15T10:00:00Z",
    "event": "challenge.created",
    "challenge_id": "chal_x7k9m2...",
    "agent_spiffe_id": "spiffe://prod.company.com/agents/crm-assistant",
    "action": "crm.contact.update",
    "risk_tier": "low",
    "requires_dual_control": false,
    "source_ip": "10.0.1.50",
    "success": true,
    "expires_at": "2026-01-15T10:05:00Z"
}
```

Step 4: Human Approval

A human approver must approve the challenge before a PoA token can be issued.

Approval Request

```
POST /v1/approve
Authorization: Bearer <approver-jwt>
Content-Type: application/json

{
    "challenge_id": "chal_x7k9m2..."
}
```

Approver Authentication

AgentAuth supports multiple authentication methods:

Method	Configuration	Use Case
EdDSA JWT	`APPROVER_ED25519_PUBLIC_KEY_PEM`	Production (recommended)
RSA JWT	`APPROVER_RSA_PUBLIC_KEY_PEM`	Enterprise SSO integration
HMAC JWT	`APPROVER_JWT_SECRET`	Development/testing
Shared Secret	`APPROVAL_SHARED_SECRET`	Legacy (not recommended)

```go
// JWT verification supports multiple algorithms
switch token.Method.Alg() {
case "EdDSA":
    // Ed25519 signature verification
case "RS256", "RS384", "RS512":
    // RSA signature verification
case "HS256", "HS384", "HS512":
    // HMAC verification
}
```

Self-Approval Prevention

By default, the accountable party cannot approve their own requests:

```go
if preventSelfApproval {
    accountableParty := req.getAccountablePartyID()
    if normalizeApproverID(approverID) == normalizeApproverID(accountableParty) {
        return "self-approval not allowed"
    }
}
```

Dual Control Approval

For high-risk actions requiring dual control:

1. First approver approves → Challenge status: 1/2 approvers
2. Second approver approves → Challenge status: 2/2 approvers (fully approved)
3. Both approvers must be distinct

```json
{
    "challenge_id": "chal_x7k9m2...",
    "requires_dual_control": true,
    "approvers_needed": 2,
    "approvers_count": 1,
    "approvers": [
        {
            "id": "manager@company.com",
            "approved_at": "2026-01-15T10:01:00Z"
        }
    ],
    "fully_approved": false
}
```

Step 5: PoA Token Issuance

Once the challenge is fully approved, the agent can exchange it for a PoA token.

Token Request

```
POST /v1/token
Content-Type: application/json

{
    "challenge_id": "chal_x7k9m2..."
}
```

Token Generation

AgentAuth generates a signed JWT with specific claims:

```
claims := PoAClaims{
    Act: challenge.Req.Act,           // Authorized action
    Con: challenge.Req.Con,           // Constraints
    Leg: challenge.Req.Leg,           // Legal basis
    RegisteredClaims: jwt.RegisteredClaims{
        Issuer:    "atb-agentauth",
        Subject:   challenge.Req.AgentSPIFFEID,
        Audience:  []string{"atb-broker"},
        ExpiresAt: jwt.NewNumericDate(now.Add(poaTTL)),
        IssuedAt:  jwt.NewNumericDate(now),
        ID:        mustRandID("poa_"),
    },
}
```

Token Signing (EdDSA)

Tokens are signed with Ed25519 for security and performance:

```
token := jwt.NewWithClaims(jwt.SigningMethodEdDSA, claims)
token.Header["kid"] = kid  // Key ID for rotation support

signedToken, err := token.SignedString(privateKey)
```

PoA Token Structure

```
Header:
```

```
{
    "alg": "EdDSA",
    "typ": "JWT",
    "kid": "abc123..."  // Key ID for verification
}

Payload:
{
    "iss": "atb-agentauth",
    "sub": "spiffe://prod.company.com/agents/crm-assistant",
    "aud": ["atb-broker"],
    "exp": 1736936700,  // 5 minutes from issuance
    "iat": 1736936400,
    "jti": "poa_xyz789...",
    "act": "crm.contact.update",
    "con": {
        "max_records": 10,
        "allowed_fields": ["email", "phone"]
    },
    "leg": {
        "basis": "contract",
        "ref": "MSA-2026-001",
        "accountable_party": {
            "type": "human",
            "id": "user@company.com"
        }
    }
}
```

Token Response

```json
{
    "poa_token": "eyJhbGciOiJFZERTQSIs...",
    "expires_at": "2026-01-15T10:05:00Z",
    "token_id": "poa_xyz789..."
}
```

Step 6: Key Rotation Support

AgentAuth supports seamless key rotation for production environments.

Multiple Signing Keys

```bash
# Primary key (used for signing)
POA_SIGNING_ED25519_PRIVKEY_PEM="..."

# Previous key (for verification during rotation)
```

```
POA_SIGNING_ED25519_PRIVKEY_PEM_PREV="..."

# Next key (pre-staged for upcoming rotation)
POA_SIGNING_ED25519_PRIVKEY_PEM_NEXT="..."
```

JWKS Endpoint

The Broker verifies tokens using the JWKS endpoint:

```
GET /.well-known/jwks.json
```

```
{
    "keys": [
        {
            "kty": "OKP",
            "crv": "Ed25519",
            "use": "sig",
            "alg": "EdDSA",
            "kid": "abc123...",
            "x": "base64url-encoded-public-key"
        },
        {
            "kty": "OKP",
            "crv": "Ed25519",
            "use": "sig",
            "alg": "EdDSA",
            "kid": "def456...",
            "x": "base64url-encoded-previous-key"
        }
    ]
}
```

Step 7: Broker Token Verification

When the agent presents the PoA token to the Broker, it is validated before proxying the request.

Broker Verification Steps

1. **Parse JWT** - Extract header, payload, signature
2. **Verify Signature** - Using JWKS from AgentAuth
3. **Check Expiration** - Token must not be expired
4. **Validate Audience** - Must include "atb-broker"
5. **Match SPIFFE ID** - Token `sub` must match client certificate
6. **Verify Action** - Requested action must match `act` claim
7. **Enforce Constraints** - Request must satisfy `con` claims
8. **OPA Policy** - Query OPA for additional policy checks

OPA Policy Evaluation

```
# From opa/policy/poa.rego
default allow := false

allow if {
    valid_signature
    not_expired
    valid_audience
    valid_action
    constraints_satisfied
}
```

Security Controls Summary

Control	Implementation
Identity	SPIFFE/SPIRE X.509-SVIDs
Input Validation	Regex, length limits, null byte checks
Rate Limiting	Per-IP and per-agent limits
Token TTL	Max 900 seconds (15 min), default 300s
Signing	Ed25519 (EdDSA) with key rotation
Self-Approval Prevention	Accountable party cannot approve
Dual Control	Two approvers for high-risk actions
Audit Logging	Structured JSON for all events
Security Headers	X-Content-Type-Options, CSP, etc.

Configuration Reference

AgentAuth Environment Variables

Variable	Default	Description
`LISTEN_ADDR`	`:9090`	HTTP listen address
`POA_ISSUER`	`atb-agentauth`	Token issuer claim
`POA_TTL_SECONDS`	`300`	Token validity (max 900)
`CHALLENGE_TTL_SECONDS`	`300`	Challenge validity (max 900)
`RATE_LIMIT_PER_IP`	`100`	Requests per minute per IP
`RATE_LIMIT_PER_AGENT`	`20`	Requests per minute per agent
`REQUIRE_JWT_AUTH`	`false`	Require JWT for approvers
`ALLOW_SELF_APPROVAL`	`false`	Allow self-approval (not recommended)
`DUAL_CONTROL_ACTIONS`	(defaults)	Comma-separated high-risk actions
`POA_SIGNING_ED25519_PRIVKEY_PEM`	(generated)	Primary signing key

Troubleshooting

Common Issues

"SPIFFE ID format invalid" - Ensure SPIFFE ID matches: `spiffe://<domain>/<path>` - No path traversal (`..`), no special characters

"rate limit exceeded" - Check `RATE_LIMIT_PER_IP` and `RATE_LIMIT_PER_AGENT` settings - Implement exponential backoff in agent

"self-approval not allowed" - The accountable party in `leg.accountable_party.id` cannot approve - Use a different approver or set `ALLOW_SELF_APPROVAL=true` (not recommended)

"challenge expired" - Challenges expire after `CHALLENGE_TTL_SECONDS` (default 5 min) - Request a new challenge

"JWT verification failed" - Check approver JWT configuration - Verify issuer is in `APPROVER_JWT_ISSUERS` list - Ensure public key matches the signing key

---

Related Documentation

- Authentication & Authorization Guide
- SPIFFE/SPIRE Integration Guide
- Security Hardening Guide
- Audit Logging
- API Reference