Advanced Security for AI Agents

Highly Regulated Identity (HRI) - Complete Guide

Table of Contents

Introduction to HRI

What is HRI (Highly Regulated Identity)?

HRI stands for **Highly Regulated Identity —** a hardened security framework designed for sensitive, high-stakes domains where standard security measures are insufficient.

Industries Requiring HRI

Organizations in these sectors face strict legal and regulatory requirements:

Finance

- Banks and credit unions
- Payment processors
- Trading platforms
- Investment firms

- Insurance companies

Healthcare

- Hospitals and clinics
- Patient record systems
- Pharmaceutical companies
- Health insurance providers
- Telemedicine platforms

Government

- Military systems
- Law enforcement databases
- Public service portals
- Tax and revenue systems
- Border and immigration control

Legal

- Law firms
- Court systems
- Legal document management
- Compliance platforms

Why These Industries Need Maximum Security

The stakes in these sectors are exceptionally high:

- **Financial impact**: Data breaches can cost millions or billions of dollars
- **Legal consequences**: Privacy violations result in massive fines and criminal charges
- **Life and death**: Healthcare security failures can endanger patient lives
- **National security**: Government system breaches threaten public safety
- **Regulatory compliance**: Failure to meet standards can shut down entire operations
- **Public trust**: Security incidents destroy organizational credibility

HRI Origins

HRI was formerly known as **FAPI** (Financial-grade API), initially developed by the financial services

industry to secure banking APIs. Due to its robust security model, it has been adopted across all highly regulated industries and renamed to reflect its broader applicability.

The Problem with Standard OAuth

What is OAuth?

OAuth is the industry-standard protocol for authorization — the technology behind "Sign in with Google," "Sign in with Facebook," and similar single sign-on experiences.

OAuth is widely used and generally effective for consumer applications, but it has inherent weaknesses when applied to high-security environments.

Three Critical Weaknesses

1. Optional Security Features

Many OAuth security measures are **optional**, not mandatory:

```
OAuth specification:
- Strong client authentication → OPTIONAL
- Token binding → OPTIONAL
- Proof of possession → OPTIONAL
- Sender constraints → OPTIONAL

Result: Developers can skip these for convenience,
        creating security vulnerabilities
```

Real-world impact:

Many implementations use the simplest, least secure options because they're easier to implement. This creates widespread vulnerabilities across the ecosystem.

---

2. Insecure Options Allowed

OAuth permits weak authentication methods that are vulnerable to attacks:

```
Allowed but insecure methods:
X Client secrets in URLs
X Tokens transmitted in URL parameters
X Long-lived tokens without rotation
X No verification of token holder identity
X Implicit flow (deprecated but still used)
```

Attack scenario:

```
1. User logs in → Receives token "abc123"
2. Token sent in URL: app.com/callback?token=abc123
3. URL logged in server logs
4. Attacker accesses logs → Steals token
5. Attacker uses token → System accepts it ⚠️
6. System cannot detect it's not the legitimate user
```

The system has no way to verify that the person using the token is the person who received it.

---

3. Too Permissive by Default

Standard OAuth scopes are often overly broad:

```
Typical scope: "read_files"

What it actually grants:
- Read ALL user files
- No time limits
- No context restrictions
- No file-specific limitations
- No conditional access

Result: Much broader access than necessary
```

Example of excessive permissions:

User request: "Let the app check my calendar for conflicts"

What standard OAuth typically grants:
✓ Read all calendar events (past, present, future)
✓ Read all attendee information
✓ Read all event details and notes
✓ Access shared calendars
✓ No time restrictions

What should be granted:
✓ Read only today's events
✓ Return only free/busy status (no details)
✓ Access expires in 1 hour

This violates the principle of least privilege and increases risk exposure.

HRI Solution: Hardened OAuth/OIDC Profile

HRI transforms OAuth from a flexible, general-purpose protocol into a rigorous, security-first framework by:

1. Removing Insecure Options

Elimination of weak methods:

❌ REMOVED in HRI:
- Implicit flow (tokens in URLs)
- Client secrets in plain text
- Weak client authentication
- Optional security features
- Bearer tokens without binding
- Long-lived access tokens

✓ REQUIRED in HRI:
- Authorization code flow with PKCE
- Strong client authentication (mTLS or private key JWT)
- Sender-constrained tokens (mandatory)
- Short-lived, rotating tokens
- Proof of possession

---

2. Requiring Security Enhancements

Mandatory protective measures:

All optional security features become **required**:

```
Standard OAuth: "You should use mTLS" (optional)
HRI:            "You MUST use mTLS" (mandatory)

Standard OAuth: "Consider token binding" (suggested)
HRI:            "Token binding is REQUIRED" (enforced)

Standard OAuth: "Rotate tokens when possible" (best practice)
HRI:            "Token rotation is MANDATORY" (automatic)
```

---

3. Implementing Zero Trust

Never trust, always verify:

Every request undergoes complete verification regardless of source:

```
Traditional model:
- Trusted network = less verification
- Inside firewall = assume safe
- Valid token once = trust continues

Zero Trust model:
- No trusted networks or zones
- Verify every request independently
- Continuous authentication
- Assume breach has occurred
- Minimize blast radius
```

---

4. Proven Security Model

HRI is not theoretical—it's **battle-tested** in the world's most security-sensitive environments:

- **Open Banking**: Secures millions of financial transactions daily
- **Payment Services Directive 2 (PSD2)**: EU regulatory compliance
- **Healthcare systems**: HIPAA-compliant patient data access
- **Government platforms**: Classified information protection

---

Key HRI Security Features

Feature 1: Sender-Constrained Tokens

The Fundamental Problem

Standard OAuth tokens are like **bearer tokens—**whoever holds the token can use it, similar to cash or a gift card.

```
Standard Token Flow:
1. User logs in → Receives token "abc123"
2. User makes API call with token
3. Attacker intercepts token (man-in-the-middle, log leak, etc.)
4. Attacker makes API call with stolen token
5. Server checks: "Is token valid?" → Yes
6. Server grants access to attacker ⚠️

Problem: Server cannot distinguish legitimate holder from attacker
```

Real-world analogy:

A standard token is like a movie ticket without your name on it. Anyone holding the ticket can enter the theater, even if they stole it.

The Solution: Cryptographic Binding

Sender-constrained tokens are **cryptographically bound** to the legitimate holder, making them useless to attackers.

```
Sender-Constrained Token Flow:
1. User logs in → Receives token bound to identity proof
2. User makes API call with token + cryptographic proof
3. Attacker intercepts token
4. Attacker makes API call with stolen token
5. Server checks: "Is token valid?" → Yes
6. Server checks: "Can you prove you're the original recipient?" → No
7. Server denies access to attacker ✓

Result: Token is worthless without cryptographic proof of identity
```

Enhanced analogy:

A sender-constrained token is like a passport. Even if someone steals it, they can't use it because their face doesn't match the photo, and they can't fake the biometric chip embedded inside.

Implementation Methods

Method 1: Mutual TLS (mTLS)

Both client and server authenticate using X.509 certificates:

```
Setup:
- Server has certificate (proves server identity)
- Client has certificate (proves client identity)
- Both certificates issued by trusted authority

Authentication flow:
1. Client: "I want to connect" + Client Certificate
2. Server: "Prove you own that certificate" + Challenge
3. Client: Signs challenge with private key
4. Server: Verifies signature with client's public key
5. Server: "I'm the real server" + Server Certificate
6. Client: Verifies server certificate
7. Connection established with mutual authentication

Token binding:
- Access token includes hash of client certificate
- Every API call must be made over same mTLS connection
- Server verifies token certificate hash matches connection
- Stolen token cannot be used on different connection
```

Security properties:

- ✓ Both parties cryptographically verified
- ✓ Token bound to specific TLS connection
- ✓ Stolen token useless without client certificate
- ✓ Certificate theft requires private key (hardware-protected)

Method 2: Proof of Possession (PoP)

Token includes cryptographic challenge requiring private key:

```
Setup:
- Client generates public/private key pair
- Public key shared with authorization server
- Private key kept secret (never transmitted)

Token issuance:
1. Client requests token
2. Server issues token containing:
   - Access permissions
   - Hash of client's public key
   - Cryptographic nonce

API request:
1. Client creates request
2. Client signs request with private key:
   Signature = Sign(Request + Token + Timestamp, PrivateKey)
3. Client sends:
   - Request
   - Token
   - Signature
4. Server verifies signature using client's public key:
   Valid = Verify(Signature, Request + Token + Timestamp, PublicKey)
5. If valid → Grant access
   If invalid → Deny access

Attack scenario:
- Attacker steals token
- Attacker cannot create valid signature (no private key)
- Server rejects all attacker requests
```

Security properties:

✓ Private key never transmitted
✓ Each request has unique signature
✓ Replay attacks prevented (timestamp included)
✓ Token useless without private key

Method 3: Private Key JWT

Client creates signed JSON Web Token for each request:

```
Setup:
- Client has public/private key pair
- Server knows client's public key
- Keys managed through registration process

Authentication:
1. Client creates JWT containing:
   {
     "iss": "client_id",          // Issuer (who created this)
     "sub": "client_id",          // Subject (who it's about)
     "aud": "https://api.server", // Audience (who should accept)
     "iat": 1702831234,           // Issued at (timestamp)
     "exp": 1702831294,           // Expires (60 seconds later)
     "jti": "unique_id_12345"     // JWT ID (prevent replay)
   }

2. Client signs JWT with private key:
   Signature = Sign(JWT_Header + JWT_Payload, PrivateKey)

3. Client includes signed JWT in request:
   Authorization: Bearer eyJhbGc...signed_jwt...

4. Server validates:
   a. Signature matches using public key
   b. Token not expired
   c. Token not used before (check jti)
   d. Audience matches this server
   e. Issue time is recent

5. If all checks pass → Grant access
```

Security properties:

- ✓ Strong client authentication
- ✓ Non-repudiation (signature proves client created request)
- ✓ Short-lived tokens (60 seconds typical)
- ✓ Replay prevention (unique jti)
- ✓ Audience restriction (token only valid for specific server)

Comparison Table

Method	Strength	Complexity	Use Case
mTLS	Highest	High	Banking, government
PoP	High	Medium	Healthcare, enterprise
Private Key JWT	High	Medium	Financial APIs, B2B

Real-World Impact

Without sender-constrained tokens:

- 2019: $1.2 billion stolen through token theft attacks
- Average breach cost: $4.24 million
- Token replay attacks: Common and effective

With sender-constrained tokens:

- Token theft becomes worthless attack vector
- Even complete token exposure causes no breach
- Dramatically reduces attack surface

Feature 2: Rich Authorization Requests (RAR)

The Limitation of Traditional Scopes

Standard OAuth uses simple, text-based scopes that are too coarse-grained:

Traditional scope: "read_files"

Problems:
❌ Applies to ALL files
❌ No time restrictions
❌ No context conditions
❌ No amount limits
❌ No transactional precision
❌ Can't specify which files
❌ Can't limit by file type
❌ Can't add business logic

Result: All-or-nothing access that violates least privilege

Real-world example:

User wants: "Let app access my latest tax document"

Traditional OAuth grants: "read_files"
- Access to ALL files
- All personal documents
- Family photos
- Business contracts
- Medical records
- Everything forever

This is like asking for your driver's license and receiving keys to your entire house.

The Solution: Rich Authorization Requests

Rich Authorization Requests (RAR) provide **structured, granular, context-aware permissions**:

json

```
{
  "type": "file_access",
  "actions": ["read"],
  "locations": ["/documents/taxes/2024"],
  "file_types": ["pdf"],
  "max_files": 1,
  "one_time": true,
  "expires": "2024-12-14T18:00:00Z",
  "purpose": "tax_filing_verification"
}
```

This precisely defines:

- **What**: Read access only
- **Where**: Specific folder only
- **Which**: PDF files only
- **How many**: Maximum 1 file
- **How often**: One-time use
- **When**: Expires in specified time
- **Why**: For tax filing verification

Banking Example: Payment Authorization

Traditional scope:

```
scope: "payments"

Grants ability to:
- Make unlimited payments
- Any amount
- To any recipient
- At any time
- Forever
```

Rich Authorization Request:

```
json
```

```
{
  "type": "payment_initiation",
  "actions": ["initiate"],
  "instructedAmount": {
    "currency": "USD",
    "amount": "99.99"
  },
  "creditorAccount": {
    "iban": "DE12345678901234567890",
    "name": "Acme Corp"
  },
  "creditorAgent": {
    "bic": "DEUTDEFF"
  },
  "remittanceInformation": {
    "reference": "Invoice-2024-12345"
  },
  "max_occurrences": 1,
  "valid_until": "2024-12-14T23:59:59Z"
}
```

This authorizes:

- Exactly one payment
- Exactly $99.99
- To specific IBAN only
- For specific invoice
- Valid only until end of day

Healthcare Example: Patient Record Access

Traditional scope:

```
scope: "read_patient_records"

Grants access to:
- All patient records in system
- All medical history
- All test results
- All prescriptions
- All diagnoses
```

Rich Authorization Request:

```json
{
  "type": "medical_record_access",
  "actions": ["read"],
  "patient_id": "MRN-12345",
  "record_types": ["lab_results"],
  "date_range": {
    "start": "2024-12-01",
    "end": "2024-12-14"
  },
  "fields": ["test_name", "result", "normal_range"],
  "exclude": ["doctor_notes", "diagnosis"],
  "purpose": "patient_portal_view",
  "purpose_code": "TREAT",
  "consent_reference": "consent-2024-456",
  "valid_for": "24_hours"
}
```

This authorizes:

- **Specific patient only** (not all patients)
- **Lab results only** (not full medical record)
- **Date-limited** (last 2 weeks only)
- **Field-limited** (results only, no notes)
- **Purpose-specified** (treatment, not research)
- **Consent-linked** (patient consent on file)
- **Time-limited** (24 hours only)

E-commerce Example: Order Management

Traditional scope:

```
scope: "orders"

Grants ability to:
- View all orders
- Modify any order
- Cancel any order
- Refund any amount
- Access all customer data
```

Rich Authorization Request:

```json
{
  "type": "order_management",
  "actions": ["read", "update_status"],
  "order_id": "ORD-2024-789",
  "allowed_status_changes": [
    {"from": "pending", "to": "processing"},
    {"from": "processing", "to": "shipped"}
  ],
  "cannot_access": ["payment_info", "customer_address"],
  "can_access": ["order_items", "shipping_status"],
  "max_updates": 5,
  "valid_for": "7_days",
  "requires_approval_for": {
    "status_change": "cancelled",
    "refund_amount": "> $0"
  }
}
```

This authorizes:

- **Specific order only**
- **Read and update status only** (not cancel/refund)
- **Specific status transitions only**
- **No payment info access**
- **Limited number of updates**
- **Time-bound authorization**
- **Requires approval for sensitive actions**

Benefits of Rich Authorization Requests

1. Precision

- Exact permissions, no ambiguity
- Transaction-specific authorization
- Eliminates over-permissioning

2. Context-Awareness

- Includes business logic and constraints
- Adapts to specific use cases
- Understands intent and purpose

3. Auditability

- Complete record of what was authorized
- Why it was authorized (purpose)
- When it expires
- Easy compliance reporting

4. Security

- Minimal permissions granted
- Time-bound access
- Reduces blast radius of compromise
- Granular revocation

5. User Trust

- Users see exactly what they're approving
- Transparent permission model
- Builds confidence in AI agents

Feature 3: Back Channel Communication

The URL Leakage Problem

Standard OAuth sends sensitive data through browser URLs, creating multiple security vulnerabilities:

Standard OAuth redirect:
https://app.com/callback?
code=SECRET_AUTHORIZATION_CODE_ABC123&
state=USER_SESSION_STATE_XYZ789&
token=SOMETIMES_ACCESS_TOKEN_HERE

Where this appears:
❌ Browser address bar (visible to user)
❌ Browser history (stored on device)
❌ Web server access logs (readable by admins)
❌ Proxy server logs (readable by network admins)
❌ Browser extensions (can intercept)
❌ Referrer headers (sent to next site)
❌ Analytics tools (tracked and stored)
❌ Screenshots and screen recordings

Attack vectors:

1. Log file exposure

Server logs contain:
2024-12-14 GET /callback?code=SECRET_CODE

If logs leaked/hacked:
- Attacker gets authorization codes
- Can exchange for access tokens
- Full account compromise

2. Browser history theft

Malware scans browser history:
- Finds OAuth callback URLs
- Extracts codes from parameters
- Uses codes before they expire

3. Referrer leakage

```
User clicks link after OAuth callback:

Referrer header sent to new site:
Referer: https://app.com/callback?code=SECRET

Third-party site now has OAuth code
```

4. **Shoulder surfing**

```
URL visible in address bar:
Nearby person sees SECRET_CODE
Types it into their own device
Gains unauthorized access
```

The Solution: Secure Back Channel

Back channel communication sends sensitive data **server-to-server** over encrypted connections, never through the browser:

HRI Back Channel Flow:

Step 1 - Browser (Front Channel):
https://app.com/callback?
  request_id=safe_public_reference_abc123

  Nothing sensitive in URL ✓
  Safe to log ✓
  Safe in browser history ✓

Step 2 - Server-to-Server (Back Channel):
POST https://auth.server.com/token
Headers:
  Content-Type: application/json
  Authorization: mTLS client certificate

Body:
{
  "request_id": "safe_public_reference_abc123",
  "client_id": "verified_client",
  "client_assertion": "signed_jwt_proof",
  "grant_type": "authorization_code"
}

Connection properties:
✓ Encrypted with TLS 1.3
✓ Server authenticated (certificate)
✓ Client authenticated (certificate)
✓ No intermediary can intercept
✓ Not logged in access logs
✓ Never touches browser

Response (Server-to-Server):
{
  "access_token": "SECRET_TOKEN_XYZ",
  "refresh_token": "SECRET_REFRESH_ABC",
  "expires_in": 3600,
  "token_type": "Bearer"
}

Tokens transmitted securely ✓
Never exposed to client browser ✓
Stored only in secure server memory ✓

Communication Channel Comparison

Aspect	Front Channel (Bad)	Back Channel (Good)
Transport	Browser URL	Server-to-Server HTTPS
Visibility	Visible to user	Hidden from user
Logging	Logged everywhere	Minimal secure logging
Interception	Easy (many points)	Very difficult
Browser history	Stored indefinitely	Never stored
Analytics tracking	Often tracked	Never tracked
Screenshot exposure	Visible in screenshots	Not visible
Third-party access	Extensions, proxies	None

Real-World Analogy

❌ Front Channel (Insecure):

```
Scenario: Restaurant payment

Bad approach:
1. Waiter asks for credit card
2. You SHOUT your credit card number across restaurant
3. Everyone hears: "4532-1234-5678-9012"
4. Number visible to all diners
5. Security cameras record it
6. Restaurant logs note the number

Risks:
- Other diners overhear
- Recorded on cameras
- Written in logs
- Anyone can steal and use
```

✓ Back Channel (Secure):

Good approach:
1. Waiter brings payment terminal to table
2. You insert card into terminal directly
3. Terminal communicates with bank securely
4. No one sees card number
5. Encrypted communication
6. Only bank and terminal interact

Security:
- Card never leaves your hand
- Number never spoken aloud
- Encrypted end-to-end
- No opportunity for theft

Back Channel Security Enhancements

1. Mutual TLS Authentication

Both client and server prove identity:

Standard TLS: Only server proves identity
Back Channel: Both parties authenticate

Client proves: "I am legitimate application server"
Server proves: "I am legitimate auth server"
Both verified before any data exchange

2. Request Encryption

All data encrypted multiple layers:

Layer 1: TLS 1.3 (transport encryption)
Layer 2: JWE (JSON Web Encryption) for sensitive fields
Layer 3: Individual field encryption for extra-sensitive data

Result: Even if TLS broken, data still protected

3. Temporary Request URIs

Short-lived references replace sensitive data:

```
request_uri: "urn:ietf:params:oauth:request:abc123"

Properties:
- Valid for 60 seconds only
- Single use (cannot be reused)
- No sensitive data embedded
- Server already has full request details
- Cannot be guessed or brute-forced
```

Impact on Security Posture

Before back channel (front channel exposure):

- 73% of OAuth security incidents involved URL parameter leakage
- Average cost of exposed credentials: $150 per user
- Log file breaches exposed thousands of tokens

After back channel implementation:

- URL-based token theft: Eliminated
- Log file breaches: No sensitive data exposed
- Browser-based attacks: No longer viable
- Compliance audit findings: Reduced by 90%

Feature 4: Dynamic Client Registration & PAR

PAR: Pushed Authorization Requests

The Traditional Flow Problem:

In standard OAuth, the authorization request is built by the client and sent through the browser:

```
Traditional OAuth Authorization Request:

User clicks "Login with OAuth"
↓
Browser redirects to:
https://auth.provider.com/authorize?
  response_type=code&
  client_id=12345&
  redirect_uri=https://app.com/callback&
  scope=read_profile+write_data&
  state=random_state_abc123&
  nonce=random_nonce_xyz789&
  code_challenge=SHA256_hash_of_verifier&
  code_challenge_method=S256

Issues:
❌ Entire request visible in browser
❌ Parameters can be manipulated
❌ Sensitive data in URL
❌ Request not validated until user action
❌ Vulnerable to parameter injection
❌ No integrity protection
```

Attack scenarios:

1. **Scope manipulation**

```
Attacker modifies URL:
scope=read_profile → scope=read_profile+admin_access

If server doesn't validate properly:
- Attacker gains elevated permissions
- User doesn't see the change
- Authorization granted with excessive scope
```

2. **Redirect URI manipulation**

```
Attacker changes:
redirect_uri=https://app.com/callback
to:
redirect_uri=https://attacker.com/steal

Result:
- Authorization code sent to attacker
- Attacker exchanges for access token
- Full account compromise
```

The PAR Solution

Pushed Authorization Requests send the authorization request **server-to-server BEFORE** involving the user:

PAR Flow:

Step 1: Application Server → Authorization Server
(Back channel, secure, encrypted)

POST https://auth.provider.com/par
Authorization: mTLS client certificate
Content-Type: application/json

```
{
  "response_type": "code",
  "client_id": "verified_app_12345",
  "redirect_uri": "https://app.com/callback",
  "scope": "read_profile write_data",
  "state": "random_state_abc123",
  "nonce": "random_nonce_xyz789",
  "code_challenge": "SHA256_hash",
  "code_challenge_method": "S256",

  // Rich authorization details
  "authorization_details": {
    "type": "account_access",
    "actions": ["read"],
    "account_id": "user123"
  }
}
```

Authorization Server validates:
✓ Client certificate authentic
✓ Redirect URI registered
✓ Scope permitted for this client
✓ Authorization details well-formed
✓ All parameters valid

Response:

```
{
  "request_uri": "urn:ietf:params:oauth:request_uri:abc123xyz",
  "expires_in": 60
}
```

Properties:
✓ Request validated before user involvement
✓ Request_uri is opaque, reveals nothing
✓ Short-lived (60 seconds)
✓ Single-use
✓ Cryptographically linked to client

```
---

Step 2: User's Browser → Authorization Server
(Front channel, but no sensitive data)

https://auth.provider.com/authorize?
  client_id=verified_app_12345&
  request_uri=urn:ietf:params:oauth:request_uri:abc123xyz

User sees in browser:
✓ Clean, simple URL
✓ No sensitive parameters
✓ Cannot be tampered with
✓ Request details already on server

Authorization Server:
1. Looks up request_uri
2. Retrieves full request submitted in Step 1
3. Shows user consent screen with accurate details
4. User approves/denies
5. Issues authorization code
```

Security Benefits of PAR

1. Parameter Integrity

Request cannot be modified:

```
Without PAR:
User sees: scope=read_profile
Attacker modifies to: scope=admin_access
Server processes modified scope ⚠️

With PAR:
Request submitted securely before user involvement ✓
Request_uri is opaque reference ✓
User cannot modify anything ✓
Server uses original, validated request ✓
```

2. Pre-validation

Errors caught before user involvement:

Without PAR:
1. User clicks login
2. Browser redirects
3. User sees consent screen
4. User approves
5. Error: Invalid redirect_uri ❌
6. Poor user experience

With PAR:
1. App submits request to server
2. Server validates immediately
3. If invalid: App gets error before user sees anything
4. App can fix and retry
5. User only sees valid, working flow ✓

3. No URL Exposure

Sensitive parameters never in browser:

Without PAR:
Browser URL: ...?scope=admin+financial_data+health_records
Anyone can see requested permissions before user approval

With PAR:
Browser URL: ...?request_uri=urn:...abc123
Opaque reference reveals nothing ✓
Actual permissions hidden until consent screen ✓

4. Replay Prevention

Request URI is single-use:

Attacker intercepts request_uri:
urn:ietf:params:oauth:request_uri:abc123

Attacker tries to reuse:
Server checks: "Already used" ❌
Request denied ✓

Additionally:
- Expires in 60 seconds
- Bound to original client
- Cannot be used from different client

Dynamic Client Registration

The Problem: Pre-registration Burden

Traditional OAuth requires manual client registration:

```
Traditional process:
1. Developer creates app
2. Developer manually registers with each OAuth provider:
   - Visits registration portal
   - Fills out forms
   - Waits for approval (hours/days)
   - Receives client_id and client_secret
3. Developer hardcodes credentials
4. Repeat for every OAuth provider

Issues:
❌ Slow and manual process
❌ Doesn't scale for AI agents
❌ Credentials often stored insecurely
❌ Difficult to update or rotate
```

The Solution: Automated Registration

HRI supports programmatic client registration:

```
Dynamic Registration Flow:

POST https://auth.provider.com/register
Content-Type: application/json

{
  "redirect_uris": ["https://app.com/callback"],
  "token_endpoint_auth_method": "private_key_jwt",
  "grant_types": ["authorization_code", "refresh_token"],
  "response_types": ["code"],
  "client_name": "AI Finance Assistant",
  "client_uri": "https://app.com",
  "logo_uri": "https://app.com
```