MCP Server Implementation Guide for ServiceNow Integration

Part 4: OAuth 2.1 Implementation

Document Series

This implementation guide is organized into five parts:

1. Part 1: Introduction and Overview
2. Part 2: Server Foundation - Core Infrastructure Setup
3. Part 3: MCP Protocol Implementation and Tools
4. **Part 4: OAuth 2.1 Implementation** (This Document)
5. Part 5: Appendices and Recommendations

Introduction

This document covers the OAuth 2.1 authentication and authorization layer that secures your MCP server for ServiceNow integration. Using the house building metaphor, this is the "roof" - the protective security layer that covers and secures the MCP protocol (walls) and tools (interior) built in Part 3.

What You'll Build:

- OAuth 2.1 authorization server with PKCE support
- Dynamic Client Registration (DCR) for automated setup
- JWT token creation, validation, and revocation
- Authorization and token exchange endpoints
- Authentication middleware for protected routes

Building on Previous Parts:

Part 2 established the server foundation (HTTP server, middleware, configuration, storage). Part 3 implemented the MCP protocol and tools. Part 4 adds the security layer that:

- Authenticates ServiceNow as a trusted client
- Issues JWT tokens for API access
- Validates tokens on every protected request
- Manages token lifecycle (issuance, refresh, revocation)

By completing this part, ServiceNow can securely connect to your MCP server and execute tools on behalf of authenticated users.

OAuth 2.1 Architecture Overview

Understanding OAuth 2.1 for MCP Servers

OAuth 2.1 provides the authorization framework that secures your MCP server, ensuring only authenticated ServiceNow instances can access your tools and capabilities. Unlike traditional OAuth flows designed for human users clicking through consent screens, the MCP-ServiceNow integration uses a machine-to-machine (M2M) pattern where ServiceNow acts as a trusted client. The flow combines OAuth 2.1's Authorization Code Grant with mandatory PKCE (Proof Key for Code Exchange) to prevent authorization code interception attacks.

OAuth Flow Overview

The complete OAuth flow for ServiceNow MCP integration:

1. Client Registration (One-Time Setup)

- ServiceNow calls your `/register` endpoint (Dynamic Client Registration)
- Your server generates `client_id` and `client_secret`
- Credentials returned to ServiceNow and stored securely

2. Authorization Request (Per Authentication)

- ServiceNow redirects to your `/oauth/authorize` endpoint
- Includes PKCE `code_challenge` (SHA-256 hash of random `code_verifier`)
- Your server validates client and generates authorization code
- Redirects back to ServiceNow with the code

3. Token Exchange (Per Authentication)

- ServiceNow calls your `/oauth/token` endpoint
- Sends authorization code, client credentials, and `code_verifier`
- Your server validates PKCE (computes hash of verifier, compares with challenge)
- Issues JWT access token and refresh token

4. Authenticated MCP Requests (Ongoing)

- ServiceNow includes access token in `Authorization: Bearer` header
- Your server validates JWT signature and expiration

- Checks token hasn't been revoked (Redis blacklist)
- Allows request if valid, rejects if invalid/expired

5. Token Refresh (When Access Token Expires)

- ServiceNow calls `/oauth/token` with refresh token
- Your server validates refresh token
- Issues new access token and rotates refresh token
- Old refresh token added to revocation blacklist

6. Token Revocation (Optional - When Disconnecting)

- ServiceNow calls `/oauth/revoke` with token to invalidate
- Your server adds token ID (jti) to Redis blacklist
- Token becomes invalid for future requests

Why PKCE is Mandatory

PKCE (Proof Key for Code Exchange) prevents authorization code interception attacks:

- **Without PKCE**: Attacker intercepts authorization code → exchanges for tokens → gains access
- **With PKCE**: Attacker intercepts code but doesn't have `code_verifier` → cannot exchange for tokens

OAuth 2.1 makes PKCE mandatory for all authorization code flows, replacing the optional nature of OAuth 2.0.

Machine-to-Machine (M2M) Authentication

ServiceNow integration uses M2M authentication pattern:

- **ServiceNow handles user authentication** (SSO, MFA, passwords, RBAC)
- **MCP server authenticates ServiceNow as a client** (OAuth client credentials)
- **Trust boundary**: MCP server trusts ServiceNow to enforce user-level authorization
- **Audit trail**: ServiceNow logs which user triggered which tool; MCP server logs which client called which tool

This division of responsibilities follows OAuth 2.1 best practices for trusted client integrations.

Important: User Authentication Division of Responsibility

In this M2M (Machine-to-Machine) architecture:

- **ServiceNow handles USER authentication** (SSO, MFA, RBAC, user sessions, access controls)

- **MCP server handles CLIENT authentication** (OAuth 2.1, client credentials, token validation)

The MCP server trusts ServiceNow as an authenticated client to enforce user-level authorization. This is why `user_id` generation is "simulated" in the code examples—ServiceNow manages actual user identity, authentication, and access controls. The MCP server authenticates that the request comes from ServiceNow (the trusted client), while ServiceNow ensures the request comes from an authorized user.

OAuth 2.1 Flow Diagram

```
┌─────────────────────────────────────────────────────────────
│           OAuth 2.1 with PKCE Flow for ServiceNow            │
└─────────────────────────────────────────────────────────────

┌──────────────┐                          ┌──────────────┐
│ ServiceNow │                          │ MCP Server │
│  Instance  │                          │            │
└──────┬───────┘                          └──────┬───────┘
  │                          │
  │ ┌──────────────────────────────────────────────────────┐ │
  │ │ STEP 1: Client Registration (One-Time Setup via DCR) │ │
  │ └──────────────────────────────────────────────────────┘ │
  │                          │
  │ POST /register                    │
  │ Authorization: Bearer {DCR_AUTH_TOKEN}          │
  │ Body: { client_name, redirect_uris, grant_types }       │
  │─────────────────────────────────────────────────────────>
  │                          │
  │        Generate client_id & client_secret       │
  │        Store client in registry         │
  │                          │
  │ 200 OK                          │
  │ { client_id, client_secret, ... }          │
  │<─────────────────────────────────────────────────────────
  │                          │
  │ [ServiceNow stores credentials in oauth_credential table]  │
  │                          │
  │ ┌──────────────────────────────────────────────────────┐ │
  │ │ STEP 2: Authorization Request (Per Authentication)   │ │
  │ └──────────────────────────────────────────────────────┘ │
  │                          │
  │ [Generate code_verifier (random string)]          │
  │ [Compute code_challenge = SHA256(code_verifier)]        │
  │                          │
  │ GET /oauth/authorize?                   │
  │   response_type=code&                   │
  │   client_id={client_id}&                   │
  │   redirect_uri={redirect_uri}&                   │
  │   code_challenge={code_challenge}&                   │
  │   code_challenge_method=S256&                   │
  │   state={state}                   │
  │─────────────────────────────────────────────────────────>
  │                          │
│              Validate client_id & redirect_uri          │
│              Validate PKCE parameters                   │
│              Generate authorization code                │
│              Store code + code_challenge                │
│                                                         │
│  302 Redirect                                           │
│  Location: {redirect_uri}?code={auth_code}&state={state}     │
│<───────────────────────────────────────────────────────────────────────
│                                                         │
│  [Receives authorization code]                          │
│                                                         │
│  ┌─────────────────────────────────────────────────────────────────┐ │
│  │ STEP 3: Token Exchange (Per Authentication)             │ │
│  └─────────────────────────────────────────────────────────────────┘ │
│                                                         │
│  POST /oauth/token                                      │
│  Body: {                                                │
│    grant_type=authorization_code,                       │
│    code={auth_code},                                    │
│    redirect_uri={redirect_uri},                         │
│    client_id={client_id},                               │
│    client_secret={client_secret},                       │
│    code_verifier={code_verifier}                        │
│  }                                                      │
│───────────────────────────────────────────────────────────────────────>
│                                                         │
│              Validate client credentials                │
│              Validate authorization code                │
│              Validate PKCE:                             │
│                SHA256(code_verifier) == code_challenge  │
│              Delete authorization code (single-use)     │
│              Generate JWT access & refresh tokens       │
│                                                         │
│  200 OK                                                 │
│  {                                                      │
│    access_token: {JWT},                                 │
│    refresh_token: {JWT},                                │
│    token_type: "Bearer",                                │
│    expires_in: 3600                                     │
│  }                                                      │
│<───────────────────────────────────────────────────────────────────────
│                                                         │
│  [ServiceNow stores tokens in oauth_credential table]   │
│                                                         │
```

```
|  ┌─────────────────────────────────────────────────────────────┐  |
|  │ STEP 4: Authenticated MCP Requests (Ongoing)              │ │
|  └─────────────────────────────────────────────────────────────┘  |
|                                 |
| POST /mcp                            |
| Authorization: Bearer {access_token}          |
| Body: { method: "tools/call", params: {...} }       |
|──────────────────────────────────────────────────────────────────>
|                                 |
|          Extract Bearer token            |
|          Verify JWT signature            |
|          Check expiration              |
|          Check not revoked (Redis)         |
|          Execute tool               |
|                                 |
| 200 OK                             |
| { jsonrpc: "2.0", result: {...} }          |
|<──────────────────────────────────────────────────────────────────
|                                 |
|  ┌─────────────────────────────────────────────────────────────┐  |
|  │ STEP 5: Token Refresh (When Access Token Expires)        │ │
|  └─────────────────────────────────────────────────────────────┘  |
|                                 |
| POST /oauth/token                        |
| Body: {                             |
|  grant_type=refresh_token,                |
|  refresh_token={refresh_token},             |
|  client_id={client_id},                 |
|  client_secret={client_secret}             |
| }                                |
|──────────────────────────────────────────────────────────────────>
|                                 |
|          Validate refresh token (JWT)        |
|          Validate client credentials        |
|          Revoke old refresh token (Redis)      |
|          Generate new access & refresh tokens    |
|          Increment rotation_count          |
|                                 |
| 200 OK                             |
| {                                |
|  access_token: {new_JWT},                |
|  refresh_token: {new_JWT},               |
|  token_type: "Bearer",                 |
|  expires_in: 3600                     |
│  }                                  │
│<──────────────────────────────────────────────────────────────────────
│                                     │
│  ┌──────────────────────────────────────────────────────────────────┐ │
│  │ STEP 6: Token Revocation (Optional)                   │ │
│  └──────────────────────────────────────────────────────────────────┘ │
│                                     │
│  POST /oauth/revoke                          │
│  Body: {                                 │
│    token={token_to_revoke},                      │
│    client_id={client_id},                      │
│    client_secret={client_secret}                   │
│  }                                  │
│──────────────────────────────────────────────────────────────────────>
│                                     │
│              Decode JWT to get jti                 │
│              Add jti to Redis blacklist            │
│              Set TTL = token remaining lifetime         │
│                                     │
│  200 OK                                │
│  {}                                   │
│<──────────────────────────────────────────────────────────────────────
│                                     │
```

Key Components

Your OAuth implementation requires these components:

- **Endpoints**: `/register`, `/oauth/authorize`, `/oauth/token`, `/oauth/revoke`, metadata endpoints
- **Storage**: Registered clients, authorization codes, revoked tokens
- **Utilities**: PKCE validation, token generation, JWT signing/verification
- **Middleware**: Bearer token extraction and validation

Utility Functions

Cryptographic and Encoding Functions

Implementation Approach

OAuth 2.1 implementation requires several cryptographic and encoding utility functions that support token generation, PKCE validation, and secure random value creation. These utilities form the foundation for all OAuth operations—generate them once and reuse throughout your implementation. Use established cryptographic libraries rather than implementing your own algorithms to ensure security and standards compliance.

Key considerations:

- Use cryptographically secure random number generators (crypto.randomBytes)
- PKCE validation must use SHA-256 hashing (S256 method)
- Base64URL encoding differs from standard Base64 (no padding, URL-safe characters)
- Generate tokens with sufficient entropy (32+ bytes recommended)
- Never use Math.random() for security-critical token generation

Pseudocode:

```
// Generate cryptographically secure random token
FUNCTION generate_secure_token(byte_length):
    random_bytes = CRYPTO_RANDOM_BYTES(byte_length)
    token = BASE64URL_ENCODE(random_bytes)
    RETURN token
END FUNCTION

// Base64URL encoding (URL-safe, no padding)
FUNCTION base64url_encode(buffer):
    base64_string = BASE64_ENCODE(buffer)

    // Replace characters for URL safety
    url_safe = base64_string
        .REPLACE('+', '-')
        .REPLACE('/', '_')
        .REPLACE('=', '')  // Remove padding

    RETURN url_safe
END FUNCTION

// Validate PKCE challenge against verifier
FUNCTION validate_pkce(code_verifier, code_challenge):
    // Compute SHA-256 hash of verifier
    hash = SHA256(code_verifier)

    // Encode as Base64URL
    computed_challenge = BASE64URL_ENCODE(hash)

    // Compare with provided challenge
    RETURN computed_challenge == code_challenge
END FUNCTION

// Example usage:
client_secret = generate_secure_token(32)  // 32 bytes = 256 bits
auth_code = generate_secure_token(32)
jwt_secret = generate_secure_token(32)

is_valid = validate_pkce(verifier_from_request, challenge_from_storage)
```

JavaScript:

```
javascript
```

```
const crypto = require('crypto');

// Generate cryptographically secure random token
function generateSecureToken(byteLength = 32) {
  const randomBytes = crypto.randomBytes(byteLength);
  return randomBytes.toString('base64url');
}

// Base64URL encoding (URL-safe, no padding)
function base64UrlEncode(buffer) {
  return buffer.toString('base64')
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=/g, '');  // Remove padding
}

// Validate PKCE challenge against verifier
function validatePKCE(codeVerifier, codeChallenge) {
  // Compute SHA-256 hash of verifier
  const hash = crypto.createHash('sha256').update(codeVerifier).digest();

  // Encode as Base64URL
  const computedChallenge = base64UrlEncode(hash);

  // Compare with provided challenge
  return computedChallenge === codeChallenge;
}

// Example usage:
const clientSecret = generateSecureToken(32);  // 32 bytes = 256 bits
const authCode = generateSecureToken(32);
const jwtSecret = generateSecureToken(32);

const isValid = validatePKCE(verifierFromRequest, challengeFromStorage);
```

TypeScript:

```
typescript
```

```
import crypto from 'crypto';

// Generate cryptographically secure random token
function generateSecureToken(byteLength: number = 32): string {
  const randomBytes: Buffer = crypto.randomBytes(byteLength);
  return randomBytes.toString('base64url');
}

// Base64URL encoding (URL-safe, no padding)
function base64UrlEncode(buffer: Buffer): string {
  return buffer.toString('base64')
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=/g, '');  // Remove padding
}

// Validate PKCE challenge against verifier
function validatePKCE(codeVerifier: string, codeChallenge: string): boolean {
  // Compute SHA-256 hash of verifier
  const hash: Buffer = crypto.createHash('sha256').update(codeVerifier).digest();

  // Encode as Base64URL
  const computedChallenge: string = base64UrlEncode(hash);

  // Compare with provided challenge
  return computedChallenge === codeChallenge;
}

// Example usage:
const clientSecret: string = generateSecureToken(32);  // 32 bytes = 256 bits
const authCode: string = generateSecureToken(32);
const jwtSecret: string = generateSecureToken(32);

const isValid: boolean = validatePKCE(verifierFromRequest, challengeFromStorage);
```

JWT Token Management

Token Creation and Validation

Why JWT Tokens:

This implementation uses JWT (JSON Web Token) for both access and refresh tokens. JWTs are stateless and self-contained, embedding all necessary claims within the token itself. This eliminates the need for database lookups on every request—the server validates tokens by verifying the cryptographic signature rather than querying a token database.

JWT vs Opaque Tokens:

- **JWT (this implementation)**: Stateless validation, horizontal scaling without shared session storage, faster authentication checks
- **Opaque tokens (alternative)**: Requires database/Redis lookup per request, simpler revocation, but adds latency and database dependency

Implementation Approach

JWT (JSON Web Token) provides stateless authentication for your MCP server—tokens are self-contained, eliminating the need for database lookups on every request. Create separate functions for access tokens (short-lived, used for API requests) and refresh tokens (long-lived, used to obtain new access tokens). Each token type includes specific claims that identify the user, client, scope, and token purpose. Sign all tokens with your JWT secret using HS256 algorithm for symmetric key cryptography.

Key considerations:

- Access tokens expire quickly (1 hour recommended) to limit exposure if compromised
- Refresh tokens live longer (30 days) allowing persistent sessions while requiring periodic re-authentication
- **Token Lifetime Rationale**: 1-hour access tokens balance security (short-lived, limited window for misuse) with usability (reduces token refresh frequency). 30-day refresh tokens enable seamless user experience across sessions while ensuring periodic credential validation.
- Include `jti` (JWT ID) claim for revocation tracking
- Include `type` claim to distinguish access vs refresh tokens
- Use `rotation_count` in refresh tokens to detect reuse attacks
- Validate `iss` (issuer) and `aud` (audience) claims during verification
- Check revocation status via Redis before accepting tokens

Pseudocode:

```
// Create access token (short-lived, for API requests)
FUNCTION create_access_token(user_id, client_id, scope):
    current_time = CURRENT_TIMESTAMP_SECONDS()

    token_payload = {
        sub: user_id,                    // Subject (user identifier)
        client_id: client_id,            // OAuth client
        scope: scope,                    // Granted permissions
        iat: current_time,               // Issued at
        exp: current_time + 3600,        // Expires in 1 hour
        iss: JWT_ISSUER,                 // Token issuer
        aud: JWT_ISSUER + "/mcp",        // Intended audience
        jti: GENERATE_UUID(),            // JWT ID (for revocation)
        type: "access"                   // Token type
    }

    // Sign with JWT secret using HS256
    jwt_token = JWT_SIGN(token_payload, JWT_SECRET, algorithm="HS256")

    RETURN jwt_token
END FUNCTION

// Create refresh token (long-lived, for obtaining new access tokens)
FUNCTION create_refresh_token(user_id, client_id, scope, rotation_count):
    current_time = CURRENT_TIMESTAMP_SECONDS()

    token_payload = {
        sub: user_id,
        client_id: client_id,
        scope: scope,
        iat: current_time,
        exp: current_time + 2592000,     // Expires in 30 days
        iss: JWT_ISSUER,
        aud: JWT_ISSUER + "/mcp",
        jti: GENERATE_UUID(),
        type: "refresh",                 // Token type
        rotation_count: rotation_count   // Track rotations
    }

    jwt_token = JWT_SIGN(token_payload, JWT_SECRET, algorithm="HS256")

    RETURN jwt_token
END FUNCTION
```

```
// Validate token (verify signature, expiration, revocation)
FUNCTION validate_token(jwt_token):
  TRY:
    // Verify signature and claims
    decoded = JWT_VERIFY(jwt_token, JWT_SECRET, {
      issuer: JWT_ISSUER,
      algorithms: ["HS256"]
    })

    // Check if revoked (async Redis check)
    is_revoked = CHECK_REDIS_REVOCATION(decoded.jti)

    IF is_revoked:
      THROW_ERROR("Token has been revoked")
    END IF

    RETURN decoded  // Token is valid

  CATCH verification_error:
    LOG_ERROR("Token validation failed: " + verification_error.message)
    RETURN null  // Token is invalid
  END TRY
END FUNCTION
```

JavaScript:

```
javascript
```

```
const jwt = require('jsonwebtoken');
const { v4: uuidv4 } = require('uuid');

// Create access token (short-lived, for API requests)
function createAccessToken(userId, clientId, scope) {
  const currentTime = Math.floor(Date.now() / 1000);

  const payload = {
    sub: userId,                  // Subject (user identifier)
    client_id: clientId,          // OAuth client
    scope: scope,                 // Granted permissions
    iat: currentTime,             // Issued at
    exp: currentTime + 3600,      // Expires in 1 hour
    iss: JWT_ISSUER,              // Token issuer
    aud: `${JWT_ISSUER}/mcp`,     // Intended audience
    jti: uuidv4(),                // JWT ID (for revocation)
    type: 'access'                // Token type
  };

  // Sign with JWT secret using HS256
  return jwt.sign(payload, JWT_SECRET, { algorithm: 'HS256' });
}

// Create refresh token (long-lived, for obtaining new access tokens)
function createRefreshToken(userId, clientId, scope, rotationCount = 0) {
  const currentTime = Math.floor(Date.now() / 1000);

  const payload = {
    sub: userId,
    client_id: clientId,
    scope: scope,
    iat: currentTime,
    exp: currentTime + 2592000,    // Expires in 30 days
    iss: JWT_ISSUER,
    aud: `${JWT_ISSUER}/mcp`,
    jti: uuidv4(),
    type: 'refresh',              // Token type
    rotation_count: rotationCount   // Track rotations
  };

  return jwt.sign(payload, JWT_SECRET, { algorithm: 'HS256' });
}
```

```
// Validate token (verify signature, expiration, revocation)
async function validateToken(token) {
  try {
    // Verify signature and claims
    const decoded = jwt.verify(token, JWT_SECRET, {
      issuer: JWT_ISSUER,
      algorithms: ['HS256']
    });

    // Check if revoked (async Redis check)
    const isRevoked = await isTokenRevoked(decoded.jti);

    if (isRevoked) {
      throw new Error('Token has been revoked');
    }

    return decoded;  // Token is valid

  } catch (error) {
    console.error('[AUTH] Token validation failed:', error.message);
    return null;  // Token is invalid
  }
}
```

TypeScript:

```
typescript
```

```
import jwt from 'jsonwebtoken';
import { v4 as uuidv4 } from 'uuid';

// JWT payload structure
interface JwtPayload {
  sub: string;
  client_id: string;
  scope: string;
  iat: number;
  exp: number;
  iss: string;
  aud: string;
  jti: string;
  type: 'access' | 'refresh';
  rotation_count?: number;
}

// Create access token (short-lived, for API requests)
function createAccessToken(userId: string, clientId: string, scope: string): string {
  const currentTime: number = Math.floor(Date.now() / 1000);

  const payload: JwtPayload = {
    sub: userId,                    // Subject (user identifier)
    client_id: clientId,            // OAuth client
    scope: scope,                   // Granted permissions
    iat: currentTime,               // Issued at
    exp: currentTime + 3600,        // Expires in 1 hour
    iss: JWT_ISSUER,                // Token issuer
    aud: `${JWT_ISSUER}/mcp`,       // Intended audience
    jti: uuidv4(),                  // JWT ID (for revocation)
    type: 'access'                  // Token type
  };

  // Sign with JWT secret using HS256
  return jwt.sign(payload, JWT_SECRET, { algorithm: 'HS256' });
}

// Create refresh token (long-lived, for obtaining new access tokens)
function createRefreshToken(userId: string, clientId: string, scope: string, rotationCount: number = 0): string {
  const currentTime: number = Math.floor(Date.now() / 1000);

  const payload: JwtPayload = {
    sub: userId,
    client_id: clientId,
    scope: scope,
    iat: currentTime,
    exp: currentTime + 2592000,    // Expires in 30 days
    iss: JWT_ISSUER,
    aud: `${JWT_ISSUER}/mcp`,
    jti: uuidv4(),
    type: 'refresh',              // Token type
    rotation_count: rotationCount   // Track rotations
  };

  return jwt.sign(payload, JWT_SECRET, { algorithm: 'HS256' });
}

// Validate token (verify signature, expiration, revocation)
async function validateToken(token: string): Promise<JwtPayload | null> {
  try {
    // Verify signature and claims
    const decoded = jwt.verify(token, JWT_SECRET, {
      issuer: JWT_ISSUER,
      algorithms: ['HS256']
    }) as JwtPayload;

    // Check if revoked (async Redis check)
    const isRevoked: boolean = await isTokenRevoked(decoded.jti);

    if (isRevoked) {
      throw new Error('Token has been revoked');
    }

    return decoded;  // Token is valid

  } catch (error) {
    console.error('[AUTH] Token validation failed:', (error as Error).message);
    return null;  // Token is invalid
  }
}
```

Authorization Server Metadata Endpoint

OAuth Discovery (RFC 8414)

Implementation Approach

The Authorization Server Metadata endpoint provides OAuth discovery functionality, allowing ServiceNow to automatically learn your server's OAuth capabilities and endpoint locations. This endpoint follows RFC 8414 (OAuth 2.0 Authorization Server Metadata) and must be publicly accessible without authentication. ServiceNow can query this endpoint to discover your authorization, token, and revocation endpoints, supported grant types, and PKCE requirements.

Key considerations:

- Must be available at `/.well-known/oauth-authorization-server` path
- No authentication required (public discovery endpoint)
- Returns JSON metadata describing OAuth server capabilities
- ServiceNow uses this for automatic endpoint discovery
- Advertise only features you actually implement
- Include all endpoint URLs (absolute URLs recommended)

Pseudocode:

```
FUNCTION handle_oauth_metadata(request, response):
    metadata = {
        issuer: JWT_ISSUER,
        authorization_endpoint: JWT_ISSUER + "/oauth/authorize",
        token_endpoint: JWT_ISSUER + "/oauth/token",
        revocation_endpoint: JWT_ISSUER + "/oauth/revoke",
        registration_endpoint: JWT_ISSUER + "/register",
        response_types_supported: ["code"],
        grant_types_supported: ["authorization_code", "refresh_token"],
        token_endpoint_auth_methods_supported: ["client_secret_post"],
        code_challenge_methods_supported: ["S256"],
        scopes_supported: ["openid", "email", "profile"]
    }

    response.JSON(metadata)
END FUNCTION

// Register endpoint
app.GET('/.well-known/oauth-authorization-server', handle_oauth_metadata)
```

JavaScript:

javascript

```javascript
// Authorization Server Metadata endpoint (RFC 8414)
app.get('/.well-known/oauth-authorization-server', (req, res) => {
  const metadata = {
    issuer: JWT_ISSUER,
    authorization_endpoint: `${JWT_ISSUER}/oauth/authorize`,
    token_endpoint: `${JWT_ISSUER}/oauth/token`,
    revocation_endpoint: `${JWT_ISSUER}/oauth/revoke`,
    registration_endpoint: `${JWT_ISSUER}/register`,
    response_types_supported: ['code'],
    grant_types_supported: ['authorization_code', 'refresh_token'],
    token_endpoint_auth_methods_supported: ['client_secret_post'],
    code_challenge_methods_supported: ['S256'],
    scopes_supported: ['openid', 'email', 'profile']
  };

  res.json(metadata);
});
```

TypeScript:

typescript

```
import { Request, Response } from 'express';

// OAuth Authorization Server Metadata structure
interface OAuthServerMetadata {
  issuer: string;
  authorization_endpoint: string;
  token_endpoint: string;
  revocation_endpoint: string;
  registration_endpoint: string;
  response_types_supported: string[];
  grant_types_supported: string[];
  token_endpoint_auth_methods_supported: string[];
  code_challenge_methods_supported: string[];
  scopes_supported: string[];
}

// Authorization Server Metadata endpoint (RFC 8414)
app.get('/.well-known/oauth-authorization-server', (req: Request, res: Response) => {
  const metadata: OAuthServerMetadata = {
    issuer: JWT_ISSUER,
    authorization_endpoint: `${JWT_ISSUER}/oauth/authorize`,
    token_endpoint: `${JWT_ISSUER}/oauth/token`,
    revocation_endpoint: `${JWT_ISSUER}/oauth/revoke`,
    registration_endpoint: `${JWT_ISSUER}/register`,
    response_types_supported: ['code'],
    grant_types_supported: ['authorization_code', 'refresh_token'],
    token_endpoint_auth_methods_supported: ['client_secret_post'],
    code_challenge_methods_supported: ['S256'],
    scopes_supported: ['openid', 'email', 'profile']
  };

  res.json(metadata);
});
```

OAuth Protected Resource Metadata Endpoint

Resource Server Discovery (RFC 8414)

Implementation Approach

The OAuth Protected Resource Metadata endpoint advertises that your MCP server is a protected resource that requires OAuth authentication. This endpoint complements the Authorization Server Metadata by helping clients discover which authorization servers protect this resource. While optional, it improves OAuth ecosystem interoperability and helps clients automatically configure authentication.

Key considerations:

- Must be available at `/.well-known/oauth-protected-resource` path
- No authentication required (public discovery endpoint)
- Declares which authorization servers protect this resource (typically points to itself)
- Lists supported scopes for this resource
- Specifies bearer token methods supported (header, body, query)
- Helps clients understand authentication requirements

Pseudocode:

```
FUNCTION handle_protected_resource_metadata(request, response):
    metadata = {
        resource: JWT_ISSUER,
        authorization_servers: [JWT_ISSUER],
        scopes_supported: ["openid", "email", "profile"],
        bearer_methods_supported: ["header"]
    }

    response.JSON(metadata)
END FUNCTION

// Register endpoint
app.GET('/.well-known/oauth-protected-resource', handle_protected_resource_metadata)
```

JavaScript:

```
javascript
```

```
// OAuth Protected Resource Metadata endpoint (RFC 8414)
app.get('/.well-known/oauth-protected-resource', (req, res) => {
  const metadata = {
    resource: JWT_ISSUER,
    authorization_servers: [JWT_ISSUER],
    scopes_supported: ['openid', 'email', 'profile'],
    bearer_methods_supported: ['header']
  };

  res.json(metadata);
});
```

TypeScript:

```typescript
import { Request, Response } from 'express';

// OAuth Protected Resource Metadata structure
interface OAuthResourceMetadata {
  resource: string;
  authorization_servers: string[];
  scopes_supported: string[];
  bearer_methods_supported: string[];
}

// OAuth Protected Resource Metadata endpoint (RFC 8414)
app.get('/.well-known/oauth-protected-resource', (req: Request, res: Response) => {
  const metadata: OAuthResourceMetadata = {
    resource: JWT_ISSUER,
    authorization_servers: [JWT_ISSUER],
    scopes_supported: ['openid', 'email', 'profile'],
    bearer_methods_supported: ['header']
  };

  res.json(metadata);
});
```

Dynamic Client Registration (DCR)

Automated Client Onboarding (RFC 7591)

Implementation Approach

Dynamic Client Registration allows ServiceNow to programmatically register as an OAuth client without manual configuration. The DCR endpoint generates unique client credentials for each ServiceNow instance, enabling automated setup and multi-tenant deployments. This endpoint must be protected with a separate DCR authorization token to prevent unauthorized client registrations. Once registered, client credentials are persisted and survive server restarts.

Key considerations:

- Endpoint secured with DCR_AUTH_TOKEN (separate from OAuth tokens)
- Accepts DCR token in Authorization header or request body
- Generates cryptographically secure client_id and client_secret
- Stores client data in registry (file-based or database)
- Returns client credentials in RFC 7591 format
- Persist clients immediately (ServiceNow needs them for OAuth flow)
- Validate required fields (client_name, redirect_uris)

Storage Note: Client registration persistence is configured in **Part 2, Section 4.2: Client Registry Setup**. The storage mechanism (file-based, Firestore, PostgreSQL, etc.) was established there.

Pseudocode:

```
FUNCTION handle_dcr_registration(request, response):
  // Extract and validate DCR authorization token
  auth_header = request.headers.authorization
  token_from_body = request.body.token_value

  provided_token = auth_header.STARTS_WITH("Bearer ")
    ? auth_header.SUBSTRING(7)
    : token_from_body

  IF provided_token != DCR_AUTH_TOKEN:
    response.STATUS(401)
    RETURN response.JSON({
      error: "invalid_token",
      error_description: "Invalid DCR authorization token"
    })
  END IF

  // Extract registration details
  client_name = request.body.client_name
  redirect_uris = request.body.redirect_uris
  grant_types = request.body.grant_types OR ["authorization_code", "refresh_token"]
  use_pkce = request.body.use_pkce OR true

  // Validate required fields
  IF client_name IS NULL OR redirect_uris IS NULL:
    response.STATUS(400)
    RETURN response.JSON({
      error: "invalid_client_metadata",
      error_description: "Missing required fields"
    })
  END IF

  // Generate client credentials
  client_id = GENERATE_UUID()
  client_secret = GENERATE_SECURE_TOKEN(32)

  // Create client data object
  client_data = {
    clientId: client_id,
    clientSecret: client_secret,
    clientName: client_name,
    redirectUris: redirect_uris,
    grantTypes: grant_types,
        usePkce: use_pkce,
        createdAt: CURRENT_TIMESTAMP()
    }

    // Store client in registry
    REGISTER_CLIENT(client_id, client_data)
    PERSIST_CLIENTS_TO_STORAGE()

    LOG("Registered new client: " + client_id + " - " + client_name)

    // Return credentials (RFC 7591 format)
    response.STATUS(201)
    RETURN response.JSON({
        client_id: client_id,
        client_secret: client_secret,
        client_id_issued_at: CURRENT_TIMESTAMP_SECONDS(),
        client_secret_expires_at: 0,  // Never expires
        redirect_uris: redirect_uris,
        grant_types: grant_types,
        token_endpoint_auth_method: "client_secret_post"
    })
END FUNCTION

// Register endpoint with rate limiting
app.POST('/register', RATE_LIMITER, handle_dcr_registration)
```

JavaScript:

```
javascript
```

```
// Dynamic Client Registration endpoint (RFC 7591)
app.post('/register', oauthLimiter, (req, res) => {
  // Extract and validate DCR authorization token
  const authHeader = req.headers.authorization;
  const tokenFromBody = req.body.token_value;

  const providedToken = authHeader?.startsWith('Bearer ')
    ? authHeader.substring(7)
    : tokenFromBody;

  if (!providedToken || providedToken !== DCR_AUTH_TOKEN) {
    return res.status(401).json({
      error: 'invalid_token',
      error_description: 'Invalid or missing DCR authorization token'
    });
  }

  // Extract registration details
  const {
    client_name,
    redirect_uris,
    grant_types = ['authorization_code', 'refresh_token'],
    response_types = ['code'],
    token_endpoint_auth_method = 'client_secret_post',
    use_pkce = true
  } = req.body;

  // Validate required fields
  if (!client_name || !redirect_uris || !Array.isArray(redirect_uris)) {
    return res.status(400).json({
      error: 'invalid_client_metadata',
      error_description: 'Missing or invalid required fields: client_name, redirect_uris'
    });
  }

  // Generate client credentials
  const clientId = uuidv4();
  const clientSecret = generateSecureToken(32);

  // Create client data object
  const clientData = {
    clientId,
    clientSecret,
    clientName: client_name,
    redirectUris: redirect_uris,
    grantTypes: grant_types,
    responseTypes: response_types,
    tokenEndpointAuthMethod: token_endpoint_auth_method,
    usePkce: use_pkce,
    createdAt: new Date().toISOString()
  };

  // Store client in registry
  registeredClients.set(clientId, clientData);
  saveClientsToFile();  // Persist to storage

  console.log(`[DCR] Registered new client: ${clientId} - ${client_name}`);

  // Return credentials (RFC 7591 format)
  res.status(201).json({
    client_id: clientId,
    client_secret: clientSecret,
    client_id_issued_at: Math.floor(Date.now() / 1000),
    client_secret_expires_at: 0,  // Never expires
    redirect_uris: redirect_uris,
    grant_types: grant_types,
    response_types: response_types,
    token_endpoint_auth_method: token_endpoint_auth_method
  });
});
```

TypeScript:

```
typescript
```

```
import { Request, Response } from 'express';

// DCR request body structure
interface DcrRequest {
  client_name: string;
  redirect_uris: string[];
  grant_types?: string[];
  response_types?: string[];
  token_endpoint_auth_method?: string;
  use_pkce?: boolean;
  token_value?: string;  // Alternative DCR token location
}

// DCR response structure
interface DcrResponse {
  client_id: string;
  client_secret: string;
  client_id_issued_at: number;
  client_secret_expires_at: number;
  redirect_uris: string[];
  grant_types: string[];
  response_types: string[];
  token_endpoint_auth_method: string;
}

// Dynamic Client Registration endpoint (RFC 7591)
app.post('/register', oauthLimiter, (req: Request, res: Response) => {
  // Extract and validate DCR authorization token
  const authHeader: string | undefined = req.headers.authorization;
  const body = req.body as DcrRequest;
  const tokenFromBody: string | undefined = body.token_value;

  const providedToken: string | undefined = authHeader?.startsWith('Bearer ')
    ? authHeader.substring(7)
    : tokenFromBody;

  if (!providedToken || providedToken !== DCR_AUTH_TOKEN) {
    return res.status(401).json({
      error: 'invalid_token',
      error_description: 'Invalid or missing DCR authorization token'
    });
  }
```

```
// Extract registration details
const {
  client_name,
  redirect_uris,
  grant_types = ['authorization_code', 'refresh_token'],
  response_types = ['code'],
  token_endpoint_auth_method = 'client_secret_post',
  use_pkce = true
} = body;

// Validate required fields
if (!client_name || !redirect_uris || !Array.isArray(redirect_uris)) {
  return res.status(400).json({
    error: 'invalid_client_metadata',
    error_description: 'Missing or invalid required fields: client_name, redirect_uris'
  });
}

// Generate client credentials
const clientId: string = uuidv4();
const clientSecret: string = generateSecureToken(32);

// Create client data object
const clientData: ClientData = {
  clientId,
  clientSecret,
  clientName: client_name,
  redirectUris: redirect_uris,
  grantTypes: grant_types,
  responseTypes: response_types,
  tokenEndpointAuthMethod: token_endpoint_auth_method,
  usePkce: use_pkce,
  createdAt: new Date().toISOString()
};

// Store client in registry
registeredClients.set(clientId, clientData);
saveClientsToFile();  // Persist to storage

console.log(`[DCR] Registered new client: ${clientId} - ${client_name}`);

// Return credentials (RFC 7591 format)
const response: DcrResponse = {
  client_id: clientId,
  client_secret: clientSecret,
  client_id_issued_at: Math.floor(Date.now() / 1000),
  client_secret_expires_at: 0,  // Never expires
  redirect_uris: redirect_uris,
  grant_types: grant_types,
  response_types: response_types,
  token_endpoint_auth_method: token_endpoint_auth_method
};

res.status(201).json(response);
});
```

Authorization Endpoint

OAuth 2.1 Authorization Code Flow with PKCE

Implementation Approach

The authorization endpoint initiates the OAuth 2.1 flow by generating an authorization code that ServiceNow will exchange for tokens. This endpoint receives the PKCE code_challenge from ServiceNow, validates the client and redirect URI, generates a secure authorization code, and stores it along with the PKCE challenge for later verification. The endpoint then redirects ServiceNow back to its callback URL with the authorization code and state parameter.

Key considerations:

- HTTP GET endpoint (receives parameters in query string)
- Validates response_type must be "code"
- Validates client_id exists in registered clients
- Validates redirect_uri matches registered URIs
- PKCE parameters are mandatory (code_challenge and code_challenge_method=S256)
- Generates cryptographically secure authorization code (32+ bytes)
- Stores code with associated data (client, redirect_uri, scope, code_challenge, user_id)
- Authorization codes expire quickly (10 minutes recommended)
- Redirects back to ServiceNow with code and state
- Apply rate limiting to prevent authorization flooding

Storage Note: Authorization code storage is configured in **Part 2, Section 4.3: Authorization Code Storage**. Codes are typically stored in-memory (short-lived, single-use).

Pseudocode:

```
FUNCTION handle_authorization_request(request, response):
    // Extract query parameters
    response_type = request.query.response_type
    client_id = request.query.client_id
    redirect_uri = request.query.redirect_uri
    scope = request.query.scope OR "openid email profile"
    state = request.query.state
    code_challenge = request.query.code_challenge
    code_challenge_method = request.query.code_challenge_method

    // Validate response_type
    IF response_type != "code":
        RETURN REDIRECT(redirect_uri + "?error=unsupported_response_type&state=" + state)
    END IF

    // Validate required PKCE parameters
    IF client_id IS NULL OR redirect_uri IS NULL OR code_challenge IS NULL OR code_challenge_method != "S256":
        RETURN REDIRECT(redirect_uri + "?error=invalid_request&state=" + state)
    END IF

    // Validate client exists
    client = GET_CLIENT(client_id)
    IF client IS NULL:
        RETURN REDIRECT(redirect_uri + "?error=unauthorized_client&state=" + state)
    END IF

    // Validate redirect_uri is registered
    IF redirect_uri NOT IN client.redirectUris:
        response.STATUS(400)
        RETURN response.JSON({
            error: "invalid_request",
            error_description: "Invalid redirect_uri"
        })
    END IF

    // Simulate user authentication (M2M pattern - auto-approve)
    // NOTE: In M2M architecture, ServiceNow handles user authentication.
    // For implementations requiring user authentication at MCP server level,
    // replace this with actual authentication logic (login form, SSO, etc.)
    user_id = GENERATE_UUID()

    // Generate authorization code
    auth_code = GENERATE_SECURE_TOKEN(32)
    // Store authorization code with PKCE challenge
    STORE_AUTH_CODE(auth_code, {
        clientId: client_id,
        redirectUri: redirect_uri,
        scope: scope,
        codeChallenge: code_challenge,
        userId: user_id,
        created: CURRENT_TIMESTAMP()
    })

    LOG("Authorization code generated for client: " + client_id)

    // Redirect back to ServiceNow with code
    redirect_url = redirect_uri + "?code=" + auth_code + "&state=" + state
    RETURN REDIRECT(redirect_url)
END FUNCTION

// Register endpoint with rate limiting
app.GET('/oauth/authorize', RATE_LIMITER, handle_authorization_request)
```

JavaScript:

```
javascript
```

```
// Authorization endpoint - OAuth 2.1 with PKCE
app.get('/oauth/authorize', oauthLimiter, (req, res) => {
  // Extract query parameters
  const {
    response_type,
    client_id,
    redirect_uri,
    scope = 'openid email profile',
    state,
    code_challenge,
    code_challenge_method
  } = req.query;

  // Validate response_type
  if (response_type !== 'code') {
    return res.redirect(`${redirect_uri}?error=unsupported_response_type&state=${state}`);
  }

  // Validate required PKCE parameters
  if (!client_id || !redirect_uri || !code_challenge || code_challenge_method !== 'S256') {
    return res.redirect(`${redirect_uri}?error=invalid_request&state=${state}`);
  }

  // Validate client exists
  const client = registeredClients.get(client_id);
  if (!client) {
    return res.redirect(`${redirect_uri}?error=unauthorized_client&state=${state}`);
  }

  // Validate redirect_uri is registered
  if (!client.redirectUris.includes(redirect_uri)) {
    return res.status(400).json({
      error: 'invalid_request',
      error_description: 'Invalid redirect_uri'
    });
  }

  // Simulate user authentication (M2M pattern - auto-approve)
  const userId = uuidv4();

  // Generate authorization code
  const authCode = generateSecureToken(32);
  // Store authorization code with PKCE challenge
  authorizationCodes.set(authCode, {
    clientId: client_id,
    redirectUri: redirect_uri,
    scope: scope,
    codeChallenge: code_challenge,
    userId: userId,
    created: Date.now()
  });

  console.log(`[OAUTH] Authorization code generated for client: ${client_id}`);

  // Redirect back to ServiceNow with code
  const redirectUrl = `${redirect_uri}?code=${authCode}&state=${state}`;
  res.redirect(redirectUrl);
});
```

TypeScript:

```
typescript
```

```
import { Request, Response } from 'express';

// Authorization endpoint - OAuth 2.1 with PKCE
app.get('/oauth/authorize', oauthLimiter, (req: Request, res: Response) => {
  // Extract query parameters
  const {
    response_type,
    client_id,
    redirect_uri,
    scope = 'openid email profile',
    state,
    code_challenge,
    code_challenge_method
  } = req.query as {
    response_type?: string;
    client_id?: string;
    redirect_uri?: string;
    scope?: string;
    state?: string;
    code_challenge?: string;
    code_challenge_method?: string;
  };

  // Validate response_type
  if (response_type !== 'code') {
    return res.redirect(`${redirect_uri}?error=unsupported_response_type&state=${state}`);
  }

  // Validate required PKCE parameters
  if (!client_id || !redirect_uri || !code_challenge || code_challenge_method !== 'S256') {
    return res.redirect(`${redirect_uri}?error=invalid_request&state=${state}`);
  }

  // Validate client exists
  const client: ClientData | undefined = registeredClients.get(client_id);
  if (!client) {
    return res.redirect(`${redirect_uri}?error=unauthorized_client&state=${state}`);
  }

  // Validate redirect_uri is registered
  if (!client.redirectUris.includes(redirect_uri)) {
    return res.status(400).json({
      error: 'invalid_request',
      error_description: 'Invalid redirect_uri'
    });
  }

  // Simulate user authentication (M2M pattern - auto-approve)
  const userId: string = uuidv4();

  // Generate authorization code
  const authCode: string = generateSecureToken(32);

  // Store authorization code with PKCE challenge
  authorizationCodes.set(authCode, {
    clientId: client_id,
    redirectUri: redirect_uri,
    scope: scope,
    codeChallenge: code_challenge,
    userId: userId,
    created: Date.now()
  });

  console.log(`[OAUTH] Authorization code generated for client: ${client_id}`);

  // Redirect back to ServiceNow with code
  const redirectUrl: string = `${redirect_uri}?code=${authCode}&state=${state}`;
  res.redirect(redirectUrl);
});
```

Token Endpoint

Token Issuance and Refresh (OAuth 2.1)

Implementation Approach

The token endpoint is the core of OAuth 2.1 token issuance, handling two distinct grant types: authorization code exchange (converting codes to tokens) and refresh token rotation (obtaining new access tokens). Both grant types require client authentication and return JWT tokens. The endpoint validates different parameters for each grant type, performs PKCE validation for authorization codes, and implements refresh token rotation for security. All responses follow OAuth 2.1 format with access_token, refresh_token, token_type, and expires_in fields.

Key considerations:

- HTTP POST endpoint accepting both JSON and URL-encoded bodies
- Client authentication required for all grant types (client_id + client_secret)
- Two grant types supported: authorization_code and refresh_token
- PKCE validation mandatory for authorization_code grant (OAuth 2.1 requirement)
- Authorization codes are single-use (deleted after successful exchange)
- Refresh tokens rotate on each use (old token revoked, new token issued)
- Return consistent response format for both grant types
- Apply rate limiting to prevent token farming attacks

Authorization Code Grant

Exchange authorization code for access and refresh tokens with PKCE validation:

Pseudocode:

```
FUNCTION handle_authorization_code_grant(request, response):
    // Extract parameters
    code = request.body.code
    redirect_uri = request.body.redirect_uri
    client_id = request.body.client_id
    client_secret = request.body.client_secret
    code_verifier = request.body.code_verifier

    // Validate client credentials
    client = GET_CLIENT(client_id)
    IF client IS NULL OR client.clientSecret != client_secret:
        response.STATUS(401)
        RETURN response.JSON({
            error: "invalid_client",
            error_description: "Invalid client credentials"
        })
    END IF

    // Retrieve authorization code data
    auth_data = GET_AUTH_CODE(code)
    IF auth_data IS NULL:
        response.STATUS(400)
        RETURN response.JSON({
            error: "invalid_grant",
            error_description: "Invalid or expired authorization code"
        })
    END IF

    // Validate authorization code belongs to this client and redirect_uri
    IF auth_data.clientId != client_id OR auth_data.redirectUri != redirect_uri:
        DELETE_AUTH_CODE(code)
        response.STATUS(400)
        RETURN response.JSON({
            error: "invalid_grant",
            error_description: "Authorization code validation failed"
        })
    END IF

    // Validate PKCE
    IF NOT VALIDATE_PKCE(code_verifier, auth_data.codeChallenge):
        DELETE_AUTH_CODE(code)
        response.STATUS(400)
        RETURN response.JSON({
```

```
        error: "invalid_grant",
        error_description: "PKCE validation failed"
      })
    END IF

    // Delete authorization code (single-use)
    DELETE_AUTH_CODE(code)

    // Generate tokens
    access_token = CREATE_ACCESS_TOKEN(auth_data.userId, client_id, auth_data.scope)
    refresh_token = CREATE_REFRESH_TOKEN(auth_data.userId, client_id, auth_data.scope, rotation_count=0)

    LOG("Tokens issued for client: " + client_id)

    // Return tokens
    response.JSON({
      access_token: access_token,
      token_type: "Bearer",
      expires_in: ACCESS_TOKEN_LIFETIME,
      refresh_token: refresh_token,
      scope: auth_data.scope
    })
  END FUNCTION
```

Refresh Token Grant

Exchange refresh token for new access token with token rotation:

Security: Refresh Token Rotation

This implementation rotates refresh tokens on every use - the old refresh token is revoked and a new one is issued with an incremented `rotation_count`. This prevents replay attacks: if an attacker steals a refresh token, it becomes useless after the legitimate client uses it once. The rotation count also enables detection of token theft (multiple rotations from different sources indicate compromise).

Pseudocode:

```
FUNCTION handle_refresh_token_grant(request, response):
    // Extract parameters
    refresh_token = request.body.refresh_token
    client_id = request.body.client_id
    client_secret = request.body.client_secret

    // Validate client credentials
    client = GET_CLIENT(client_id)
    IF client IS NULL OR client.clientSecret != client_secret:
        response.STATUS(401)
        RETURN response.JSON({
            error: "invalid_client",
            error_description: "Invalid client credentials"
        })
    END IF

    // Validate refresh token
    decoded = VALIDATE_TOKEN(refresh_token)
    IF decoded IS NULL OR decoded.type != "refresh":
        response.STATUS(400)
        RETURN response.JSON({
            error: "invalid_grant",
            error_description: "Invalid refresh token"
        })
    END IF

    // Validate token belongs to this client
    IF decoded.client_id != client_id:
        response.STATUS(400)
        RETURN response.JSON({
            error: "invalid_grant",
            error_description: "Client mismatch"
        })
    END IF

    // Revoke old refresh token (add to token blacklist)
    // Storage configured in Part 2, Section 4.4: Token Revocation Storage
    token_expiry = decoded.exp - CURRENT_TIMESTAMP_SECONDS()
    REVOKE_TOKEN(decoded.jti, expires_in=token_expiry)

    // Generate new tokens with incremented rotation count
    new_access_token = CREATE_ACCESS_TOKEN(decoded.sub, client_id, decoded.scope)
    new_refresh_token = CREATE_REFRESH_TOKEN(
        decoded.sub,
        client_id,
        decoded.scope,
        rotation_count=(decoded.rotation_count OR 0) + 1
    )

    LOG("Tokens refreshed for client: " + client_id)

    // Return new tokens
    response.JSON({
        access_token: new_access_token,
        token_type: "Bearer",
        expires_in: ACCESS_TOKEN_LIFETIME,
        refresh_token: new_refresh_token,
        scope: decoded.scope
    })
END FUNCTION
```

Combined Token Endpoint Handler:

Pseudocode:

```
FUNCTION handle_token_endpoint(request, response):
    grant_type = request.body.grant_type

    // Route based on grant type
    IF grant_type == "authorization_code":
        RETURN handle_authorization_code_grant(request, response)
    ELSE IF grant_type == "refresh_token":
        RETURN handle_refresh_token_grant(request, response)
    ELSE:
        response.STATUS(400)
        RETURN response.JSON({
            error: "unsupported_grant_type",
            error_description: "Grant type not supported"
        })
    END IF
END FUNCTION

// Register endpoint with rate limiting
app.POST('/oauth/token', RATE_LIMITER, handle_token_endpoint)
```

JavaScript:

javascript

```
// Token endpoint - handles authorization_code and refresh_token grants
app.post('/oauth/token', oauthLimiter, (req, res) => {
  const {
    grant_type,
    code,
    redirect_uri,
    client_id,
    client_secret,
    code_verifier,
    refresh_token
  } = req.body;

  // Validate client credentials
  const client = registeredClients.get(client_id);
  if (!client || client.clientSecret !== client_secret) {
    return res.status(401).json({
      error: 'invalid_client',
      error_description: 'Invalid client credentials'
    });
  }

  // Handle authorization_code grant
  if (grant_type === 'authorization_code') {
    const authData = authorizationCodes.get(code);

    if (!authData) {
      return res.status(400).json({
        error: 'invalid_grant',
        error_description: 'Invalid or expired authorization code'
      });
    }

    // Validate authorization code
    if (authData.clientId !== client_id || authData.redirectUri !== redirect_uri) {
      authorizationCodes.delete(code);
      return res.status(400).json({
        error: 'invalid_grant',
        error_description: 'Authorization code validation failed'
      });
    }

    // Validate PKCE
    if (!validatePKCE(code_verifier, authData.codeChallenge)) {
```

```
    authorizationCodes.delete(code);
    return res.status(400).json({
      error: 'invalid_grant',
      error_description: 'PKCE validation failed'
    });
  }

  // Delete code (single-use)
  authorizationCodes.delete(code);

  // Generate tokens
  const accessToken = createAccessToken(authData.userId, client_id, authData.scope);
  const newRefreshToken = createRefreshToken(authData.userId, client_id, authData.scope);

  console.log(`[OAUTH] Tokens issued for client: ${client_id}`);

  return res.json({
    access_token: accessToken,
    token_type: 'Bearer',
    expires_in: ACCESS_TOKEN_LIFETIME,
    refresh_token: newRefreshToken,
    scope: authData.scope
  });
}

// Handle refresh_token grant
if (grant_type === 'refresh_token') {
  validateToken(refresh_token).then(decoded => {
    if (!decoded || decoded.type !== 'refresh') {
      return res.status(400).json({
        error: 'invalid_grant',
        error_description: 'Invalid refresh token'
      });
    }

    if (decoded.client_id !== client_id) {
      return res.status(400).json({
        error: 'invalid_grant',
        error_description: 'Client mismatch'
      });
    }

    // Revoke old refresh token
    const tokenExpiry = decoded.exp - Math.floor(Date.now() / 1000);
```

```
      revokeToken(decoded.jti, tokenExpiry > 0 ? tokenExpiry : 1);

      // Generate new tokens with rotation
      const newAccessToken = createAccessToken(decoded.sub, client_id, decoded.scope);
      const newRefreshToken = createRefreshToken(
        decoded.sub,
        client_id,
        decoded.scope,
        (decoded.rotation_count || 0) + 1
      );

      console.log(`[OAUTH] Tokens refreshed for client: ${client_id}`);

      return res.json({
        access_token: newAccessToken,
        token_type: 'Bearer',
        expires_in: ACCESS_TOKEN_LIFETIME,
        refresh_token: newRefreshToken,
        scope: decoded.scope
      });
    }).catch(error => {
      console.error('[OAUTH] Refresh token validation error:', error);
      return res.status(400).json({
        error: 'invalid_grant',
        error_description: 'Token validation failed'
      });
    });

    return;  // Exit early for async handling
  }

  // Unsupported grant type
  return res.status(400).json({
    error: 'unsupported_grant_type',
    error_description: 'Grant type not supported'
  });
});
```

TypeScript:

```
typescript
```

```
import { Request, Response } from 'express';

// Token request body structure
interface TokenRequest {
  grant_type: 'authorization_code' | 'refresh_token';
  code?: string;
  redirect_uri?: string;
  client_id: string;
  client_secret: string;
  code_verifier?: string;
  refresh_token?: string;
}

// Token response structure
interface TokenResponse {
  access_token: string;
  token_type: string;
  expires_in: number;
  refresh_token: string;
  scope: string;
}

// Token endpoint - handles authorization_code and refresh_token grants
app.post('/oauth/token', oauthLimiter, (req: Request, res: Response) => {
  const {
    grant_type,
    code,
    redirect_uri,
    client_id,
    client_secret,
    code_verifier,
    refresh_token
  } = req.body as TokenRequest;

  // Validate client credentials
  const client: ClientData | undefined = registeredClients.get(client_id);
  if (!client || client.clientSecret !== client_secret) {
    return res.status(401).json({
      error: 'invalid_client',
      error_description: 'Invalid client credentials'
    });
  }
```

```
// Handle authorization_code grant
if (grant_type === 'authorization_code') {
  const authData: AuthCodeData | undefined = authorizationCodes.get(code!);

  if (!authData) {
    return res.status(400).json({
      error: 'invalid_grant',
      error_description: 'Invalid or expired authorization code'
    });
  }

  // Validate authorization code
  if (authData.clientId !== client_id || authData.redirectUri !== redirect_uri) {
    authorizationCodes.delete(code!);
    return res.status(400).json({
      error: 'invalid_grant',
      error_description: 'Authorization code validation failed'
    });
  }

  // Validate PKCE
  if (!validatePKCE(code_verifier!, authData.codeChallenge)) {
    authorizationCodes.delete(code!);
    return res.status(400).json({
      error: 'invalid_grant',
      error_description: 'PKCE validation failed'
    });
  }

  // Delete code (single-use)
  authorizationCodes.delete(code!);

  // Generate tokens
  const accessToken: string = createAccessToken(authData.userId, client_id, authData.scope);
  const newRefreshToken: string = createRefreshToken(authData.userId, client_id, authData.scope);

  console.log(`[OAUTH] Tokens issued for client: ${client_id}`);

  const response: TokenResponse = {
    access_token: accessToken,
    token_type: 'Bearer',
    expires_in: ACCESS_TOKEN_LIFETIME,
    refresh_token: newRefreshToken,
    scope: authData.scope
  };

  return res.json(response);
}

// Handle refresh_token grant
if (grant_type === 'refresh_token') {
  validateToken(refresh_token!).then((decoded: JwtPayload | null) => {
    if (!decoded || decoded.type !== 'refresh') {
      return res.status(400).json({
        error: 'invalid_grant',
        error_description: 'Invalid refresh token'
      });
    }

    if (decoded.client_id !== client_id) {
      return res.status(400).json({
        error: 'invalid_grant',
        error_description: 'Client mismatch'
      });
    }

    // Revoke old refresh token
    const tokenExpiry: number = decoded.exp - Math.floor(Date.now() / 1000);
    revokeToken(decoded.jti, tokenExpiry > 0 ? tokenExpiry : 1);

    // Generate new tokens with rotation
    const newAccessToken: string = createAccessToken(decoded.sub, client_id, decoded.scope);
    const newRefreshToken: string = createRefreshToken(
      decoded.sub,
      client_id,
      decoded.scope,
      (decoded.rotation_count || 0) + 1
    );

    console.log(`[OAUTH] Tokens refreshed for client: ${client_id}`);

    const response: TokenResponse = {
      access_token: newAccessToken,
      token_type: 'Bearer',
      expires_in: ACCESS_TOKEN_LIFETIME,
      refresh_token: newRefreshToken,
      scope: decoded.scope
    };
```

```
        return res.json(response);
      }).catch((error: Error) => {
        console.error('[OAUTH] Refresh token validation error:', error);
        return res.status(400).json({
          error: 'invalid_grant',
          error_description: 'Token validation failed'
        });
      });

      return;  // Exit early for async handling
    }

    // Unsupported grant type
    return res.status(400).json({
      error: 'unsupported_grant_type',
      error_description: 'Grant type not supported'
    });
  });
```

Token Revocation Endpoint

Token Invalidation (RFC 7009)

Implementation Approach

The token revocation endpoint allows ServiceNow to invalidate tokens when disconnecting or detecting security issues. Following RFC 7009, this endpoint accepts either access or refresh tokens and adds their JWT ID (jti) to the Redis blacklist with a TTL matching the token's remaining lifetime. The endpoint always returns 200 OK regardless of whether the token was valid, preventing attackers from probing for valid tokens.

Key considerations:

- HTTP POST endpoint accepting both JSON and URL-encoded bodies
- Optional client authentication (client_id + client_secret)
- Accepts any token type (access or refresh)
- Decode JWT to extract jti claim (don't verify signature for revocation)
- Add jti to Redis blacklist with TTL matching token's natural expiration
- Always return 200 OK per RFC 7009 (even if token invalid or already revoked)

- Apply rate limiting to prevent revocation flooding
- Log revocation events for audit trail

Pseudocode:

```
FUNCTION handle_token_revocation(request, response):
    // Extract parameters
    token = request.body.token
    client_id = request.body.client_id
    client_secret = request.body.client_secret

    // Optional client authentication
    IF client_id IS NOT NULL AND client_secret IS NOT NULL:
        client = GET_CLIENT(client_id)
        IF client IS NULL OR client.clientSecret != client_secret:
            response.STATUS(401)
            RETURN response.JSON({ error: "invalid_client" })
        END IF
    END IF

    // Decode token to get jti (don't verify - we're revoking it anyway)
    TRY:
        decoded = JWT_DECODE(token)  // Decode without verification

        IF decoded.jti EXISTS:
            // Calculate token's remaining lifetime
            token_expiry = decoded.exp - CURRENT_TIMESTAMP_SECONDS()

            // Add to Redis blacklist with TTL
            REVOKE_TOKEN(decoded.jti, expires_in=MAX(token_expiry, 1))

            LOG("Token revoked: " + decoded.jti.SUBSTRING(0, 10) + "...")
        END IF
    CATCH decode_error:
        // Per RFC 7009, ignore errors and return 200 OK
        // Token might be malformed, already invalid, etc.
    END TRY

    // Always return 200 OK (RFC 7009 requirement)
    response.STATUS(200)
    RETURN response.JSON({})
END FUNCTION

// Register endpoint with rate limiting
app.POST('/oauth/revoke', RATE_LIMITER, handle_token_revocation)
```

JavaScript:

javascript

```javascript
// Token revocation endpoint (RFC 7009)
app.post('/oauth/revoke', oauthLimiter, async (req, res) => {
  const { token, client_id, client_secret } = req.body;

  // Optional client authentication
  if (client_id && client_secret) {
    const client = registeredClients.get(client_id);
    if (!client || client.clientSecret !== client_secret) {
      return res.status(401).json({
        error: 'invalid_client'
      });
    }
  }

  // Decode token to get jti (don't verify - we're revoking it anyway)
  try {
    const decoded = jwt.decode(token);

    if (decoded && decoded.jti) {
      // Calculate token's remaining lifetime
      const tokenExpiry = decoded.exp - Math.floor(Date.now() / 1000);

      // Add to Redis blacklist with TTL
      await revokeToken(decoded.jti, Math.max(tokenExpiry, 1));

      console.log(`[OAUTH] Token revoked: ${decoded.jti.substring(0, 10)}...`);
    }
  } catch (error) {
    // Per RFC 7009, ignore errors and return 200 OK
    // Token might be malformed, already invalid, etc.
  }

  // Always return 200 OK (RFC 7009 requirement)
  return res.status(200).json({});
});
```

TypeScript:

typescript

```
import { Request, Response } from 'express';
import jwt from 'jsonwebtoken';

// Token revocation request structure
interface RevokeRequest {
  token: string;
  client_id?: string;
  client_secret?: string;
}

// Token revocation endpoint (RFC 7009)
app.post('/oauth/revoke', oauthLimiter, async (req: Request, res: Response) => {
  const { token, client_id, client_secret } = req.body as RevokeRequest;

  // Optional client authentication
  if (client_id && client_secret) {
    const client: ClientData | undefined = registeredClients.get(client_id);
    if (!client || client.clientSecret !== client_secret) {
      return res.status(401).json({
        error: 'invalid_client'
      });
    }
  }

  // Decode token to get jti (don't verify - we're revoking it anyway)
  try {
    const decoded = jwt.decode(token) as JwtPayload | null;

    if (decoded && decoded.jti) {
      // Calculate token's remaining lifetime
      const tokenExpiry: number = decoded.exp - Math.floor(Date.now() / 1000);

      // Add to Redis blacklist with TTL
      await revokeToken(decoded.jti, Math.max(tokenExpiry, 1));

      console.log(`[OAUTH] Token revoked: ${decoded.jti.substring(0, 10)}...`);
    }
  } catch (error) {
    // Per RFC 7009, ignore errors and return 200 OK
    // Token might be malformed, already invalid, etc.
  }

  // Always return 200 OK (RFC 7009 requirement)
  return res.status(200).json({});
});
```

Authentication Middleware

Bearer Token Validation

Implementation Approach

Authentication middleware intercepts requests to protected endpoints, extracts the Bearer token from the Authorization header, validates the JWT, and attaches the decoded token data to the request object for use by route handlers. This middleware enforces authentication requirements transparently—route handlers simply check for req.user without implementing validation logic. The middleware supports both JWT tokens (primary) and API keys (development fallback).

Key considerations:

- Extract Bearer token from Authorization header
- Validate JWT signature, expiration, and revocation status
- Attach decoded token to request object (req.user) for route handlers
- Return 401 Unauthorized for missing or invalid tokens
- Support API key fallback for development/testing
- Middleware is async due to Redis revocation check
- Apply selectively to routes that require authentication (not initialize)
- Clear error messages for debugging authentication failures

Pseudocode:

```
FUNCTION authenticate_request(request, response, next):
    // Extract Authorization header
    auth_header = request.headers.authorization

    IF auth_header IS NULL OR NOT auth_header.STARTS_WITH("Bearer "):
        response.STATUS(401)
        RETURN response.JSON({
            error: "Unauthorized",
            message: "Missing or invalid Authorization header"
        })
    END IF

    // Extract token (remove "Bearer " prefix)
    token = auth_header.SUBSTRING(7)

    // Validate JWT token
    decoded = VALIDATE_TOKEN(token)  // Async - checks signature + Redis

    IF decoded IS NOT NULL:
        // JWT token is valid
        request.user = decoded
        RETURN next()  // Proceed to route handler
    END IF

    // Fallback: Check if it's a valid API key (development only)
    IF token == MCP_API_KEY:
        request.user = {
            sub: "api-key-user",
            client_id: "api-key-client"
        }
        RETURN next()
    END IF

    // Token is invalid
    response.STATUS(401)
    RETURN response.JSON({
        error: "Unauthorized",
        message: "Invalid or expired token"
    })
END FUNCTION

// Apply to protected routes:
app.POST('/mcp', FUNCTION(request, response):
    method = request.body.method

    IF method == "initialize":
        // Public - skip authentication
        RETURN handle_initialize(request, response)
    ELSE:
        // Protected - require authentication
        authenticate_request(request, response, FUNCTION():
            // Handle authenticated request
            HANDLE_MCP_METHOD(method, request, response)
        END FUNCTION)
    END IF
END FUNCTION)
```

JavaScript:

```
javascript
```

```
// Authentication middleware - validates Bearer tokens
function authenticateRequest(req, res, next) {
  // Extract Authorization header
  const authHeader = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return res.status(401).json({
      error: 'Unauthorized',
      message: 'Missing or invalid Authorization header'
    });
  }

  // Extract token (remove "Bearer " prefix)
  const token = authHeader.substring(7);

  // Validate JWT token (async due to Redis check)
  validateToken(token).then(decoded => {
    if (decoded) {
      // JWT token is valid
      req.user = decoded;
      return next();  // Proceed to route handler
    }

    // Fallback: Check if it's a valid API key (development only)
    if (token === MCP_API_KEY) {
      req.user = {
        sub: 'api-key-user',
        client_id: 'api-key-client'
      };
      return next();
    }

    // Token is invalid
    return res.status(401).json({
      error: 'Unauthorized',
      message: 'Invalid or expired token'
    });
  }).catch(error => {
    console.error('[AUTH] Authentication error:', error);
    return res.status(401).json({
      error: 'Unauthorized',
      message: 'Authentication failed'
    });
  });
}

// Example usage in MCP endpoint:
app.post('/mcp', mcpLimiter, async (req, res) => {
  const { method } = req.body;

  if (method === 'initialize') {
    // Public - skip authentication
    return handleInitialize(req, res);
  } else {
    // Protected - require authentication
    authenticateRequest(req, res, async () => {
      // req.user is now available with decoded token data
      // Handle authenticated MCP request
      switch (method) {
        case 'tools/list':
          return handleToolsList(req, res);
        case 'tools/call':
          return handleToolsCall(req, res);
        default:
          return res.status(200).json({
            jsonrpc: '2.0',
            error: { code: -32601, message: `Method not found: ${method}` },
            id: req.body.id
          });
      }
    });
  }
});
```

TypeScript:

```
typescript
```

```
import { Request, Response, NextFunction } from 'express';

// Extend Express Request type to include user property
declare global {
  namespace Express {
    interface Request {
      user?: JwtPayload | { sub: string; client_id: string };
    }
  }
}

// Authentication middleware - validates Bearer tokens
function authenticateRequest(req: Request, res: Response, next: NextFunction): void {
  // Extract Authorization header
  const authHeader: string | undefined = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    res.status(401).json({
      error: 'Unauthorized',
      message: 'Missing or invalid Authorization header'
    });
    return;
  }

  // Extract token (remove "Bearer " prefix)
  const token: string = authHeader.substring(7);

  // Validate JWT token (async due to Redis check)
  validateToken(token).then((decoded: JwtPayload | null) => {
    if (decoded) {
      // JWT token is valid
      req.user = decoded;
      return next();  // Proceed to route handler
    }

    // Fallback: Check if it's a valid API key (development only)
    if (token === MCP_API_KEY) {
      req.user = {
        sub: 'api-key-user',
        client_id: 'api-key-client'
      };
      return next();
    }
    // Token is invalid
    res.status(401).json({
      error: 'Unauthorized',
      message: 'Invalid or expired token'
    });
  }).catch((error: Error) => {
    console.error('[AUTH] Authentication error:', error);
    res.status(401).json({
      error: 'Unauthorized',
      message: 'Authentication failed'
    });
  });
}

// Example usage in MCP endpoint:
app.post('/mcp', mcpLimiter, async (req: Request, res: Response) => {
  const { method } = req.body;

  if (method === 'initialize') {
    // Public - skip authentication
    return handleInitialize(req, res);
  } else {
    // Protected - require authentication
    authenticateRequest(req, res, async () => {
      // req.user is now available with decoded token data
      // Handle authenticated MCP request
      switch (method) {
        case 'tools/list':
          return handleToolsList(req, res);
        case 'tools/call':
          return handleToolsCall(req, res);
        default:
          return res.status(200).json({
            jsonrpc: '2.0',
            error: { code: -32601, message: `Method not found: ${method}` },
            id: req.body.id
          });
      }
    });
  }
});
```

Part 4 Summary

This document has covered the complete OAuth 2.1 implementation for ServiceNow MCP integration:

✅ **OAuth 2.1 Architecture Overview** - Flow diagram, M2M pattern, PKCE rationale ✅ **Utility Functions** - Secure token generation, PKCE validation, Base64URL encoding ✅ **JWT Token Management** - Access token, refresh token creation and validation ✅ **Authorization Server Metadata** - RFC 8414 discovery endpoint ✅ **OAuth Protected Resource Metadata** - Resource server discovery ✅ **Dynamic Client Registration** - RFC 7591 automated client onboarding ✅ **Authorization Endpoint** - OAuth flow initiation with PKCE ✅ **Token Endpoint** - Code exchange and refresh token grants ✅ **Token Revocation Endpoint** - RFC 7009 token invalidation ✅ **Authentication Middleware** - Bearer token validation for protected routes

Implementation Notes:

The OAuth implementation follows OAuth 2.1 best practices with:

- Mandatory PKCE (S256 method) for all authorization code flows
- JWT tokens for stateless authentication
- Refresh token rotation to prevent token reuse
- Redis-based token revocation for persistent blacklist
- Rate limiting on all OAuth endpoints
- Comprehensive audit logging

This OAuth layer provides the security "roof" that protects your MCP server's tools and capabilities, ensuring only authenticated ServiceNow instances can access your resources.

Document Status

- **Part:** 4 of 5
- **Version:** 1.0
- **Last Updated:** January 30, 2026
- **Status:** Complete