OAuth 2.1 + PKCE Authentication Guide

OAuth 2.1 + PKCE Authentication Guide

Security-First Authentication Implementation

Modern OAuth 2.1 with PKCE (Proof Key for Code Exchange) implementation following RFC 7636 and security best practices.

Core OAuth 2.1 Client

PKCE Implementation

```python
import secrets
import base64
import hashlib
from typing import Tuple

class PKCEChallenge:
    """RFC 7636 compliant PKCE implementation."""

    @staticmethod
    def generate_challenge() -> Tuple[str, str]:
        """Generate code challenge and verifier pair."""
        # Generate cryptographically secure 64-byte verifier
        code_verifier = secrets.token_urlsafe(64)[:128]

        # Create S256 challenge from verifier
        code_challenge = base64.urlsafe_b64encode(
            hashlib.sha256(code_verifier.encode()).digest()
        ).decode().rstrip('=')

        return code_challenge, code_verifier
```

OAuth Client with Type Safety

```python
from pydantic import BaseModel, Field, HttpUrl
import time

class OAuth2Token(BaseModel):
    """OAuth 2.1 token with expiration tracking."""

    access_token: str
    token_type: str = "Bearer"
    expires_in: int = Field(default=3600, gt=0)
    refresh_token: str | None = None
    scope: str | None = None
    created_at: float = Field(default_factory=time.time)

    @property
    def is_expired(self) -> bool:
        """Check if token is expired with 60-second buffer."""
        return time.time() > self.created_at + self.expires_in - 60

class OAuth2Client:
```

```
    """Production OAuth 2.1 client with PKCE support."""

    def __init__(
        self,
        client_id: str,
        token_endpoint: HttpUrl,
        authorization_endpoint: HttpUrl,
        redirect_uri: HttpUrl,
        scope: str = "openid profile",
    ) -> None:
        self.client_id = client_id
        self.token_endpoint = str(token_endpoint)
        self.authorization_endpoint = str(authorization_endpoint)
        self.redirect_uri = str(redirect_uri)
        self.scope = scope
        self.http_client = httpx.AsyncClient()
```

Dynamic Client Registration

RFC 7591 Implementation

```python
class DynamicClientRegistration:
    """Automated OAuth client registration per RFC 7591."""

    async def register_client(
        self,
        registration_endpoint: str,
        server_info: MCPServerInfo
    ) -> ClientCredentials:
        """Register OAuth client dynamically."""
        registration_request = {
            "client_name": f"MCP Vacuum - {server_info.name}",
            "client_uri": "https://github.com/your-org/mcp-vacuum",
            "redirect_uris": [
                "http://localhost:8080/oauth/callback",
                "urn:ietf:wg:oauth:2.0:oob"
            ],
            "grant_types": ["authorization_code", "refresh_token"],
            "response_types": ["code"],
            "token_endpoint_auth_method": "none",  # Public client
            "scope": "mcp:tools mcp:resources mcp:prompts"
        }

        async with httpx.AsyncClient() as client:
            response = await client.post(
                registration_endpoint,
                json=registration_request,
                headers={"Content-Type": "application/json"}
            )

            if response.status_code != 201:
                raise RegistrationError(f"Registration failed: {response.text}")

            return ClientCredentials.parse_obj(response.json())
```

Secure Token Storage

Encrypted Storage Implementation

```python
from cryptography.fernet import Fernet
import keyring

class SecureTokenStorage:
    """Encrypted token storage using system keyring."""

    def __init__(self, service_name: str = "mcp_vacuum") -> None:
        self.service_name = service_name
        self.fernet = self._initialize_encryption()

    async def store_token(self, server_id: str, token: OAuth2Token) -> None:
        """Store encrypted token in keyring."""
        # Serialize and encrypt token
        token_data = token.json().encode()
        encrypted_data = self.fernet.encrypt(token_data)

        # Store encrypted token in keyring
        keyring.set_password(
            self.service_name,
            f"oauth2_token:{server_id}",
            base64.b64encode(encrypted_data).decode()
        )

    async def get_token(self, server_id: str) -> OAuth2Token | None:
        """Retrieve and decrypt token from keyring."""
        encrypted_data = keyring.get_password(
            self.service_name,
            f"oauth2_token:{server_id}"
        )

        if not encrypted_data:
            return None

        try:
            encrypted_bytes = base64.b64decode(encrypted_data)
            decrypted_data = self.fernet.decrypt(encrypted_bytes)
            return OAuth2Token.parse_raw(decrypted_data)
        except Exception:
            await self.delete_token(server_id)
            return None
```

Token Management & Auto-Refresh

Production Token Manager

```python
class TokenManager:
    """Automatic token management with refresh capabilities."""

    def __init__(
        self,
        oauth_client: OAuth2Client,
        storage: SecureTokenStorage
    ) -> None:
        self.oauth_client = oauth_client
```

```
        self.storage = storage
        self.token_cache: dict[str, OAuth2Token] = {}

    async def get_valid_token(self, server_id: str) -> OAuth2Token:
        """Get valid access token, refreshing if necessary."""
        # Check memory cache first
        cached_token = self.token_cache.get(server_id)
        if cached_token and not cached_token.is_expired:
            return cached_token

        # Load from secure storage
        stored_token = await self.storage.get_token(server_id)
        if not stored_token:
            raise AuthenticationError(f"No token found for server {server_id}")

        # Refresh if expired
        if stored_token.is_expired:
            refreshed_token = await self._refresh_token(stored_token)
            await self.storage.store_token(server_id, refreshed_token)
            self.token_cache[server_id] = refreshed_token
            return refreshed_token

        self.token_cache[server_id] = stored_token
        return stored_token
```

Security Best Practices

Implementation Guidelines

1. **PKCE Mandatory**: All authorization flows must use PKCE
2. **Short-lived Tokens**: Access tokens expire in 15-60 minutes
3. **Exact Redirect URI**: No pattern matching for security
4. **State Parameter**: CSRF protection for all flows
5. **Secure Storage**: Encrypt refresh tokens at rest
6. **Memory-only Access**: Never persist access tokens

Security Audit Points

```
class SecurityAuditor:
    """Security compliance checker for OAuth implementation."""

    def audit_oauth_flow(self, client: OAuth2Client) -> SecurityReport:
        """Audit OAuth implementation for security compliance."""
        issues = []

        # Check PKCE implementation
        if not self._validates_pkce_challenge(client):
            issues.append("PKCE challenge validation missing")

        # Check redirect URI validation
        if not self._validates_exact_redirect_uri(client):
            issues.append("Exact redirect URI validation missing")

        # Check token storage security
        if not self._validates_secure_storage(client.storage):
            issues.append("Token storage not encrypted")

```

```
        return SecurityReport(
            compliant=len(issues) == 0,
            issues=issues
        )
```

Next Steps

1. **Implement OAuth Client**: Start with basic OAuth 2.1 + PKCE flow
2. **Add Dynamic Registration**: Implement RFC 7591 client registration
3. **Secure Storage**: Add encrypted token storage with keyring
4. **Token Management**: Implement automatic refresh capabilities
5. **Security Audit**: Validate implementation against security checklist