PKCE in OAuth2.1

Proof Key for Code Exchange

Understanding OAuth's Critical Security Extension

The Problem

OAuth's authorization code flow has a vulnerability:

Authorization codes can be intercepted

- Codes travel through the browser via redirect URIs
- Malware or compromised apps can capture them
- Mobile apps are especially vulnerable
- Once intercepted, attacker can exchange code for tokens

This is called an **authorization code interception attack**

Attack Scenario

1. User initiates OAuth flow

Legitimate app requests authorization

2. User approves

Authorization server generates code

3. Attacker intercepts redirect

Malicious app captures code from redirect URI

4. Attacker exchanges code

Uses stolen code to get access token

5. Unauthorized access

Attacker accesses protected resources

Why Client Secrets Don't Help

Traditional OAuth 2.0 solution:

Use a client secret when exchanging the authorization code for a token

Problem: Public clients can't keep secrets

- Mobile apps: Secret embedded in app binary (easily extracted)
- Single-page apps: Secret visible in JavaScript
- Desktop apps: Secret stored on user's machine
- Any attacker with the app has the secret

PKCE: The Elegant Solution

Core Idea:

Create a dynamic, one-time secret for each authorization request that only the legitimate client knows

Code Verifier

Random secret generated by client (kept private)

Code Challenge

Hash of verifier sent in auth request (public)

Authorization server verifies the client by checking verifier matches challenge

How PKCE Works: High Level

Before authorization request:

Client generates random code verifier and creates challenge from it

During authorization request:

Client sends code challenge (not the verifier) to auth server

Authorization server stores:

Links the authorization code with the code challenge

During token exchange:

Client sends code verifier (original secret)

Authorization server validates:

Hashes verifier and compares to challenge

Step 1: Generate Code Verifier

Requirements:

- Cryptographically random string
- 43 to 128 characters long
- Characters: A-Z, a-z, 0-9, -, ., _, ~
- Generated fresh for each authorization request

Example:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

Store securely - never send in auth request

Step 2: Create Code Challenge

Transform the verifier:

Apply SHA-256 hash and Base64URL encode

challenge = BASE64URL(SHA256(verifier))

Code challenge method:

S256 (SHA-256) - recommended and required by OAuth 2.1

There's also "plain" method (verifier = challenge) but it's deprecated

Result:

`E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM`

Step 3: Authorization Request

Send the code challenge and method to the authorization server:

```
https://auth.example.com/authorize?
 response_type=code&
 client_id=YOUR_CLIENT_ID&
 redirect_uri=https://app.example.com/callback&
 scope=openid profile&
 code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM&
 code_challenge_method=S256
```

The authorization server stores the challenge with the authorization code it will issue

Step 4: Token Exchange with Verifier

Exchange authorization code for tokens, including the verifier:

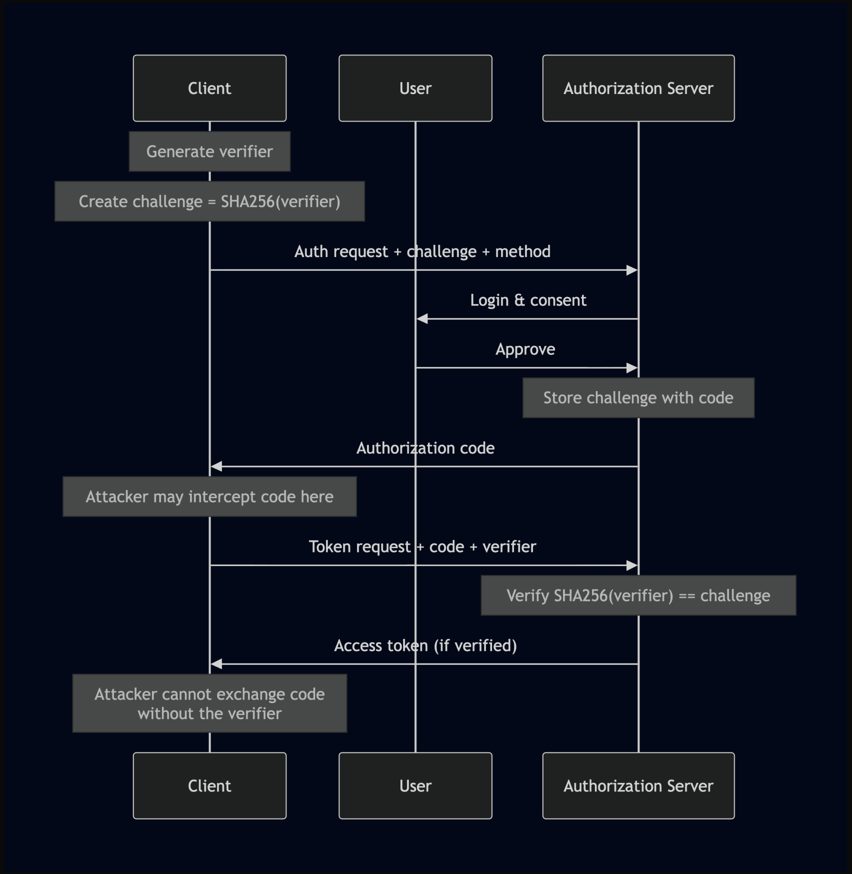
```
POST https://auth.example.com/token


 grant_type=authorization_code&
 code=AUTHORIZATION_CODE&
 redirect_uri=https://app.example.com/callback&
 client_id=YOUR_CLIENT_ID&
 code_verifier=...
```

Server validates:

SHA256(verifier) == stored challenge

Only the legitimate client has the verifier that produces the challenge

Complete PKCE Flow



What PKCE Prevents

Authorization Code Interception

Intercepted code is useless without the verifier

Authorization Code Injection

Attacker cannot inject their own code into victim's flow

Malicious Apps on Same Device

Each app generates unique verifier per request

Replay Attacks

Authorization code bound to specific challenge/verifier pair

Implementation Best Practices

Use cryptographically secure random

secrets.token_bytes() in Python, crypto.randomBytes() in Node.js

Always use S256 method

Plain method is deprecated and insecure

Store verifier securely during flow

Memory, secure storage, or session - never expose in URLs

Generate fresh verifier per request

Never reuse verifiers across authorization attempts

Use OAuth libraries

Most modern OAuth libraries handle PKCE automatically

Test with interception scenarios

Verify that intercepted codes cannot be exchanged

Key Takeaways

PKCE solves a fundamental problem:

Public clients can't keep static secrets, but PKCE creates dynamic, one-time secrets

How it works:

- Client generates random verifier
- Creates challenge by hashing verifier
- Sends challenge in auth request
- Sends verifier in token request
- Server validates they match

Required for:

All OAuth 2.1 clients - public and confidential