Authorization Code with PKCE

Overview

PKCE (Proof Key for Code Exchange) is an extension to the OAuth 2.0 Authorization Code Grant (RFC 7636) that provides enhanced security, especially for public clients like single-page applications (SPAs) and mobile apps that cannot securely store a client_secret. PKCE is now recommended by OAuth 2.0 best practices and is already supported in Genesys Cloud.

Status: Implicit Grant deprecated (May 2027 deadline), PKCE is the secure replacement.

Why PKCE?

The Problem PKCE Solves:

Authorization Code Can Be Intercepted:
├─ Attacker monitors network traffic
├─ Captures authorization code
├─ Attempts to exchange code for token
└─ Without PKCE: Attacker succeeds

Public Clients Cannot Store Secrets:
├─ Browser-based apps: No server backend
├─ Mobile apps: Can be reverse engineered
├─ Desktop apps: Can be analyzed
├─ Cannot safely store client_secret
└─ Traditional approach inadequate

PKCE Solution:

Add Proof to Authorization Code:
├─ Generate random code_verifier
├─ Compute code_challenge (hash)
├─ Send code_challenge to auth server
├─ Auth server stores it
├─ Only original verifier can exchange code
└─ Attacker lacks verifier → cannot exchange

Proof Cannot Be Reversed:
├─ code_challenge = SHA256(code_verifier)
├─ Hash is one-way function
├─ Cannot reverse-engineer verifier from hash
├─ Even if code intercepted: useless without verifier
└─ Verifier never sent over network

Complete PKCE Flow

Step 1: Generate Proof Strings

Your Application Generates:

code_verifier:
├─ Random string, 43-128 characters
├─ Cryptographically secure (use crypto random)
├─ Unrepeatable (different each request)
├─ Only stored in memory
├─ Example: "E9Mrozoa2owusvxrFHo89ejyK3OMVZZWhtbQrHfl"

code_challenge:
├─ SHA256 hash of code_verifier
├─ BASE64-URL encoded
├─ Sent to authorization server
├─ Example: "47DEQpj8HBSa-_TImW-5JCeuQeRkm5NMpJWZG3hSuFU"

code_challenge_method:
├─ "S256" (SHA256 hash)

├─ Only recommended method
├─ "plain" exists but deprecated
└─ Always use "S256"

Pseudo Code:

```
code_verifier = generateRandomString(128)
code_challenge = BASE64URL(SHA256(code_verifier))
code_challenge_method = "S256"
```

Step 2: Redirect to Authorization Endpoint

Redirect user browser to:

```
https://login.mypurecloud.com/oauth/authorize
  ?client_id=YOUR_CLIENT_ID
  &response_type=code
  &redirect_uri=https://yourapp.com/callback
  &scope=conversations:readonly
  &code_challenge=47DEQpj8HBSa-_TImW-5JCeuQeRkm5NMpJWZG3hSuFU
  &code_challenge_method=S256
  &state=random_state_string
```

Parameters:

client_id:
├─ Your app's public identifier
└─ Can be embedded in SPA code

response_type:
├─ Must be "code"
└─ Returns authorization code

redirect_uri:
├─ Where user is redirected
├─ Can be embedded in SPA code
├─ Example: https://yourapp.com/callback
└─ Must be registered in OAuth client

scope:
├─ Requested permissions
├─ Space-separated list
└─ Example: "conversations:readonly"

code_challenge (PKCE):
├─ SHA256 hash of random string
├─ Sent to auth server
├─ Auth server stores it
└─ Cannot derive original verifier

code_challenge_method (PKCE):
├─ "S256" (recommended and required)
├─ Indicates SHA256 method used
└─ Only secure method

state (CSRF Protection):
├─ Random string
├─ Prevents CSRF attacks
└─ Verified in callback

Step 3: User Authenticates & Consents

Same as Authorization Code Grant:

1. User sees Genesys Cloud login
2. User enters credentials
3. User sees permission consent screen
4. User grants permission
5. Auth server generates authorization code
6. Auth server stores code_challenge with authorization code

Step 4: Callback with Authorization Code

Auth server redirects to callback:

https://yourapp.com/callback
?code=AUTH_CODE_12345abcde67890
&state=random_state_string

In Callback Handler:

1. Verify state parameter (CSRF check)
2. Retrieve authorization code
3. Verify you have code_verifier in memory
4. Proceed to Step 5 (exchange code)

Never:
├─ Do NOT send code_verifier in callback
├─ Do NOT include code_verifier in URL
├─ Do NOT exchange code in browser
└─ Do NOT expose code_verifier to user

Step 5: Exchange Code with PKCE Proof

Now you have:
├─ Authorization code (from Step 4)
├─ code_verifier (generated in Step 1, stored in memory)
└─ client_id (public, stored in app)

Exchange Code:

POST https://login.mypurecloud.com/oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=AUTH_CODE_12345abcde67890
&client_id=YOUR_CLIENT_ID
&redirect_uri=https://yourapp.com/callback
&code_verifier=E9Mrozoa2owusvxrFHo89ejyK3OMVZZWhtbQrHfl

Parameters:

grant_type:

├─ "authorization_code"
└─ Standard OAuth parameter

code:
├─ Authorization code from Step 4
└─ Single-use only

client_id:
├─ Your public client ID
└─ Can be public

redirect_uri:
├─ Must match redirect_uri from Step 2
└─ Confirms code ownership

code_verifier (PKCE):
├─ Original random string from Step 1
├─ Proves you own the authorization code
├─ MUST match the code_challenge sent in Step 2
└─ Server recomputes SHA256(code_verifier) and compares

NOTE: NO client_secret needed!
├─ PKCE replaces need for client_secret
├─ Perfect for public clients
├─ Proof of ownership is cryptographic
└─ Signature is binding

Step 6: Server Validates & Issues Token

Genesys Cloud Authorization Server:

1. Receives code_verifier
2. Retrieves authorization code from storage
3. Retrieves stored code_challenge
4. Computes: SHA256(code_verifier) → new_hash
5. Compares: new_hash == stored_code_challenge?

If MATCH (✓):

├─ code_verifier is correct
├─ Same application that requested code
├─ Issue access token
└─ Return token in response

If NO MATCH (✗):
├─ code_verifier is wrong
├─ Likely code theft attempt
├─ Reject request with error
└─ Attack prevented!

Response on Success:

HTTP 200 OK
{
  "access_token": "abc123xyz789...",
  "token_type": "bearer",
  "expires_in": 3600,
  "scope": "conversations:readonly"
}

Response on Failure:

HTTP 400 Bad Request
{
  "error": "invalid_grant",
  "error_description": "code_verifier invalid"
}

Step 7: Use Access Token

Same as Authorization Code Grant:

GET /api/v2/conversations
Authorization: Bearer abc123xyz789...

Response:
HTTP 200 OK
{...conversation data...}

JavaScript Implementation Example

```
// Configuration
const CLIENT_ID = 'your_client_id';
const REDIRECT_URI = 'https://yourapp.com/callback';
const SCOPES = 'conversations:readonly users:readonly';
const GENESYS_REGION = 'mypurecloud.com';

// Step 1: Generate PKCE code challenge
async function generatePKCE() {
  // Generate random code_verifier
  const array = new Uint8Array(64);
  crypto.getRandomValues(array);
  const codeVerifier = btoa(String.fromCharCode.apply(null, array))
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=/g, '');

  // Compute code_challenge = BASE64URL(SHA256(code_verifier))
  const encoder = new TextEncoder();
  const data = encoder.encode(codeVerifier);
  const hashBuffer = await crypto.subtle.digest('SHA-256', data);

  const hashArray = Array.from(new Uint8Array(hashBuffer));
  const hashString = String.fromCharCode.apply(null, hashArray);
  const codeChallenge = btoa(hashString)
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=/g, '');

  return { codeVerifier, codeChallenge };
}

// Step 2: Redirect to authorization
async function login() {
```

```
  const { codeVerifier, codeChallenge } = await generatePKCE();

  // Store verifier in memory (or sessionStorage for SPA)
  sessionStorage.setItem('pkce_verifier', codeVerifier);

  // Generate state for CSRF protection
  const state = btoa(Math.random().toString()).substring(0, 40);
  sessionStorage.setItem('pkce_state', state);

  // Redirect to authorization endpoint
  const authUrl = new URL(`https://login.${GENESYS_REGION}/oauth/authorize`);
  authUrl.searchParams.append('client_id', CLIENT_ID);
  authUrl.searchParams.append('response_type', 'code');
  authUrl.searchParams.append('redirect_uri', REDIRECT_URI);
  authUrl.searchParams.append('scope', SCOPES);
  authUrl.searchParams.append('code_challenge', codeChallenge);
  authUrl.searchParams.append('code_challenge_method', 'S256');
  authUrl.searchParams.append('state', state);

  window.location.href = authUrl.toString();
}

// Step 4: Handle callback
async function handleCallback() {
  const urlParams = new URLSearchParams(window.location.search);
  const code = urlParams.get('code');
  const state = urlParams.get('state');
  const error = urlParams.get('error');

  // Check for errors
  if (error) {
    console.error('Authorization error:', error);
    return false;
  }

  // Verify state (CSRF protection)
  const storedState = sessionStorage.getItem('pkce_state');
  if (state !== storedState) {
    console.error('State mismatch - CSRF attack detected');
    return false;
```

```
}

// Retrieve code_verifier from memory
const codeVerifier = sessionStorage.getItem('pkce_verifier');
if (!codeVerifier) {
  console.error('Code verifier not found');
  return false;
}

// Step 5: Exchange code for token
try {
  const response = await fetch(`https://login.${GENESYS_REGION}/oauth/token`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded'
    },
    body: new URLSearchParams({
      grant_type: 'authorization_code',
      code: code,
      client_id: CLIENT_ID,
      redirect_uri: REDIRECT_URI,
      code_verifier: codeVerifier  // PKCE proof
    })
  });

  if (!response.ok) {
    throw new Error('Token exchange failed');
  }

  const { access_token } = await response.json();

  // Step 7: Store token and use it
  sessionStorage.setItem('access_token', access_token);

  // Clean up PKCE values
  sessionStorage.removeItem('pkce_verifier');
  sessionStorage.removeItem('pkce_state');

  console.log('Login successful');
  return true;
```

```
  } catch (error) {
    console.error('Token exchange error:', error);
    return false;
  }
}

// Use access token
async function callAPI(endpoint) {
  const token = sessionStorage.getItem('access_token');

  if (!token) {
    console.error('No access token found');
    return null;
  }

  try {
    const response = await fetch(`https://api.${GENESYS_REGION}/api/v2${endpoint}`, {
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json'
      }
    });

    if (response.status === 401) {
      console.error('Token expired, user must log in again');
      login();
      return null;
    }

    if (!response.ok) {
      throw new Error(`API error: ${response.status}`);
    }

    return await response.json();

  } catch (error) {
    console.error('API call failed:', error);
    return null;
  }
```

```
}

// Example: Login button click handler
document.getElementById('loginBtn').addEventListener('click', login);

// Example: Handle callback on return from auth server
if (window.location.pathname === '/callback') {
  handleCallback().then(success => {
    if (success) {
      window.location.href = '/dashboard';
    } else {
      window.location.href = '/error';
    }
  });
}

// Example: Call API
async function getConversations() {
  const data = await callAPI('/conversations?pageSize=100&pageNumber=1');
  console.log('Conversations:', data);
}
```

Why Migrate from Implicit Grant to PKCE?

```
Implicit Grant (DEPRECATED):
├─ Status: Deprecated November 2025
├─ New clients: Blocked March 2026
├─ Existing clients: Must migrate by May 2027
├─ Issues: Token in URL, browser history, no protection
└─ Replacement: PKCE

PKCE (RECOMMENDED):
├─ Status: Supported and recommended
├─ Security: Enhanced via proof mechanism
├─ Suitable for: All client types
```

├─ Implementation: Slightly more complex
└─ Future-proof: Long-term standard

Migration Path:

Step 1: Update OAuth Client
├─ Delete old Implicit Grant client (or keep if needed)
├─ Create new Authorization Code + PKCE client
└─ Note new client_id

Step 2: Update Application Code
├─ Implement PKCE proof generation
├─ Add code_challenge to authorization
├─ Include code_verifier in token exchange
└─ Remove implicit grant references

Step 3: Test Thoroughly
├─ Test login flow end-to-end
├─ Test token exchange
├─ Test API calls
└─ Verify error handling

Step 4: Deploy
├─ Update production code
├─ Verify working in production
├─ Monitor for errors
└─ Keep old client available briefly

Step 5: Cleanup
├─ Remove old Implicit Grant client
├─ Update documentation
├─ Notify users if applicable
└─ Archive old implementation

Timeline:
├─ Now (March 2026): Implement PKCE
├─ Before May 2027: Migration required
├─ May 2027: Implicit Grant stopped working
└─ Plan ahead to avoid outages!

PKCE vs Implicit Grant

Security Comparison:

Token Exposure:
├─ Implicit: Token in URL fragment (visible)
├─ PKCE: Token in body, 200 response (hidden)
└─ PKCE wins: Less exposed

Token Lifetime:
├─ Implicit: No refresh, static lifetime
├─ PKCE: Can have refresh tokens
└─ PKCE wins: Better UX

Browser History:
├─ Implicit: Token in URL history (risk)
├─ PKCE: No URL tokens (safe)
└─ PKCE wins: No history exposure

Code Interception:
├─ Implicit: Not applicable (no code)
├─ PKCE: Protected by proof (secure)
└─ PKCE wins: Protected exchange

CSRF Protection:
├─ Implicit: State parameter only
├─ PKCE: State + cryptographic proof
└─ PKCE wins: Multiple protections

Standards Alignment:
├─ Implicit: Deprecated OAuth 2.0
├─ PKCE: Modern OAuth 2.0 best practice
└─ PKCE wins: Future-proof

Complexity:
├─ Implicit: Simple (but insecure)
├─ PKCE: Slightly more complex (much more secure)

└─ Trade-off: Worth the extra complexity

Security Checklist for PKCE

PKCE Implementation Security:

□ Generate cryptographically secure code_verifier
  ├─ Use crypto.getRandomValues() (not Math.random)
  ├─ 64 bytes minimum (not shorter)
  └─ Different every request

□ Compute SHA256 hash of verifier
  ├─ Use crypto.subtle.digest()
  ├─ Encode to BASE64URL format
  └─ Do not use plain method

□ Store code_verifier securely
  ├─ Memory only (not localStorage)
  ├─ Clear after token exchange
  ├─ Not persisted
  └─ Lost on page reload

□ Use S256 method always
  ├─ Never use "plain" method
  ├─ Only S256 recommended
  └─ Specify in authorization

□ Include state parameter
  ├─ Separate CSRF protection
  ├─ Random and unpredictable
  ├─ Verify in callback
  └─ Different from verifier

□ Validate SSL certificates
  ├─ HTTPS always
  ├─ Check cert validity
  └─ Reject self-signed

□ Do not expose secrets

├─ code_verifier: Memory only

├─ access_token: SessionStorage or memory

├─ Never in localStorage

└─ Clean up after logout

□ Handle errors gracefully

├─ Catch network errors

├─ Retry on failure

├─ Don't expose verifier in errors

└─ Log safely

Key Takeaways: Chapter 4

- **Enhanced Security** - Cryptographic proof prevents authorization code interception
- **No Client Secret** - Suitable for public clients (SPAs, mobile, desktop)
- **Proof Mechanism** - Verifier prevents code theft (cannot reverse-engineer from hash)
- **RFC 7636 Standard** - Modern OAuth 2.0 best practice
- **Implicit Replacement** - Use PKCE instead of deprecated Implicit Grant
- **Migration Deadline** - May 2027 cutoff for Implicit Grant
- **Slightly Complex** - More code than Implicit, but much more secure
- **Future-Proof** - Long-term standard, recommended by OAuth 2.0 experts

---

Interview Prep: PKCE

Question	Answer
What is PKCE?	Proof Key for Code Exchange - enhanced OAuth Code grant security
Why PKCE needed?	Prevents authorization code interception attacks
code_verifier?	Random string (43-128 chars) generated by client, never sent over network
code_challenge?	SHA256(code_verifier), BASE64URL encoded, sent to auth server
How prevent intercept?	Only original code_verifier can exchange the code, attacker lacks verifier

Question	Answer
Why not reverse?	SHA256 is one-way hash function, cannot derive verifier from challenge
S256 vs plain?	S256 (SHA256) is secure, plain is deprecated, always use S256
When use PKCE?	Public clients (SPAs, mobile, desktop) that cannot store client_secret
Migration deadline?	May 2027 for Implicit Grant existing clients
State parameter?	Separate CSRF protection, still needed with PKCE

Document Version

Chapter: 4 of 8
Last Updated: March 2026
Status: Current with RFC 7636
Scope: PKCE Flow, Security, Implementation, Migration from Implicit

Revision #1
Created 14 March 2026 19:32:10 by Cesar Gzz
Updated 14 March 2026 19:32:29 by Cesar Gzz