Using PKCE to make Zendesk OAuth access tokens more secure

ON THIS PAGE

This article explains how to add Proof Key for Code Exchange (PKCE) to the Zendesk authorization code grant flow. The authorization code grant flow is used by developers to get OAuth access tokens to authenticate Zendesk API requests. PKCE adds an additional layer of security to the grant flow, reducing the risk of attackers getting access tokens.

Note: This article applies only to the authorization code grant flow for the Zendesk Ticketing, Help Center, and Voice APIs. PKCE is not supported in the authorization code grant flows for Chat, Conversations, or Sell.

Updating the grant flow to add PKCE

The authorization code grant flow consists of four steps (see Authorization code grant flow in Zenbdesk help). To enable PKCE, you must add three additional properties to the HTTP requests in steps 1 and 3 of the grant flow.

Updating step 1 of the grant flow

In the first step of the grant flow, you define a URL that the user clicks to go to the Zendesk authorization page. The URL includes a number of required query parameters. See Step 1 - Send the user to the Zendesk authorization page in Zendesk help.

To enable PKCE, specify the following two additional query parameters for the URL:

- `code_challenge` - A string representing a code challenge derived from a code verifier. To create the string, see Generating the code_challenge value.
- `code_challenge_method` - The method used to derive the code challenge. Specify "S256" as the value.

Example:

```
// generate the code_challenge value
const codeVerifier = getCodeVerifier();
const codeChallenge = generateCodeChallenge(codeVerifier);

app.get("/zendesk/auth", (req, res) => {
  const queryParams = querystring.stringify({
    response_type: "code",
    redirect_uri: REDIRECT_URI,
    client_id: ZENDESK_CLIENT_ID,
    scope: "read",
    code_challenge_method: "S256",
    code_challenge: codeChallenge
  })

  res.redirect(
    `https://${ZENDESK_SUBDOMAIN}.zendesk.com/oauth/authorizations/new?${queryParams}`
  )
})
```

Updating step 3 of the grant flow

In step 3 of the authorization code grant flow, you make a POST request to the following Zendesk endpoint:

```
https://{subdomain}.zendesk.com/oauth/tokens
```

The request body includes a number of required properties, including the authorization code you received after the user granted access. See Step 3 - Get an access token from Zendesk in Zendesk help.

To enable PKCE, specify the following additional property in the body of the request:

- `code_verifier` - The string you used to generate the `code_challenge` value in step 1. See Generating the code_challenge value.

Zendesk uses the `code_verifier` value to validate the `code_challenge` value sent with the URL in step 1. Even if an attacker intercepts the authorization code or the `code_challenge` value from the URL in step 1, they won't be able to obtain an access token without the `code_verifier` value.

Generating the code_challenge value

You should generate new `code_verifier` and `code_challenge` values each time your application requests an access token.

The `code_challenge` value is derived from a `code_verifier` string. The `code_verifier` string should be between 43 to 128 characters in length. One approach to creating the `code_verifier` string is to generate a random string and Base64-encode it.

The `code-challenge` value is generated from the `code_verifier` string using the following operation:

```
BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))
```

The following Python, JavaScript, and Ruby examples generate `code_verifier` and `code_challenge` values. Most languages also have libraries and packages for generating these PKCE values.

Python

```python
# length is the number of characters to use
verifier_base = os.urandom(length)
code_verifier = base64.urlsafe_b64encode(verifier_base).rstrip(b'=').decode('utf-8')

hashed_verifier = hashlib.sha256(code_verifier.encode('utf-8')).digest()
code_challenge = base64.urlsafe_b64encode(hashed_verifier).rstrip(b'=').decode('utf-8')
```

JavaScript

```javascript
// Function to generate a random string of given length
function generateRandomString(length) {
  const array = new Uint8Array(length);
  window.crypto.getRandomValues(array);
  return btoa(String.fromCharCode.apply(null, array))
```

```
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=+$/, '');
}

// Function to generate SHA-256 hash and encode it in base64
async function sha256(base64String) {
  const encoder = new TextEncoder();
  const data = encoder.encode(base64String);
  const hash = await window.crypto.subtle.digest('SHA-256', data);
  let binary = '';
  const bytes = new Uint8Array(hash);
  for (let i = 0; i < bytes.byteLength; i++) {
    binary += String.fromCharCode(bytes[i]);
  }
  return btoa(binary)
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=+$/, '');
}


const codeVerifier = generateRandomString(length);
const codeChallenge = await sha256(codeVerifier);
```

Ruby

```
# length is the number of characters to use
verifier_base = SecureRandom.random_bytes(length)
code_verifier = Base64.urlsafe_encode64(verifier_base).delete_suffix('=')

hashed_verifier = OpenSSL::Digest::SHA256.digest(code_verifier)
code_challenge = Base64.urlsafe_encode64(hashed_verifier).delete_suffix('=')
```

Using PKCE to migrate from the implicit grant flow

Using the implicit grant flow to get access tokens is no longer recommended because of security concerns. Instead, using the authorization code grant flow with PKCE is recommended.

The implicit grant flow seems useful in some cases because it doesn't take a `client_secret` value in step 3 of the flow. This is useful because the client app doesn't have to securely store a `client_secret` value. The authorization code grant flow normally requires a `client_secret` value in step 3, but with PKCE you can optionally omit the `client_secret` property.

Not using a `client_secret` is also useful for public OAuth clients such as native apps and single-page apps that cannot securely store the `client_secret`.

For confidential OAuth clients currently using the authorization code grant flow, including all three parameters -- `client_secret`, `code_challenge`, and `code_verifier` -- is recommended because it provides an additional layer of security.

Client types

Zendesk Ticketing OAuth clients feature a `kind` property that can be set to either `Public` or `Confidential`. Public OAuth clients operate in environments where credentials cannot be securely stored, such as mobile and web applications, and must use PKCE. Confidential OAuth clients run on secure servers that can safely store credentials, allowing them to use PKCE, a client secret, or both.

For more information, see Using OAuth authentication with your application.