ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/arun-neelan/538

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 1 & Iss: 3

Research Article

Fortifying Web and Mobile Applications with PKCE and State Parameters in OAuth 2.0

Arun Neelan*

Citation: Neelan A. Fortifying Web and Mobile Applications with PKCE and State Parameters in OAuth 2.0. *J Artif Intell Mach Learn & Data Sci 2023* 1(3), 2515-2519. DOI: doi.org/10.51219/JAIMLD/arun-neelan/538

Received: 02 September, 2023; **Accepted:** 18 September, 2023; **Published:** 20 September, 2023

***Corresponding author:** Arun Neelan, Independent Researcher, PA, USA

ABSTRACT

As the digital landscape rapidly expands, secure authorization has become essential for safeguarding user data in apps and online services, whether on mobile apps, websites or APIs. OAuth 2.0 has become the standard framework for allowing third-party services to access user data without needing to share passwords. However, OAuth 2.0 is not without its security challenges, especially when used by public clients like mobile apps or single-page web applications. This paper examines how OAuth 2.0's security can be enhanced by focusing on two key features: Proof Key for Code Exchange (PKCE) and the State Parameter. It begins by explaining OAuth 2.0's role and its inherent vulnerabilities, especially when used by public clients. The paper then explores how PKCE strengthens security by preventing authorization code interception attacks, contrasting it with the traditional OAuth 2.0 flow and the role of the State Parameter in mitigating Cross-Site Request Forgery (CSRF) attacks. Furthermore, the review highlights essential security considerations, best practices and the challenges developers face while implementing these features. In conclusion, it emphasizes how these enhancements significantly improve OAuth 2.0's security and underscores the need for continued development to address emerging threats.

Keywords: OAuth2.0, Secure authorization, API security, OAuth flows, PKCE, OAuth 2.0 PKCE, OAuth 2.0 state parameter

1. Introduction

OAuth 2.0 is a widely used protocol that allows applications to access user data across different platforms without exposing sensitive login credentials. It does this by using access tokens, which define key details such as the scope, duration, permissions, validity, approved scopes and the context in which the token was issued-information essential for making authorization decisions for protected resources[7]. This offers a secure and flexible way to manage data access. In simple terms, OAuth 2.0 enables a user to grant a client application access to their data on a resource server, without the need to share their login credentials.

As outlined in[1] OAuth 2.0 relies on four key roles:

- **Resource owner:** The user who owns the data and grants access.
- **Resource server:** The server that hosts the protected data or resources.
- **Client:** The application requesting access to the resource server on behalf of the resource owner.
- **Authorization server:** The server that authenticates the resource owner and issues access tokens to the client.

While OAuth 2.0 provides a strong foundation for authorization, it doesn't fully address several critical security concerns, such as authorization code interception, user impersonation and token leakage. These vulnerabilities are particularly concerning public clients, like mobile apps and single-page web applications, which are unable to securely store client secrets. Since these clients often run on platforms such as browsers or mobile devices, where secrets can be easily accessed

or reverse-engineered, they are more vulnerable to attacks. These risks can be mitigated by implementing PKCE (Proof Key for Code Exchange) and the State parameter in OAuth 2.0.

2. Authorization Code Interception Attack

In the past, public clients often used implicit flow, which issued an access token directly without needing an authorization code exchange. However, due to security concerns, the authorization code grant is now the preferred approach, even though it involves a few extra steps. That said, the authorization code grant is still vulnerable to interception attacks, as briefed in[2].

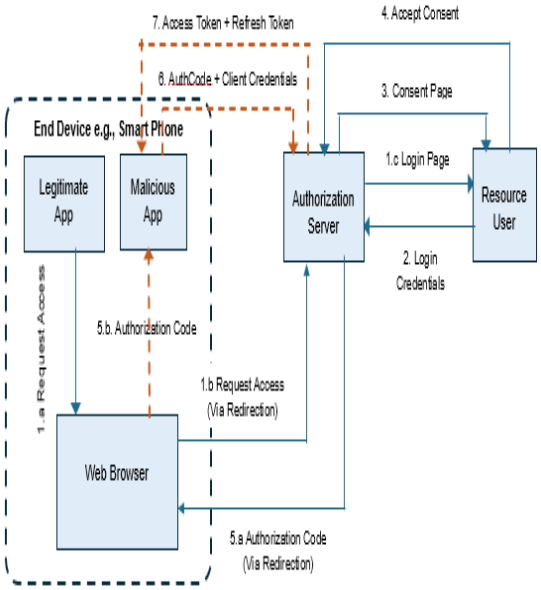

Figure 1: Authorization Code Interception Attack Flow.

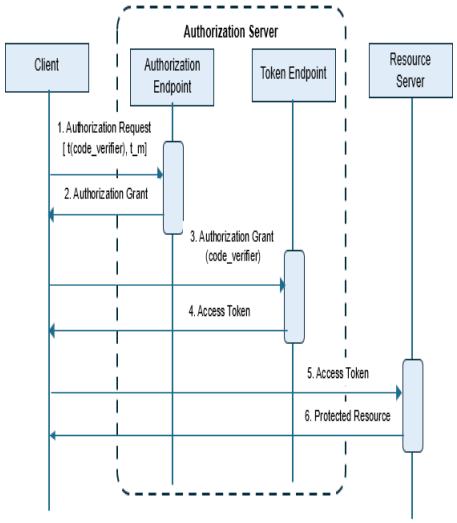

Figure 2: OAuth PKCE Abstract Flow.

A. Flow overview

The client (Legitimate App) directs the resource owner (user) to the authorization server's authorization endpoint.

- The user authenticates (e.g., via username and password).
- The authorization server validates the credentials and requests consent to access specific resources.
- The user grants permission to the authorization server to let the client (Legitimate App) access specific resources.
- The authorization server redirects the user back to the client, passing an authorization code as part of the URL. This code represents the user's consent. Although the intent is to redirect the user back to the client, it is important to note that a malicious app can register itself as a handler for the same custom scheme as the legitimate app. Once this occurs, the malicious app gains the ability to intercept the authorization code.
- The malicious app then exchanges the authorization code for an access token by making a request to the authorization server's token endpoint, including the client credentials for authentication.
- The authorization server provides the malicious app the access token and refresh token to access the resources. Since the malicious app now has access token and refresh token, it can impersonate the legitimate app and access protected resources.

3. Proof Key for Code Exchange (PKCE)

This attack can be mitigated through the technique Proof Key for Code Exchange (PKCE, pronounced "pixy") technique[2].

The OAuth 2.0 PKCE flow is described in [2], where:

- The client generates a random value called the "code_verifier" and derives a transformed version, "t(code_verifier)", known as the "code_challenge", using a transformation method (or hashing algorithm) "t_m." Both the "code_challenge" and the transformation method "t_m" are then included in the Authorization Request. t_m usually is SHA256. If the client can't support SHA256, then the client can send the code_verifier itself where t_m then is assumed as plain.
  - **plain:** code_challenge = code_verifier
  - **SHA256:** code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))
- The Authorization Endpoint validates the request, stores the "code_challenge" and the transformation method and then returns the authorization code.
- The client requests an Access Token from the Authorization Server by sending the "code_verifier," the random value generated during the Authorization Request.
- The authorization server transforms the 'code_verifier' into a 'code_challenge' using the 't_m' sent during the Authorization Request and compares the resulting value with the 'code_challenge' sent in the same request. If the values match, Access Token is returned.
  - **plain:** code_verifier == code_challenge.
  - **SHA256:**BASE64URL-ENCODE(SHA256(ASCII(code_verifier))) == code_challenge
- The client requests the protected resource from the resource server and authenticates by presenting the access token.
- The resource server validates the access token and, if valid, serves the request.

In this flow, PKCE adds an extra layer of security by requiring the code verifier to be sent during the token exchange. This ensures that even if an attacker intercepts the authorization

code, they cannot use it to obtain an access token and ensures the entity requesting the access token is same as the one that initiated the authorization request. Additionally, PKCE eliminates the need to securely store client secrets for public clients, which are vulnerable to reverse engineering[2].

A. OAuth 2.0 vs OAuth 2.0 PKCE Flow

Table 1: OAuth 2.0 And OAuth 2.0 PKCE Comparison.

Flow	OAuth 2.0 vs OAuth 2.0 PKCE	
	OAuth 2.0	**OAuth 2.0 PKCE**
Auth Request	The client sends the client_id, redirect_uri and client_secret.	The client sends code_challenge, transformation method (t_m) along with client_id, redirect_uri.
	The Authorization Server validates the request and returns the authorization code.	Authorization Server validates the request, stores code_challenge and t_m and returns the authorization code.
Token Request	The client sends the authorization code and client_secret to the Authorization Server.	The client sends authorization code and code_verifier to the Authorization Server.
	The Authorization Server validates client_secret and authorization code and if they are valid, it returns an access token.	The Authorization Server derives the code_challenge by applying t_m to the code_verifier and validates it against the value sent in the Authorization Request Flow. If they are valid, it validates the authorization code and returns an access token.
Resource Access Request	The client uses the access token to access protected resources.	The client uses the access token to access protected resources.
Overall Security	OAuth 2.0 relies on client_secret, susceptible to interception attacks if not protected well.	Relies on dynamic verification mechanism using code_verifier and code_challenge, mitigating interception attacks, primarily in public clients. E.g., mobile or browser apps.

B. Security considerations and best practices

- **Entropy of code_verifier value:** The code verifier must be generated in a manner that ensures it is cryptographically random and has high entropy, making it impractical for an attacker to guess[1,4].
- **Secure storage of the code verifier and client secret:** Though client_secret is not required in PKCE, when used client_secret should also be securely stored[2].
- **Use of strong cryptographic methods:** The SHA-256 code challenge method or another cryptographically secure mechanism should be used. If the code challenge is plain, alternative mechanisms, such as encrypting and decrypting the code verifier may be employed than relying on transacting with just the plain value[2].
- **Always Use TLS (https):** Clients must always use TLS (https) or equivalent transport security when making requests with tokens. Failing to do so exposes the token to numerous attacks that could give attackers unintended access[3,4].

C. Challenges and limitations

Though PKCE mitigates the risk of interception authorization code attacks, it does come with challenges and limitations.

- **Implementation complexity & additional latency:** Technologists need to design, implement and thoroughly test the creation, exchange and verification of code challenges. This introduces additional complexity, including future enhancements, compatibility with existing systems and other factors resulting in increased latency and response time.
- **Secure storage of code verifiers:** Code verifiers and details should be stored securely to avoid leaks or misuse.
- **Additional protection to tokens:** PKCE helps secure the process until the access and/or refresh token is obtained. However, once an attacker intercepts or steals either token, they can impersonate the user and access protected resources. This highlights the need for additional security measures, such as using short-lived tokens, securely storing them, implementing token revocation mechanisms and adding extra validations for both access and refresh tokens on the server to further protect user data[5,6].

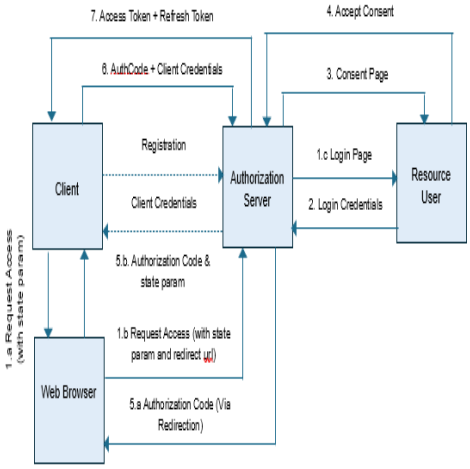

Figure 3: OAuth Code Grant with State Parameter.

4. Proof Key for Code Exchange (PKCE)

Cross-Site Request Forgery (CSRF) is an attack that tricks an authenticated user into performing unwanted actions on a web application. In the case of the Authorization Code Grant flow, CSRF could involve scenarios like stealing the authorization code during the process and using it to access protected resources or sending the attacker's authorization code to a malicious client redirect URL[4].

The state parameter mechanism plays a crucial role in preventing these types of attacks by ensuring that the callback from the authorization server matches the original request made by the client.

The following section explains the authorization code grant flow, as outlined in[1], including the use of the state parameter when a client logs into a legitimate application. This process is a prerequisite for the attacks discussed earlier. The client first registers with the authorization server, providing redirect URLs where the authorization code can be sent.

A. Flow overview

- The client directs the resource owner (user) to the authorization server's authorization endpoint. During this process, the client sends state param to the authorization server along with the redirect url to which the authorization code should be sent upon user consent. The redirect url sent in the request must be one of the urls provided during the registration process.

- The user authenticates (e.g., via username and password).
- The authorization server validates the credentials and requests consent to access specific resources.
- The user grants permission to the authorization server to let the client access specific resources.
- The authorization server redirects the user back to the client, passing an authorization code and state param as part of the redirect url. This code represents the user's consent.
- The client checks that the state parameter matches the one sent in the original request. If it's valid, the client then exchanges the authorization code for an access token by sending a request to the authorization server's token endpoint, along with its client credentials for authentication.
- The authorization server provides the client with the access token and refresh token to access the resources.

In addition, the following sections outline the consequences of each CSRF attack scenario and explain how the state parameter along with redirect url validation can help address these issues. For both scenarios, its assumed that after the client logs into the application and the session is still active, the attacker sends them a malicious link that contains an authorization code request and the client unknowingly clicks on the link and visits the page from where the attacker takes the control.

Client has not registered the redirect urls with authorization server: In this scenario, it becomes much easier for an attacker to hijack the entire flow once the user clicks on the malicious link. The malicious app can send an authorization request with a redirect URL that's different from the client's, allowing the attacker to capture the authorization code from the callback and access the user's resources. While we'll dive deeper into the state parameter later, this section highlights the importance of validating the redirect URL to ensure the callback goes to the correct location[2]. The next section will focus on the need for the state parameter.

Client has registered the redirect urls with authorization server: The authorization server will reject requests if the redirect URL provided by a malicious application differs from the registered one. However, the malicious app can provide a valid redirect url in the authorization request with different client credentials that will result in a different, but a valid authorization code sent to client. Additionally, without a secure transport layer, the attacker could modify the response, say., authorization code. Both these attacks can cause potential harm.

B. Security considerations and best practices

- **Enforce state parameter:** Implementing state parameter will prevent clients from using the invalid authorization code to obtain access token. In this case, the client will check if the state parameter value that comes in the callback matches with the one sent in the authorization request earlier. If no match, the client will not proceed with the flow[1,4].
- **Entropy of code_verifier value:** Make sure the value of the state parameter has high entropy so that it's difficult to predict for attackers. To enforce high security, the state parameter value can be encrypted at the client and decrypted at the authorization server so only these two parties know how to interpret the value[2].
- **Register and Validate redirect URLs:** Make sure valid redirect URLs are provided during registration process and authorization server performs validations with the one sent during the authorization request process. The authorization server should send the response only if the validations are through.
- **Always use TLS (HTTPS):** Usage of HTTPS ensures the requests and responses can't be intercepted or altered, protecting applications from potential attacks[3,4].
- Limited Time Usage of Auth Code: Restrict the usage of authorization code to obtain access tokens to prevent brutal force attacks or unauthorized access attempts[7].

C. Challenges and limitations

Additional logic, maintenance and testing are required to ensure

- Valid redirect urls are stored and validated with the one that comes with the authorization request.
- The state param value is managed securely so that the correct state param value (the one that came with the authorization request) is sent in the callback along with authorization code.
- Encryption keys should be properly managed, shared and rotated in case the state parameter value is to be exchanged securely than just plain value.

PKCE helps ensure that only authorized clients can exchange authorization codes for tokens, while the State Parameter protects against cross-site request forgery (CSRF) attacks that could steal tokens. Using both together helps mitigate the risk of token leakage. As mentioned earlier, following best practices alongside PKCE and the State Parameter is crucial to achieving the best results.

5. Conclusion

PKCE helps protect against authorization code interception attacks by introducing a dynamic verification process with the code_verifier and code_challenge. This is particularly useful for public clients that can't securely store client secrets. The State Parameter, on the other hand, helps prevent Cross-Site Request Forgery (CSRF) attacks by ensuring the callback from the authorization server matches the original client request.

While these techniques significantly improve security, additional measures are still needed to better protect user data. These include using short-lived tokens, enabling token revocation, validating access and refresh tokens on the server, checking redirect URLs and ensuring TLS/HTTPS is used. However, these added protections come with trade-offs, like increased complexity, potential delays and the need for secure storage of code verifiers.

In conclusion, PKCE and the State Parameter significantly strengthen OAuth 2.0's security. However, as attackers continuously seek out new vulnerabilities, it is essential to continually refine these mechanisms to stay ahead of emerging threats, all while ensuring they remain simple to implement.

6. References

1. Hardt D. The OAUTH 2.0 Authorization Framework, 2012.
2. Bradley J and Agarwal N. Proof key for code exchange by OAuth public clients, 2015.
3. Dierks T and Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.2, 2008.

4. McGloin M and Hunt P. OAUTH 2.0 threat model and security Considerations, 2013.
5. Jones M and Hardt D. The OAUTH 2.0 Authorization Framework: Bearer Token usage, 2012.
6. Dronia S and Scurtescu M. OAUTh 2.0 token revocation, 2013.
7. OAUth 2.0 token introspection, 2015.