Uncover the Secrets of PKCE: Elevating OAuth2.0 for security of native clients

Literature study

Student: Alina Boshchenko (2732782)

Supervisor: Dr. A. Belloum, *University of Amsterdam*

July 21, 2023

Contents

Abstract. *This study examines the OAuth2.0 extension known as Proof Key for Code Exchange (PKCE) and its role in safeguarding native client applications against code interception attacks. OAuth2.0 is an industry-standard authorization framework that facilitates federated authentication, and PKCE has emerged as a crucial enhancement to address specific security challenges faced by native applications. This study provides a comprehensive overview of OAuth2.0, followed by a detailed exploration of the PKCE mechanism. It also analyzes potential vulnerabilities and limitations in PKCE's implementation, offering insights into mitigation strategies and best practices to enhance the security of native OAuth2.0 applications.*

1 Introduction

OAuth2.0 is an industry-standard authorization framework that conquered the federated authentication niche. There are other ways of providing federated authentication, but they are way less generic, if we take the SAML protocol [1] as an example it would be way more enterprise-oriented. OAuth2.0 establishes a trusted relationship between the resource owner, typically the user, the application seeking access (client), the authentication server, and the server holding the resources (resource server) [2]. There are several possible flows for the Oauth2.0 mechanism, figure 1 shows the general concept. In OAuth 2.0, the client first registers with the server using a client ID and (optionally) client secret. To make the whole scheme secure, each pair client id/client secret should be registered in the authorization server in advance. When the user wants to authorize the client, they are redirected to the authorization server's authentication page. The user authenticates themselves, and after that server asks the user to approve giving certain permissions to the client app. Once the user grants consent, the server issues an access token. Later client can present this access token to the resource server, which verifies its and grants access to the resources. The killer feature is the lifespan of the access token. The access token always lives for a very short period of time, usually from 3 minutes to 1 hour.

As we mentioned before, there are several possible OAuth flows, we now look a bit more in detail into them.

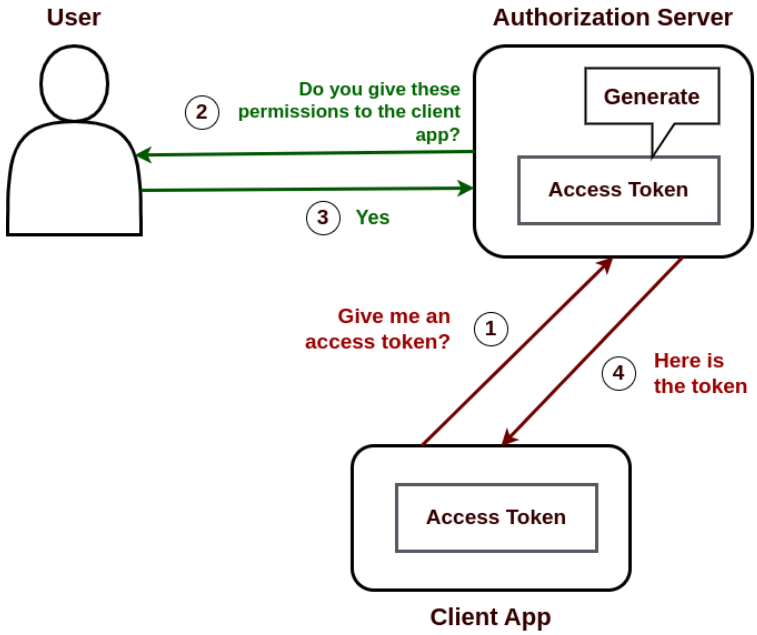

Figure 1: RFC 6749: The OAuth 2.0 Authorization Framework [3]

2 OAuth2.0 authorization flows

Let us briefly describe the authorization flows, emphasizing the authorization code flow, which will be the basis for further study.

Client Credentials Flow

This flow allows the application to pass the client secret and client id to an authorization server without user intervention [3].

1. The application authenticates with the authorization server, passing the client secret and client id.

2. The authorization server checks the client secret and client id and returns an access token to the application.

Resource Owner Password Flow

The Resource Owner Password Flow requires users to submit their credentials via a form. Credentials are transferred to the backend of the application and may be retained for future use before an Access Token is granted. It's essential that the app is completely trusted. Therefore, this flow is generally

not recommended [3].

1. The user clicks a login link in the application and enters credentials into a form managed by the app.
2. The application stores the credentials and passes them to the authorization server.
3. The authorization server validates credentials and returns the access token

Implicit Flow with Form Post

This flow uses OIDC to implement a web sign-in that functions like WS-Federation and SAML. The web app requests and receives tokens via the front channel, without requiring extra backend calls or secrets [3].

Hybrid Flow

This flow allows the application to obtain immediate access to an ID token while enabling retrieval of additional access and refresh tokens [3].

Device Authorization Flow

This flow allows to authenticate users without asking for their credentials. It provides a better user experience on the devices where it is hard to enter credentials [3].

1. The user starts the app on the device.
2. The device app requests authorization from the authorization server using its client id.
3. The user is asked to authorize the device by following the link on their computer or smartphone.
4. If the consent was granted, he application requests the token endpoint.
5. Authorization server responds with an access token.

Authorization Code Flow

This flow is the key flow for this study as PKCE was built as an extension to it and it is used further in this report to describe the attacks and mitigation measures [3], [2].
The distinguishable feature of this flow is the presence of the access code, which is exchanged for the access token. The flow works as follows:

1. A user tries to access the client and initiates the authorization flow.

2. The client application calls the authorization server's authorization endpoint.
3. The authorization server responds with the redirect URI.
4. The user gets redirected to the consent form.
5. The user authenticates with their identity source and gives their consent.
6. The authorization server issues an authorization code for the client application.
7. The client application requests the token endpoint to exchange the authorization code for the authorization token.
8. The authorization server validates the authorization code, client ID, and client secret, and if everything is valid returns the token.
9. The client application requests protected resources from the resource server and submits the token it received in the previous step.
10. The resource server validates the token and responds with the requested resources.

Note that the token introspection endpoint needs to be able to return information about a token, so most likely it would be located at the same place as the token endpoint.
The request will be a POST request containing just a parameter named "token". It is expected that this endpoint is not made publicly available to developers. Applications should not be allowed to use this endpoint since the response may contain privileged information that developers should not have access to. One way to protect the endpoint is to put it on an internal server that is not accessible from the outside world, or it could be protected with HTTP basic authorization.
Also, according to the specification, Oauth 2.0 should be performed over a secured connection such as TLS only.

3 Authorization code flow on native clients

Considering the authorization code flow, we now want to focus on the part of this flow responsible for the issue of the authorization code which will be exchanged for an access token to get access to the data. Let's discuss why this complication is required and why the auth code is needed. If we take a look at arrow 6 in Figure 2, we can notice that these mechanics are processed via browser (HTTP) redirects (so as 2,3 arrows as well). So in such redirects, there are not many ways to keep

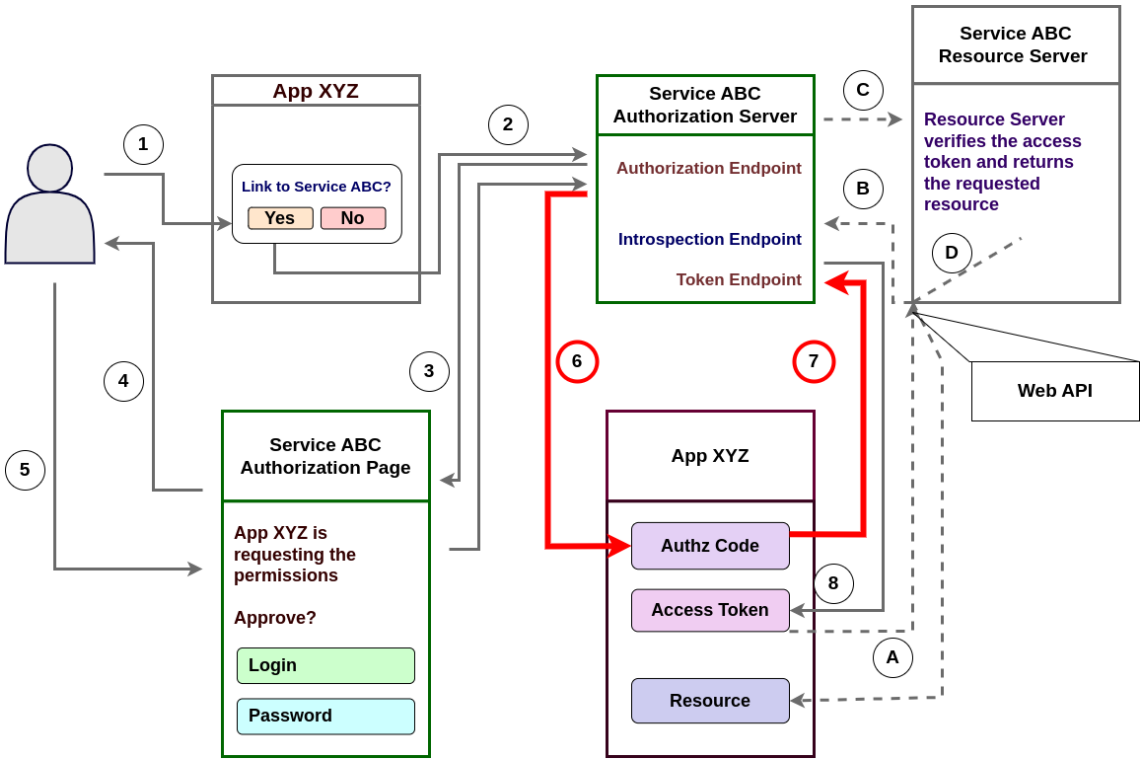

Figure 2: Authorization Code Flow, RFC 6749, 4.1, https://www.authlete.com/

anything secret. It is technically possible to add a token to the URL as it will survive the redirect, but any proxy or reverse proxy will write all those URLs to access logs, so anyone that has access to logs will be able to use those tokens. An alternative can be headers, however, they are also visible, even in the Chrome Developer Tools [4].

Therefore, we need a one-time auth code (OTP) that lives for a very very short time before the exchange to a token which lowers the chances of it being compromised. It is considered secure due to the change of code to the token being usually performed on a request from backend to backend, not on the level of the HTTP requests so the browser is not involved here. It's also clear in what network those backends are located.

Another point of interest here is the ability of the authorization server to know where to redirect back the URL with code to get to the client application. As an attacker, I can technically construct a URL that would impersonate the client application to authorize in authorization server and instruct the server to return the code to my personal server [5]. There are, for example, no signatures on step 2,3,6 in Figure 2(unlike SAML), only HTTP redirects, so there is no trust, and therefore for each

client application, there is a list of addresses on which it is possible to return code/token depending on the flow.

Redirects

I would like to elaborate more on the topic of redirects on native clients. Figure 3 illustrates the case of custom redirect URIs - when the user clicks on a link and instead of opening the browser window it opens the app. If an app controls the domain name "app.example.com" it is possible to use "com.example.app" as their scheme. Some authorization servers assign client identifiers based on domain names, for example, "client1234.example.net", which can also be used if reversed in the same manner. A scheme such as "myapp", would not work as it is not based on a domain name [6].

A limitation of using private-use URI schemes for redirect URIs is that multiple apps can typically

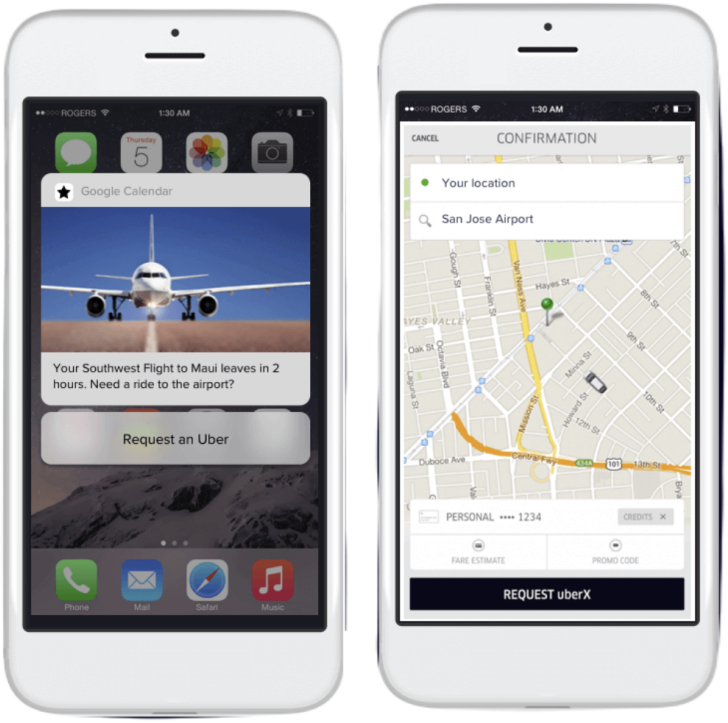

Figure 3: Illustration of the case of custom redirect URIs - when the user clicks on a link and instead of opening the browser window it opens the app.

register the same scheme, which makes it hard to determine which app will receive the authorization. An interesting point here is that according to the OAuth2.0 flow, two different apps can split the

responsibility - one makes the authorization request, and another - handles code and token, it is not prohibited by the protocol.

When an app wants to register itself as a handler for a custom scheme, it typically interacts with the operating system's APIs or configuration files. For example, in Android, the app registers itself as a handler for the custom scheme in the app's manifest file (AndroidManifest.xml). By adding an intent filter with the appropriate scheme in the manifest, the app declares its ability to handle incoming requests with that scheme.

It is important to say that the ability to register custom scheme handlers is a necessary functionality provided by operating systems to enable legitimate app interactions. However, attackers can abuse this functionality if they can install a malicious app on a user's device and register themselves as a handler for a custom scheme associated with a widely used OAuth 2.0 app. Later we will see how this limitation can become a perfect surface for the code interception attack [7].

4 Proof Key for Code Exchange

PKCE (RFC 7636) is an extension to the Authorization Code flow to prevent CSRF and authorization code injection attacks. PKCE was originally designed to protect the authorization code flow in mobile apps, but its ability to prevent authorization code injection makes it useful for every type of OAuth client, even web apps that use client authentication [8].

Authorization code interception attack

In the authorization code interception attack, the attacker intercepts the authorization code returned from the authorization endpoint within a communication path not protected by TLS such as inter-application communication within the client's operating system. Once the attacker has gained the authorization code, it can use it to obtain the access token.

1. In step 1 of Figure 4, the native app running on the end device, such as a smartphone, issues an OAuth2.0 Authorization Request via the browser/operating system. The Redirection Endpoint URI in this case typically uses a custom URI scheme. Step 1 happens through a secure API that cannot be intercepted. However, it may potentially be observed in advanced attack scenarios.
2. The request then gets forwarded to the authorization server in step 2. Because OAuth requires the use of TLS, this communication is protected by TLS and cannot be intercepted.

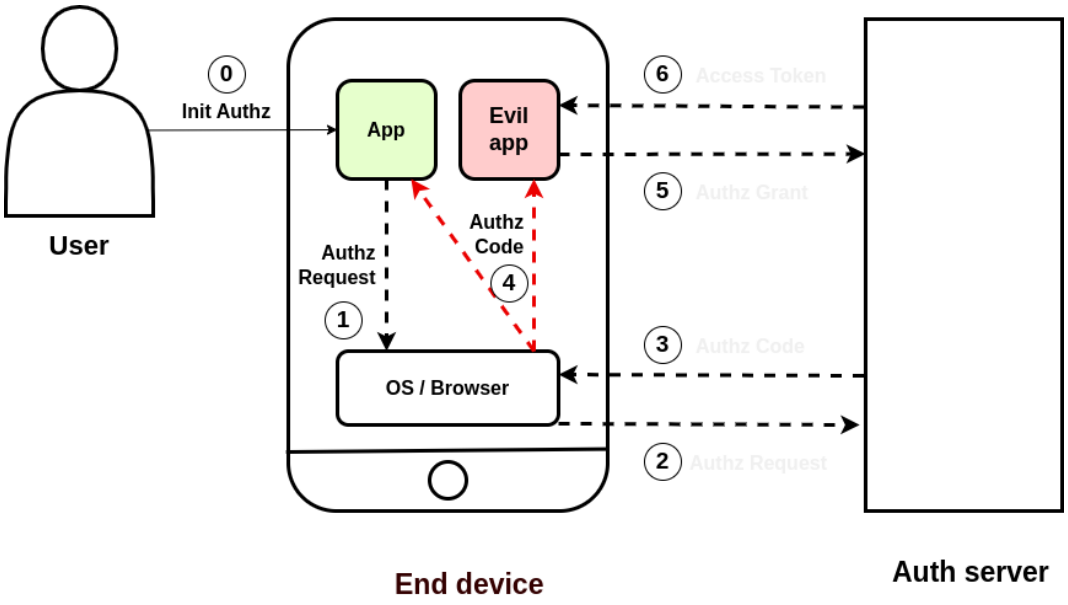

Figure 4: Authorization Code Interception Attack flow

3. The auth server returns the auth code in step 3.

4. In step 4, the Auth Code is returned to the requester via the Redirection Endpoint URI provided in step 1.

This way, if a malicious app registers itself as a handler for the custom scheme in addition to the legitimate app it is able to intercept the authorization code in step 4.
It seems that even with the access code there is another barrier preventing the attacker from gaining access – the client credentials. However, if the target application is a mobile app or a Single Page Web App (SPA) chances are that they will be using the same client credentials for every instance of the app and the credentials are hardcoded into the apps. These are the kinds of apps that are known as public clients. It is not possible to ensure that those credentials have been kept secret and no one else already had them. For public clients, it is recommended not to perform any actions by trusting the secret [9], [8]. This allows the attacker to easily request and obtain an access token in steps 5 and 6.

Mitigation

The basic idea behind PKCE is proof of possession. The client application should give proof to

the authorization server that the authorization code belongs to it in order for the server to issue an access token for this client. The mitigation technique is presented in Figure 5 and is described below.

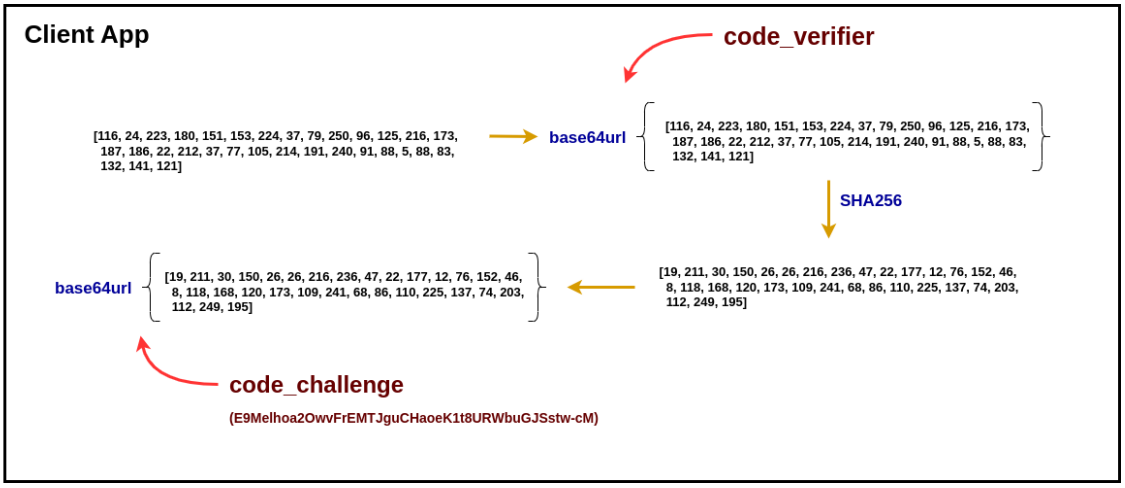

Figure 5: New values generation with PKCE

1. The client uses the output of a random generator to create a 32-octet sequence. It is the secret value and it's crucial to make it hardly predictable.
2. Encoding this sequence as base64 provides the value of the so-called code_verifier.
3. The code_verifier is then hashed via the code_challenge_method hash function, in most cases it's SHA256, to produce a new sequence.
4. The sequence from step 3 is encoded with base64 and it provides the value of the so-called code_challenge.

The client sends the authorization request with the **code_challenge** and the **code_challenge_method**. The authorization server then records the code_challenge and **code_challenge_method** along with the authorization code that is granted to the client. If the consent is granted, client receives the access code. After that, the client sends an access token request along with the **code_verifier**. The authorization server retrieves the information for the code grant. Based on the recorded **code_challenge_method**, it hashes and encodes the value of provided code_verifier. The calculated value is then compared with the value of **code_challenge**.

The key contribution of this approach is that the code_challenge or the code_verifier cannot be intercepted as the communication between the client and authorization server should be conducted through a secured TLS channel. The code_verifier and the code_challenge should only be used once per token requesting cycle. Every time an authorization request is made a new code challenge should be sent.
Since the code challenge is sent through the browser it is highly important that the code challenge is not the same as the code_verifier. Otherwise, if by some means an attacker has access to the HTTP request (for example through HTTP logs) then the use of PKCE will be useless. Due to those measures we bind together the client app and authorization code and mitigate the code interception attack [8], [10].

5 Limitations

Judging by the previous section it seems that the mitigation was found. But let's now discuss the limitations of this approach. Let us introduce a PKCE bypass with an app impersonation attack.

PKCE bypass via app impersonation attack.
Let's consider a target native app and an attacker, who managed to install a malicious app on the same device. They are able to steal the client_id of the target app and register the same redirect URI [11]. The illustration of the attack is provided in Figure 6.

1. The evil app initiates the authorization flow and generates the Authorization Request. The Authorization Request contains the client_id and redirect_uri of the target app. This way the user and the authorization server cannot recognize that the malicious app initiates this request.
2. The browser is called and the Authorization Request is sent to the authorization server (Facebook, Google, etc.).
3. We assume that the user is authenticated and they authorize access to the requested resources. As a result, the Authorization Response containing the code is sent back to the browser.
4. Now, the browser calls the malicious app registered handler and it receives the code.
5. In step 5 of Figure 2 the malicious app sends the code_verifier used for the computation of the code_challenge. Thus, the stolen code can be successfully redeemed and the malicious app receives the access_token and id_token.

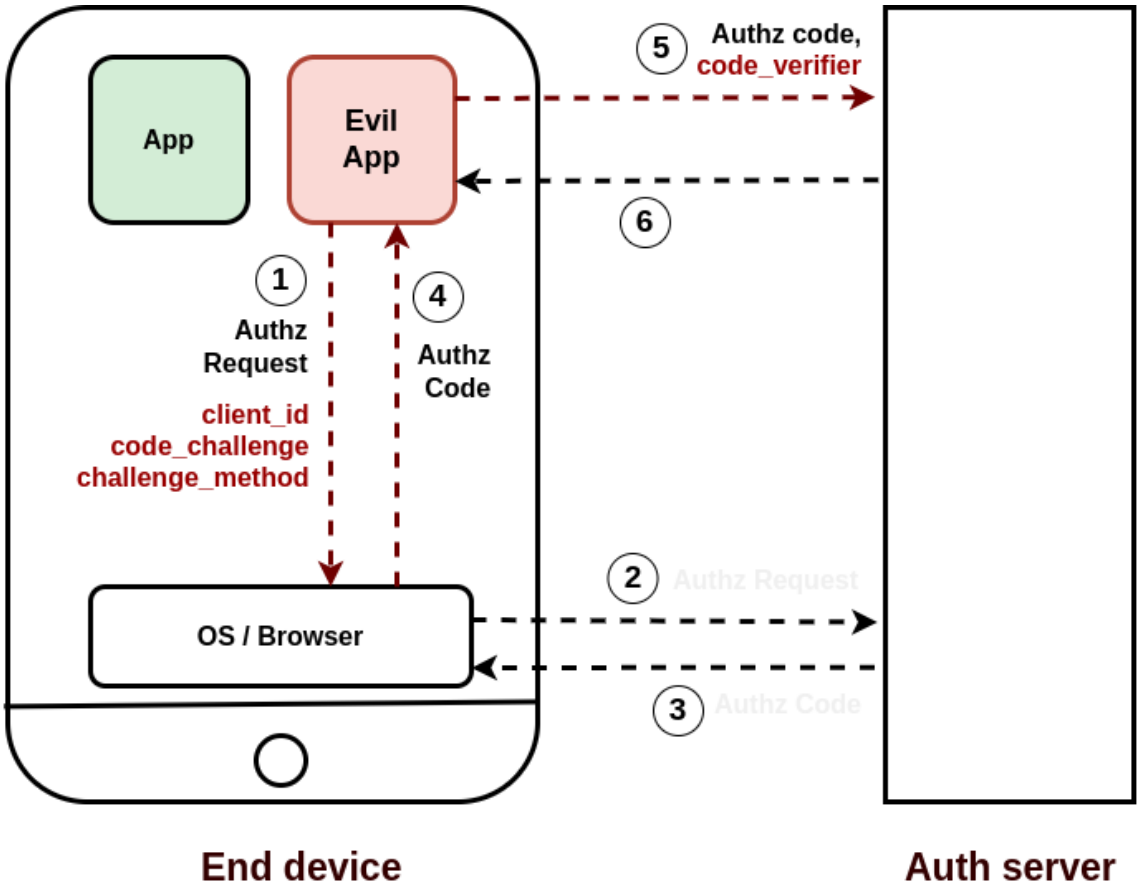

Figure 6: App impersonation attack flow

This way the attacker again manages to get access to the resources.

Mitigation For the attack described above to be successful we need to fulfill a number of important requirements.

First, we need to ensure that the malicious app has higher priority when handling the same Redirect URL, which is possible to do according to the specification of the OS.

Second, in some cases stealing the client id and client secret requires a lot of effort. Sometimes it requires traffic sniffing, which is under the HTTPS. However, we cannot say it is the guarantee as there are cases of public networks or unsafe actions of the user.

But let's consider the biggest difference between the code interception attack (Figure 3) that led to the development of PKCE and current attack (Figure 6). The thing that immediately catches the

eye is the user. In the first attack user is the one who initiates the request and they expect to see a pop-up asking for a confirmation. In the second scenario, the app itself initiates the authorization flow introducing the need for phishing tecniques to make the user approve the permissions. Without PKCE it would not be needed. Therefore, most importantly, it's crucial to trick the user into authorizing access to the requested resources.
Obviously these factors does not completely mitigate the attack, but they significantly lower the chances of success.
Let's now discuss a bit more ways of mitigation.
We already discussed that certain types of callback URL can be registered by more than one app, meaning only a client ID needs to be stolen in order to impersonate a real app. However, this still requires a malicious app to trick the user to sign in before tokens can be retrieved. It is considered to be a best practice for an apps to only us the scopes they need, preferably read-only ones. After that it depends on the type of native app.

Mobile applications

A mobile app can use Claimed HTTPS Schemes via an HTTPS callback URL to overcome the problem of redirects. It is backed by App Links on Android or Universal Links on iOS. Even if a malicious app uses the correct client id, it cannot receive the login response with the authorization code, because it will be received on a URL like this, and the mobile OS will only pass this to the app that has proved ownership of the domain via the deep linking registration process [6].

Desktop applications

For desktop apps the situation is a bit more complicated, since only Loopback and Private URI Scheme callback URLs are allowed to be used. It relies on users to avoid installing malicious apps. One of the good practices would be to only install apps from stores that require code signing, which also inform the user who the publisher is. It is generally considered that if users install malicious apps then perhaps they have more serious problems.

6 Best practices

In this section we will summarise the best practices and recommendations.

- For the refresh tokens it is best to use PKCE in conjunction with the Refresh token rotation,

meaning that any time a refresh token is used to get a new access token it should be replaced with a new one, so every time we generate a new pair of refresh and access tokens.

- Let us introduce two types of user agents. An external user agent is typically the device's native browser with a separate security domain from the native app. Embedded user agents, on the other hand, are browsers integrated within other software. They are designed to render web content within a specific application.

  RFC documents two approaches for native apps to interact with the authorization endpoint. This best practice requires that native apps **must not** use embedded user-agents to perform authorization requests and allows that authorization endpoints **may** take steps to detect and block authorization requests in embedded user-agents [5].

  In typical web-view-based implementations, they can record every keystroke entered in the login form to capture usernames and passwords, automatically submit forms to bypass user consent, and copy session cookies and use them to perform authenticated actions as the user. Even when used by trusted apps belonging to the same party as the authorization server, embedded user-agents violate the principle of least privilege as they can access the user's full authentication credential, not just the OAuth2.0 authorization grant that was intended for the app. It makes the whole OAuth2.0 concept useless and increases the attack surface.

- It's a good practice not to use the hardcoded constant client id, but rather replace it with a client id per app instance. It's not possible for all kinds of apps, but if the architecture allows, it would be definitely beneficial as it requires more effort to get non-constant client id.

- There are OAuth2.0 flows that allow applications to authorize access without user consent, an example could be the client credentials flow. Those flows are definitely not advised to be used in native clients, even with PKCE.

7 Conclusion

In conclusion, PKCE has become the industry standard in native OAuth2.0 applications. PKCE addresses the security challenges specific to these types of applications, providing an additional layer of protection against code interception and unauthorized access. Its widespread adoption and support by major OAuth2 providers make PKCE a reliable and recommended choice for developers looking to enhance the security of their native apps. By implementing PKCE, organizations can

ensure that user authentication and authorization processes remain more secure, bolstering the overall trust and confidence in their applications. In conclusion we would like to say that the development of PKCE made a huge contribution in the OAuth2.0 on native apps security, but even it can't become a silver bullet.

References

[1] B. Campbell, C. Mortimore, and M. Jones, "Security assertion markup language (saml) 2.0 profile for oauth 2.0 client authentication and authorization grants." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7522

[2] T. Lodderstedt, M. McGloin, and P. Hunt, "Oauth 2.0 threat model and security considerations," RFC 6819 (Informational), Tech. Rep., January 2013.

[3] E. D. Hardt, "Rfc6749: The oauth 2.0 authorization framework." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6749

[4] D. Fett, R. Kusters, and G. Schmitz, "A comprehensive formal security analysis of oauth 2.0," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016.

[5] R. Yang, C. L. Wing, and T. Liu, "Signing into one billion mobile app accounts effortlessly with oauth2.0," The Chinese University of Hong Kong, 2015.

[6] OAuth.com. Redirect uris for native apps. [Online]. Available: https://www.oauth.com/oauth2-servers/redirect-uris/redirect-uris-native-apps/

[7] M. Shehab and F. Mohsen, "Securing oauth implementations in smart phones," in Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, ser. CODASPY 2014. ACM, 2014.

[8] E. Sakimura, J. Bradley, and N. Agarwal. Rfc7636: Proof key for code exchange by oauth public clients. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7636

[9] F5 Labs. Securing apis in banking with oauth and pkce. [Online]. Available: https://www.f5.com/labs/articles/cisotociso/securing-apis-in-banking-with-oauth-and-pkce

[10] RingCentral. Auth code pkce flow. [Online]. Available: https://developers.ringcentral.com/guide/authentication/auth-code-pkce-flow

[11] V. Mladenov and C. Mainka. (2017) Pkce: What can(not) be protected. [Online]. Available: https://web-in-security.blogspot.com/2017/01/pkce-what-cannot-be-protected.html