CURITY

Financial Grade APIs Using OAuth and OpenID Connect

Financial Grade APIs Using OAuth and OpenID Connect

Table of Contents

Introduction

In this white paper, we describe various aspects of OAuth and OpenID Connect that can be used to conform to the revised Directive on Payment Services (PSD2) and the General Data Protection Regulation (GDPR). Though these regulations are mandated by the European Union (EU), the applicability of the techniques described in this paper transcends regulatory regimes. For this reason, readers in other parts of the world who are building APIs that expose high-worth data will also find this paper useful. Much of the content is industry agnostic, but various banking and finance examples are provided. Some of the techniques will be described in the context of PSD2 and GDPR, but an in-depth, or even introductory, knowledge of these regulations is not required.

Overview

OAuth and OpenID Connect are international standards that define methods for securing APIs. They are developed in an open and transparent manner. They have been designed by skilled practitioners who authored preceding, related standards and can be used to solve various use cases. They are governed by open communities with clear Intellectual Property Regimes (IPR). For these and other reasons, various vendors, including Curity, have implemented them. Certification and testing processes exist from various community members, including the OpenID Foundation, the Open Banking Implementation Entity (OBIE) in the UK, and the GSMA.

This is an ideal ecosystem from which to obtain guidance, standards, and software to complying with PSD2, GDPR, and similar regulations. In fact, certain European groups and initiatives have stipulated how OAuth and OpenID Connect should be used to conform to PSD2. This refinement or constraining of the specifications forms what is called a “profile”. The Berlin Group and STET, in particular, have created such profiles. Similarly, the UK Open Banking initiative has mandated the use of an existing profile from the OpenID Foundation called the Financial-grade API (FAPI). This profile stipulates how OpenID Connect must be used in order to provide banking-grade security. This particular usage of OpenID Connect has also been formally analyzed for security, and reviewed by a large group of experts, implementers, cryptographers, and white-hat hackers.

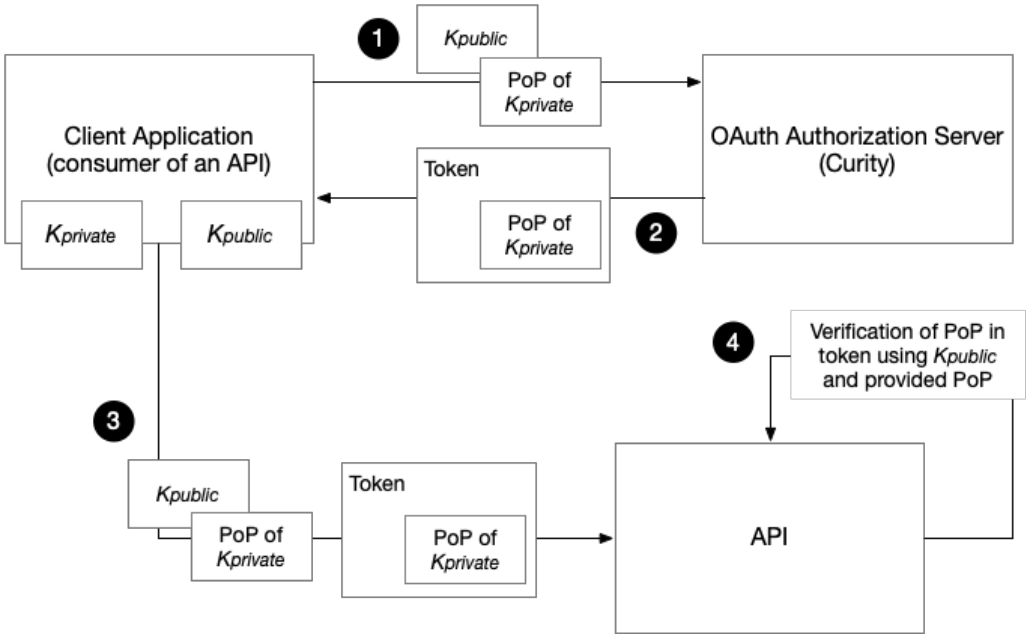
Much of the techniques described in this paper are part of the FAPI profile of OpenID Connect. In particular, the following topics will be discussed:

- The general notion of a proof key and the use of certificates, per se, to bind tokens to a particular key
- Signed requests and responses
- Strong authentication and how certain authentication methods or classes of login methods can be requested
- The role of dynamic client registration is conforming to PSD2 and the extra defenses provided by non-static clients
- How a user can authorize a particular value, like a payment ID, using the OAuth concept of scope
- What interactive user consent is and how to implement it in a user-friendly manner while also being secure
- How to preserve privacy by using pseudonyms instead of an actual user's ID (like their email or social security number)
- How to reduce the regulated space of a deployment by using "phantom tokens"

To start with, we'll discuss the idea of proof of possession keys.

Securely Binding Token Obtainment to Presentation

When exposing data via APIs, the caller and the person operating that client must be authenticated before the request can be authorized. For APIs, this authentication is represented using an access token which the client obtains using OAuth or OpenID Connect. In high-security scenarios like banking (and wherever else high-worth data is exposed via APIs), it is important to ensure that the API consumer is the same entity to whom the token was issued. If the token is not tied in some way to the client, it is possible that an unauthorized entity could steal an access token and call the API, masquerading as the authenticated client. The technique for binding a token to a particular entity is done by the client proving possession of a secret that only it has. This secret is used when authenticating to the OAuth server (i.e., Curity Identity Server) and again when calling the API. This is done in a way that the API can check some value in the presented token to



Proof of Possession

see that the same key was originally used. This technique is illustrated in the following figure:

This illustration shows one variant of this technique which is called “proof of possession” or PoP; it is named such because the client proves it possesses a private key. Firstly, the client authenticates to the OAuth server (shown in step one in the figure above). This will typically be to the OAuth server’s token endpoint, for instance, after an authorization flow was performed with the end user involved. In this call, the client sends a proof of its possession of a private key (PoP of Kprivate). It also includes the corresponding public key (Kpublic). After authenticating the client, the Curity Identity Server will issue a new token, embedding (directly or by reference) the proof of the client’s usage of Kprivate. In step three, the client then calls an API with the access token previously issued. In addition to sending the token, the client sends another proof of its possession of Kprivate. Finally, the API can verify that the caller possesses the private key by comparing the included proof with the one embedded in the token. If this succeeds, then the caller is the same entity to whom the token was issued.

NOTE: This technique doesn’t authenticate the client nor does it necessarily imply that the token is valid. Those are handled by other techniques that the API will also perform. PoP only verifies that the presenter of a token is in possession of a secret that should only be had by the entity to whom the token was issued. (This proof is only as strong as the certainty the receiver has in the client’s ability to keep its secret safe.)

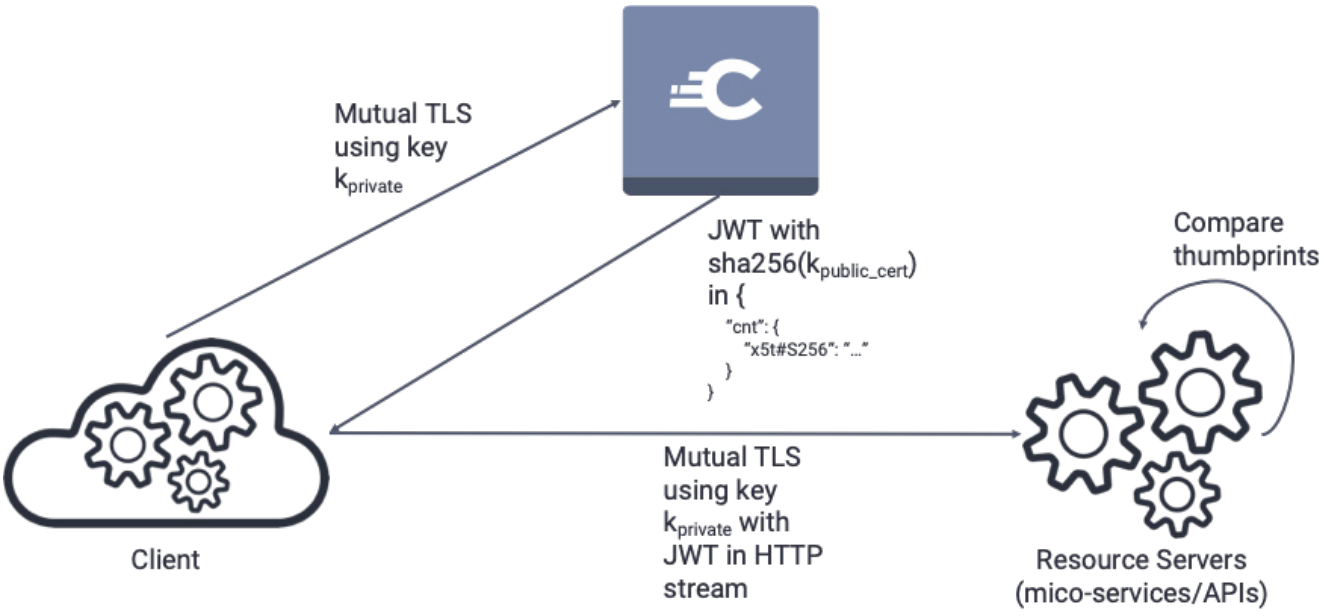
There are various uses of PoP in an OAuth deployment. A couple will be described in this paper,

starting with a very practical one.

Certificate-bound Access Tokens

One application of PoP that is relatively easy to deploy is a technique that uses X.509 certificates. Using this standard, the client has an asymmetric key pair, meaning it possessesa private and public key. The client transmits the public portion to the OAuth server and the API in a certificate that is either signed by the client itself or by a Certificate Authority (CA). The later case leverages a Public Key Infrastructure (PKI) to establish trust and verify the validity of the client's public key; it can also be used to ensure that the client's key pair has not been revoked, and that the client doesn't continue to use the same key for longer than expected.

The reason that this application of PoP is relatively simple to setup is because the transmission and proof of possession is done using mutual TLS (i.e., 2-way SSL). Bi-directional authentication over TLS usually gets a bad rap, and is often described as anything but simple to deploy. The reason it is easy in this case though is because the client and web server technologies are wide-spread and understood by developers and system administrators. Setting up the TLS tunnel, configuring the PKI, certificate revocation checks, and key distribution are not unique to OAuth. Also, many organizations (especially in banking and finance) already have a PKI setup, so leveraging it to secure OAuth access tokens is simpler than message-level PoP which would require application-level development. Lastly, this technique doesn't involve the end users' browsers, bounding key distribution and making it manageable.



PoP using certificates

The message exchange is depicted in the following figure:

This flow is similar to the general one previously described, but, in this implementation, the client begins by performing the first PoP by setting up a mutually-authenticated TLS tunnel. This is only possible if:

- The client has a private key that is signed by an entity (typically a CA) that the OAuth server trust.
- The client's key isn't expired or revoked.

NOTE: This first leg of the flow is done server-to-server and does not involve the end user's browser or its (in)ability to perform mutual TLS.

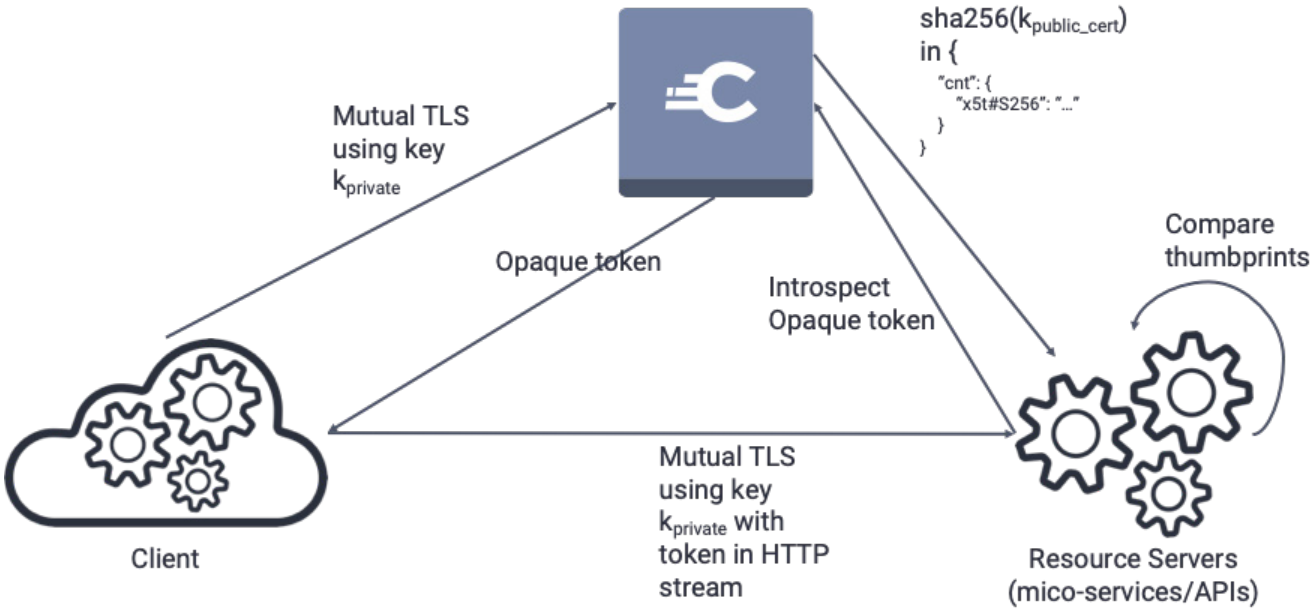
Mutual TLS to the OAuth server's token endpoint isn't enough though. This only authenticates the client; it does not bind the access token to this entity, nor does it prevent others from presenting the token to an API as if they were the one to whom the token was issued. To accomplish this, a hash of the certificate is embedded in the token (directly, as shown in the figure, or by reference). The OAuth server returns the token containing this hash (which is the PoP).

Next, the client makes the API call. The trick is that this call is also done over mutual TLS. This 2-way SSL tunnel is setup with the same key. As a consequence:

- The client must be in possession of the private key, since the communication channel was successfully setup.
- The API has access to the client's public key (included in the certificate).

Lastly, the API verifies that the presenter of the token is the same entity to whom it was issued by computing a hash of the client's certificate (which it has available since the SSL tunnel was set up correctly). It then compares this hash to the one in the access token. If they are the same, the caller is the holder of the key, and the proof is complete.

As mentioned, the flow described above can use reference tokens (i.e., handle tokens) instead of tokens that contain all the claim values (including the PoP). The flow is very similar except that the OAuth server stores the PoP and issues a reference to the identity data (including the PoP). Because the access token is opaque to the client and the API, it must be first dereferenced. This is done using a standard API called introspection, which the Curity OAuth server exposes. When the API calls this, it receives the PoP and can do the same comparison as previously described.



Certificate-constrained reference tokens

This alternative implementation is shown in the following figure:

In the context of PSD2, the client in the above flows is the Third-party Payment Provider (TPP). This organization has been certified by authorities to have the right to initiate payments and/or access end-user accounts. These rights are encoded in an X.509 certificate signed by eIDAS, the EU authority for electronic identification. This certificate and the associated key are the ones the client uses to establish the two-way SSL tunnel to the Curity OAuth server's token endpoint. This endpoint is set up to only allow connections from clients that have a valid certificate signed by eIDAS. As a bonus, eIDAS (or its delegates) will manage the life cycle of certificates and handle revocation.

Using a Proof Key to Protect Authorization Codes

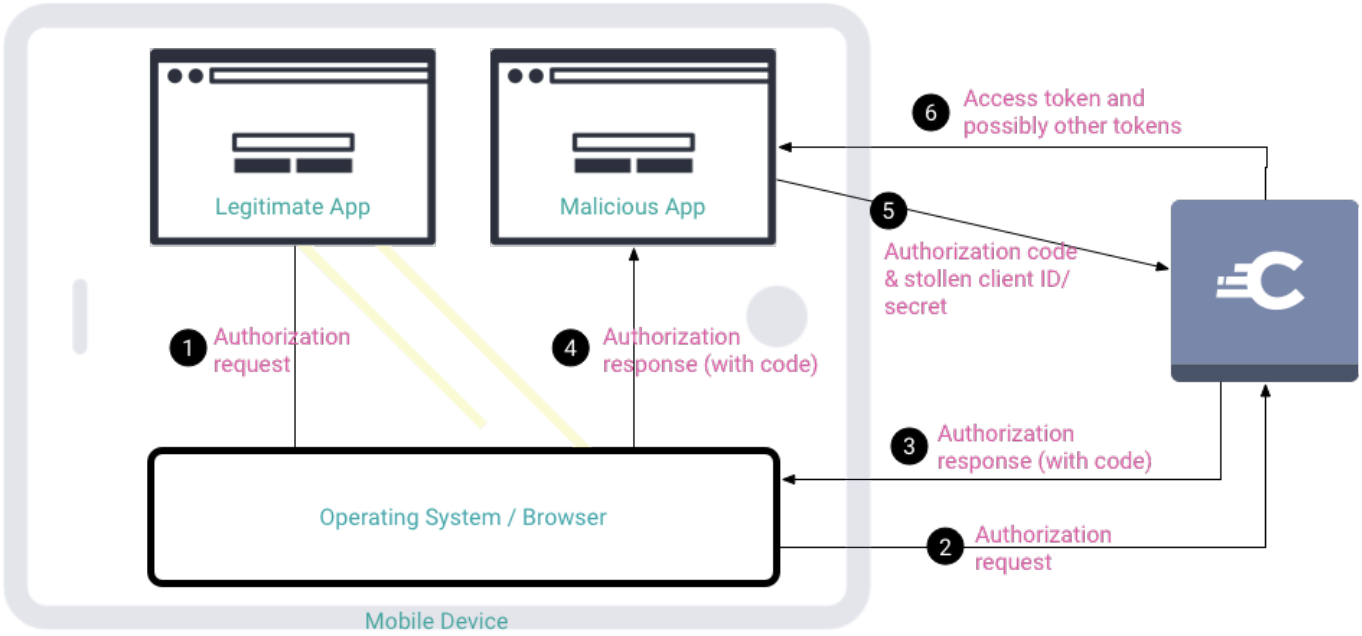
To provide banking-grade security, proof keys are also used in other parts of the OAuth flow. When the end user (the Payment Service User or PSU in PSD2 terminology) starts an authorization request in their browser or from a mobile application, it is important that this flow cannot be intercepted along the way by an attacker. There are many safeguards against this, but one that is important and easy to add is what is called Proof Key for Code Exchange (PKCE or "pixie"). This standard uses a PoP to protect the authorization code that is returned to the app after the user

has logged in and authorized it.
Usually, this vulnerability doesn't exist because:

1. The token endpoint requires authentication and the attacker doesn't have the credential.
2. The redirect Uniform Resource Identifier (URI) of the client application is protected by TLS, meaning the code is confidential and not available to the attacker.
3. The redirect URI is unique and under the control of the client, not the attacker.

These conditions often do not hold, however, when the client is a mobile application where an attacker may have intercepted the credentials used at the token endpoint, and registered a malicious app using the same scheme on the redirect URI (e.g., victim-app://).

To see how PKCE uses a PoP to avoid attacks where the above protections are not enough, examine the vulnerable case firstly:



Vulnerability Necessitating PKCE

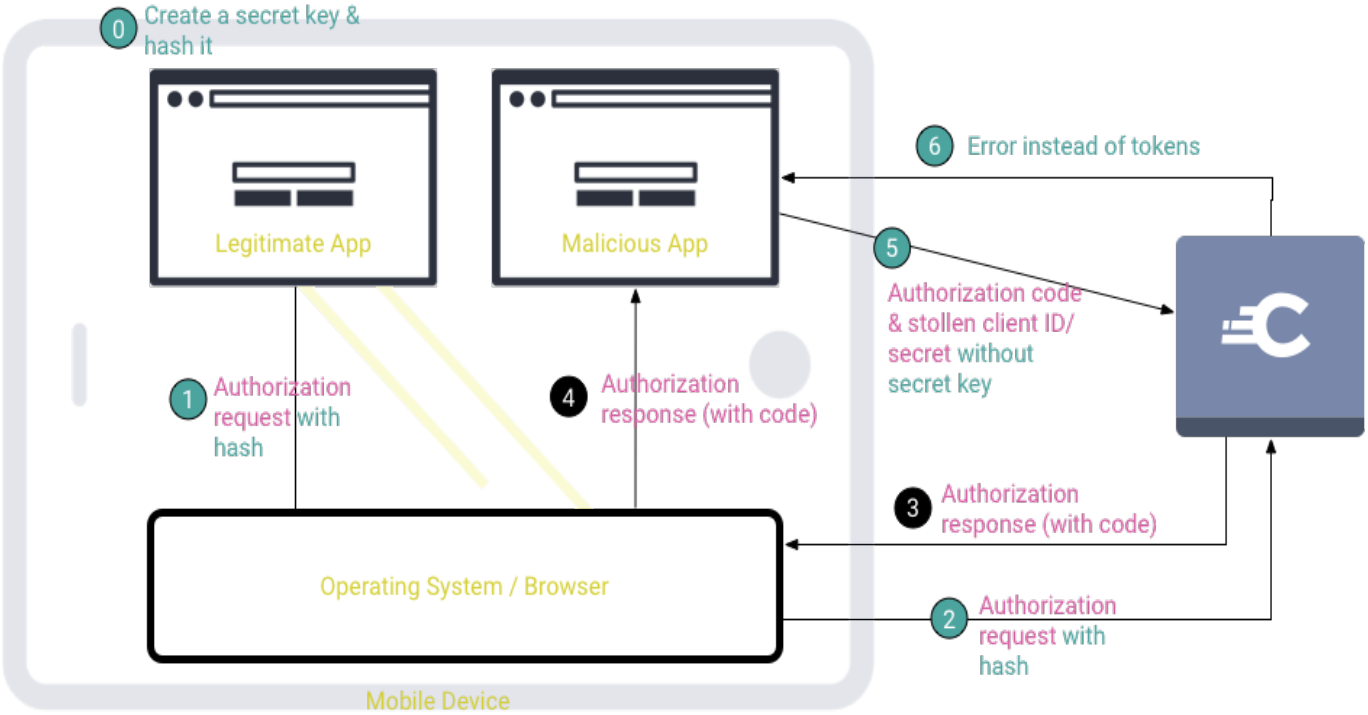
In this figure, the flow starts like normal (1) where an authorization request is sent to the OAuth server's authorization endpoint (2). The user logs in and delegates their rights to the client application (not shown). Then, the OAuth server responds as normal with an authorization code (3). If a malicious application has managed to register with the host operating system using the same scheme (i.e., URL protocol), the attacker's application may be used to handle the callback

instead of the legitimate app (4). If it is and if the malicious app has managed to steal the credentials of the actual app, then the attacker can redeem the authorization code (5) and obtain an access token (6). With this, it could call APIs as the end user, for instance, to initiate a payment (also not shown)

.

That is a lot of ifs, but when dealing with money or other high-worth resources we can't leave this open. To close this attack vector, the OAuth server needs to ensure that the presenter of the code is the same entity to whom it was issued. This is the same case as above except that the API in this case is the OAuth server's token endpoint. Therefore, a PoP can be used in this case as well to tie obtainment of the authorization code together with its presentation. To do this, the flow is altered slightly:

SIDENOTE: An extra defense that can be erected to protect against the callback URL being intercepted is to use deep links (i.e., Universal Links as Apple calls them or what Google refers to as App Links). These use the HTTPS scheme, and the mobile operating system opens the appropriate app if it is installed. For this to happen, the application has to be compiled with the HTTP scheme in its manifest and the mobile OS will authenticate the server at that URL using TLS or by some other form of authentication. This makes it a lot harder for a malicious app to get control of the



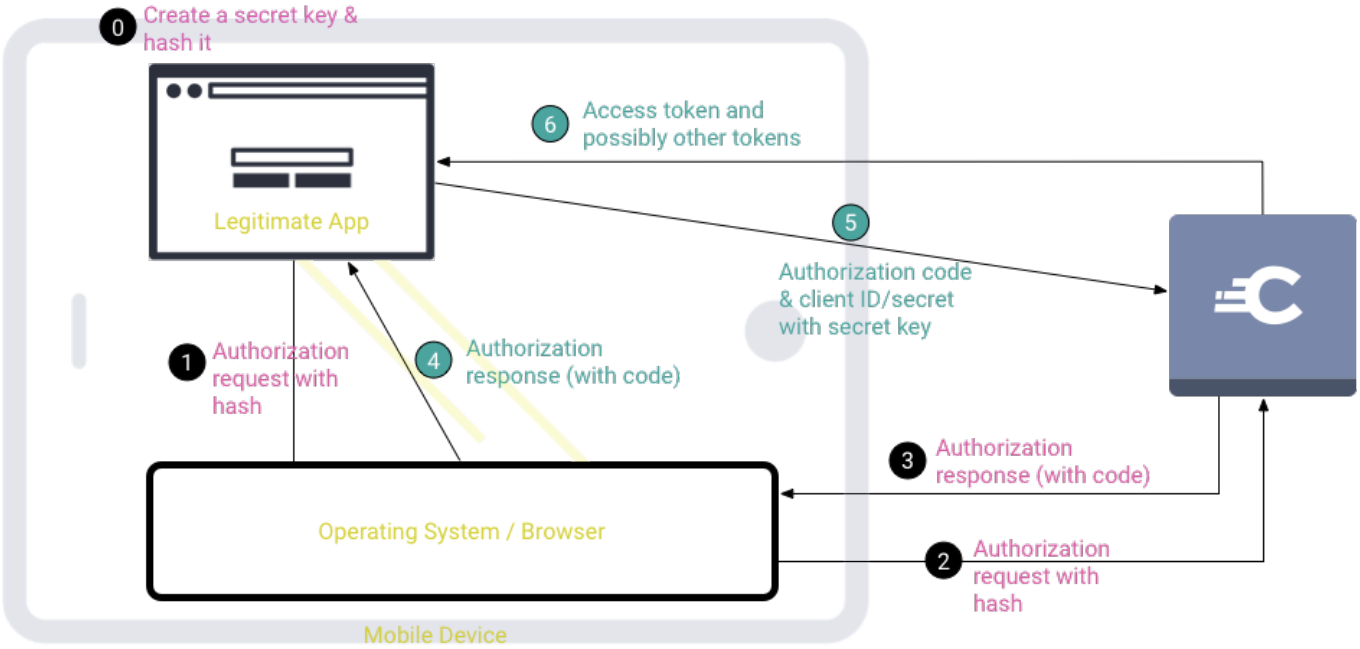
No Vulnerability Using PKCE

redirect URI.

This flow uses a PoP key according to the protocol set forth in the PKCE standard. Unlike the techniques described above, this method uses a symmetric key. Initially (0), the client generates a new random, symmetric key that it uses for just one authorization request. With this new key, it

hashes this PoP key itself, and sends this digest together with the hashing algorithm that was used to the OAuth server's authorization endpoint (1 and 2). Again, login, consent and the reply is made as normal (3). Like before, the nefarious application intercepts the callback (4) and makes a call to the token endpoint (5). This call now requires a PoP which the malicious application cannot perform because it does not have the secret key. As a result, it gets an error instead of a token (6).

At the same time, the legitimate application can redeem the authorization code because it does



Legitimate Use with PKCE

possess the proof key. This case is shown in the following illustration:

When the legitimate app gets the callback with the authorization code (4), it sends the per-transaction secret key to the OAuth server's token endpoint together with the code (5). The OAuth server uses the hashing algorithm sent in step (1) to hash this secret key. If it matches the one previously provided (also in step 1), then the presenter of the code is the same entity that requested it.

For more info about PKCE and how to set it up in the Curity Identity Server, refer to the PKCE tutorial on the developer portal.

PKCE, certificate-constrained tokens, and PoP, in general, are important techniques for deploying financial-grade APIs. These overcome OAuth's Achilles' heel – bearer tokens. They tie the presenter of a token to the requester, creating a much safer and secure environment. There are

parameter in the request object supersedes those provided on the query string. The request object and the query string must contain both response_type and client_id and they must have the same value. Parameters that vary per authorization request (e.g., state, nonce, etc.) will typically not be included in the request object because the client will usually compute the signed JWT request object and reuse it for a time; if the JWT is recomputed with each request, the per-request values should be included in the JWT as well.

Similar to the request object, the OAuth server can also be instructed to sign its responses to the authorization request to obtain message-level integrity in both directions. The mechanisms for this is drafted in JWT Secured Authorization Response Mode for OAuth 2.0 (JARM). It defines three new response modes for OAuth, query.jwt, fragment.jwt, and form_post.jwt. Using one of these three will make the server return a JWT in a response parameter, as opposed to returning the bare parameters. The three have different use cases. If the client is using code flow, the client can decide if it wants the response in the form of a redirect, query.jwt, or to get the response using a form post, form_post.jwt. fragment.jwt is for clients using implicit flow. JARM also defines a fourth response mode for convenience, jwt, which means the client will use the response mode that is default for the requested flow.

When using query.jwt the response will be a redirect to the redirect_uri of the client, with a response parameter in the uri containing a signed JWT. The payload of the JWT will look something like this:


```
{
  "iss":"https://login.curity.se",
  "aud":"s6BhdRkqt3",
  "exp":1311281970,
  "code":"PyyFaux2o7Q0YfXBU32jhw",
  "state":"S8NJ7uqk5fY4EjNvP_G_FtyJu6pUsvH9jsYni9dMAJw"
}
```


Since all the parameters are inside the JWT, and the client can validate the signature using the public key of the OAuth server, the client can be fully sure that the response comes from the

correct source. And combining this with a request object signed by the client, we can achieve perfect message level integrity.

Strong Authentication

One of the main requirements of PSD2 is Strong Customer Authentication (SCA). With SCA, the user must be authenticated using two forms or factors of identification:

- Something the user knows (like a password or PIN)
- Something the user has (like a phone or key fob)
- Something the user is (like their voiceprint or fingerprint)

By combining at least two factors of identification, it is harder for a malicious attacker to steal the identity of the user and login as this victim.

The problem is that neither OAuth nor OpenID Connect, per se, stipulate how a user must authenticate. They require user authentication of some kind, but they do not mandate the use of Multi-factor Authentication (MFA). On the other hand, FAPI, a profile of OpenID Connect mentioned above, does stipulate that MFA must be used. It says that the user must be authenticated at Level of Assurance (LoA) 2 as defined by another standard called the Entity Authentication Assurance Framework (X.1254) from the International Telecommunication Union (ITU), a UN agency.

According to this specification, LoA 2 permits single-factor authentication. This is OK according to PSD2 in some cases where the risk of fraud is quite low. Usually, however, a higher level of assurance is needed. LoA 3 is “used where substantial risk is associated with erroneous authentication”, says the X.1254 spec. This is the case whenever a transaction involves even a moderate amount of worth. In such cases, LoA 3 requires MFA. Therefore, most financial API transactions will require the user to prove something they know together with something they have or are.

To accomplish this, it is important to allow various methods of login. These must be chained together in various ways and consider contextual information when determining which factor of authentication is required. Some of these will be determined at the application if it finds that too few factors of authentication were used to complete the current transaction. Other times, they will be determined during login based on previous login geographies or login frequency. For these and other reasons, login becomes a nuanced and important part of any PSD2 or financial-grade API solution.

To address these challenges, the Curity Identity Server includes not only an OAuth and OpenID Connect server, but also a feature-rich authentication service. With its support for dozens of authentication methods, it is possible to support not just 1- or 2-factor authentication but any number of factors. To cope with PSD2 and avoid fraud, it is important that this authentication be tied into Security Information and Event Management (SIEM) systems. By taking a holistic view of the risks and compensating by denying login or challenging with additional authentication methods, users, reputations, and assets can be protected. The Curity Identity Server allows for this by including in its authentication service various features such as:

- Configuration of custom workflows of authentication actions that run after login or Single Sign-on (SSO)
- Ability to add in steps to these processes using the support SDK
- Out of the box actions that trigger other authentication methods, redirection of the login flow to other services, and the possibility to abort the login

By combining a powerful login service with a standard compliant OAuth and OpenID Connect server, providing SCA that conforms to PSD2 is straightforward.

Dynamic Registration Vis-à-vis PSD2

PSD2 mandates that any organization (a TPP) that has been certified to initiate payments or access accounts should be able to. To utilize this right with any Account Servicing Payment Service Provider (ASPSP), typically a bank, the TPP organization needs to obtain access tokens to invoke the provider's API. A prerequisite to obtaining tokens is to be a registered client of the OAuth server. There are two ways that this registration can take place: statically or dynamically.

PSD2 also says that there must not be any waiting period for the TPP to become a registered client. This process has to be completely automated and completed in short order. For this reason, statically registering the OAuth clients isn't typically an option. This means that clients must be able to register at will. This is possible when the OpenID Connect provider, the bank, supports the dynamic profile of that standard or the related OAuth dynamic client registration protocol (of which the OpenID Connect profile is a subset). Either of these coupled with Dynamic Client Registration Management (DCRM) API support allows any TPP to register as an OAuth

many other techniques that also need to be deployed in order to protect the system at various other points. Some of these are described in the following sections.

Signed Requests and Responses

As shown in the above diagrams, some aspects of the OAuth and OpenID Connect flow travel through the user's browser. This application is inherently untrusted and typically runs on third-party networks. It is prudent to assume, therefore, that the browser and network are compromised. Because of this, it is possible that the user agent is infected with malware, and that there is a "man in the browser". To protect against this, the request and response can be signed. This will ensure that they are not tampered with.

NOTE: If there is no man-in-the-browser or other intermediary, the requests and responses should not be vulnerable because they are sent over a confidential channel using TLS. This should not be assumed though in high-worth use cases.

How this message-level integrity can be added is defined in the OpenID Connect core specification and the OAuth auxiliary draft standard, JWT Secured Authorization Request (JAR). (The latter is compatible with the former and was drafted after OpenID Connect.) In these, the client creates a "request object" which is a JSON object containing all of the authorization parameters; the app then signs or encrypts this and encodes it as a JSON Web Token (JWT). This is transmitted either on the query string (through the user's browser) or as a URI (also passed via the browser). If the request URI technique is used, the OAuth server will fetch it, obtaining a request object which is also signed or encrypted. If the request object is encrypted, this should be done using the OAuth server's public key. This is rarely the case, however, and signing is enough to close the attack vector laid out above.

The use of the request URI allows the OAuth server to authenticate the client using SSL. The API can also be protected using OAuth, basic authentication, mutual TLS or whatever else is fitting. This technique does impose the extra point-to-point connect though; this results in latency but also requires a line of sight between the OAuth server and the client, which isn't always possible (e.g., on a mobile device where running a Web server is cumbersome at best). This technique is called "passing by reference" whereas sending the request object via the query string is what is referred to as "passing by value".

Whichever method is used to transmit the request object and whether it is signed or encrypted, the Curity OAuth server follows certain rules about what values must be present in the request object and what to do if they are also included in the query string. The specs say that any

client and make updates as needed. The combination delivers a standard compliant Create, Read, Update, and Delete (CRUD) API for dynamic clients.

To ensure that only authorized organizations are allowed to register, some form of authentication is needed on the DCR endpoint. Since we have the PKI in place, it makes sense to use mTLS and allow any client with a certificate from one of the PSD2 issuers to register. This will allow for a completely automated registration, since the TPP can register directly after they've been issued a certificate by eIDAS, and we can also allow them to update the client settings using the DCRM API. Since the TPP is registering and requesting tokens using their certificate, the tokens can be bound to the certificate.

Summary

Complying with the new regulations might not be an easy task. But, by creating your API platform using standardized building blocks, your journey will be a lot easier.

In this article we've shown you how to combine multiple standards, and using their strengths to cover all the use cases of the new regulations. We've shown you how to leverage the PKI of eIDAS to make the trust model more robust, and hopefully sleep better at night knowing that only approved TPPs are allowed to register and gain access to your APIs.

The standards mentioned are all open and can be implemented by anyone. Curity Identity Server