



MOBILE API SECURITY

Using Keys and Tokens



INTRODUCTION

Mobile apps commonly use APIs to interact with backend services and information. In 2016, time spent in mobile apps grew an impressive 69% year to year, reinforcing most company's mobile-first strategies, while also providing fresh and attractive targets for cybercriminals. As an API provider, protecting your business assets against information scraping, malicious activity, and denial of service attacks is critical in maintaining a reputable brand and maximizing profits.

Properly used, API keys and tokens play an important role in application security, efficiency, and usage tracking. Though simple in concept, API keys and tokens have a fair number of gotchas to watch out for.

CHAPTER 1

We'll start off with a very simple example of API key usage and iteratively enhance its API protection.

CHAPTER 2

We will move from keys to JWT tokens within several OAuth2 scenarios

CHAPTER 3

We will remove any user credentials and static secrets stored within the client and, even if a token is somehow compromised, we can minimize exposure to a single API call.

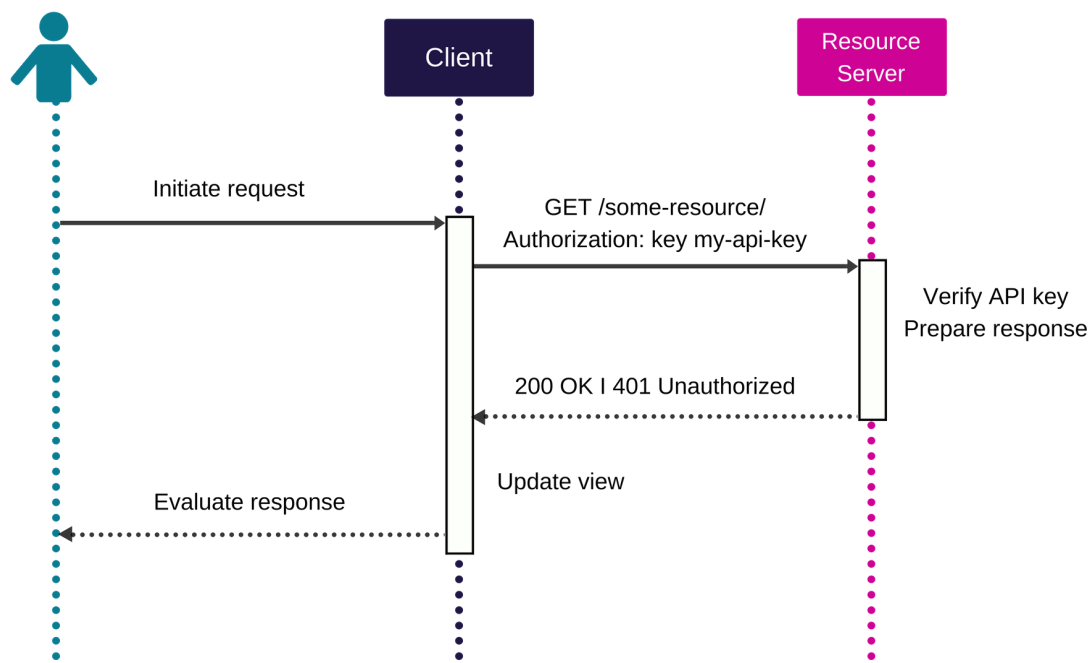


Mobile apps commonly use APIs to interact with backend services and information. In 2016, time spent in [mobile apps grew an impressive 69% year to year](#), reinforcing most company's mobile-first strategies, while also providing fresh and attractive [targets for cybercriminals](#). As an API provider, protecting your business assets against information scraping, malicious activity, and denial of service attacks is critical in maintaining a reputable brand and maximizing profits.

Properly used, API keys and tokens play an important role in application security, efficiency, and usage tracking. Though simple in concept, API keys and tokens have a fair number of gotchas to watch out for.

START WITH A SIMPLE APP ID KEY

The simplest API key is just an application or developer ID string. To use an API, the developer registers his application with the API service and receives a unique ID to use when making API requests.



In the sequence diagram, the *client* is a mobile application. The *resource owner* is the application user, and a *resource server* is a backend server interacting with the client through API calls. We will use [OAuth2](#) terminology as much as possible.

With each API call, the client passes the API key within the HTTP request. It is generally preferred to send the API key as part of the authorization header, for example:

```
authorization: key some-client-id
```

URLs are often logged, so if the API key is passed as a query parameter, it could show up in client logs and be easily observed, as demonstrated by this past [Facebook vulnerability](#).

This initial API key approach offers some basic protection. Any application making an API call will be rejected if the call does not contain a recognized ID. Different applications with different keys could also have different permission scopes associated with those keys; for example, one app could have read-only access while another may be granted administrative access to the same backend services.

Keys can be used to gather basic statistics about API usage such as call counting or traffic sourcing, perhaps rejecting calls from non-app user agents. Importantly, most API services use calling statistics to enforce rate limits per application to provide different tiers of service or reject suspiciously high frequency calling patterns.

One obvious weakness with this simple approach is that the API call and key are passed in the clear. A [man in the middle attack](#) could successfully modify any API call or reverse engineer the API and use the observed API key to make its own malicious API calls. The compromised API key cannot be blacklisted without breaking existing application instances and requiring an upgrade of the entire installed base.

SECURE THE COMMUNICATION CHANNEL

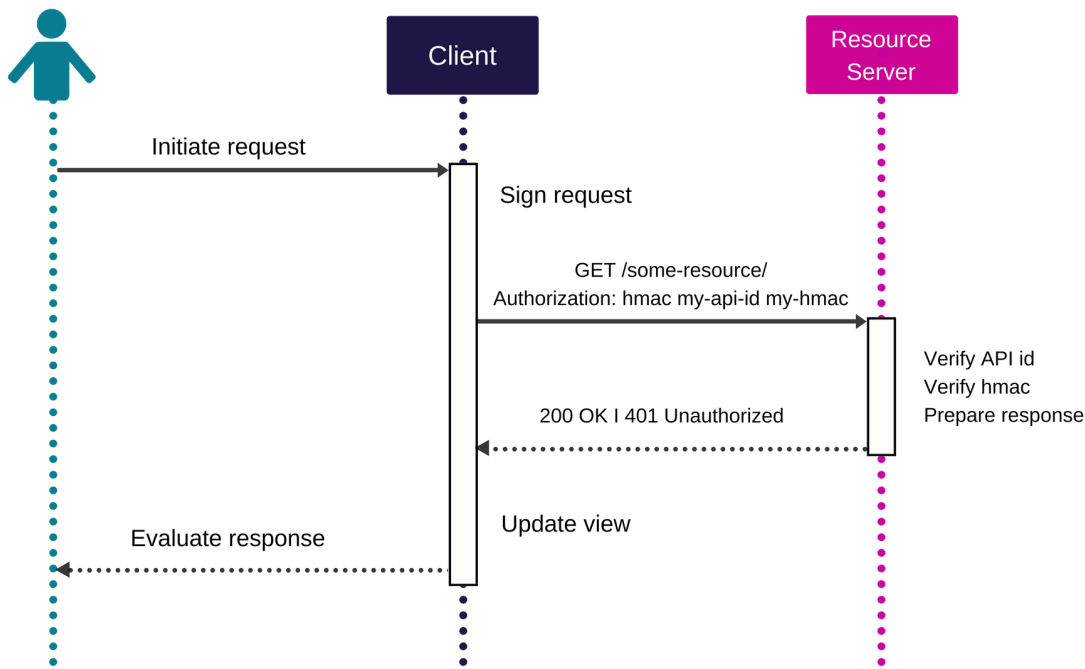
[Transport Level Security](#) (TLS) is a standard approach to securing an HTTP channel for confidentiality, integrity, and authentication. With [mutual TLS](#), client and server exchange and verify each other's public keys. With [certificate pinning](#), the client and server know which public keys to expect, and they compare the actual exchanged keys with the expected ones, rather than verifying through a hierarchical chain of certificates. The client and server must keep their own private keys secure. Once the keys are verified, the client and server negotiate a shared secret, message authentication code (MAC), and encryption algorithms.

When running on an uncompromised mobile device, client traffic over TLS is reasonably safe from man in the middle attacks. Unfortunately, if an attacker can install your client application on a device he controls, he can use a packet sniffer to observe the public key exchange, and use that knowledge to decrypt the channel to observe the API key and reverse engineer your APIs. While he may not be able to observe traffic on other clients, he can now create his own malicious app, freely calling your API over a TLS-secure channel. So even when using TLS, you'll need additional security to prevent APIs called from unauthorized applications.

PREVENT API CALL TAMPERING

One of the first improvements we can make is to separate the API key into an ID and a shared secret. As before, the ID portion of the key is passed with each HTTP request, but the shared secret is used to sign and/or encrypt the information in transit.

To ensure message integrity, the client computes a message authentication code (MAC) for each request using the shared secret with an algorithm such as [HMAC SHA-256](#). Using the same secret, the server computes the received message MAC and compares it with the MAC sent in the request.



Though the secret is known by both client and server, that secret is never present in the communication channel. An attacker might somehow see the ID, but without the secret, he cannot properly sign the request. As it stands, an attacker can still deny or replay the request, but he cannot alter it. Examples built around this scheme include the [HAWK](#) HTTP authentication specification or the [Amazon S3 REST API signing and authorization scheme](#).

To further protect critical information from being observed, all or portions of a message can be encrypted before signing using key material derived from the shared secret.

SECURE THE SECRETS

We are starting to accumulate secrets on the client. We have the shared API secret and the client's private TLS key.

In its basic form, a secret will be a static constants with the installed application package using developer-friendly names like `SHARED_SECRET`. It won't take a junior hacker much time to extract that constant, and once he has it, your backend is compromised. As a first step, use [code obfuscators](#), to make it harder to locate and extract a secret constant. To go a bit further, consider [encoding a static secret in some computationally simple way](#), cut that encoding into small segments, and distribute them around the binary. Reassemble and decode the secret in memory as needed; never save it in persistent storage.

Though the public keys are not actually secrets, you want to obfuscate them as well. Their values can be observed, so if they are not obfuscated, they can be easily found and changed, making it easy to disable or spoof back-end traffic.

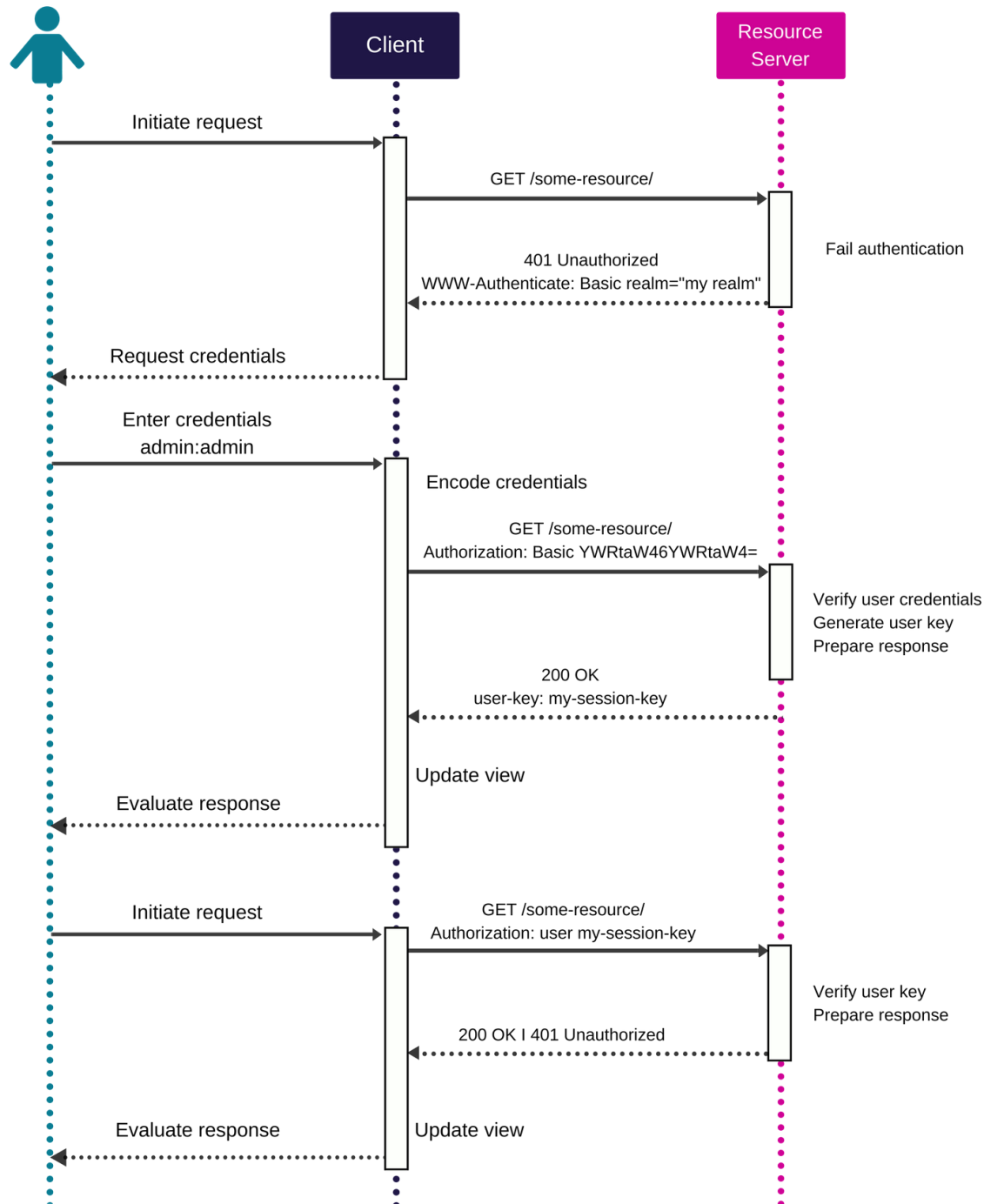
Regardless of your efforts, it is not a matter of if a secret will be stolen, but if the time and effort to steal it is worth the return. Make it as difficult as you can afford. If an API secret is stolen, we have the same revocability issues as before; all app instances will be compromised until we upgrade the entire installed base with a new secret and a new technique to obscure it.

We will return to this challenge again in [Chapter 2](#), when we discuss approaches which remove the secret from the app altogether.

HANDLE USER CREDENTIALS

We have enhanced API security using application keys, but we have not considered how to handle user credentials.

Starting simple, a client requests a user to provide user ID and password. Using [basic access authentication](#), the client encodes and passes the credentials to the server which verifies them. If the credentials are valid, the server can start a user session and return a user session key. Multiple authentications using the same credentials should always return different key strings.



Like we saw for application keys, we can use user keys to gather statistics and set authorization levels, but now we can do it with user granularity. Assuming we are using both app and user keys, the authorization levels for a user will be a function for both app and user; for example, a user may have administrative authorizations on one app while having only read permission on a different app, even though they are talking to the same backend server.

Similar to when using [HTTP cookies](#), session state likely must be maintained on the server. This may decrease server scalability, and if multiple servers can handle a user request, session data must be synchronized between them. We'll address this with user tokens in [Chapter 2](#).

So far, our application keys are static and therefore have infinite lifetimes. By contrast, user keys are created on the server, and they can and should expire. When a user key expires, the user must reauthenticate to continue making API calls, and session state is lost. Users do not like to logon repeatedly, so a policy decision needs to be made on key lifetimes. The longer the lifetime, the more user convenience, but if a user key should be compromised, it could be used maliciously for a longer time as well.

If a key can last longer than an application instance, then it must be stored in persistent storage on the client between app invocations. This is inherently less secure than if the key only exists in memory. Use secure storage such as [Keychain Services](#) for IOS and consider [SharedPreferences](#) for Android.

Unlike an application key, a user key can be revoked without breaking installed applications.

SUMMARY

We started off with a very simple example of API Key usage and iteratively enhanced its API protection to secure the communication channel and authorize both clients and users using API keys. In [Chapter 2](#), we will move from keys to JWT tokens within several OAuth2 scenarios, and in our final chapter, we will remove any user credentials and static secrets stored within the client and, even if a token is somehow compromised, we can minimize exposure to a single API call.

CHAPTER 2

API Tokens, OAuth2, and Disappearing Secrets

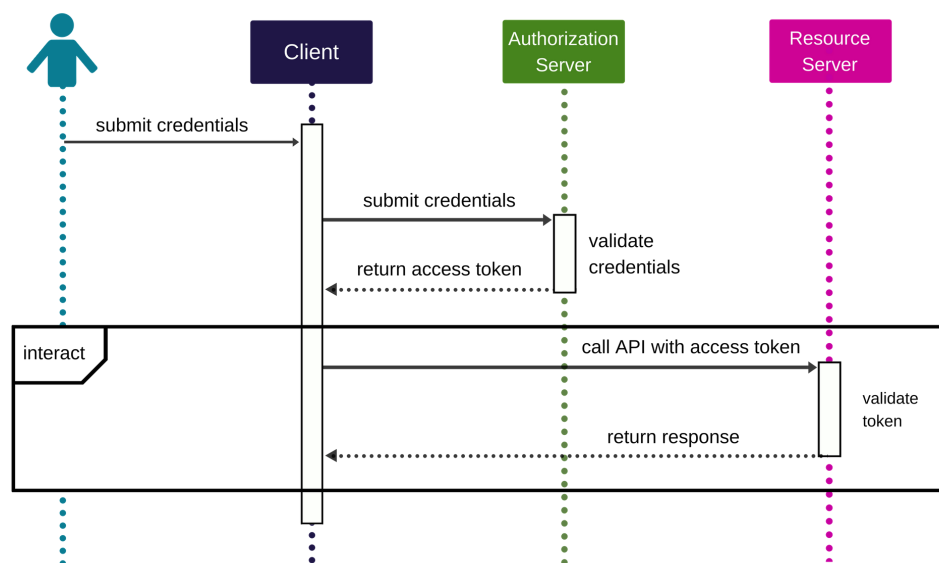
For all following scenarios, we assume that [TLS techniques](#) are used to keep the communications channel secure and we will use [OAuth2](#) terminology as much as possible.

At the end of [Chapter 1](#), we used [basic access authentication](#) to verify user credentials and start a user session on a server. If authentication succeeds, the server returns a session key to the client. The client adds the session key to API calls, and the server checks that the session key is currently valid and uses it as a key to look up any session state it is storing for the client.

SWITCH TO AN AUTHORIZATION TOKEN

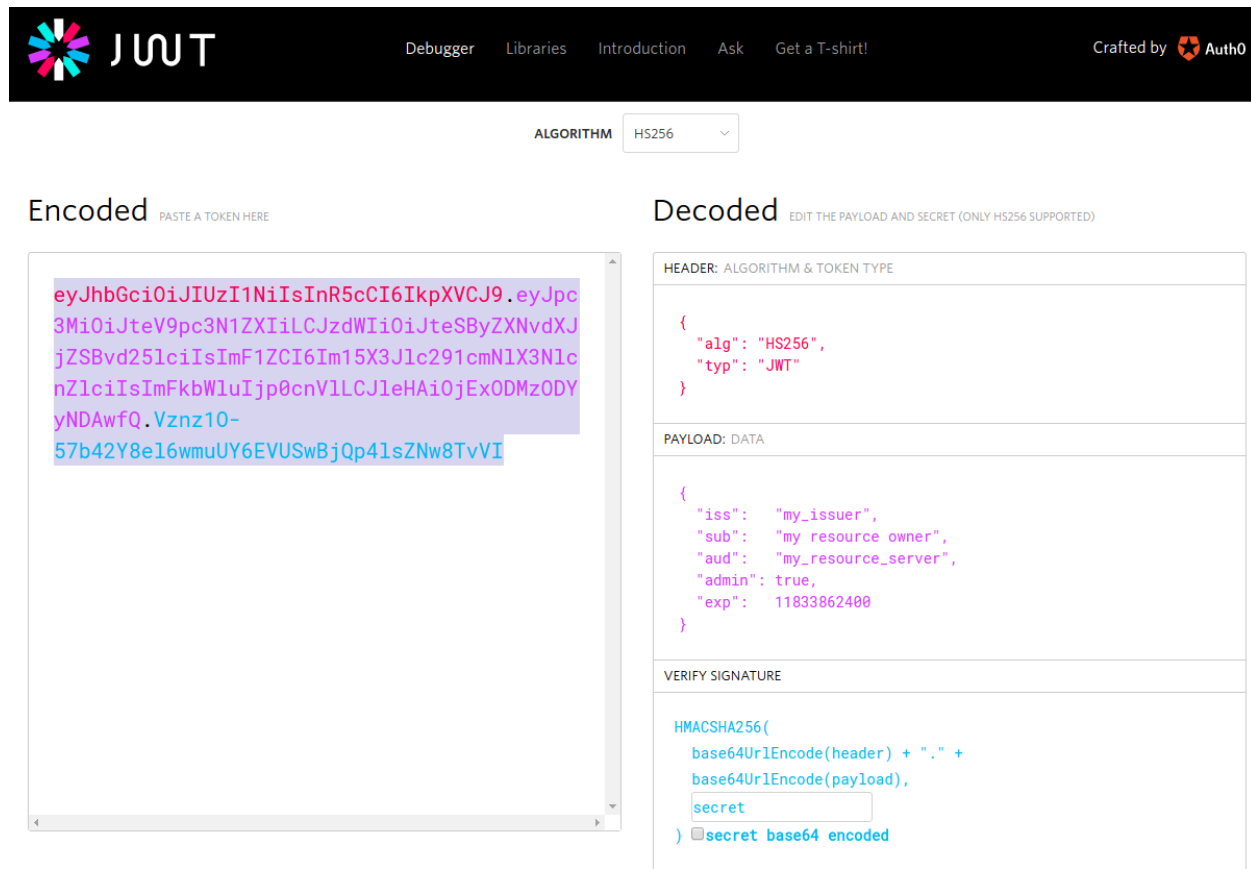
[OAuth2](#) has become a popular way to authorize user access to protected resources. The [OAuth2 authorization framework](#) defines several authorization grant flows. Though most service providers follow the spirit of the specifications, they often choose to implement just a part of the full specifications or sometimes implement capabilities differently than specified.

The OAuth2 flow which most closely resembles basic access authentication is called resource owner password credentials grant. In this flow, the mobile client directly obtains the resource owner's id and password credentials and passes them to its back-end resource server. The back-end server validates the credentials, and returns an access token to the client.



The returned access token looks just like a session key, and it is used by the client in the same way; the access token is provided in any http request requiring authorization to make an API call. As with basic access authentication, the access token stands in for the resource owner's credentials, so once the client has received an access token, the client should discard the credentials. The resource owner must trust that the client is not retaining these credentials.

Where an access token differs from a session key is in how the token is interpreted. [JSON Web Tokens \(JWT\)](#) are a secure, URL-safe method for representing claims and are often used as OAuth2 access tokens. The [JWT.io site](#) provides a convenient place to experiment with tokens.



The screenshot shows the JWT.io website interface. At the top, there's a navigation bar with links: Debugger, Libraries, Introduction, Ask, Get a T-shirt!, and a note 'Crafted by Auth0'. Below the navigation bar, there's a dropdown menu for 'ALGORITHM' set to 'HS256'. The main content area is split into two panels: 'Encoded' and 'Decoded'.

Encoded Panel: Labeled 'PASTE A TOKEN HERE'. It contains a text area with a JWT token: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJteV9pc3N1ZXIiLCJzZWUiOiJteSB5ZXNvdXJjZSBvd25lciIsImF1ZCI6Im15X3Jlc291cmNlX3NlcjZlciIsImFkbWluIjp0cnVlLCJleHAiOjExODMzODYyNDAwfQ.Vznz10-57b42Y8e16wmuUY6EVUSwBjQp41sZNw8TvVI`.

Decoded Panel: Labeled 'EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)'. It shows the decoded structure of the token:

- HEADER: ALGORITHM & TOKEN TYPE**

```
{  "alg": "HS256",  "typ": "JWT"}
```
- PAYLOAD: DATA**

```
{  "iss": "my_issuer",  "sub": "my_resource_owner",  "aud": "my_resource_server",  "admin": true,  "exp": 11833862400}
```
- VERIFY SIGNATURE**

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  secret  ) secret base64 encoded
```

A JWT contains a JSON formatted payload describing a set of claims. Common claims include:

- “iss” - identifies who issued the token
- “sub” - the principal subject of the claims, often the resource owner
- “aud” - the intended audience for the claims, often the resource server
- “exp” - the expiration timestamp of the claims

The access token is also called a bearer token and is passed with every API call, typically as an HTTP request header:

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJteV9pc3N1ZXliLCJzdWIiOiJteSByZXNvdXJjZSBvd25lcilslmF1ZCI6Im15X3Jlc291cmNIX3NlcnZlcilslmFkbWluljp0cnVILCJleHAiOiJExODMzODYyNDAwfQ.Vznz1O-57b42Y8el6wmuUY6EVUSwBjQp4lsZNw8TvVI

There are different ways that the token can be validated by the resource server. One common approach is to sign the JWT token using a secret known to both the resource server and the authorizing service, which in this case is the resource server itself. An attacker cannot modify a token's claims without invalidating the signature. The client can read the claims if necessary, but it does not know the secret, so it cannot verify the token itself. Neither the user credentials nor the signing secret are stored on the client, so they cannot be extracted through reverse engineering the application.

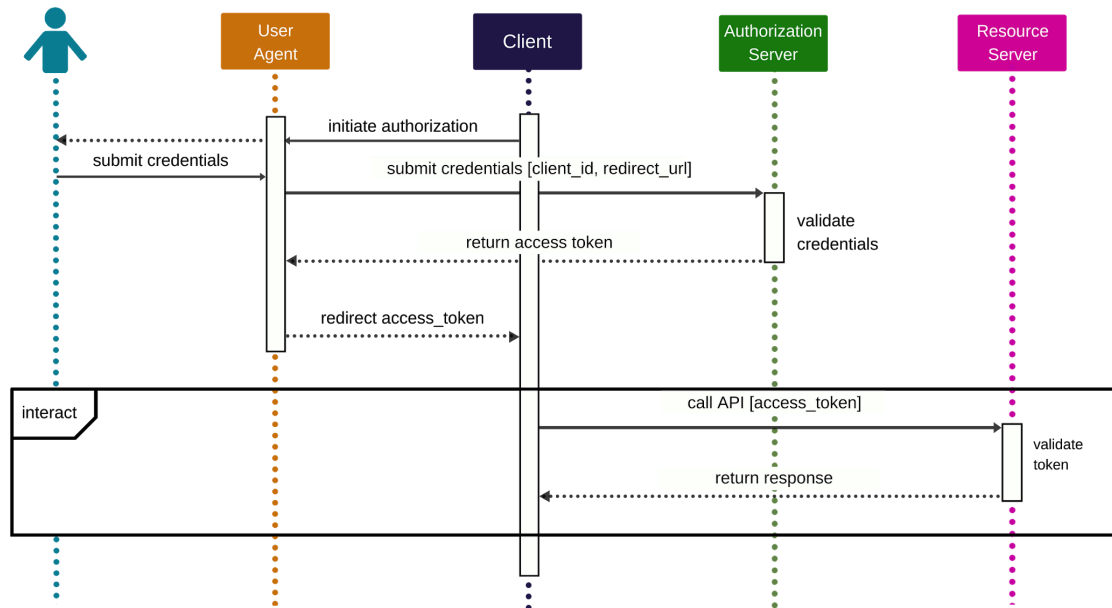
The server can validate the signature and check that the claims have not expired. With basic access authentication, the session key is used to retrieve state stored in the back end. If there are multiple servers which can service a request, then accessing this state and synchronizing it between servers can be a performance bottleneck. With access tokens, authorization state, such as the subject, is stored on the client, and that state is provided with every call. If the client maintains and provides the equivalent of all session state with every call, then the API protocol is stateless, and any server is free to handle a request independently of other servers, greatly improving system scalability.

Since the signing secret is not stored on the client, the secret can be changed on the server without requiring changes to the client. Access tokens are valid until they expire, so if the expiration window is long, a stolen token could be used successfully by an attacker for quite a while. If suspicious behavior is suspected, a token can be blacklisted on a group of servers. A revoked token will fail validation, triggering a new password credential sequence for the resource owner.

SEPARATE USER AUTHORIZATION FROM SERVICE

Security can be enhanced if resource authorization is separated from resource access. If separated, then only the authorization server needs to handle user credentials. The user credentials are never exposed to the client or the resource server.

In OAuth's implicit grant type, the authorization and resource servers are separated. The client sends the resource owner, through redirection, to the authorization server's website. The local user-agent, usually a browser, separate from the client, submits the credentials. The authorization server validates the credentials and redirects the access token through the user agent and back to the client.



Along with the user credentials, the authorization server can receive a client identifier and a requested scope. The authorization server can use the resource owner's id, the client's id, and the requested scope to determine which resources the client is allowed to access and how the client can access or modify those resources. A single authorization server can therefore manage the security policies for many users across many clients and many resources.

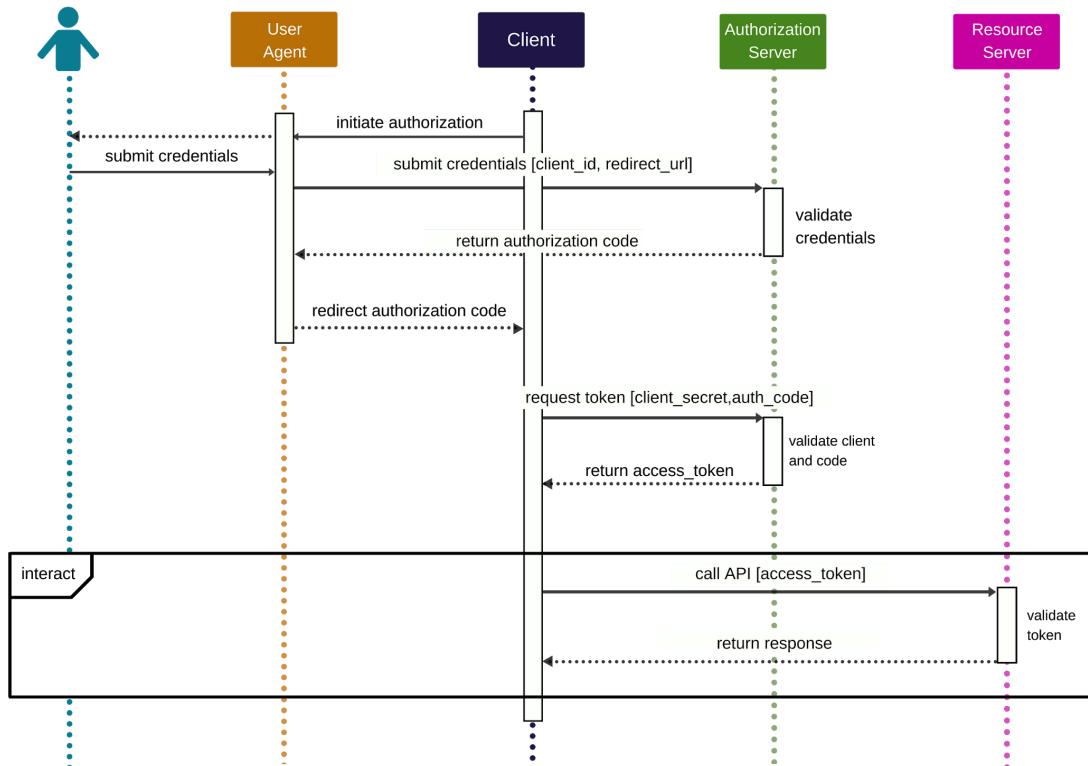
In the implicit grant type flow, the user is sent via redirect to the auth provider's web site, the user submits credentials, the auth provider verifies them for your app, and then the provider redirects back to your web application. The access token is directly passed in the URL #hash and can be extracted with a little JavaScript code. As the client is not able to keep a secret, there is none. Also, there is no such thing as a refresh token; the client web app will have to ask for a new token.

Access tokens passed from client to resource server can be verified by the resource server using the same secret used to sign them. Both authorization and resource servers share this secret, but this secret is never exposed to the client or user agent. A separate system administrates user credentials, client ids, resource access scope, and shared secrets.

This grant type is considered implicit because the resource server implicitly trusts that the requester submitting the access token is the client. This can be a risky assumption. For example, if an attacker can compromise the user-agent, the attacker may be able to view the access token and subsequently use the token to make valid but malicious API calls.

AUTHENTICATE THE APP, NOT JUST THE USER

The authorization grant type adds client app authentication to the implicit flow.



Authorization is split into two steps. In the first step, the resource owner provides credentials through the user agent, but this time only an authorization code is returned to the client. The client calls back to the authorization server with the authorization code and an authentication secret. The authorization server then returns the access token directly to the client.

By separating the authorization process into two steps, the access token does not flow through the user agent which is a big improvement in security. If the authorization code is exposed by the user agent, an attacker cannot make use of that code unless he can authenticate himself using the client secret.

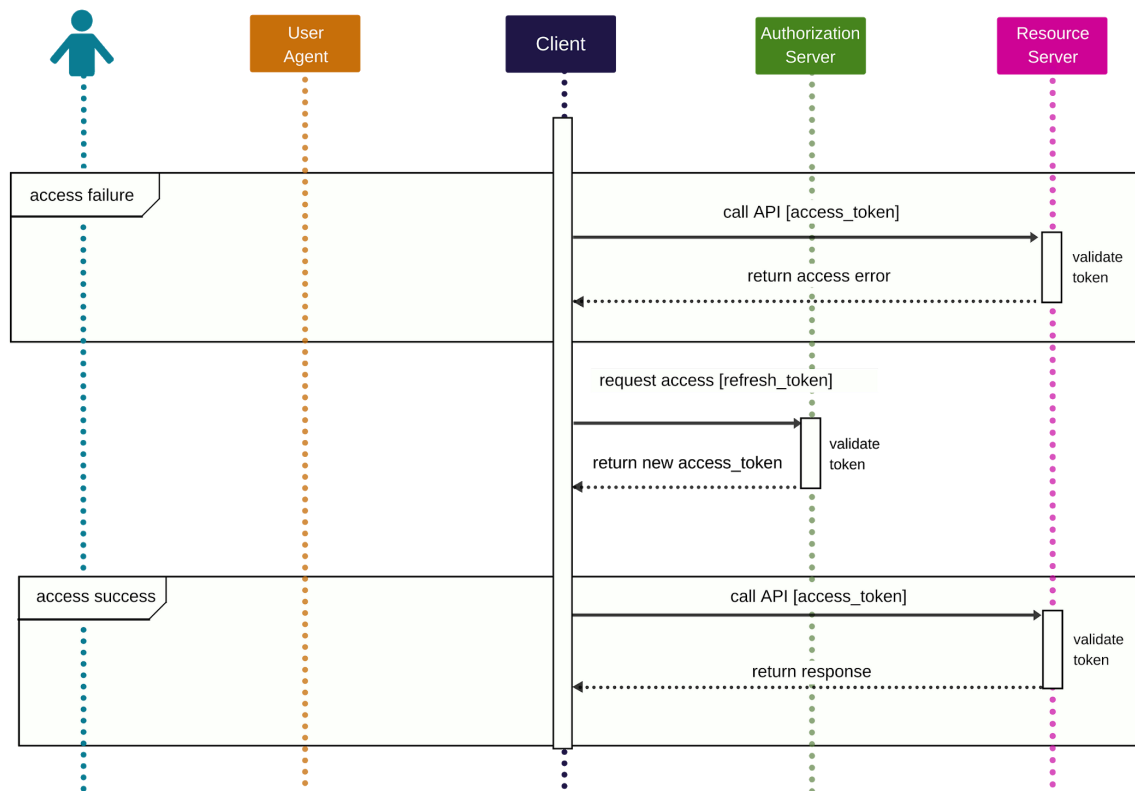
As we know from [Chapter 1](#), a static secret stored on the client is hard to hide from a determined attacker. We will discuss how to remove static secrets from the client shortly.

The OAuth2 spec does not require client app authentication beyond the authorization flow, but ideally both user access, through the access token, and app access, through the client secret, should be sent with each API call. The resource server should validate both before allowing access to resources.

SHORTEN TOKEN LIFETIMES

One great thing about access tokens is that they have an expiration date. If they are somehow exposed, they are only useful for a limited amount of time. Since the resource owner has to enter his credentials each time to get a token, if lifetimes are short then users will get annoyed at having to repeatedly reauthenticate. Conversely, if lifetimes are long, an exposed token can do a lot of damage before it expires. A token suspected of being stolen can be revoked, but this will not undo any damage which occurred before detection.

With the authorization grant type, OAuth2 optionally allows the use of refresh tokens. A refresh token can be received along with an access token during the initial authorization grant. Now an access token can be given a short expiration window, and when it expires, the refresh token can be sent to receive a fresh access token.



Refresh tokens have longer lifetimes than access tokens. The resource owner will not need to reauthenticate until the refresh token expires. If an access token is compromised, then its malicious use is limited to a short time. If a refresh token is compromised, it has a longer lifetime and can be used to generate additional access tokens. As such, refresh tokens are usually subject to strict storage requirements to ensure they are not leaked. They can also be blacklisted by the authorization server, which will trigger a new resource owner credentials session. I

strongly recommend authenticating the client app during any refresh token operation, but this is not required by the OAuth2 spec.

With each refresh, in addition to the new access token, a new refresh token can also be sent. The old refresh token can be immediately blacklisted, or more complicated token rotation schemes can be used to frustrate the malicious use of any individual token.

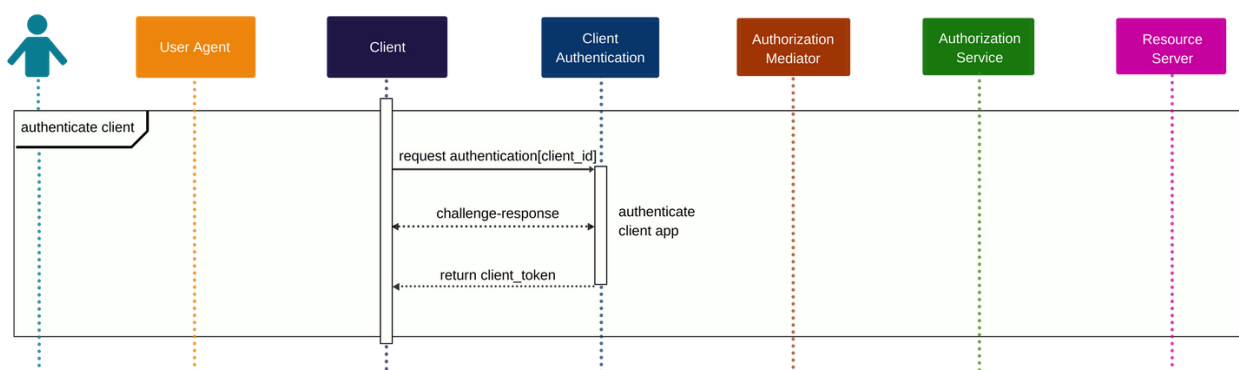
REMOVE THE CLIENT SECRET

The client secret should be used to authenticate the client app for:

- initial access token grant
- every access token refresh
- every API call

Many people do not send the client secret with every API call, arguing that since the client was authenticated using the secret during the authorization token grant, it is redundant. I prefer to include it as one additional check so you authenticate that it is still the client who is using the access token.

Unfortunately, the client secret is statically stored in the client app, and as such, it is vulnerable. We can remove the static secret from the app by following a playbook similar to how OAuth removes the resource owner's credentials from the client, by delegating to an app authentication service. In this case, the client must present its app credentials to the authentication service, the service authenticates the app, and the client receives its own authentication token. Instead of authorizing user access to the resources, this token authorizes client access to the resources.



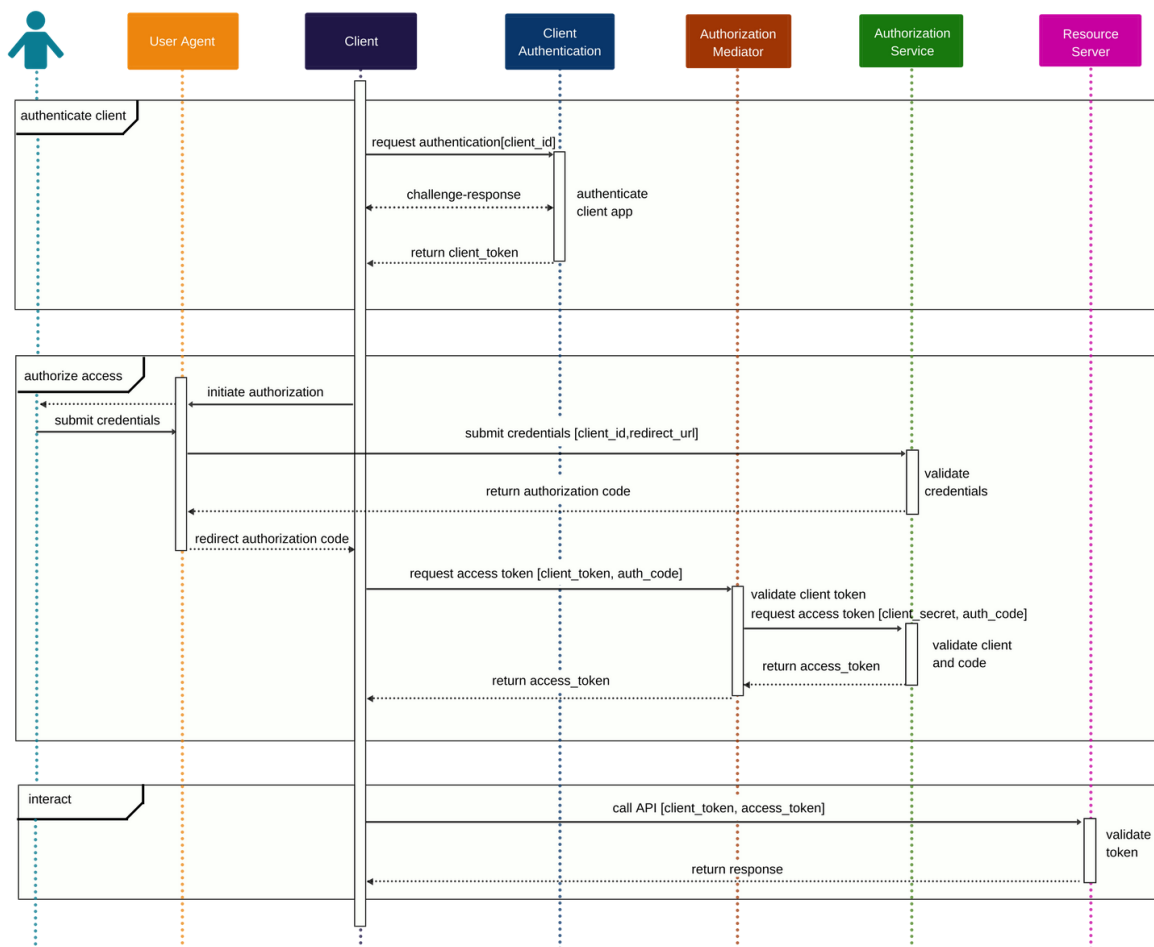
An app authentication service uses the unique characteristics of the client app to attest the app's integrity and authenticity. An example of a unique characteristic might be a simple hash of the application package. The integrity of this simple attestation depends on the integrity of the hashing computation. Such a simple scheme might be fairly easy to spoof.

A more robust attestation service might use a random set of high-coverage challenges to detect any app replacement, tampering, or signature replay. An example of this type of service is [Approov](#). If the responses satisfy the challenges, the authentication service returns an authenticating, time-limited client integrity token signed by the client secret. The token can be verified by the resource server, which also knows the client secret. If the attestation fails, the service still returns a time-limited token, but it will fail to be verified by the resource server. The attestation service and the resource server share the client secret, but it is no longer stored in the client.

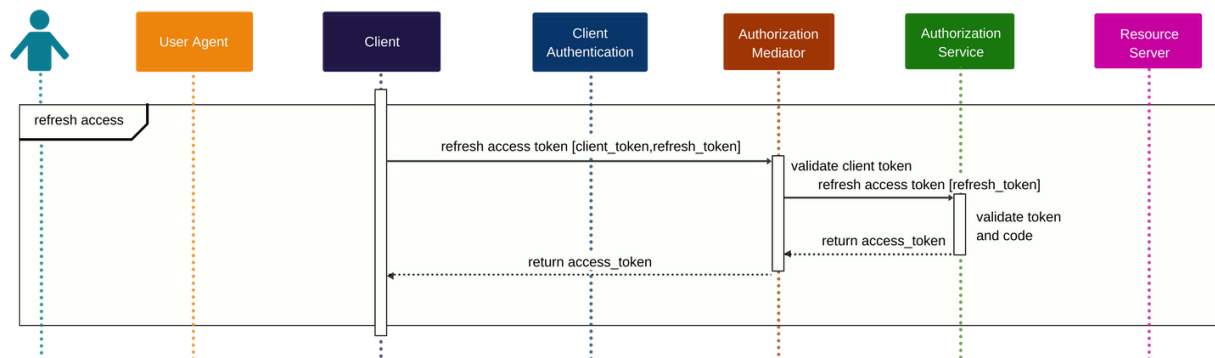
With every API call, both the client integrity token and the OAuth2 access token are sent in the request headers. The resource server is modified to validate both tokens before handling the request.

Unlike user authentication, client authentication requires no user interaction, so client integrity token lifetimes can be extremely short, and no refresh tokens are needed.

Since the client secret is no longer stored in the app, if the client secret is somehow exposed, it can be replaced with a fresh secret without requiring any changes to the installed client base.



The initial authorization grant flow must be modified to add app authentication. Since the grant spec requires a static client secret, a mediation server is introduced. The client authenticates itself with the app authentication service and receives a client integrity token. As before, it receives an authorization code from the resource owner's authentication. It sends both to the mediation server. If the client integrity token is valid, the mediator sends the authorization code and the client secret to the authorization server which returns the user access token which is then returned to the client, to be used for subsequent API calls until a new user access token is needed.



To refresh the user access token, the same mediation server can be used. This time if the client integrity token is valid, the refresh token is passed on to the authorization server and if valid, a fresh user access token is returned. Even though user access token refresh does not require app authentication, we were able to strengthen the refresh checks by adding the client integrity token validation to the mediator.

Using this approach, we are able to protect every API call made to our resource server validating both user and app authenticity, and we did it all without exposing user credentials or the client secret to the client itself.

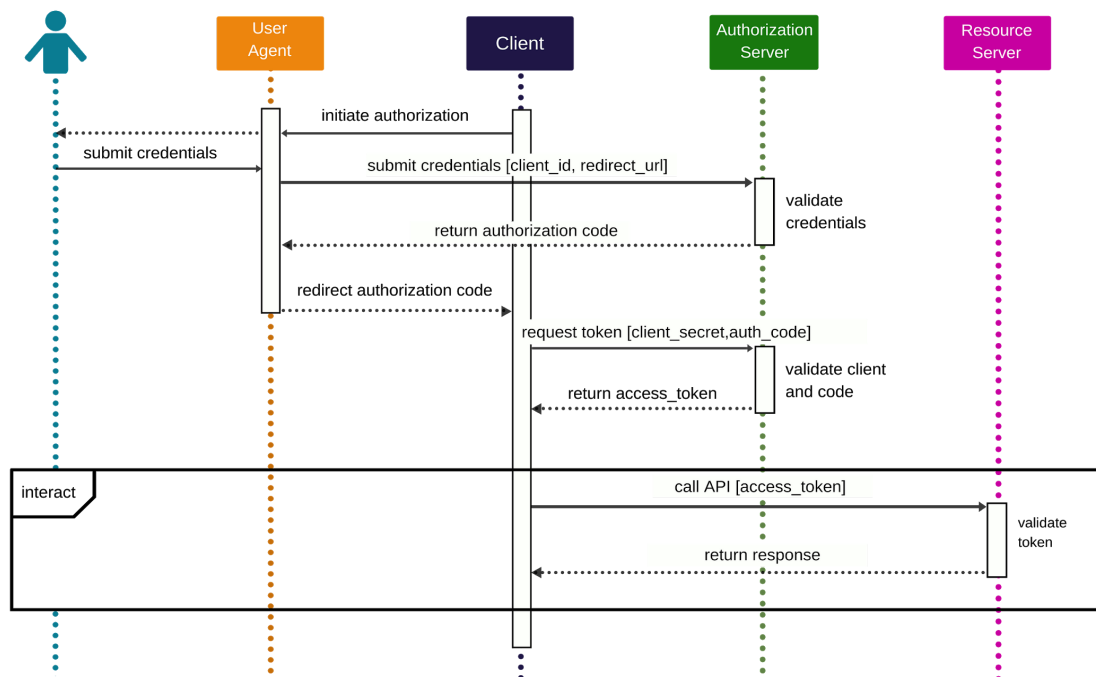
CHAPTER 3

OAuth2 Attacks, Protections, and Mediators

In this chapter, we discuss several attacks on OAuth2 authorization grant flows and common mitigations for them. We finish by extending the authorization mediator pattern introduced in [Chapter 2](#) to offer best practice security, while adapting to less secure Oauth2 implementations.

THE BASIC OAUTH2 AUTHORIZATION GRANT FLOW

Here's the basic OAuth2 authorization grant flow:



Authorization is a two step process. In the first step, the client delegates to a user agent, usually a browser. Through the user agent, the user as resource owner approves the client and presents credentials to an authentication server. The server returns an authorization code which is redirected back to the client. In the second step, the client authenticates itself and presents the authorization code back to the authorization server which, if satisfied, returns an access token.

OAuth2 had a rather [drawn out specification phase](#), and as a result, implementations vary between different service providers. Some optional features and newer extensions, which we'll describe below, have become important in preventing attacks which exploit the authorization grant flow.

For all scenarios, we assume that [TLS techniques](#) discussed in [Chapter 1](#) are used to keep the communications channels secure.

SPECIAL CONSIDERATIONS FOR MOBILE CLIENTS

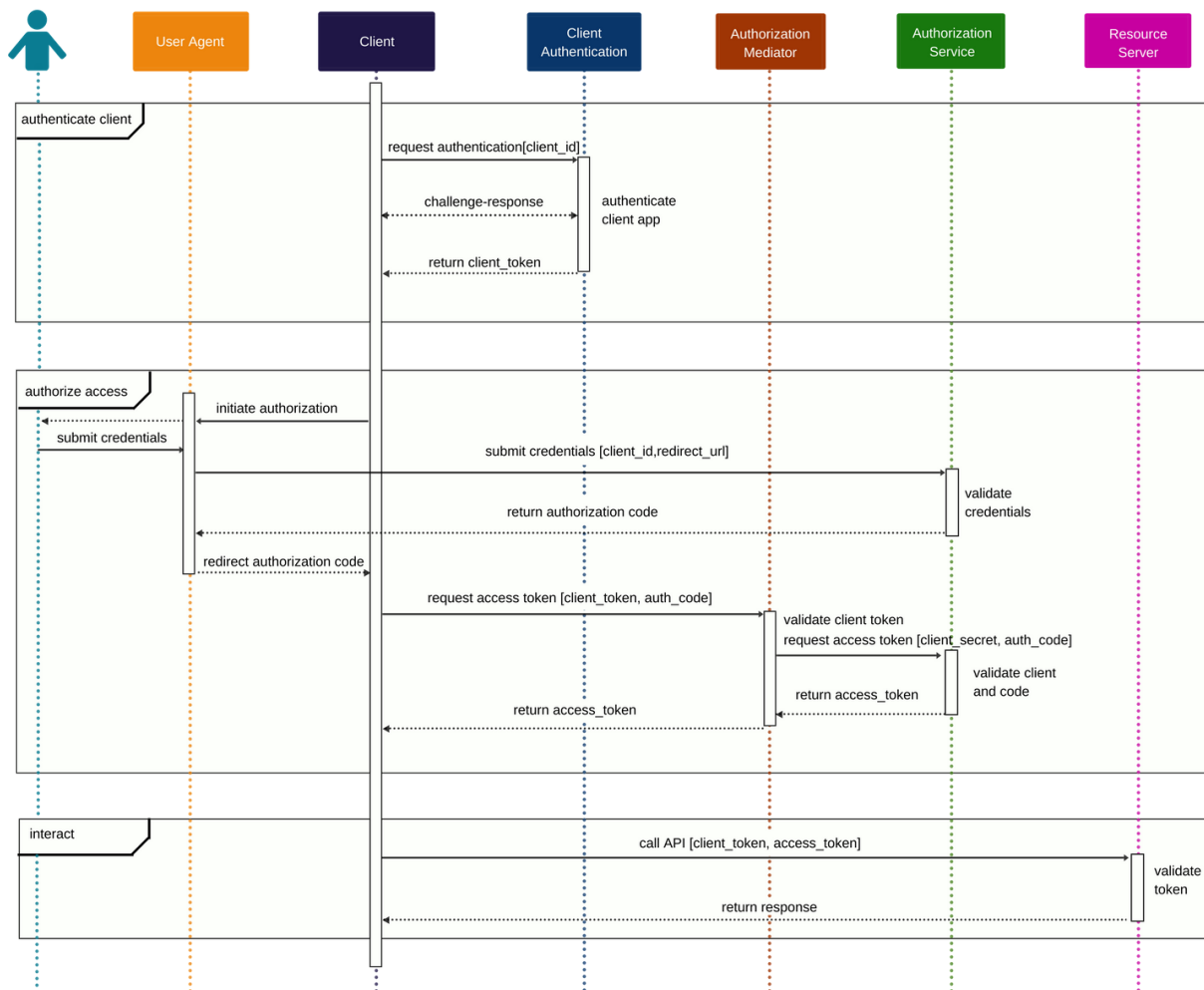
OAuth2 distinguishes between public and confidential clients. A confidential client is able to protect a secret, while a public client cannot make that guarantee. The original OAuth2 spec, [RFC6749](#), recommended that only confidential clients use the authorization grant flow, which uses a client secret for authentication, and allows the use of refresh tokens. Since public clients cannot protect a secret, they must use an implicit grant flow which does not authenticate the client nor allow refresh tokens.

OAuth2 considers mobile apps to be public clients. Despite [RFC6749](#)'s recommendations, most authorization service providers elected to implement the authorization grant flow for their mobile apps, and recent [IETF drafts for native OAuth2 clients](#) now appear to require such an authorization grant flow. This provides the user convenience of refresh tokens but at greater risk unless an alternative to client secrets is used for client authentication.

I KNOW YOUR SECRET

Consider a straightforward authorization grant implementation. It relies on a client id, client secret, and, optionally, a fixed set of redirect URIs shared between client, authorization server, and resource server. As these are all static values within the client app, they cannot be considered secure. If an attacker reverse engineers these values, it is a simple matter to construct a fake app which looks both cosmetically genuine and which appears perfectly authentic to the OAuth2 authorization grant flow. As [There's a Fake App for That](#) suggests, there are plenty of these apps causing mischief in the app stores.

In [Chapter 2](#), we discussed removing the client secret and replacing it with a dynamic attestation service.



In this flow, an authorization mediator checks the attestation service client integrity token for the authorization service. We will expand on this pattern later.

A dynamic attestation service, such as [Approov](#), provides extremely reliable positive authentication of untampered apps. Being dynamic, this service authenticates the app during both phases of the authorization grant flow as well as frequently during authorized operation. As a result and without relying on client secrets, every authorized API call is made by an authentic app for an authorized user.

I SEE YOUR AUTH CODE

Several attacks on grant authorization involve observing the authorization code. One approach relies on modifying the redirect URI which is used to redirect the authorization code back through the user agent to the client. The modified URI returns the authorization code to a malicious client instead. Client administrators optionally register a whitelist of redirect URIs with the authorization service to prevent this.

With mobile apps, things are more complicated. Mobile app redirect URIs typically use a custom URI scheme. For example, the URI

```
com.example.awesome:/redirect/here
```

might redirect from a mobile browser to example.com's awesome application running on the same device. These schemes must be registered with the operating system, and multiple applications can register against the same scheme. So even though the URI is preregistered with the authorization server, there is no guarantee that the correct application will receive the redirection.

With optional or exploitable static secrets, a malicious app could successfully convert the redirected authorization code into access and refresh tokens and start accessing the resource server. Imagine launching a real banking app, granting your permissions, presenting login credentials, and then wondering why your app seems stuck. While you are waiting, your authorization code has been stolen, and a malicious app is making API calls to empty your bank account.

CODE EXCHANGE PROTECTION

To mitigate authorization code interception scenarios, the [Proof Key for Code Exchange \(PKCE\) extension](#) was added as a requirement for OAuth2 public clients.

When using the PKCE extension, the client creates a cryptographically random code verifier key when initiating authorization. Two parameters, a code challenge and a code challenge method, are added to the initial client authorization request. For the “plain” method, the code challenge value is the same as the code verifier key value, and the code challenge method is a simple comparison.

When receiving the request, the authorization server notes the code verifier challenge and method and returns the authorization code as usual.

To convert the received authorization code to a token, the client must present its original code verifier along with the authorization code. Using the code challenge method, comparison for the plain case, the server will validate the code challenge before returning valid access and refresh tokens.

So even if a malicious client observes or receives the authorization code, it will be unable to present the correct code verifier to the authorization server.

If the authorization server wishes to remain stateless, it is acceptable for the server to cryptographically encode the code challenge and code challenge method into the authorization code itself.

Though the plain code challenge method is the default, it is really only for backward compatibility with initial implementations. The plain method fails if the attacker can observe the initial client request, which includes the code verifier in the clear.

An alternative and preferred challenge method is “S256”. With S256, the code challenge is a base 64 encoding of the SHA2 256 bit hash of the code verifier plain text.

Since the SHA2 transformation is practically irreversible, even if the code challenge is observed by the attacker, he cannot provide the original plain text code verifier required to obtain the access code.

DYNAMIC ATTESTATION PROTECTION

If PKCE is not implemented, dynamic attestation provides pretty good defense against hijacked authorization codes. Using the dynamic attestation flow above, whichever app receives the authorization code must properly attest in order to convert the authorization code into an access token. Assuming that only one instance of the authentic app can be running on the device, then only that authentic app will be able to convert the authorization code successfully.

It is possible that the malicious app could store the hijacked code and try to use it later or even pass the authorization code off device. Since it can only convert the authorization code to an access token from an attested client, it must take control of that client somehow and is limited to normal API call sequences through that client. Authorization code expiration times, device fingerprinting, and replay defenses can be added to frustrate these attacks.

THE EXTENDED AUTHORIZATION MEDIATOR

We introduced an authorization mediator in [Chapter 2](#) to adapt the authorization grant flow to use a dynamic attestation service. This moved the vulnerable shared secret off the client and provided a reliable positive authentication approach without requiring any changes to the existing OAuth2 authorization service.

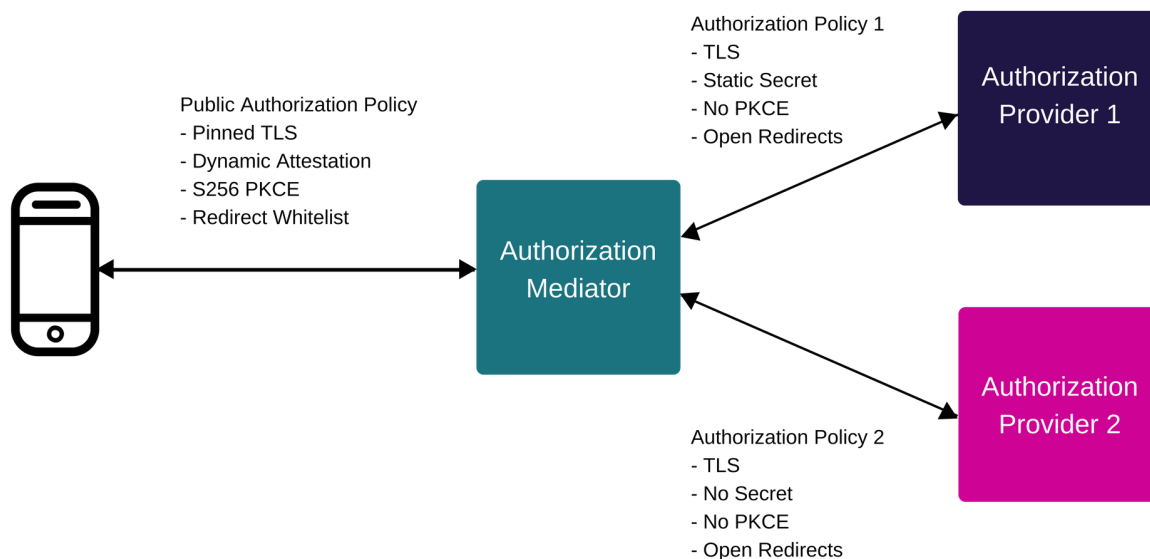
One of the challenges we alluded to earlier was the variety of inconsistent OAuth2 service implementations. Each service may ignore some required elements and/or require some optional ones. Implementations may also add some capabilities, like PKCE, over time.

If you want to directly interface with an authorization service, you are bound to whatever security policy it implements. As native apps are public, a “lite” implementation could be very insecure.

If you want to directly interface with multiple authorization services, then your client authorization library/SDK will grow and become difficult to manage. Any policy changes will require app updates.

Alternatively, if you use an authorization mediator, you can implement a single strong authorization policy between client and mediator, while the mediator itself can bridge to whatever policies are required by different OAuth2 authorization services. A consistent, best practice authorization policy is used by the public clients, where it is most needed, while the often weaker authorization provider policies are used in private behind the authorization mediator.

Service policy upgrades and additional service providers can be added at the mediator without requiring client upgrades.



If using a dynamic attestation service, the mediator could be used as a front end to both authorization and resource servers. For resource servers, it would filter out any invalid client requests before they hit the backend resource servers. This mediator could be integrated with existing [WAF](#) or [API gateway](#) solutions.

Using this approach, we are able to protect every API call made to our resource servers validating both user and app authenticity, independent of the capabilities of the authorization service used.

CONCLUSION

In [Chapter 1](#), we demonstrated use of client secrets and basic user authentication to protect API usage. In [Chapter 2](#), we introduced JWT tokens and described several OAuth2 user authentication schemes. On mobile devices, static secrets are problematic, so we replaced static secrets with dynamic client authentication, again using JWT tokens. Combining both user and app authentication services provides a robust defense against API abuse. In [Chapter 3](#), we discussed a few threat scenarios and extended the authorization mediator to provide strong authorization regardless of the strength of the supported OAuth2 providers.

ABOUT THE AUTHOR



Skip Hovsmith is a developer and customer advocate at CriticalBlue focusing on Mobile API Security.

CriticalBlue is a well-established part of the Edinburgh technological scene. We launched Approov to help our customers protect the business value that flows through their API channels. With Approov you control which apps access your mobile API in a secure and easily deployable manner.