



Optimizing External M2M Token Usage

Customized Solution Design

Contents

Document Control	1
Executive Summary	2
Problem	2
Solution	2
Architecture	3
Baseline: Direct Communication with Auth0 Endpoint	3
Use Case Scenario #1: Passthrough Middleware	4
Considerations	4
Use Case Scenario #2: Cache Tokens Using Middleware	5
Considerations	6
Proof-of-Concept	6
server.js	6
cache.js	7
Incorporating IP Address Allow List	8
Additional Considerations	8
Incorporating Client Secret Encryption	9
Additional Considerations	9
Use Case Scenario #3: Restricting the use of the /oauth/token endpoint	10
Appendix A	11

Document Control

Revision

Author	Date	Version	Comment
Peter Fernandez	17th February 2023	2.5	Updated document for publication

Executive Summary

This document is meant to examine various scenarios concerning the external use of access tokens obtained via a Client Credentials exchange. Discussion in this document is primarily aimed at use cases for Customer Identity Cloud (CIC) consumers who want to avoid running through their allotted Machine-to-Machine (M2M) token quota when utilizing externally implemented services. The document has been created because there is currently no out-of-the-box solution that CIC provides, and running through tokens quickly and using them ineffectively can be a pain point for our customers. Not to mention a costly one.

The solution options provided in this document typically address business-to-business (B2B) use cases, and also present some generic best practices on what to consider. They are also presented in a progressive manner, starting with some basic designs and then moving to more advanced implementations that explore different ways to restrict direct interaction with CIC's `/oauth/token` endpoint. For each option, we'll review guidance on implementation, discuss known issues and risks, and highlight any expected maintenance.

Problem

The [Client Credentials Flow](#) is a commonly used scenario in which a Machine-to-Machine (M2M) application passes along its *Client ID* and *Client Secret* to the Auth0¹ `/oauth/token` endpoint to obtain a new Access Token. The frequency of this transaction, precipitated by the external misuse of token lifespan, can mean M2M usage can quickly become expensive for a customer. To mitigate this, it can be necessary to implement countermeasures - such as token throttling and/or token caching - in order to force better utilization of the full potential and lifespan of an allocated token.

Solution

The guidance provided is intended to examine various scenarios for the caching, et al, of these valuable token resources. The goal is to get more use out of the tokens during their lifespan (by default, 24 hours) and to help customers decelerate external token requests in an effort to avoid hitting their allotted quota too quickly. See the [Architecture](#) section below for more details.

¹ Auth0 is synonymous with [Okta CIC](#).

Architecture

Below, we explore 3 use case scenarios that include various degrees of monitoring, caching, and/or restricting on the `/oauth/token` endpoint. The designs progress from simple to more complex and offer general approaches that have pros and cons to each. One solution may fit your use case better than the rest. For this reason, each must be considered in detail before implementation. The guidance provided can be helpful for a variety of scenarios. See below for a few examples in which an Auth0 customer may consider implementing some form of monitoring, caching, and/or restricting of the `/oauth/token` endpoint.

- An Auth0 customer with multiple teams who are responsible for products that have an M2M interaction between them.
- An Auth0 customer with multiple partners/clients that interact with their APIs using M2M tokens.
- An Auth0 customer with their own implementation that is requesting more tokens than needed.

First, we'll review a basic out-of-the-box flow to make sure we have familiarity with how a consumer can interact directly with the Auth0 `/oauth/token` endpoint. Then we'll look into implementing monitoring. Followed by token caching, where a consumer still has access to the Auth0 endpoints. And finally, for some more advanced scenarios, we examine ways in which to also restrict the use of the Auth0 `/oauth/token` endpoint. In these use case scenarios, middleware and caching can be implemented with different technologies, so each design can be modified to fit something you may already be using.

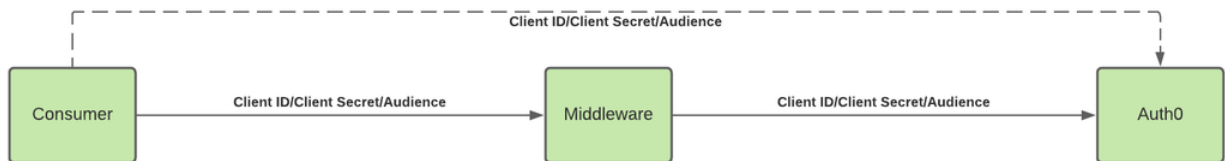
Baseline: Direct Communication with Auth0 Endpoint

The diagram below illustrates a very basic out-of-the-box flow where a consumer interacts directly with the Auth0 `/oauth/token` endpoint. Here the consumer sends [Client ID, Client Secret, Audience, and Grant Type](#) to get back an Access Token with an Expiration (in seconds). In this basic flow, both the consumer and Auth0 know the *Client Secret*. The consumer will be responsible for applying best practices by reusing the Access Token as much as possible. This will represent our starting point.



Use Case Scenario #1: Passthrough Middleware

In this scenario, we'll introduce a middleware that will sit between the consumer and the Auth0 endpoint for token allocation. Here the consumer sends the *Client ID*, *Client Secret*, and *Audience* through to the middleware, and the middleware will then interact with Auth0's `/oauth/token` endpoint to retrieve the Access Token. Once the token has been issued, it will then get sent back to the consumer. See the diagram below for the flow



This scenario does not offer guidance for caching/re-using tokens, nor does it quell token over usage. Instead, we simply introduce middleware as a mechanism to start monitoring token usage and also evaluate the habits of the consumer. There is a chance that the consumer could already be reusing their unexpired Access Tokens, but if that is not the case, this can be a good starting point in which to determine if there are additional steps that need to be taken (i.e. in order to stay within the bounds of your allotted M2M token quota).


Considerations

- Although the primary goal is to have all interactions between the consumer and the Auth0 `/oauth/token` endpoint pass through the middleware, the consumer also has the ability to bypass and interact directly with Auth0's `/oauth/token` endpoint. This will prevent you from getting an accurate picture of what interactions are taking place.
- An additional attack vector is introduced here since we are now sharing the *Client ID*, *Client Secret* and *Audience* with the middleware as well.
- Quota monitoring will require some maintenance and planning. It will be helpful to plot out how the gathered data will be used and what the thresholds are for acting on it.

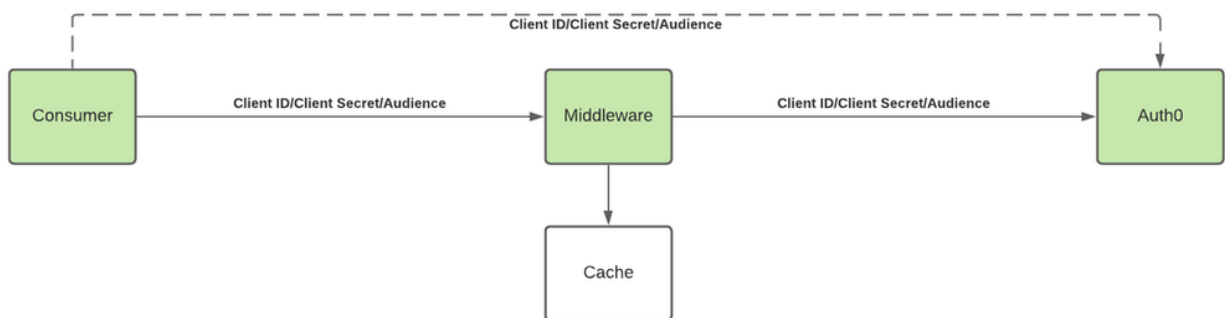
Use Case Scenario #2: Cache Tokens Using Middleware

Here we build upon [Use Case Scenario #1](#). In this scenario, we will introduce a cache to store the tokens so they can be reused by the consumer. In an ideal world, in-application caching - see [Appendix A](#) for further details - should be adopted as the preferred caching method of choice. In reality, though, the ability to enforce this is almost impossible. The notion then of building some form of independent cache mechanism - through which all (3rd party) M2M token allocation is directed - sounds appealing. However, the challenges associated with doing this can be prohibitive if best practice guidance is not adopted.

In this flow, the consumer sends the request to middleware, and the middleware is responsible for returning the cached response if the token is found. Where a valid cached token is found no additional request is sent to Auth0. If no valid token is found, the middleware is responsible for making the `/oauth/token` request and caching the response prior to returning it to the consumer. The token should be cached using the expiry provided by the `/oauth/token` response and the middleware should use a hash of the consumer's request as the key for the cache record. See the diagram below.



Whilst the Client Secret will still pass through the middleware, using a hash of the consumer's request also mitigates the need to store it in any (semi) persistent form.



The following can also be made to further restrict the access available to the consumer:

- Restrict access to the `/oauth/token` endpoint by enforcing an IP allowlist through a `credentials-exchange` Action. [See below for more details](#). This ensures that no M2M tokens can be generated without using the middleware.
- Encrypting the response in the cache. [See below for more details](#). This prevents tokens from being leaked by unauthorized access to the cache.

Considerations

- Although the primary goal is to have all interactions between the consumer and the Auth0 `/oauth/token` endpoint pass through the middleware, the consumer also has the ability to bypass and interact directly with Auth0's `/oauth/token` endpoint. This will prevent you from getting an accurate picture of what interactions are taking place.
- Additional attack vectors have been introduced here since we are now sharing the *Client ID*, *Client Secret* and *Audience* with the middleware and the cache will know the Access Token. See the guidance in [Appendix A](#) for additional advice on prospective countermeasures.
- Although we are implementing a cache to store and reuse tokens, some consumers may know how to bypass the middleware and abuse the `/oauth/token` endpoint. Because of this, it will be beneficial to continue monitoring of token quota as discussed in the section [above](#).
- Expect to monitor and clear your cache periodically to keep data from becoming stale and accumulating.

Proof-of-Concept

The sample code below provides a Proof-of-Concept (POC) example for storing the tokens in a memory cache on your system. Please note the POC is provided purely by way of illustration; no warranty - either assumed or otherwise - is provided. Though not included here, you will need to make a call to the `/oauth/token` endpoint as part of any implementation in order to get the token so it can be cached. You should also keep in mind that any memory cache will get wiped on termination, and factor that into your solution.

server.js

```
const app = require('express');
const getToken = require('./token');
const getCache = require('./cache');
const singleAudience = require('./singleAudience');

const router = app.Router();

router.post('/cached', singleAudience, async function (req, res) {
  console.log('Passthrough/Cached');

  res.json(await getCache(
    req.body.client_id,
    req.body.client_secret,
    req.body.audience,
```

```

    });
  })

  module.exports = router;

```

cache.js

```

const config = require('../config/config.json');
const NodeCache = require('node-cache');
const getToken = require('./token');
const time = require('./time');
const cache = new NodeCache();
const expiresInPercentage = (expiresIn, percentage) => {
  return expiresIn * percentage / 100;
};

const getCached = async (client_id, client_secret, audience) => {
  const keyCode = new Buffer.from(client_id + client_secret +
audience).toString('base64');

  const cached = cache.get(keyCode);

  if (cached) {
    return cached;
  }

  const response = await getToken({
    client_id,
    client_secret,
    audience,
  });

  cache.set(
    keyCode,
    response,
    response.expiration_timestamp - time.secondsToMilliseconds(
      expiresInPercentage(
        response.expires_in,
        config.expiresInPercentage
      )
    )
  );

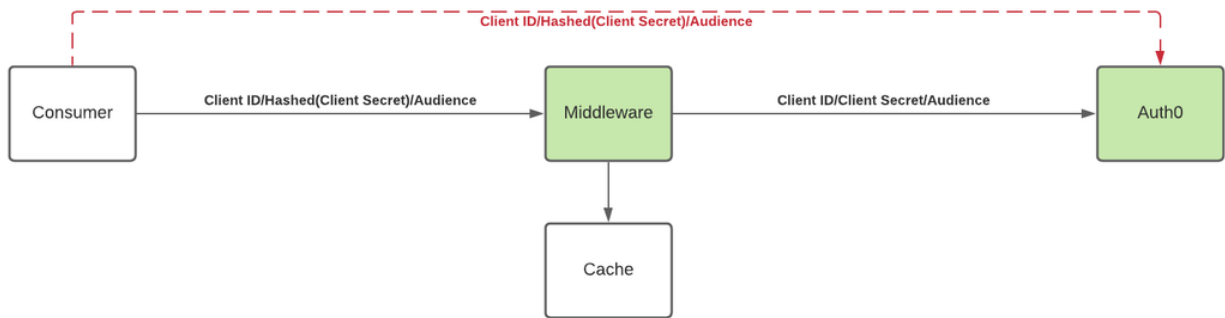
  return response;
};

module.exports = getCached;

```


Incorporating IP Address Allow List

In order to further restrict undesired M2M access to the `/oauth/token` endpoint, a `credentials-exchange` Action can be introduced to restrict access to the address space used by the caching middleware. The Action can also have a list of Resource Servers that should skip the IP Address Allow List check. A sample implementation of this action can be found below.



```

/**
 * Handler that will be called during the execution of a Client Credentials exchange.
 *
 * @param {Event} event - Details about client credentials grant request.
 * @param {CredentialsExchangeAPI} api - Interface whose methods can be used to change
 the behavior of client credentials grant.
 */
exports.onExecuteCredentialsExchange = async (event, api) => {
  const resourceServerDenylist = [
    '[https://[TENANT_DOMAIN]/api/v2/'
  ]

  if (resourceServerDenylist.includes(event.resource_server.identifier)) {
    return;
  }

  const ipAllowlist = [
    '[IP_ADDRESS]'
  ];

  if (!ipAllowlist.includes(event.request.ip)) {
    api.access.deny('invalid_request', "NO NO NO!");
  }
};

```

Additional Considerations

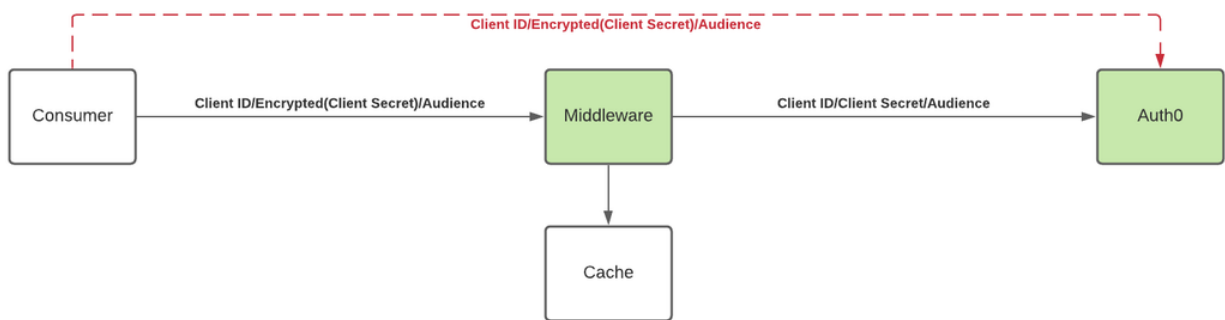
- Maintain the Allow Listed IP addresses if more than Middleware is going to do interaction with Auth0.

- Maintain the Blacklisted Resource Servers as APIs that require it are added in the Tenant
- Expect to monitor and clear your cache periodically to keep data from becoming stale.

Incorporating *Client Secret* Encryption

The scenario below offers another option for restricting direct access to the `/oauth/token` endpoint. By adding this restriction, the *Client Secret* has fewer components handling it and more security as a result.

The consumer will send an encrypted *Client Secret* through to the middleware, and then the middleware can decrypt the value and proceed with the process of calling the `/oauth/token` endpoint. The consumer does not know the *Client Secret* in this scenario.



Additional Considerations

- The *Client Secret* can be decrypted over time, but this shouldn't be a concern as the consumers are not expected to look for it in a B2B scenario.
- Additional processing to decrypt the *Client Secret* could cause issues if not done correctly.
- Additional work to deliver the encrypted *Client Secrets* to the consumers.
- Expect additional maintenance around
 - Encryption Secret Rotation
 - Delivering encrypted *Client Secrets* to consumers securely

Use Case Scenario #3: Restricting the use of the /oauth/token endpoint

It may be an option to *throttle* 3rd party token allocation via the use of some form of Access Token restriction mechanism. Using either the Auth0 [Client Credentials Exchange Hook](#) - or preferably the [credentials-exchange](#) Actions Trigger - noisy 3rd parties who make excessive M2M Access Token calls to Auth0 could be detected, and call from these offenders throttled by rejecting Access Token allocation. This would directly help address the problems described in the section [above](#), whilst at the same time forcing 3rd parties who do not want to be “penalized” to address matters - ideally by implementing their own in-application caching as discussed in [Appendix A](#).

Appendix A

In-application caching is essentially the process of using temporary storage within an M2M *Application* context, in which to store an *Access Token*. In-application caching is often, and most easily, implemented via the use of some technology stack compatible library/middleware which ideally should implement the caching mechanism in such a way as to observe the following:

- **Indexing that takes into consideration the context for Access Token use.** Access Tokens are designed to allow authorized access to an API and, as such, carry scope - the standards-based token claim that essentially defines what information can be obtained from an API. For M2M communication, scope directly relates to the access permission(s) granted to the token. Following the Principle of Least Privilege then (see [here](#)), any caching mechanism should prevent a consumer from obtaining a cached token that may have elevated access permissions.
- **Secure handling of the *Client Secret*.** Client Credentials flow makes use of a *Client ID* and a *Client Secret*. Whilst the *Client ID* is typically a public asset, the *Client Secret* is typically only known by an M2M Application and Auth0. The use of a library/middleware implementation reduces the risk of a *Client Secret* being misappropriated as, technically, execution is still within the application context. However, care should still be taken with the *Client Secret* value as it is a piece of security-sensitive information. Additionally, standard best practice should be observed by the M2M Application, where the *Client Secret* is stored in a secured environment space rather than embedding it directly into the application (direct embedding into the application can lead to the *Client Secret* being leaked via source control).
- **Temporary storage that is highly secure.** Whilst not excessively so in an in-application context, temporary storage does typically become a central location for multiple Access Tokens with varying degrees of access permission. So temporary storage needs to be highly secure in order to prevent any attack resulting in Access Token leakage (which could have serious security implications), and mechanisms such as the following should be considered:
 - Prefer to use in-memory/non-persistent cache storage over storage that is persistent/disk-based. This limits the potential attack surface.
 - If persistent/disk-based storage must be used - such as in scenarios where an M2M application is distributed in nature (i.e. multi-instance) - prefer to limit access from authorized endpoints only. Such as via IP Address AllowListing, etc. Again, this limits the surface for a potential attack.