BACHELOR'S THESIS 2021

Access Management in Cloud Networks

Hannes Larsson

ISSN 1651-2197
LU-CS/HBG-EX: 2021-03

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

Access Management in Cloud Networks

Comparing OAuth 2.0-Based Services On Securing Serverless Applications

LTH School of Engineering at Campus Helsingborg
Department of Computer Science

Bachelor thesis:
Hannes Larsson

Copyright Hannes Larsson
LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden
LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg
Printed in Sweden
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2021

Abstract

OAuth 2.0 is an open standard proposed by the IETF in 2012 to securely handle delegation of access, and it has since been integrated into a majority of systems worldwide to provide users with the ability to manage data access across platforms. One such platform that has been on the rise in recent years is the cloud infrastructure AWS and the deployment of serverless applications. However, due to cloud and serverless being young concepts and also very different from standard server infrastructure, security has to be reviewed more frequently so that unknown risks don't arise.

This thesis work aims to compare the cloud-native OAuth implementation Cognito with Curity, an OAuth implementation that was originally built as a standard server application. In order to make this comparison this thesis investigates notable differences between Cognito and Curity, shows examples of how to compare OAuth implementations in different scenarios and results in a presentation of situations where Curity is advantageous to Cognito.

While Cognito may be the native application for OAuth operations in AWS and a powerful tool for its simplicity, it is somewhat lacking compared to Curity as a standalone security service and quickly becomes pricey. This is partly due to OAuth features that haven't been developed in Cognito, and partly due to Cognito relying on other security services in AWS for certain features. Overall, Curity is worth the effort of deploying on the cloud in systems where Cognito is lacking in flexibility, lacking in features, handling a large amount of users or traffic, and when handling sensitive data.

This thesis contributes to research on the topic of access management in cloud environments, a field that has not been investigated much.

Keywords: Cloud, OAuth, AWS, Cognito, Curity

Sammanfattning

OAuth 2.0 är en öppen standard föreslagen av IETF under 2012 för att säkert hantera delegering av åtkomst, och den har sedan dess blivit integrerad i en stor mängd system över hela världen för att ge användare möjlighet att hantera dataåtkomst mellan olika plattformar. En sådan plattform som har stigit i populäritet under senare år är den molnbaserade infrastrukturen AWS och användningen av serverless applikationer. Men, eftersom att moln och serverless är relativt nya koncept och dessutom även väldigt annorlunda från standard serverstrukturer så behöver säkerhet undersökas mer frekvent så att okända säkerhetsrisker förhindras att uppstå.

Det här examensarbetet jämför den cloud-baserade OAuth implementationen Cognito med Curity, en OAuth-implementation som först utvecklades som en standard serverapplikation. För att göra denna jämförelse så undersöks märkbara skillnader mellan Curity och Cognito, exempel ges på hur OAuth-implementationer kan jämföras i olika scenarion. Detta resulterar i en presentation av situationer där Curity kan vara fördelaktig för Cognito.

Medan Cognito må vara standardapplikationen för OAuth-baserad aktivitet i AWS och ett kraftfullt verktyg trots dess enkelhet, så har den svagheter som Curity täcker och blir särskilt kostsamt i större system. Detta är delvis för att en del viktig OAuth-funktionalitet ej har implementerats i Cognito, men även för att Cognito förlitar sig mycket på andra säkerhetssystem i AWS för annan funktionalitet. Överlag så är Curity värd ansträngningen att distribuera i molnet när systemet behöver funktionalitet eller flexibilitet som Cognito saknar, när systemet hanterar stora mängder trafik och när systemet har handa om känslig data.

Det här examensarbetet ger ett bidrag till kunskapen om åtkomsthantering i molnmiljöer, ett område som ej har utforskats mycket.

Nyckelord: Moln, OAuth, AWS, Cognito, Curity

Foreword

I would like to thank Per-Gustaf Stenberg, Viraj Padte and Christian Nyberg for their contribution and support over the duration of this thesis, without their help I would not have been able to finish.

I would also like to thank Data Ductus for their financial and moral support in supplying licenses, a place to work, a supervisor to guide me and colleagues with an interest on the topic.

Finally I would like to thank Per-Gustaf again for all the pushes and help he has given me while I’ve been stressed with all there is to learn.

Table of contents

1 Introduction

1.1 Background

This thesis will be carried out in collaboration with Data Ductus. Data Ductus is a swedish consulting company based in Skellefteå, with several offices located both in and outside of Sweden. They provide several services, e.g. IoT services and management, as well as specially tailored solutions for high-security customers such as banks. The thesis will build upon their SaaS in AWS.

AWS (Amazon Web Services) is an IaaS Cloud Management System provided by Amazon that gives its customers the option of deploying their services on a serverless solution. This thesis work is focused on AWS Lambda, a service where customers deploy code for their serverless applications. AWS has a cloud-native IAM-service (identity access management) called Cognito which manages different aspects of security, such as authorization flows.

Cognito is based on the standard OAuth 2.0, a protocol for authorization, as well as OpenID Connect, an extension of OAuth 2.0 which handles authentication. These two protocols let users authenticate themselves in a large variety of ways, such as username/password or federated identities.

Although Cognito is a powerful tool, it was built to handle security in a large cloud network with a large amount of different services. According to Data Ductus Cognito is not flexible enough in handling security for serverless applications, and therefore wishes to explore the use of a different OAuth-based service, specifically Curity. Curity is a non-cloud hybrid IAM and API management service, also based on the OAuth 2.0 standard.

In order to compare Cognito and Curity, they wish to deploy Curity on the AWS network by developing an API that can be integrated into the API Gateway, and handle authorization of serverless applications with lambda's support for custom authorizers.

See figure 1.0.1 for the proposed system.

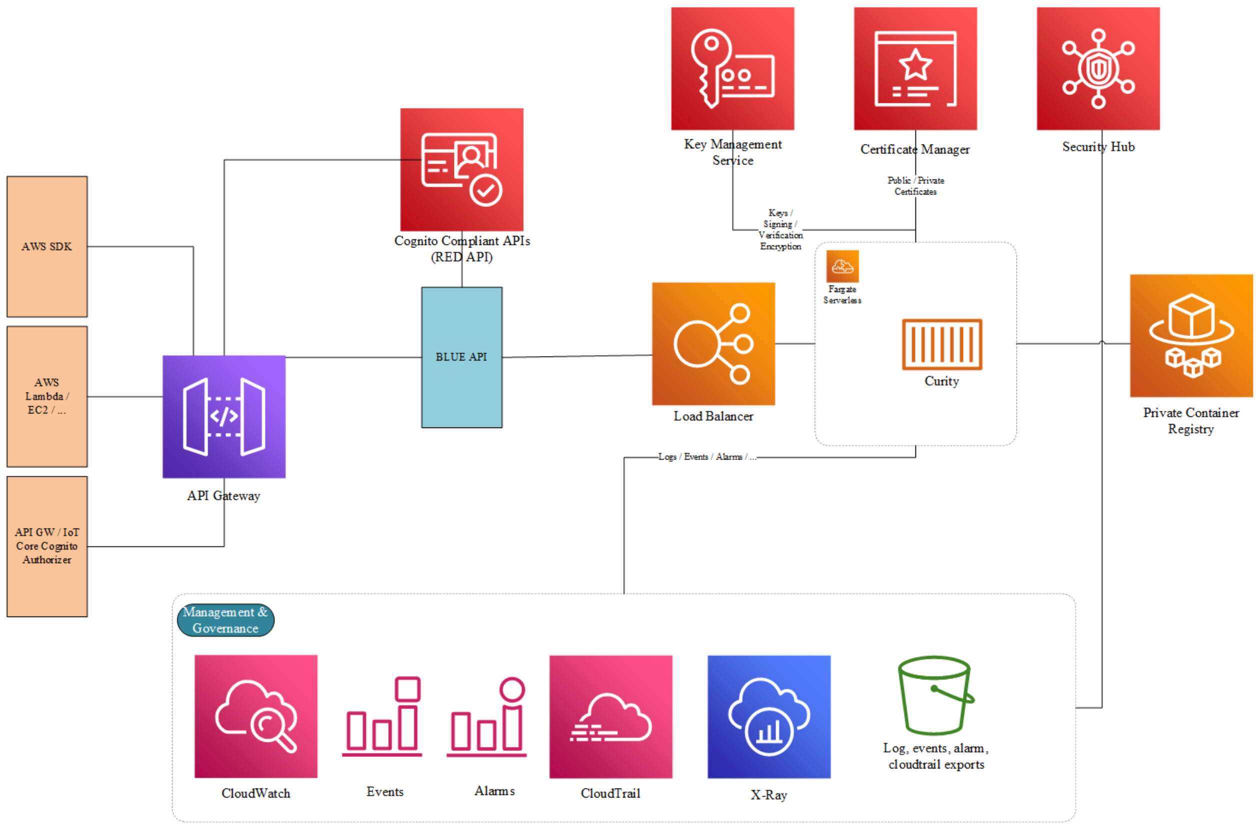
Key Management Service
Certificate Manager
Security Hub
Public / Private Certificates
Keys / Signing / Verification Encryption
Cognito Compliant APIs (RED API)
AWS SDK
AWS Lambda / EC2 / ...
API GW / IoT Core Cognito Authorizer
API Gateway
BLUE API
Load Balancer
Fargate Serverless
Curity
Private Container Registry
Logs / Events / Alarms / ...
Management & Governance
CloudWatch
Events
Alarms
CloudTrail
X-Ray
Log, events, alarm, cloudtrail exports

Figure 1.0.1 - Proposed system to handle authenticationflow via Curity.

1.2 Purpose

The purpose of this thesis is to determine whether Curity could be used as a suitable replacement for Cognito in order to handle security for serverless applications in an AWS cloud network.

1.3 Goal

This thesis will deploy a configuration of Curity on the AWS network in order to compare Cognito and Curity regarding maintaining security in serverless applications. The end goal of the thesis is to detail differences between Cognito and Curity, compare them and give answers as to when Curity might be advantageous over Cognito.

1.4 Problem Definitions

The thesis will answer the following questions:

1. What are key differences between Cognito and Curity?
2. How should Cognito and Curity be evaluated when deciding which is better at securing serverless applications?
3. In what situations would Curity be advantageous to Cognito?

1.5 Motivation

Using Curity instead of Cognito allows the architecture to be more cloud agnostic, allowing Data Ductus to use other or private cloud infrastructures if the need arises. Additionally, if the company becomes more efficient and flexible it can adhere to more of their customers' needs. Finally, as this thesis regards applications who's purpose is to maintain security, the company has a moral obligation to strive to maintain the highest standard. This thesis is motivated by an interest in security, and the lack of research on the topic.

1.6 Delimitations

The thesis is delimited to comparing the two services Cognito and Curity. The primary area of concern will be OAuth 2.0 authorization code flow.

1.7 Resources

To complete the thesis, the following resources will be used, either as research material or for practical use:

- Computer
- GitHub
- AWS SDK
- AWS reinvent
- AWS Documentation
- Curity Documentation
- OAuth Documentation
- JWT Documentation
- Supervisor at Data Ductus
- Supervisor at LTH

2 Technical Background

Before Cognito and Curity can be explored and compared, several services will have to be detailed as to what they are, what they do and why they are relevant. Due to both the complexity and size of OAuth and AWS, a brief overview of relevant services will be provided in this segment.

2.1 OAuth 2.0

OAuth 2.0 is an internationally standardized protocol based on its predecessor OAuth 1.0, and it was created in order to handle access to resources without having to pass credentials between systems. Understanding the intent behind OAuth is best done by describing a scenario of traditional client-server based authorization from before OAuth. See figure 2.1.1.

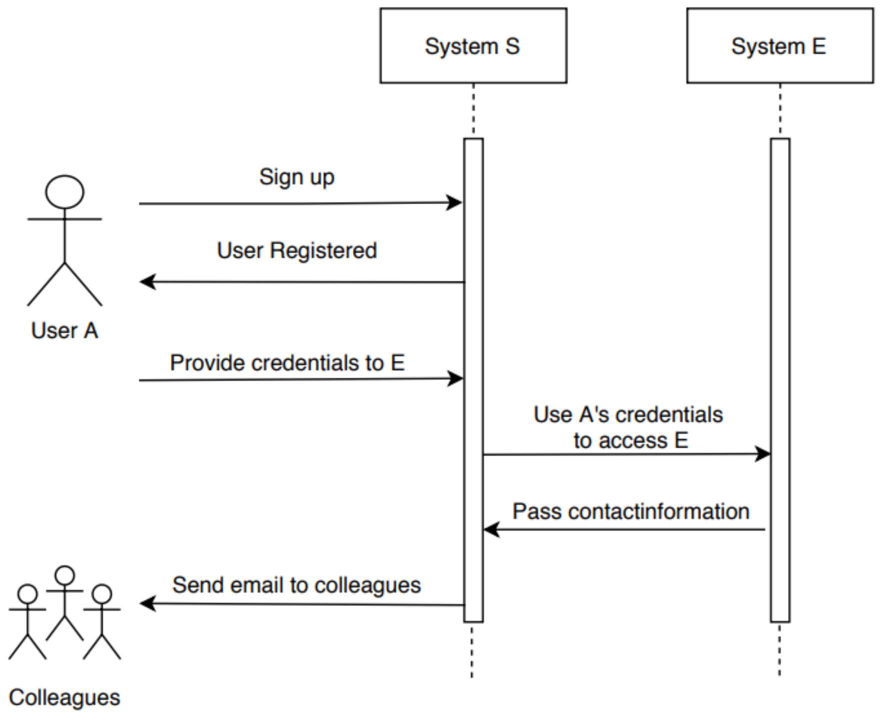

Figure 2.1.1 - Traditional client-server authorization.

Scenario: *User A has just signed up to Social Network S. S asks A if he would like to send an email to invite his colleagues to join his group in S, which A does. S prompts A to supply credentials to A's account on Email Service E, then proceeds to use A's credentials to access the account, fetch contactinformation and email an invitation to A's colleagues.*

This dated authorization-model is very insecure and inefficient for all parties involved. Examples of issues that this model brings are, as listed in the OAuth Authorization Framework RFC 6749, the following: [1]

1. Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.
2. Servers are required to support password authentication, despite the security weaknesses inherent in passwords.
3. Third-party applications gain overly broad access to the resource owner's protected resources without the resource owner being able to regulate it to a set duration or limited subset of resources.
4. Resource owners cannot revoke access to an individual third party without revoking access to all third parties, and must do so by changing the third party's password.
5. Compromise of any third-party application results in compromise of the end-user's password and all of the data protected by that password.

OAuth 2.0 solves these problems by instead of passing credentials to gain access to a resource, the client is granted access with a token. What kind of token, how to get one and how it is used depends on the grant type and the OAuth flow, however for this thesis the "authorization code flow" will be the one used. The authorization code flow uses the following steps to provide an access token, and to make it easier to understand, the same scenario as before will be described: [1]

1. Client S asks Resource Owner A, usually with a pop-up, if they wish to invite their colleagues to S.
2. Resource owner A clicks 'yes', and the client proceeds to make an authorization request to resource server E.
3. Resource server E returns an authorization page to client S.
4. Client S displays the authorization page to resource owner A.
5. The authorization page shows resource owner A what resource client S is trying to access, in this case the contactbook, and resource owner A provides credentials and approves the authorization request. Key difference here is that the credentials are not passed from Resource Owner A to Client S to Resource Server E, but instead supplied directly from A to E.

6. A short-lived authorization code is passed from resource server E to client S via resource owner A’s browser.
7. Client S uses a predefined back-channel communication to trade the authorization code for an access token, which contains information on what resources client S is allowed to access.
8. Client S provides the token to resource server E to gain access to resource owner A’s contactbook.

2.1.1 OpenID Connect

OpenID Connect is an open standard published in 2014, and an extension of the OAuth 2.0 protocol, which is designed to also include the use of the protocol for authentication. [12] OpenID Connect was developed because people were using OAuth for authentication despite that the access tokens on which the protocol is built contain no information on who the user is, and therefore aren't suited for it.

The extension provided by OpenID Connect comes in the form of an additional token sent in the process, the ID token, which contains information about the user. It also adds a UserInfo endpoint as an additional option, where one could store/fetch additional non-vital information about the user. Due to OpenID Connect being built on top of OAuth 2.0 it is commonly supported in OAuth 2.0 based systems. Both Cognito and Curity support OpenID Connect. [2][3]

2.1.2 PKCE

PKCE, or “Proof Key for Code Exchange” is an extension on top of the OAuth protocol that was introduced as an additional security layer in the code grant flow in order to prevent malicious activity originating from an intercepted authorization code. [13]

2.2 Amazon Web Service

Amazon Web Service, or AWS, is a cloud IAAS (Infrastructure as a service) which allows customers to host serverless digital solutions. Serverless here means that the customers do not need to maintain their own data centers but instead host their applications on Amazon's servers, with the advantage of only having to pay for the traffic they consume.

AWS is a large platform with more than 170 subservices, ranging from data storage to hosting websites, security maintenance, physical data transfer and more. The key services that will need to be understood will be listed here. [2]

2.2.1 AWS Identity Access Manager

The AWS IAM is a service that handles access to the AWS account, both authentication to access the account and the authorization to access its resources. Primary functions it serves are: [2]

1. The ability to add / remove users from an AWS account.
2. Create roles, groups and policies, in order to manage large user pools.
3. Support MFA (Multi-Factor Authentication) on the account.
4. Deploying resources onto the account.

2.2.2 AWS Fargate

Fargate is a scalable, pay-as-you-go service which deploys and runs cluster applications. The cluster applications consist of tasks, which are Docker containers with a configured amount of computational resources. Fargate can be configured to scale up or down these clusters according to demand, while only billing for the amount of computational resources used. [2]

2.2.3 AWS Elastic Container Service

AWS Elastic Container Service, or ECS, is a service which allows users to set up and deploy applications on the AWS cloud using containers. Containers allow us to package all parts of an application and deploy it as a single entity. ECS is similar to EC2 in that it handles instances of virtual machines, however ECS instead deploys a cluster of machines which serves as the deployment ground of your apps. [2]

2.2.4 AWS Elastic Container Registry

AWS ECR is Amazon's equivalent of the Docker Hub, a service that lets developers upload and manage Docker images. ECR is primarily used as the origin from which ECS instances are launched. [2]

2.2.5 AWS API Gateway

AWS API Gateway is, put simply, an API Gateway. An API Gateway is by definition an API management tool located inbetween a client and a collection of backend services. By using this service a developer can create, publish, maintain, monitor and secure APIs on any scale. The types of APIs that it supports are REST, HTTP and Websocket APIs. The APIs can in turn be used to access AWS resources/services and other web services, as well as data stored in the cloud. The API Gateway also lets developers create APIs that can be used in client applications, or alternatively make the APIs available to third-party app developers. [2]

2.2.6 AWS Lambda

AWS Lambda is AWS's primary serverless computing service. Serverless in this context means that the users do not need to provision or manage servers, instead all that is automatically handled serverside by Amazon. AWS Lambda is eventdriven, meaning that the code is run only on demand and it also scales automatically. The primary reasoning behind this is to only bill users for the amount of computing required, thus not paying for unnecessary uptime for servers. This type of service also has drawbacks, such as being prone to "cold starts" (increased computing time caused by initialization of an inactive function). [2]

This is the primary service where Curity's and Cognito's efficiency to secure serverless applications will be compared.

2.2.7 AWS Cognito

AWS Cognito is an access management service that handles access to individual applications or websites that are available on the AWS account. As has been mentioned in the introduction, Cognito is based on OAuth 2.0 and additionally supports OpenID Connect as well as SAML. SAML is basically a standard in exchanging authentication and authorization data between a service provider and an identity provider. Cognito supports these features with two different services, which are User Pools and Identity Pools. For a typical scenario of how these are used, see figure 2.2.1. [2]

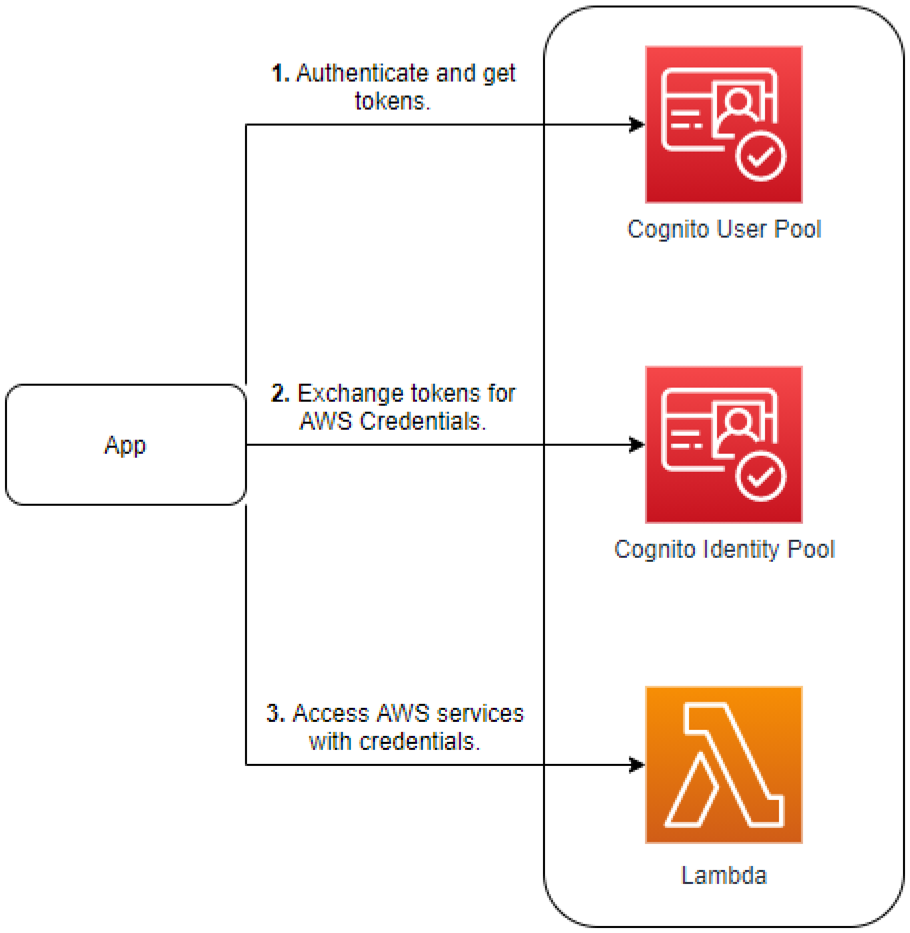
1. Authenticate and get tokens.
Cognito User Pool
App
2. Exchange tokens for AWS Credentials.
Cognito Identity Pool
3. Access AWS services with credentials.
Lambda

Figure 2.2.1 - Standard scenario of a user accessing an application hosted on AWS.

In the scenario above, a user authenticates through a Cognito user pool, receives a JWT, exchanges the token for temporary AWS credentials, then uses those credentials to access services within AWS.

The user pool is a user directory in Cognito. It allows users to sign in to an application through Cognito, either with a username/password combination or through a third party, such as Facebook, Google and SAML identity providers. The primary features of user pools are sign-up and sign-in services, a default but customizable web UI to sign in users, support for federated identities, user directory management and user profiles, customized workflows and user migration through lambda triggers. After authentication, the user is provided with a JWT (JSON Web Token) which can either be used to give secure access to applications and APIs, or be exchanged for temporary AWS credentials to access the account. If the token is traded for AWS Credentials, it is done so with the use of an identity pool.

Cognito identity pools, or federated identities, are used to create unique identities and fedeare users with identity providers. What this means is that you can grant users limited access to services on the AWS account. The access can be configured in a large number of ways, and the identity providers that can be supported are SAML and OpenID Connect providers, Google, Apple, Amazon, Facebook, Cognito User Pools and developer authenticated identities.

2.2.8 AWS Cloudwatch

AWS Cloudwatch is a log-service that is used to monitor applications and infrastructure in the cloud. This service will likely be used in logging data processing in order to compare Cognito and Curity, and also in troubleshooting errors. [2]

2.2.9 AWS S3 Buckets

Amazon simple storage, or Amazon S3, is a storage service. A very simplified description would be that it is effectively a very large Key-Value structure, with the key referencing the file name and the value referencing the contents of the file. S3 buckets might be used to host data for the lambda applications, logs from cloudwatch etc. [2]

2.3 Curity

Curity Identity Server, or just Curity, is an OAuth-based service that was developed to allow companies to secure their applications using configuration instead of code, thus effectively reducing complexity and maintenance. In addition to OAuth 2.0, it was also built with OpenID Connect and SCIM (System for Cross-Domain Identity Management) in mind, SCIM being a standard for automating the exchange of user identity information between identity domains. [3]

Curity is divided into three major modules, namely the authentication service, the token service and the user management service. The Authentication Service is a flexible framework handling multiple means of MFA, SSO and process workflows. The token service provides the foundation of the security aspects, such as highly customizable token management, token scope, claims and policies. In his review *API Management and Security* Alexei Balaganski highlights four strengths of Curity, and they are as follows: [10]

1. Comprehensive support for OAuth & OIDC open standards.
2. Combines flexible authentication with token-based API security-controls.
3. Reference "Phantom Token" architecture for privacy protection.
4. Modular API-driven architecture.

Since the initial release of the OAuth 2.0 framework it has been updated on a regular basis, examples being OAuth 2.0 Token Introspection from 2015 which is a standardized way to define a method for a protected resource to query an OAuth 2.0 authorization server to determine the active state of a token, and OAuth 2.0 Token Revocation from 2013 which proposes an additional endpoint for authorization servers which clients can use to notify the server when a token is no longer needed. What point 1 above refers to is the expansive support for updated features in the OAuth and OIDC open standards.

2.4 JSON Web Token

JSON Web Token, or JWT as it will be referred to from now on, is a standard format and is defined as “a string representing a set of claims encoded in a JWS or JWE, enabling the claims to be digitally signed or MACed and/or encrypted”. The claims in this context can be of different types however, they are always a pair representing a name/value combination, where the name is always a string and the value can be any JSON value. [11] A claim is an assertion that one subject, for example a user or a server, makes about itself or another subject. An example of this would be the “name” claim in an ID token that contains the name of a user. Claims are not to be confused with OAuth’s “scope”, which is effectively a group of claims.

The claims themselves can be grouped into two types, namely reserved and custom. Reserved are claims defined by the JWT-documentation, used to ensure interoperability with third-party, or external, applications, and custom are claims defined and used in an internal system. Custom claims need to be carefully named in order not to collide with reserved claims.

JWTs are most commonly used as a means of, but not limited to, authenticating or authorizing an entity, and are divided into two types as well, namely signed and encrypted tokens. JWTs that use JWS are signed tokens, and JWTs that use JWE are encrypted tokens. When authenticating a user the ID token is used, and while authorizing an entity the Access token is used.

The relevance JWTs hold in this thesis is that both OAuth 2.0 and OIDC are heavily dependent on this standard.

2.5 Docker

Docker is a platform used to develop, ship and run isolated applications, in the shape of containers. The containers are built from images, which in turn are a read-only template that contains a set of instructions to launch a system that can run the application. [14]

3 Method

This section will describe how the thesis was concluded. The thesis was divided into three phases; information gathering, analysis and results. Figure 3.0.1 provides an overview of the workflow for the thesis. The thesis work followed the phases Information Gathering, Analysis and Results as outlined in figure 3.0.1, and due to the nature of the thesis being a comparison between services, the first phase at times overlapped with the second and third phase. The phases underwent iterations where a singular piece of information was studied, for example a specific flow or security feature.

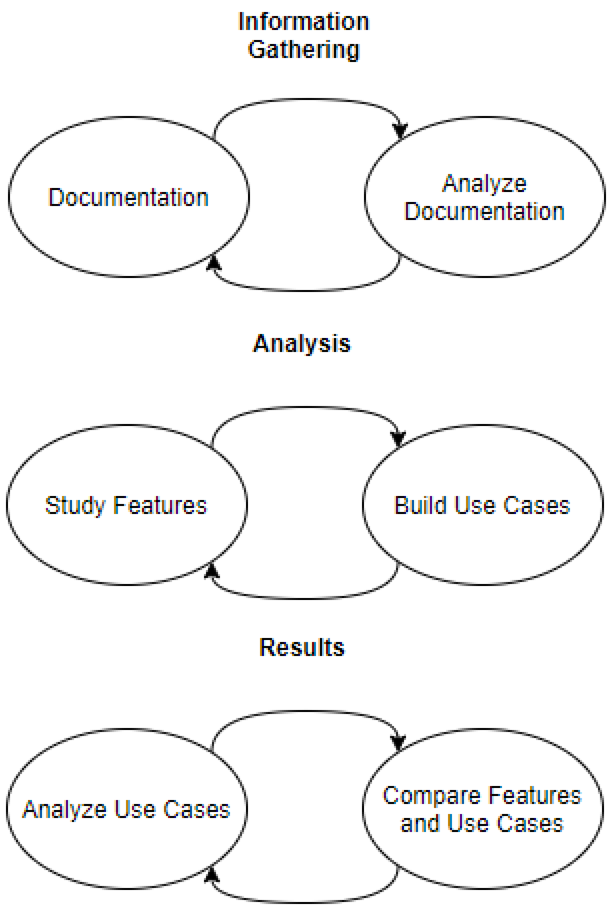

Figure 3.0.1 - Project Workflow

Here’s a brief description of the phases:

- **Information Gathering -** The majority of the services were initially identified, then the services were studied and summarized, so that it could be used as a reference to fall back on. In later iterations specific features were identified and studied, giving a basis for analysis.
- **Analysis -** Features and services gathered during the first phase were studied. Common use cases and security concerns regarding serverless and OAuth applications were identified, giving a basis for comparison between Cognito and Curity. Additionally areas of interest for comparison were identified. Development of use cases will be presented as a post-analysis in chapter 3.3, functioning as a bridge between analysis and results.
- **Results -** The use cases built and the features gathered during the second phase were analyzed resulting in a comparison between Curity and Cognito.

3.1 Phase One - Information Gathering

During phase one, or the information gathering phase, most of the work was done from home (due to Covid-19) and the time was spent identifying and learning about the primary services that will come into play when comparing Cognito and Curity. In order to maintain communication with the supervisors at Data Ductus, a slack group was created. Additionally, an account was configured on AWS, with unique users for the examinee and each of the supervisors at Data Ductus.

Due to the lack of research on the topics of OAuth-implementations in the cloud, Curity, serverless and AWS, information was primarily collected from raw documentation and articles on security. The documentation was summarized and used in the thesis document (such as technical background & tables of features) if considered important to understand the thesis, otherwise it was kept in local documents or memorized.

The first two months, the examinee spent the majority of the time learning about OAuth, Curity and AWS, partly by reading the documentation, partly by trying them out in practice. Below is shown flowcharts for two sample applications of the ‘Code Grant Flow’ in Cognito and Curity respectively. Since both systems were developed with the intent to learn, only a brief explanation of how they work will be provided. See figure 3.1.1.

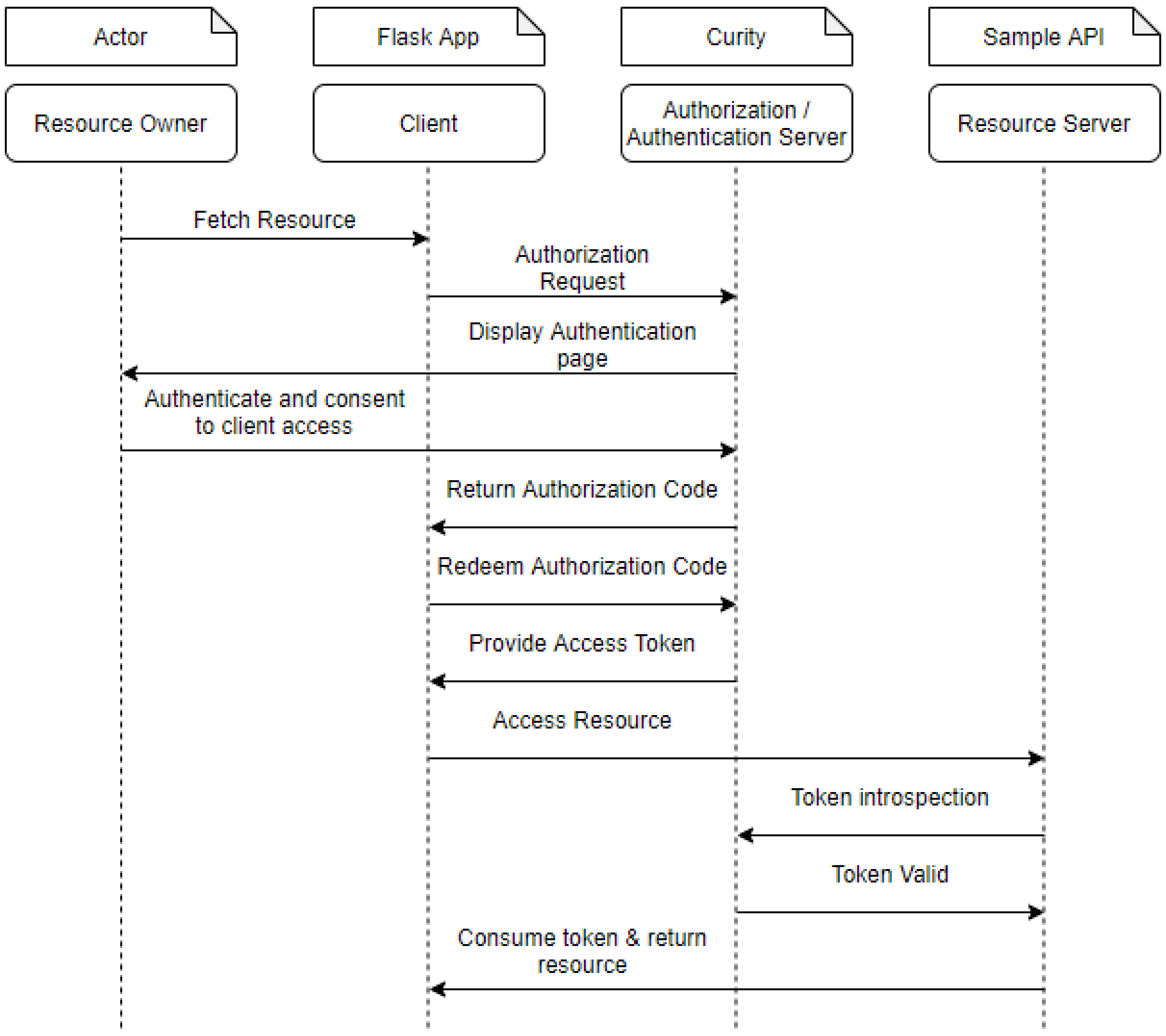
Actor
Flask App
Curity
Sample API
Resource Owner
Client
Authorization / Authentication Server
Resource Server
Fetch Resource
Authorization Request
Display Authentication page
Authenticate and consent to client access
Return Authorization Code
Redeem Authorization Code
Provide Access Token
Access Resource
Token introspection
Token Valid
Consume token & return resource

Figure 3.1.1 - Code Grant Flow (Curity), basic implementation on localhost

The system consists of a Flask application (hosted on localhost) as the client, Curity as the authorization server (also hosted on localhost) and a Python API combined with a SQLite database as the resource server. The resource owner (the examinee) starts the flow from the flask app, which in turn sends a request to the authorization endpoint in Curity with the parameters client ID, client secret, state, scope and redirect URI. Curity redirects to the authentication endpoint (in this implementation Curity's default authentication was used) and displays it to the user, which promptly logs in and consents to the client accessing resources. Curity then redirects to the previously provided redirect URI, where the client retrieves the authorization code from the URL and passes it to the token endpoint on Curity. Curity returns an access token, which the client passes to the API. The API verifies the token lifetime, scopes and access code, then consumes the token and returns the requested resources.

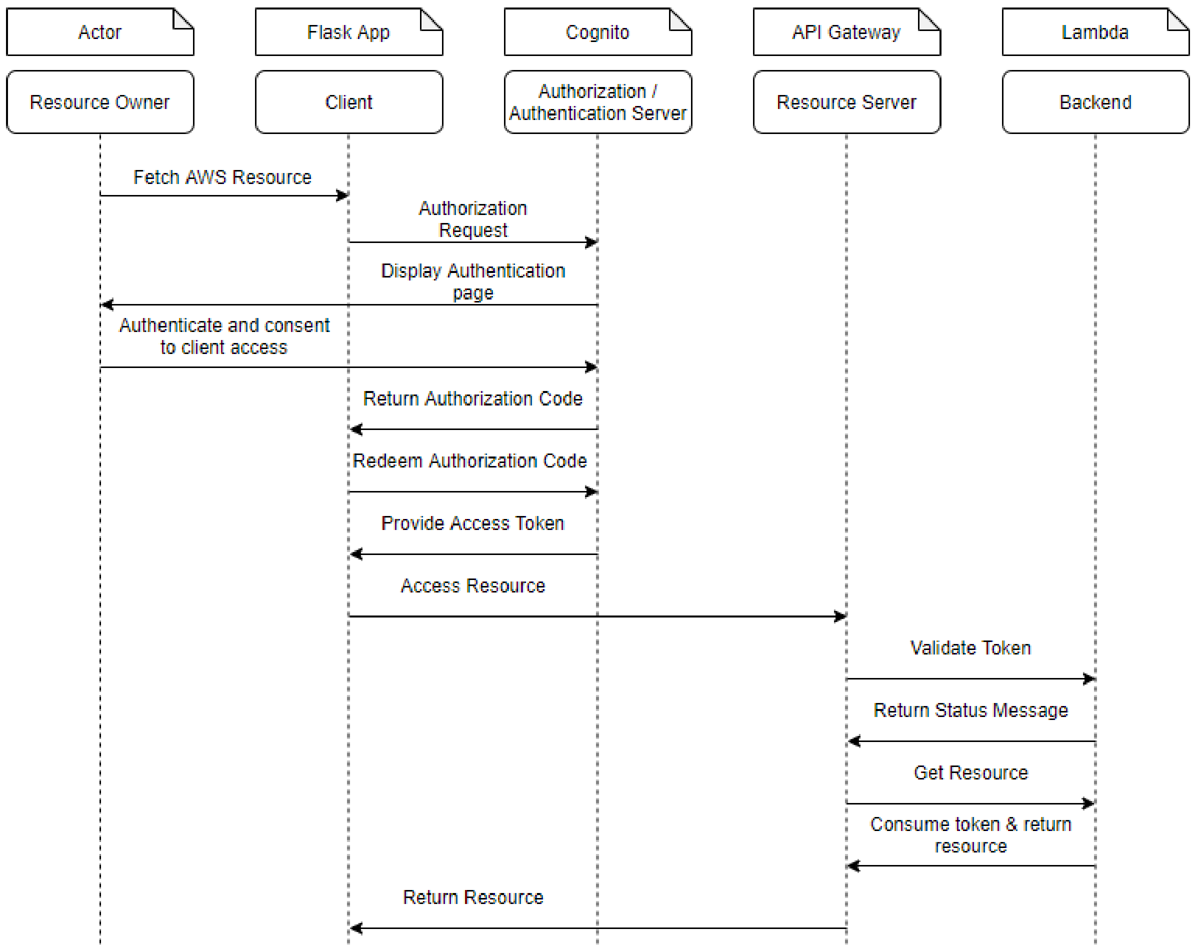

Figure 3.1.2 - Code Grant Flow (Cognito), basic implementation

The system in AWS using Cognito was similarly implemented to Curity. The flask application was edited to instead communicate with the Cognito endpoint, and the resource server was exchanged with an API deployed on the API Gateway fetching resources with lambdas. Token validation was processed in a triggered lambda to compensate for the lack of introspection. For an overview of the system, see figure 3.1.2.

After having gathered enough information to understand the concepts, the examinee and supervisors at Dataductus started discussing how to best compare Curity and Cognito. Due to time and budget constraints, large scale case studies were excluded from consideration. Basing the comparison on existing research and studies was considered, however due to OAuth and Serverless being relatively new concepts, the number of published studies in these fields is small, and therefore this approach was also excluded. Finally, it was decided that common use cases would be studied and the comparison would be based on these, as well as security and business concerns regarding deployment.

After the approach had been decided, the next step was to deploy Curity on AWS. Two alternatives were considered:

1. Deploying Curity as an application on a cluster of EC2 (virtual machines).
2. Deploying Curity as a cluster of ECS (docker) containers.

Due to the high price of virtual machines and the low system requirements of running docker containers, it was decided to deploy Curity on an ECS cluster managed by Fargate. For a simplified overview of the system, see figure 3.1.3.

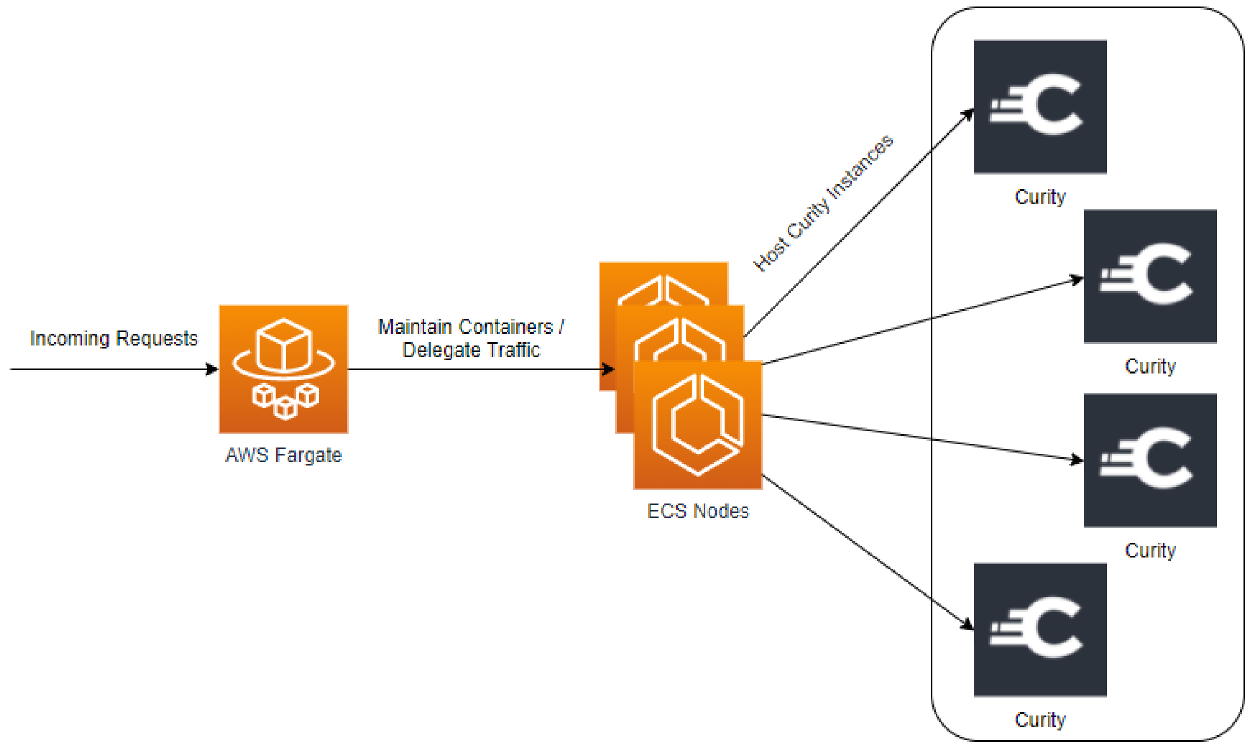

Figure 3.1.3 - ECS Cluster hosting Curity

After having deployed Curity and documented key features and services, it was time to move on to the next phase, and start analyzing Curity and Cognito.

The direct result of this phase in the thesis documentation is primarily the technical background.

3.2 Phase Two - Analysis

During the analysis stage the information gathered and learned during the information gathering phase was analyzed in order to map out differences between Curity and Cognito. The use cases are derived from the results of the analysis, and will, in union with the information analyzed, be the basis for the conclusion of the thesis.

Initially, the OAuth and OIDC capabilities Curity and Cognito support respectively were identified in order to compare how much of the OAuth protocol they've implemented each. Table 3.2.1 shows what flows are supported:

Supported Flows	**Curity**	**Cognito**
Code Grant Flow	Yes	Yes
Implicit Flow	Yes	Yes
Introspection Flow	Yes	No
Hybrid Flow	Yes	No
Client Credentials Flow	Yes	Yes
Refresh Token Flow	Yes	Yes
Token Revocation Flow	Yes	*
Resource Owner Password Flow	Yes	No
On-Behalf-Of Flow	Yes	No

Table 3.2.1 - OAuth & OIDC capabilities

* Token revocation flow for Cognito can not be summarized as a yes or no, and the presence of the feature will therefore have to be explained. The token revocation flow is primarily used as a means to revoke access and refresh tokens, and Cognito does not have a flow which does that. However, the AWS SDK has a method which logs out the user from all devices and invalidates all of the user's tokens. This feature is limited to User Pool users, and is therefore not available for federated identity solutions. In short, the use of token revocation is dependent on the scenario.

Next, security concerns regarding OAuth and Serverless were considered in order to list relevant security features. The listed features are primarily derived from expansions to the OAuth protocol and protection against common serverless and OAuth attack patterns, and are present in **at least** either Curity or Cognito. For a brief overview of some common security risks in serverless and OAuth applications, refer to '*OWASP top ten serverless*' and '*Attacking and Defending OAuth 2.0*' in references. [4] [5] Presence of features is listed in table 3.2.2.

Another important reason why some of these features were considered is human error being a large factor in the occurrence of security breaches. This is stated in a large number of articles on the topic of security, such as "*Three common mistakes that lead to a security breach*". [7].

Security Features	**Curity**	**Cognito**
CSRF Protection	Yes	Yes
PKCE	Yes	Yes
Federated Identities	Yes	Yes
Geolocation Tables	Yes	*
Redirect URI Validation	Yes	Yes
Origin URI Validation	Yes	No
Token Validation	Yes**	No**

Table 3.2.2 - Security features

* Geolocation tables are used to implement specific behaviour depending on the location of the user, such as deny requests from a specific country or similar actions. Cognito does not have this feature, however it is possible to implement a similar solution in union with AWS route53 and AWS WAF.

** Token validation is the act of validating scopes, expiration etc of tokens procured from client applications. Curity simplifies this via token introspection, which is a flow that returns a phantom token if the token is active and status if it's invalid. Even if it's simplified though, developers are still required to verify scope against resources.
Cognito on the other hand relies entirely on self-validation, which means that developers have to manually implement the entirety of the token-validation process.
The reason for each of the features being selected for comparison is listed below:

- **CSRF-Protection:** CSRF attacks are a major threat if not protected against in OAuth clients, and therefore are considered a **MUST** in the RFC documentation of OAuth. This is especially true in serverless applications where scalability is a major feature, and with open APIs catering to large audiences. In order to ensure this feature exists, it had to be confirmed that both Cognito and Curity have implemented it. For information on how CSRF attacks work and how to protect against them, read 10.12 in the RFC document.
- **PKCE:** Proof Key for Code Exchange is an extension of the OAuth protocol. It is an important feature used to prevent injection attacks in the code grant flow, and the presence of this feature is therefore being compared partly to ensure it is implemented, and partly to discover potential discrepancies in following updates in the OAuth protocol.
- **Federated Identities:** Federated identities are a direct result of the OIDC protocol, and one of the primary ways to reduce price in applications handled by Cognito. The prevalence of this feature should therefore be considered in Curity.
- **Geolocation Tables:** Geolocation tables are not only used to tailor UX to regions, but can also be used as an additional security layer, especially in device flows. An example would be having devices located in Malmö, using geolocation tables it can be ensured that only IP-addresses based in Malmö are allowed to call a specific endpoint. This is especially true in the cloud, where applications can easily be migrated to servers in different regions.
- **Redirect URI validation:** Validating the redirect URI is another **MUST** in the RFC documentation. What's being considered here however is whether Curity and Cognito handles that automatically or whether it has to be implemented.
- **Origin URI validation:** Origin URI validation is not listed in the OAuth specification but it is a feature that can add additional security layers to OAuth flows. It is basically a whitelist of origins that can initiate flows to an endpoint. It's prevalence was known in Curity, and it was therefore investigated whether Cognito had this feature as well.
- **Token validation:** Validating tokens is one of the most important parts of the OAuth flow, and the lack of assistance in this endeavour could increase the risk of human error.

Next up, some customization options were considered. While developing the test applications in the information gathering phase, features that made Curity and Cognito easier to manage were identified. It was then investigated whether both services had these features implemented. See table 3.2.3.

Customization	**Curity**	**Cognito**
Endpoint Customization	Yes	No
Custom Identity Providers	Yes	Yes*
Flow Customization	Yes	Yes
UI customization	Yes	Yes
Template Client Registration	Yes	No

Table 3.2.3 - Customization & Management features in Cognito and Curity.

* Custom identity providers are featured in Curity as a redirection to an implemented OIDC authentication endpoint, while in Cognito it is featured by fetching identity from an existing User Pool.

What beneficial effect these features had is listed below:

- **Endpoint Customization:** This feature makes it easier to test out endpoints, micromanage clients and separate client access.
- **Custom Identity Providers:** Should be implemented in OIDC solutions as it could potentially negate the need to migrate or create new users when registering a new client with a large existing user base.
- **Flow Customization:** Useful feature in micromanaging steps during the OAuth flow, such as storing tokens and validating scopes.
- **UI Customization:** Not necessarily related to the security of the system, however customizing the UI does separate the app from other apps.
- **Template Client Registration:** A very useful feature in handling registration of new clients, especially when attempting to automate it.

Another important factor when deciding what OAuth service should be used is the price, however estimating price is quite difficult and has to weigh in a lot of factors. To simplify, the act of performing the ‘Code Grant Flow’ was isolated, however it will be noted where cost can vary significantly.

Since Curity is being deployed on an ECS cluster, the price paid is dependent on the amount of computing power reserved for it. Table 3.2.4 shows data gathered from Data Ductus about the computing power necessary for Curity to support the specified amount of grants per day.

Token Grants / Day	Computing Power Reserved
Less than 100 000	Three nodes with 4 GB ram & 1 vcore reserved each.
Less than 1 000 000	Three nodes with 8 GB ram & 2 vcores reserved each.
More than 1 000 000	Four nodes with 16 GB ram & 4 vcores reserved each.

Table 3.2.4 - Computing Power required for Curity

It should be noted that these numbers were gathered when Data Ductus hosted Curity on virtual machines, and these were directly translated into nodes on the ECS cluster. It is highly likely that the price can be optimized by temporarily launching new nodes when traffic is high instead of increasing reserved computing power. However, in this study, the supplied data will be used and it is assumed that these nodes are kept running 24 hours a day.

Following this format, and using the price documented on AWS ECS listed pricings, Curity’s price was calculated based on the following equation:

$$(VCpuPrice\ +\ GBPrice)\ *\ nbrOfInstances\ *\ 730hrs$$

Meanwhile, Cognito’s price is difficult to estimate because it is based on several assumptions. First, it is assumed in the price that no MAU’s (monthly active users) are paid for, which is calculated on the number of users registered in Cognito User Pools. Note, the user pools in Cognito is one of its primary features, and a mere 50 000 active users registered would increase the monthly price by around 270 dollars. Secondly, actions customizing the OAuth flow such as pre-authentication, are performed using lambda triggers, and the price per token grant is entirely based on the number of lambdas triggered. The number of triggers performed can vary a lot depending on factors such as the level of security necessary for the client initiating the flow and the amount of UX customization done in the flow. It should be noted that there are up to twelve lambda triggers in common use.

It should also be noted that the price for other services in AWS to which access is protected via Curity or Cognito can not feasibly be calculated into the price, and therefore is excluded.

As such, five lambda triggers are used in order to estimate the average price for Cognito. At least the following triggers are used:

- A trigger to initiate auth challenge.
- A trigger to validate the user.
- A trigger to validate the scope.
- A trigger to grant the access token.
- A trigger to validate the access token.

With these assumptions made, Cognito’s price is calculated from the following equation:

$$LambdaPrice * 5 * nbrOfTokenGrants$$

While Cuity has a fixed price for uptime, and Cognito has a scaling price by number of grants, a middleground based on the number of daily grants Curity can serve is assumed. This results in table 3.2.5:

Token Grants / day	**Curity**	**Cognito**
50 000	127.5 $	10.8 $
500 000	255.18 $	108.9 $
5 000 000	680.476 $	1087.8 $

Table 3.2.5 - Estimated Price for Curity and Cognito

With the analysis of factors completed, the final comparison between Curity and Cognito will be discussed under results, but first the development of use cases is discussed.

3.3 Post-Analysis - Development of Use Cases

With the analysis of features completed, the final step before comparison was the development of use cases. Two use cases were identified early on, one being the code grant flow providing an access token to a client, the second being the completion of the authorization flow to consume the access token and retrieve a protected resource via an API. The use cases are of obvious importance due to it being the basis upon which OAuth was implemented. In order to find more use cases, advantages of deploying serverless applications were considered so that common scenarios could be identified. Serverless has the potential to scale “indefinitely” if price is not an issue, keeps the price down for small scale applications and function-as-a-service systems, has a global availability zone and handles traffic load balancing excellently. To narrow down the scenarios the use cases were additionally limited to OAuth applications. From this process, three additional use cases were identified:

- **Authorizing a public client** - When authorizing an application, the client commonly provides credentials to authenticate towards the authorization server, however in the case of public clients (clients who are incapable of maintaining confidentiality of their credentials) the authorization must proceed differently. An example of a public client would be a SPA (single page application). SPAs are quite popular to host on AWS at the moment due to the ease of updating content on the page using lambdas instead of relying entirely on Javascript. This makes the authorization of a public client important when comparing Cognito and Curity.
- **Authorizing an IoT device** - The OAuth authorization code is commonly granted via the URL from the response of the authorization server, however for devices that don’t have a browser or where input from the user is difficult, the flow has to be modified. IoT devices, such as smart TVs or smart locks could have this issue, but would otherwise be easy to manage in a cloud environment. Therefore, this use case should be investigated.
- **Dynamic Client Registration** - DCR, or dynamic client registration, is an extension on the OAuth protocol that provides a way to programmatically register clients. This could be useful in two common use cases for serverless - when scaling an application that needs to support managing an increasing number of clients, and providing installationspecific credentials to instances of mobile applications.

The first use case will grant an access token, and when applicable will be used as a basis to build the remaining use cases. Additionally, the first use case will be heavily detailed with example links in order to make it easier to understand, while later use cases will refer to the names of the endpoints.

3.3.1 Use Case 1: Grant Access Token

First, let's start by explaining why this was separated from the second use case. From the experience gathered while implementing the test applications it was assumed that the code grant flow up to the point of providing an access token would be very similar, so in order to simplify later comparison the overall scenario was split into two use cases.

Before mapping out how Cognito and Curity implement this, it is stated what attributes they both share and require in table 3.3.1. Note, the values used in table 3.3.1 are examples and therefore do not represent a real implementation.

Key	Value	Exchanged via
Client ID	App	URI
Redirect URI	https://app.com/callback	URI
Response Type	code	URI
Scope	db_read	URI
State	hash(redirect uri)	URI
Client Secret	test1234!?#=	HTML Body (Post)

Table 3.3.1 - Shared attributes required for Code Grant Flow.

The first step that differs between Cognito and Curity is the endpoint that is called to initialize the flow. The authorization endpoints have these URLs:

Curity - https://(Fargate_IP):8443/oauth2/oauth_authorize
Cognito - https://mydomain.auth.region.amazoncognito.com/oauth2/authorize

The full URL to start the Code Grant Flow will look like this:

endpoint?response_type=code&redirect_uri=https://app.com/callback&scope=db_read&state=state&client=app

Note, “endpoint” in the link above is exchanged for Curity’s & Cognito’s endpoints. Next, the use cases for Cognito and Curity will be described.

UC 1 Grant access token - Client initiates code grant flow.

For Curity:

1. Client sends a GET request to https://(Fargate_IP):8443/oauth2/oauth_authorize?response_type=code&redirect_uri=https://app.com/callback&scope=db_read&state=state&client=app
2. Curity verifies that the client is registered.
3. Curity verifies that origin is whitelisted (optional).
4. Curity verifies that the redirect URI is whitelisted.
5. Curity verifies that the scope is whitelisted for the client.
6. Curity redirects back to https://app.com/callback?code=authorizationcode&state=state, authorization code being passed in the URI.
7. Client sends a POST request to https://(Fargate_IP):8443/oauth2/oauth_token, with body containing client_id, client_secret, grant_type, state, scope, redirect_uri and authorization code.
8. Curity verifies client secret with client id.
9. Curity verifies that the authorization code is valid and not expired.
10. Curity responds with POST request to https://app.com/callback, body containing access token and refresh token.

For Cognito:

1. Client sends a GET request to https://mydomain.auth.region.amazoncognito.com/oauth2/authorize?response_type=code&redirect_uri=https://app.com/callback&scope=db_read&state=state&client_id=app
2. Cognito verifies the client is registered.
3. Cognito verifies redirect URI is whitelisted.
4. Cognito redirects back to https://app.com/callback?code=authorizationcode&state=state, authorization code being passed in the URI.
5. Client sends a POST request to https://mydomain.auth.region.amazoncognito.com/oauth2/token, with body containing grant_type, state, scope, redirect_uri and authorization code. Client id and secret are passed via headers.
6. Cognito verifies client secret with client id.
7. Cognito verifies authorization code is valid and not expired.
8. Cognito responds with POST request to https://app.com/callback, body containing access token and refresh token.

3.3.2 Use Case 2: Authorizing a standard client

For this use case, the access token procured in the previous use case will be used to make an authorized request to access a protected resource. Both Cognito and Curity will keep the values from the previous configuration. For the overall system, see figure 3.3.1.

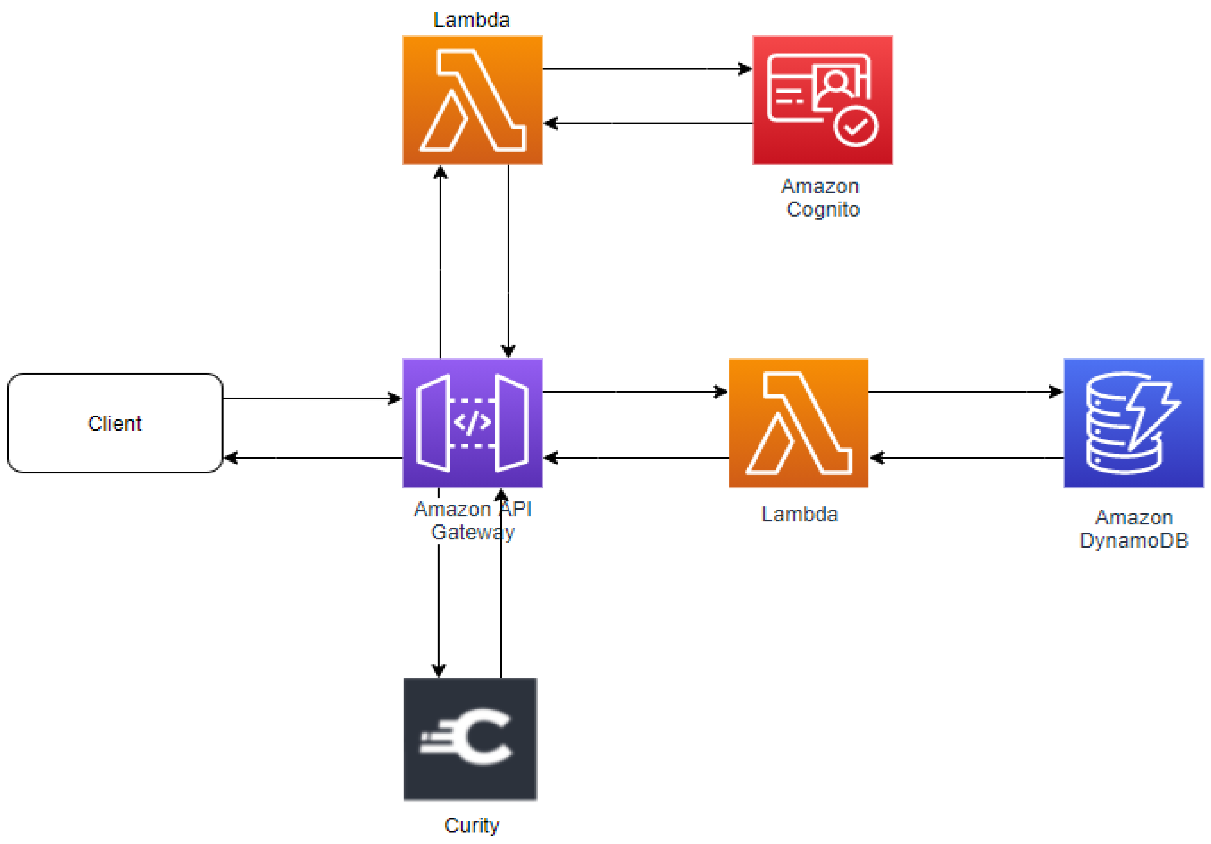

Figure 3.3.1 - Overview of Code Grant Flow for a standard client.

Let's first examine the scenario. A standard client means that the client is registered, performs the code grant flow via the browser and it has the capability to hold a secret. Because of this, the authorization flow from use case 1 does not have to be modified. After having received the access token, the client will attempt to call an API method that in this case returns a value from a database. The client will pass the access token along with the request, and the API will have to validate the token. After the token has been validated, the API will execute a lambda function that retrieves the requested entry in the database, and will finally return it to the client.

With the scenario examined, it can be concluded that differences may apply in how the token is validated. As such, the second use case will be isolated to validating the access token.

UC 2 Authorizing a standard client - Validating an access token received from a client.

For Curity:

1. Client makes a POST request to the API, passing access token, secret and client id in body.
2. API Gateway passes the body to https://(fargate_ip):8443/oauth2/introspection.
3. Curity verifies the access token is active.
4. Curity returns a JWT copy of the token to API Gateway.
5. API Gateway verifies scope from the JWT.
6. API Gateway proceeds to execute lambda that fetches database entry.
7. API Gateway passes the resource to the client.

For Cognito:

1. Client makes a POST request to the API, passing access token, secret and client id in body.
2. API Gateway executes lambda authorizer to validate token.
3. Lambda authorizer verifies JWT structure.
    a. Verifying that there are three segments.
    b. Parse JWT to extract components.
    c. Decode body and header.
4. Lambda authorizer verifies signature on access token.
    a. Retrieve and verify the algorithm.
    b. Verify signature on token with public key from Cognito.
5. Lambda authorizer verifies standard claims.
    a. Verify token is not expired.
    b. Verify token issuer.
6. Lambda authorizer verifies scope of the token.
7. Lambda authorizer returns success code.
8. API Gateway proceeds to execute lambda that fetches database entry.
9. Api Gateway passes the resource to the client.

3.3.3 Use Case 3: Authorizing a public client

As mentioned before, public clients are incapable of maintaining confidentiality of credentials, however that is not the only issue that can arise from this scenario. In addition to not being able to keep secrets, the following practices should be followed as well:

1. **Tokens available in the browser** - Due to a public client not using a backend, it does not have a safe place to store tokens, and thus stores them in the browser. This leaves the client vulnerable to OWASP-defined attacks such as XSS (cross-site scripting attacks) and CSRF (cross site request forgery attacks). It has already been ensured that both Cognito and Curity have implemented protection against CSRF attacks, and XSS attacks can be protected against using AWS WAF (web application firewall). Additionally, both Cognito and Curity can configure the behaviour of the authorization flow to scan input from the client.
2. **Token lifetime needs to be short** - Since public clients cannot safely store tokens, the lifetime of tokens should be as short as possible, to reduce the risk of a token being stolen by a malicious party.
3. **CORS and PKCE must be enabled** - CORS, or cross-origin resource sharing, is a security measure that acts as a whitelist for origins that are allowed to call upon a resource. This, and PKCE, must be enabled while authorizing a public client, in order to reduce the risk of impersonation attacks.
4. **Client Authentication must be toggled off** - Since clients will not be able to hold confidential information, they will not be able to authenticate themselves, and therefore must not be required to authenticate.

It should be noted that Cognito has not implemented CORS, however API Gateway has. Due to limited time and resources, the effects this may have on security, price and efficiency has not been tested in an implementation, but possible implications will be discussed in results.

For this use case, a SPA is used as an example of a public client. Implicit flow was originally intended as a solution to public clients, however current best practices according to the IETF suggest Code Grant Flow with PKCE as the best solution, and as such that pattern will be followed. For more information on the topic, read *OAuth for Browser-Based Apps* in references. [8].

UC 3 Authorizing a public client - Authorizing a Single Page Application.

For Curity:

1. Client generates PKCE challenge and hashes it.
2. Client makes a GET request to API Gateway with the PKCE challenge code as value to the parameter code_challenge along with the hashing method used. Additional parameters are state, client id, redirect uri, scope and response type.
3. API Gateway redirects the request to Curity.
4. Curity validates client id.
5. Curity validates origin.
    a. CORS is enabled here for Curity.
6. Curity validates redirect uri.
7. Curity returns authorization code in header.
8. Client makes a POST request to Curity with the body containing plaintext of secret, authorization code, state, client id and redirect uri.
9. Curity verifies plaintext with the hash.
10. Curity returns an access token.

For Cognito:

1. Client generates PKCE challenge and hashes it.
2. Client makes a GET request to API Gateway with the PKCE challenge code as value to the parameter code_challenge along with the hashing method used. Additional parameters are state, client id, redirect uri, scope and response type.
3. API Gateway validates origin, then redirects to Cognito.
    a. CORS is enabled here for Cognito.
4. Cognito validates client id.
5. Cognito validates redirect uri.
6. Cognito returns authorization code in header.
7. Client makes a POST request to Cognito with the body containing plaintext of secret, authorization code, state, client id and redirect uri.
8. Cognito verifies plaintext with the hash.
9. Cognito returns an access token.

3.3.4 Use Case 4: Authorizing an IoT device

The popularity of IoT devices has been rising vastly over the past few years, and in an infrastructure like AWS they can easily be managed. The issue with IoT devices when it comes to OAuth is that IoT devices usually do not have access to a browser, either because input from users would be too clunky or because the specifications don't leave room for it. Examples of IoT devices would be printers, smartTVs and smart appliances. To allow for secure communication with IoT devices, OAuth suggested the device flow, which is an extension to the standard protocol. The device flow is not implemented by Cognito as seen in table 3.2.1, and as such Cognito cannot participate in this use case. See figure 3.3.2 for an overview of device flow in Curity.

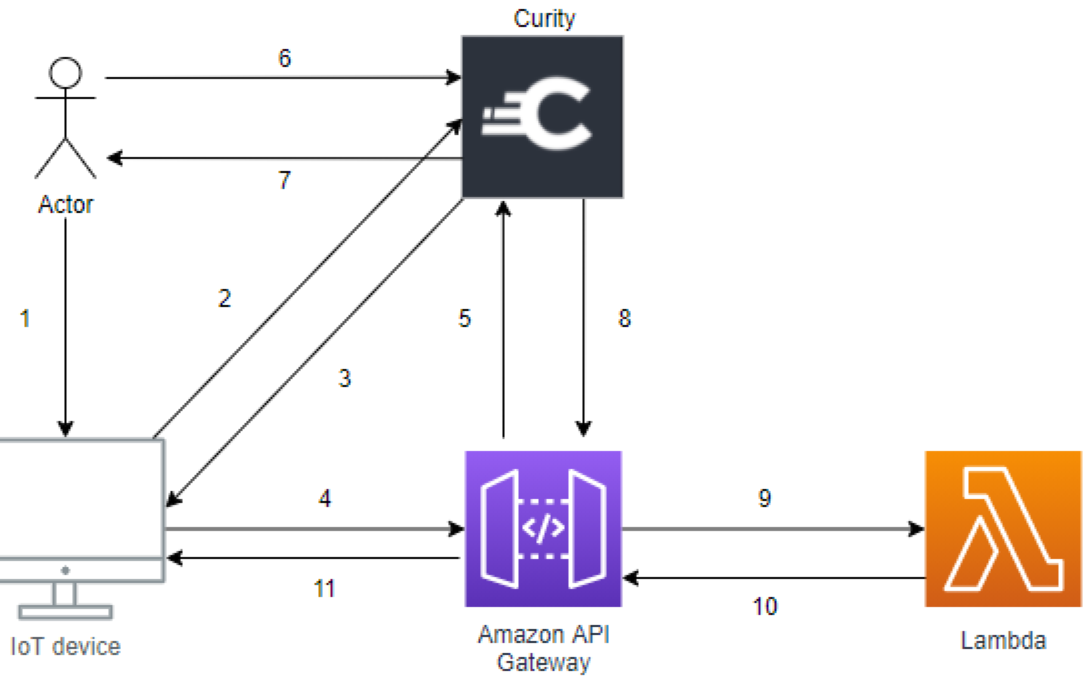

Figure 3.3.2 - Device Flow in Curity.

First, let's explain how the device flow differs from the Code Grant Flow. Instead of supplying the client with an authorization code, the authorization server (in this case Curity) supplies the client with a device code and a user code. The user is asked to supply the user code when consenting to the client accessing resources on the resource server, and the device code is supplied when fetching the access token. Next up, defining the use case.

UC 4 Authorizing an IoT device - Client initiates device flow.

For Curity:

1. Client sends an authorization request to the device code endpoint on Curity.
2. Curity validates redirect URI and client ID.
3. Curity responds with device code and user code.
4. Client attempts to poll Curity for an access token with a regular interval.
5. User redirects to the link supplied with the response and enters the user code.
6. User consents to the client accessing scoped resources.
7. Curity returns an access token after being polled by the client.
8. Curity marks the device as authorized.
9. Client sends access token along with request for resource to API Gateway.
10. API Gateway calls on Curity's introspection endpoint to validate the token.
11. Curity validates the token.
12. API Gateway proceeds to return the requested resource to the client.

For Cognito:

A solution could not be found for Cognito since it has not implemented device flow.

3.3.5 Use Case 5: Dynamic Client Registration

As mentioned in the motivation, DCR is an extension to OAuth which allows developers to register clients via code. The suggested protocol was developed in order to provide a standardized and secure method for the client to obtain necessary metadata from the authorization server. While gathering information on the topic, it became apparent that Cognito has not implemented the protocol, therefore the focus will be on Curity's implementation.

There are four ways to configure DCR in Curity:

1. **Open Registration -** DCR can be enabled and any client can register without authenticating.
2. **Client Authenticated Registration -** Only a client with an initial access token obtained from client credential flow is allowed to register a client. In this case, only the identity of the client is verified.
3. **User Authenticated Registration -** Client registration is limited to clients with an access token obtained from an authenticated user. In this case, the client is identified but only the identity of the user is verified.
4. **No Registration -** DCR can be completely denied. This is the default configuration.

For this use case, the focus will be on the scaling potential of serverless applications and therefore use DCR for automating client registration. This use case can easily be implemented in Curity using the templates feature, and a workaround solution is possible for Cognito.

UC 5 Dynamic Client Registration - Automated client registration.

For Curity:

1. Client fills in a form with specifications about the type of client it needs to register.
2. Resource server returns a template id.
3. Client initiates client credential flow.
    a. Client makes a POST request to Curity, passing client ID, client secret, grant type and scope. Grant type has to be client_credentials and scope has to be DCR.
4. Curity verifies client id, client secret, redirect uri and origin.
5. Curity returns a single-use access token.
6. Client makes POST request to Curity's DCR endpoint, header containing access token and client id, body containing template id.
7. Curity verifies token, client id and template id.
8. Curity registers the new client.
9. Curity returns a JSON object containing metadata for the client, such as client id and client secret.
10. A server side script adds allowed redirect uri to the client.

For Cognito:

Cognito has not implemented DCR, however it would be possible to solve this particular use case anyways using the AWS SDK. The metadata from the form would instead have to be passed through a serverside shellscript that would execute the necessary commands to register a new client in Cognito. This would be quite complex, and would most likely be approached differently depending on the developer, and as such will not be further investigated. The possible implications this would have on security will be discussed in results.

4 Results

The results from gathering information, analyzing the features and developing the use cases will be discussed in this chapter, resulting in a comparison between Cognito and Curity and concluding phase three. This chapter will start with comparing the features presented in analysis, and then move on to a more detailed analysis of the use cases. In the final part, advantages and disadvantages of using Cognito and Curity are presented and in what scenarios one would be advantageous to the other.

4.1 Implemented OAuth and OIDC Flows

As seen in table 3.2.1, it was investigated which flows presented in the OAuth and OIDC specifications that had been implemented. This is useful, partly because it tells us how diligent Cognito and Curity are in implementing and updating software according to the protocol, but also because it tells us some of the weak points they might have. Let us first list off the most important flows, which would be Code Grant Flow, Client Credentials Flow and Refresh Token Flow. Both Cognito and Curity have implemented these flows, and they account for the majority of OAuth scenarios. While these flows can be used to implement most solutions, there are other flows that can provide better security, easier implementation and additional possible solutions. Curity has implemented these flows while Cognito hasn't:

- Introspection Flow
- Hybrid Flow
- Token Revocation Flow
- Resource Owner Password Flow
- On-Behalf-Of Flow

Let us start with the Hybrid, Resource Owner Password and On-Behalf-Of flows. These flows are either rarely used or were implemented for highly specific use cases, or both, and therefore will only be briefly mentioned.

The Hybrid Flow is a mix between Code Grant and Implicit Flow, which allows us to get information about a user from the authorization endpoint while the access token is procured via a back channel. This could be used to prevent IdP mix-up attacks, which is a risk in OIDC solutions when a client has registered multiple identity providers. It should be noted that this attack is very unlikely to occur due to a large number of assumptions that has to be made, of which several are the misconfiguration of OAuth and OIDC due to human error. For further understanding of the attack, read *IdP mix-up attack on OAuth* in references. [16].

The Resource Owner Password flow is a grant that allows the client to impersonate the resource owner, and could be used to integrate legacy applications with OAuth solutions. This flow can be misused and should if possible be avoided, as it leaves the user with no control over the authorization process and it leaves much room for human error, and because of this it will not be further discussed.

The On-Behalf-Of flow is a grant that allows a user to impersonate another. This can be used in, for example, IT solutions where an admin is required to help an end-user with solving a problem. An admin being forced to receive authorization to impersonate an end-users account would provide an additional layer of security to ensure admins do not misuse administrative privileges, however with proper policy use and access tracking similar solutions can be implemented in AWS.

While it is good that Curity has implemented these flows, they do not provide a notable advantage to Cognito in most applications. However, the introspection flow and token revocation flow, which will be discussed, do.

The introspection flow is a grant that takes an access token as input, verifies its existence and lifetime, then returns a phantom token. Both the input and output can be either opaque tokens or JWTs, the difference being that opaque tokens do not contain information about the claims and the user, while the JWT contains information about user identity and can be customized to return additional information. First, this leaves much room for customization, such as returning specific user information depending on the client and the type of access token. Second, this flow can easily be used to validate access tokens, which is of primary concern in OAuth applications. The issue with Cognito not implementing introspection is that, as mentioned in analysis, the developer has to perform the entire validation process manually. Because of this, third party libraries are commonly used to perform the validation process, and the use of external sources increase the attack surface of an application. If developers instead stick to implementing the validation process manually, it leaves a large margin for human error when attempting to manage multiple user pools with different signatures and verifying scopes from multiple clients. As such, this could be of major concern when using Cognito in large scale applications.

The lack of the token revocation flow is another grant that could potentially harm use of Cognito. The token revocation flow is not only the revocation of an access token, it can be used to revoke the entirety of the token hierarchy. Use cases for this would be when a user has logged out or disconnected from an application, or uninstalled it entirely, and therefore there shouldn't be any authorization left active that the user is unaware of. As mentioned in the notes to table 3.2.1, depending on the scenario, it is possible to invalidate the tokens of User Pool users via the AWS SDK. However, there are use cases where this is not enough. One possibility would be a client retrieving sensitive userinformation hosted on AWS over a federated identity. When that client has perfomed the necessary operations, in Cognito, the token will stay alive until expired. The token revocation flow allows for proper consumption in Curity, invalidating the token after use. Additionally it could be used to ensure a single use of access tokens in public applications, and while it may primarily be used in edge cases, it is still an important feature in systems with GDPR information and other sensitive data.

While Cognito may have implemented the most commonly used flows, Curity has the advantage of a more complete implementation in this area, and the lack of these OAuth flows cannot be complemented with other AWS security services. From the implementation of flows alone, Cognito is good enough for a majority of OAuth applications, however Curity could be a more suitable replacement for higher security and more flexibility in authorization and authentication. Next the security features from which can be seen in table 3.2.2 will be discussed.

4.2 Implemented Security Features

The motivation for the features in table 3.2.2 was discussed under 3.2 Analysis, and will not be covered in this section. For this part, the most important features are CSRF protection, PKCE, redirect URI validation and token validation. Both Cognito and Curity have implemented CSRF, PKCE and redirect URI validation. The features that differed were the following:

- Geolocation Tables
- Origin URI Validation
- Token Validation
- *Dynamic Client Registration
- *CORS

While DCR and CORS were not brought up in the initial part of the analysis, they were identified as important security features later on during the development of the use cases, and as such they will be discussed briefly. Token validation has already been covered in the previous discussion about introspection and will therefore not be mentioned here.

Let us start with the geolocation tables. While they are most commonly used for customization of UX and splitting up the application into country specific modules, they also provide a layer of security. A quick example would be forcing attackers from another country, region or even city to use proxies or vpn services to attack an application, thereby reducing attack surface. Curity has implemented this feature, and the feature can be implemented in AWS WAF combined with route53 for Cognito. While the desired effect can be reached in both services, it is easier to isolate geolocation tables in Curity to specific clients.

Next up is origin URI validation. While it is not a necessary feature in most use cases, it can be very useful to provide additional security where it is otherwise lacking, and Curity relies on it to implement CORS. Due to Cognito relying on API Gateway for CORS, the lack of this feature is not of a major concern, however it might make it more difficult to separate client access to APIs as CORS is validated in the API call instead of the OAuth flow.

Last is DCR, which is an important feature to securely handle programmed registration of clients. While it is possible to register clients via the AWS SDK, it does not implement the transmission of metadata between client and authorization server, which would have to be done in another way. As such, and for security reasons that will be discussed in the results of use case five, the lack of DCR in Cognito should be considered a major flaw.

Overall, Cognito does have a few shortcomings, of which most can be made up for with the use of other security services in AWS. Curity does however cover security better, and leaves more room for micromanagement and customization.

4.3 Implemented Customization Features

Next some of the customization features listed in table 3.2.3 will be discussed. The differences in implementation gathered from the table are the following:

- Endpoint Customization
- Custom Identity Providers
- Template Client Registration

While these features don't act as major keystones in security, they still contribute to ease of use and flexibility in the application.

First, endpoint customization. In Cognito, you are limited to a single domain for an entire User Pool, and all clients registered in that pool call upon the same endpoint. This makes it more difficult to track traffic, test implementations of individual clients and so on. In Curity, you can create custom endpoints and limit clients to access specific ones, thus simplifying an otherwise complex networkhierarchy. Additionally, this could make it easier to separate client data and access, potentially adding additional security layers.

Second is the Custom Identity Providers. While Cognito technically supports this feature, it is limited to federating custom identities listed in Cognito User Pools. In Curity, the Custom Identity Providers can be integrated to point to existing OIDC authentication server implementations, leaving more room for flexibility in OIDC solutions.

Last is Template Client Registration. Templates are part of the DCR implementation in Curity, and are primarily used to simplify registration of clients. The lack of this feature in Cognito is only natural since DCR is not implemented. As such, Curity gains some extra advantage in flexibility.

Overall, Curity does have many advantages in flexibility, partly from the extensive OAuth implementation as seen in *Implemented OAuth and OIDC flows* and *Implemented security features*, but also because of the additional customization options available.

4.4 Pricing

As mentioned during the analysis, the pricing for Cognito is very hard to predict, however with the assumptions made costs are estimated in table 3.2.5. If the values from the table are plotted in a graph, the average monthly price for Curity and Cognito will resemble figure 4.4.1.

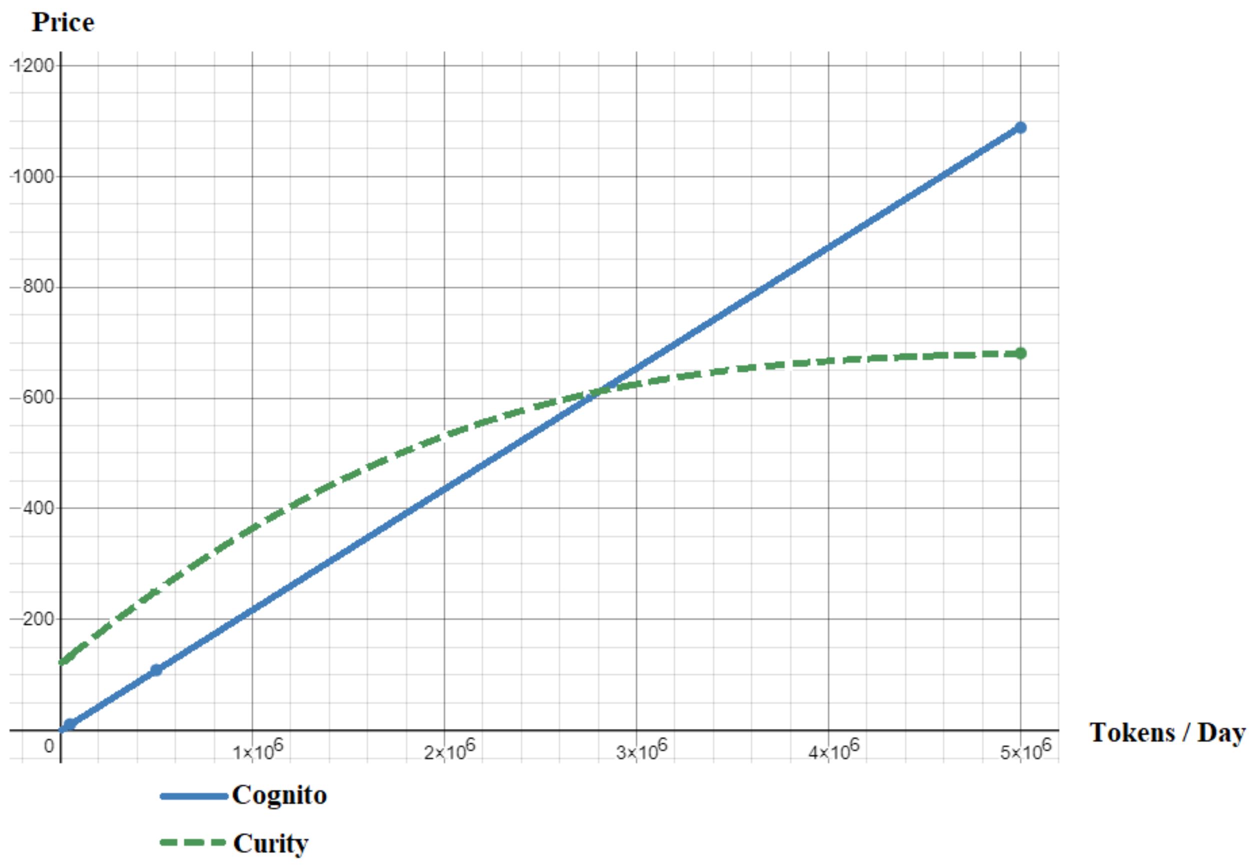

Figure 4.4.1 - Average Monthly Price for Curity (green) and Cognito (blue).

As can be seen from the table and the graph, Curity's price scales better despite several price reductions on Cognito, such as minimal flow customization and not including the cost of monthly active users. Cognito is, however, superior in price by a far margin when it comes to handling small scale applications.

4.5 Results of the Use Cases

This section will compare the results of the use cases for Curity and Cognito one by one. First out is the Code Grant Flow which was split into two use cases.

4.5.1 UC 1 Grant access token

For this use case there will not be much to discuss. As mentioned in the background to the use case, the flow is performed with almost no differences. There is the case of CORS being an alternative in Curity's flow, while Cognito would have to rely on API Gateway, and Curity allowing for endpoint customization, however these have already been discussed. The final difference that was identified is that Cognito accepts the secret via the header of the authorization request, while Curity accepts the secret via the body of the authorization request. This should however not have any implications on security for the system.

4.5.2 UC 2 Authorizing a standard client

For the second use case, the token validation process of the Code Grant Flow was isolated for comparison, however the implications of the lack of introspection in Cognito has already been discussed. In short, self validation leaves room for human error or, if relying on a third party library, increases the attack surface of the system. For more details, refer to the discussion about the introspection flow under *Implementation of OAuth and OIDC flows*.

4.5.3 UC 3 Authorizing a public client

This use case, like the first two, is handled very similarly for Cognito and Curity; however, as mentioned before, they integrate CORS differently. In short, Cognito relies on API Gateway to perform CORS while Curity has it implemented in the OAuth flow. This allows for Curity to more easily separate access and data between clients. For more details, refer to the discussion about CORS under *Implementation of Security Features*.

4.5.4 UC 4 Authorizing an IoT device

For this use case, one can see the benefit Curity has over Cognito in certain scenarios due to Curity's wider implementation of the OAuth protocol. As explained in the introduction to the use case, some devices might not have a browser available or input from the user might be awkward, and as such authorization requests would require the device flow. Curity can securely delegate access with browserless clients via the device flow, while Cognito has not implemented the flow and therefore cannot fulfill the use case. On this point, Curity's wider implementation gives it a clear advantage over Cognito.

4.5.5 UC 5 Dynamic Client Registration

Like the previous use case, DCR is another example where Curity gains an advantage over Cognito due to more implemented features. The lack of DCR in Cognito has already been discussed under implemented security features and will therefore not be covered here. Refer to *results of implemented security features* for more details on the topic.

4.6 Summary of results

Cognito and Curity are both powerful tools, as is the OAuth protocol overall. Comparing the two services is difficult and complex at best, mostly due to the final decision coming down to the fine details of the role the authorization server would play and the level of security it would have to implement, however there are a few points where a comparison for certain can be done.

1. Price - While Cognito is much cheaper to use in small scale applications with either a small userbase or an OIDC federated identities solution, in larger applications Curity is much more price efficient.
2. OAuth implementation - Curity has overall done a better job at adhering to the OAuth protocol that the IETF designed, and as such can cover a wider range of applications. This does however also make it more difficult to learn, with the addition of having to manage deployment on the cloud. Cognito has implemented the base features of the OAuth protocol and can as such be used for many standard implementations.
3. Security - While Curity has implemented many more security features, several of these can be covered with other security services in AWS. There are however a couple of security features, such as DCR and token validation, that can not be covered up by the use of other AWS services, and there are also more levels of customization that add additional layers of security in Curity that gives it a competitive edge over Cognito.
4. Customization - The final point would be customization. In customizing UX Cognito and Curity are quite equal, they have both implemented ways to customize all pages that are displayed to end-users, however Curity has more features that customize flows, flexibility, data storage, user migration, client management and so on. Overall, Curity is more flexible to manage than Cognito.

With this in mind, the final results of when Curity or Cognito would be best to use can be formulated. Cognito is good enough to use when managing applications without a lot of traffic, with small userbases and standard scenarios. Additionally, Cognito does not require alot of knowledge about the OAuth protocol and is suitable as an introductory authorization service. Curity on the other hand is advantageous when used in large scale applications, applications with edge cases or where more flexibility is needed, and when handling sensitive data such as GDPR-protected user information. Curity also gives more freedom to developers but requires some prior knowledge about the OAuth protocol, otherwise the large amount of features can be overwhelming.

5 Conclusion

In this final part of the thesis, ethical aspects and future research on the topic will be discussed. However first a conclusion summarizing the thesis with answers to the defined problems in the introduction will be covered.

5.1 Conclusion

While Cognito and Curity are both powerful OAuth implementations, there are scenarios where one could be advantageous to the other. In order to determine if and when Curity would be worth deploying on AWS as a replacement for Cognito, information was gathered on what differentiated the two services so that they could be compared. One important observation is that Curity has a wider range of capabilities and features than Cognito, of which some, but not all, could be covered by other AWS security services. The key differences between Cognito and Curity primarily comes from implementing the OAuth and OIDC protocol, where Curity has an advantage over Cognito in OAuth capabilities and security features. Curity also allows for more flexibility in OAuth related aspects, such as flow behavior and client management. Security features that aren't covered by the OAuth specification which Curity had implemented and Cognito had not could often be covered by other security services in AWS. This concludes in the following notable differences:

1. What are key differences between Curity and Cognito?

    ○ Dynamic Client Registration is implemented in Curity.
    ○ Curity has a larger variety in OAuth capabilities such as device flow and introspection flow.
    ○ Curity is more flexible in micromanaging behaviour.
    ○ The cost of Curity is lower in larger applications, while the cost of Cognito is lower in smaller applications.
    ○ Cognito relies on self-validation of tokens, while Curity has implemented support for token validation.
    ○ Curity has implemented many security features not related to OAuth that Cognito relies on other security services in AWS to cover.

With the information gathered from analyzing OAuth, AWS, Cognito and Curity it was possible to proceed with comparing Cognito and Curity. The lack of research on serverless applications and comparing OAuth implementations resulted in comparing Curity and Cognito on how they handled scenarios that might appear in securing serverless applications. OAuth was developed to securely delegate access, and serverless applications give the benefits of availability, “limitless” scaling, flexibility, and cost efficiency. Analysis suggested the following areas for comparison:

2. How should Cognito and Curity be evaluated when deciding which is better at securing serverless applications?

    - OAuth capabilities and features.
    - Security.
    - Scalability and price.
    - Flexibility and customization.

From this, five use cases were used along with the differences in OAuth and security features to compare Curity and Cognito in different OAuth scenarios. Of these use cases, Curity allowed for implementing all five while Cognito’s lack of key OAuth features only allowed for the completion of three use cases. This leads to the final conclusion of the thesis, which answers what situations Curity would be advantageous to Cognito. The results of the use cases and the analyzed features leads to Curity being advantageous in the following scenarios:

3. In what situations would Curity be advantageous to Cognito?

    - Applications with large amounts of token grants per day.
    - Applications handling GDPR sensitive data.
    - Applications requiring support for OAuth flows or features not present in Cognito.
    - Applications where high levels of micromanagement is beneficial.
    - Applications requiring external federated identities not present in Cognito.
    - Applications requiring flexibility in modifying user attributes.
    - Applications requiring flexibility in management of OAuth related aspects, such as clients, endpoints and flow behaviour.

Overall, Cognito is a useful security service in AWS however it was primarily implemented to allow for easy user integration in applications, and as such lacks certain OAuth features. Due to this, there are scenarios where Curity should be used as either a complement or a replacement of Cognito.

5.2 Ethical aspects

While working on the thesis, no sensitive data was used except for licenses in AWS and Curity, which were managed properly. For AWS, extra precautions were taken when managing credentials, such as generating a long string of random characters and forcing the use of MFA. In the case of Curity, extra steps were taken to keep the license out of the Docker build process to ensure that the container would not contain the license JSON file if it were to become publicly available.

As for the overall ethical aspects of the thesis, it is obviously important with security, especially in cloud infrastructure where access can spread more easily than on isolated domestic servers. While one should always maintain the highest level of security possible, one must also account for costs and knowledge that these systems require. Curity may be advantageous in many situations, but faulty configuration due to lack of experience could leave more holes open than covered, and the additional cost for small applications may not be worth the extra security.

Overall, security should only come second to maintaining the life of the application, except in the case of handling GDPR-classified data where a breach can lead to long-term complications not only for the company, but their user base as well. In these systems security must be considered of the highest priority at all times.

5.3 Future Work

As there is not much research material available on the topic, there are several points that should be considered for future research.

AWS and Curity, and OAuth overall, are complex systems that more often than not have full time specialists with years of experience managing these, and the lack of material on prior research leaves this thesis with a margin of error. Therefore, it is highly possible that the writer of this thesis has either misunderstood or completely missed documentation of certain features, risks and implementations, and selected sub optimal use cases to compare Curity and Cognito on. As such, one possible and recommended topic of future research would be verifying the contents of this thesis. Additionally, if that is not enough, one could add other OAuth implementations to compare and use the process of this thesis as a guideline.

Second, while working on this thesis the lack of research on serverless applications was a major inhibiting factor. Research that would assist this topic would be common uses, benefits and risks of using serverless applications.

Third, a topic of interest would be comparing integration of Curity and Cognito with IAM to achieve ABAC (Attribute Based Access Control) via resource policy lists. Using OIDC claims to provide additional security layers to access would differ between Curity and Cognito and would also be an interesting topic.

The last suggestion for future research would be to implement and compare the use cases presented in this thesis in deployment. While the overall process has been described in this thesis, due to a lack of time and resources they were implemented with minimal configuration and lacking features, and as such comparison has been done on the use cases in theory rather than in practice. This comparison therefore lacks insight on potential issues such as bottlenecking and IAM role delegation.

6 Information Evaluation

Due to the lack of research on this specific topic, information was primarily gathered from documentation. These sources are AWS developer guide, Curity developer guide, OpenID Connect specification and Docker developer guide. [2, 3, 12, 14] These sources should be considered reliable as it is documentation for their own service. All sources published or approved by IETF should also be considered reliable, as this is a highly trusted and reputable organisation. [1, 8, 11, 13] Finally, OWASP is a non-profit organisation dedicated to security with a high trust factor, and should be considered a reliable source as well. [4]

The source reference to Okta should be considered reliable as it is a company with a focus on security services, however as the article is not documentation of their own service it could be biased information. [7]

The source reference to Colby Morgan's article should be considered reliable, as he is an experienced data security engineer and the article has clear references to the RFC documentation, however it should also be considered that he is an independent entity with potential for bias. [5,6]

The source reference to Nat Sakimura's article should be considered reliable, [9] as he is an experienced engineer and has published multiple articles on the topic of OAuth, some of which have been published by IETF. [11] Howerver, the article is a translation of a japanese blog post on the topic, which leaves room for translation errors and misunderstandings. [15]

Finally, source reference to Alexei Balaganski's article should be considered reliable as he is being published by Kuppingercole, and he has written multiple articles on API management and security. [10]

7 Sources

7.1 Digital Sources

[1] **https://tools.ietf.org/html/rfc6749** (November 2020)
RFC 6749 - The OAuth 2.0 Authorization Framework authored by D. Hardt, approved by IETF, ISSN 2070-1721 October 2012.

[2] **https://docs.aws.amazon.com/** (November 2020)
AWS Developer Guide published by Amazon.

[3] **https://curity.io/docs/idsvr/latest/developer-guide/index.html** (December 2020)
Curity Developer Guide published by Curity.

[4] **https://owasp.org/www-project-serverless-top-10/** (February 2021
OWASP top 10 serverless interpretation published by OWASP.

[5] **https://www.praetorian.com/blog/attacking-and-defending-oauth-2-0-part-1/** (February 2021)
Attacking and defending OAuth 2.0 part 1 published by Colby Morgan, August 2020.

[6] **https://www.praetorian.com/blog/attacking-and-defending-oauth-2/** (February 2021)
Attacking and defending OAuth 2.0 part 2 published by Colby Morgan, August 2020.

[7] **https://www.okta.com/identity-101/mistakes-that-lead-to-security-breach/** (March 2021)
Three common mistakes that lead to a security breach published by Okta, (date missing).

[8] **https://tools.ietf.org/html/draft-ietf-oauth-browser-based-apps-06** (March 2021)
OAuth for Browser-Based Apps published by IETF, April 2020.

[9] **https://nat.sakimura.org/2016/01/15/idp-mix-up-attack-on-oauth-rfc6749/** (March 2021)
IdP Mix-Up Attack on OAuth published by Nat Sakimura, January 2016.

[10] **https://plus.kuppingercole.com/reprints/40682b01cadfb22c6303a6c517d104eb** (November 2020)
API Management and Security by Alexei Balaganski, KuppingerCole, December 2019.

[11] **https://tools.ietf.org/html/rfc7519** (December 2020)
RFC 7519 - JSON Web Token (JWT) authored by M. Jones, J. Bradley, N. Sakimura, approved by IETF, ISSN 2070-1721 May 2015.

[12] **https://openid.net/connect/** (November 2020)
OpenID Connect specification published by OpenID Foundation.

[13] **https://tools.ietf.org/html/rfc7636** (April 2021)
Proof Key for Code Exchange by OAuth Public Clients published by IETF, september 2015.

[14] **https://docs.docker.com/** (January 2020)
Docker Developer Guide published by Docker.

[15] **https://oauth.jp/blog/2016/01/12/oauth-idp-mix-up-attack/** (April 2021)
OAuth IDP Mix-Up Attack published by Nov Matake, January 2016.

8 Terminology

Authentication - The act of verifying an identity.
Authorization - The act of verifying permissions of an identity.
Resource Owner - An entity capable of granting access to a protected resource.
Resource Server - The server hosting the protected resource, capable of accepting and responding to resource requests using access tokens.
Client - An application making requests to access the protected resource on behalf of the resource owner and with its authorization.
Authorization Server - The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.
Access Token - Token containing key information to gain access to a resource.
Refresh Token - Token which can be used to update the lifetime for an accesstoken.
Multi-Tenancy - Support for several tenants using the same service from different clients.
Virtual Machine - OS run on virtualized hardware.
JSON Web Encryption (JWE) - JSON data structures representing encrypted content.
JSON Web Signature (JWS) - JSON data structures representing MAC / Digital Signature.
Standardized Format - Description of how data is structured.
Standardized Protocol - Description of how structured data is procedurally handled.
Docker Image - A template with a set of instructions to launch a Docker container.
Docker Container - Isolated application executed from a Docker image.
CSRF - Cross-Site Request Forgery.
DCR - Dynamic Client Registration.
SPA - Single Page Application.
JWT - JSON Web Token.

9 Appendix

9.1 IAM Role Example

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::my-bucket-name-region/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::my-bucket-name-region/*"
        }
    ]
}
```

9.2 Lambda Example

```
import boto3
import json
print('loading function')
def respond(err, res=None):
    return {
        'statusCode': '400' if err else '200' ,
        'body': err.message if err else json.dumps(res),
        'headers': {
            'Content-Type': 'application/json',
        },
    }
def hello(event, context):
    body = {
        "message": "Executed Successfully",
        "input": event
    }
    response = {
        "statusCode": 200,
        "body": json.dumps(body)
    }
    string = ""
    for x in "abcdefghijklmnopqrstuvwxyz":
        string += x
    encoded_string = string.encode("utf-8")
    file_name = "Test.txt"
    bucket_name = "my-bucket-name"
    lambda_path = "/tmp/" + file_name
    s3_path = "s3-bucket/" + file_name
    s3 = boto3.resource("s3")
    s3.Bucket(bucket_name).put_object(Key=s3_path,
Body=encoded_string)

    return response
```

9.3 OAuth Token Example

```
{
     "Id_token":
"eyJraWQiOiItMzgwNzQ4MTIiLCJ4NXQiOiJNUi1wR1RhODY2UmRaTGpONlZ3cmZheTkwN2ciLCJhbG
ciOiJSUzI1NiJ9.eyJhdF9oYXNoIjoiNEt1SXFOUmFsWWFlZUFYZnlSM21TdyIsImFjciI6InVybjpz
ZTpjdXJpdHk6YXV0aGVudGljYXRpb246aHRtbC1mb3JtOmh0bWwxIiwiYXpwIjoid3d3IiwiYXV0aF9
0aW1lIjoxNTIwMzIxOTEyLCJleHAiOjE1MjAzMjYzNzQsIm5iZiI6MTUyMDMyMjc3NCwianRpIjoiZm
I2MGZiNDYtNGYwZS00NmI2LTkyNzQtYzJlZDBhMzE0MDU2IiwiaXNzIjoiaHR0cHM6Ly9zcHJ1Y2U6O
DQ0My9-IiwiYXVkIjoid3d3Iiwic3ViIjoiam9obmRvZSIsImlhdCI6MTUyMDMyMjc3NCwicHVycG9z
ZSI6ImlkIn0.D_vHKt1rRwqIXX5VumzFkweiTKWykx7X6Wv7LLYSgAgNoq67ews6PoLlWTnviMNSYXh
PV4xpsEqt4b-lMdG53I8g_tslrxVOI3FOy5mysZIub74wkkE0J6Qgba3s8DlbWhj9h4zIO3MNkhfdUR
J2PJ6GY6kwc_8Eril0ilNZ8TU_puT8bQHJ_QWxghY3XpeQHtCyzuVDgVv6q7gfcGoy1JxZaLoXNSh02
ZIpp7thVrgEAWAiWo7v46HJFiBNpyPnJfzDRwbTIdPFMEKoHOLjUCczsii_4akCb97IVPz5I3bRWAST
yig7P_Q0646cNHsHZM-pan7cl5bYb42JI0ykCw",
     "token_type": "bearer",
     "access_token": "858a3746-ebbd-473c-9af7-f74bedd114c5",
     "refresh_token": "3aba4d9b-6339-442e-9c05-126d9cce29da",
     "scope": "openid",
     "expires_in": 299
}
```