Internet-Scale analysis of AWS Cognito Security

Andrés Riancho
andresriancho.com
2019

Introduction

This white-paper contains the methodology and results of an internet-scale security analysis of AWS Cognito configurations. This research identified 2500 identity pools, which were used to gain access to more than **13000 S3 buckets** (which are not publicly exposed), **1200 DynamoDB tables and 1500 Lambda functions**.

Table of Contents

Summary

The paper begins with an introduction to the AWS Cognito service and how it can be configured by the developers to give end-users direct access to AWS resources such as S3 and DynamoDB. Access is restricted by IAM policies which are under the developer's control and, in many cases, do not follow the least privilege principle.

Because Cognito identity pool IDs are randomly generated (UUID4) it was necessary to download thousands of APKs from the Google Play store, decompile them and extract the identifiers. Other sources such as Common Crawl were also used to identify valid identifiers. The tools used to perform these tasks are now open source and referenced below.

Each AWS Cognito identity pool that was configured with an unauthenticated role was analyzed using an in-depth permission enumeration tool that identifies potential breaches to least privilege principle.

The paper ends with recommendations for developers that want to configure the service in a secure manner, and an analysis of potential reasons for this widespread issue such as poor official documentation and missing restrictions.

Introduction to AWS Cognito

According to the [AWS Cognito main site]():

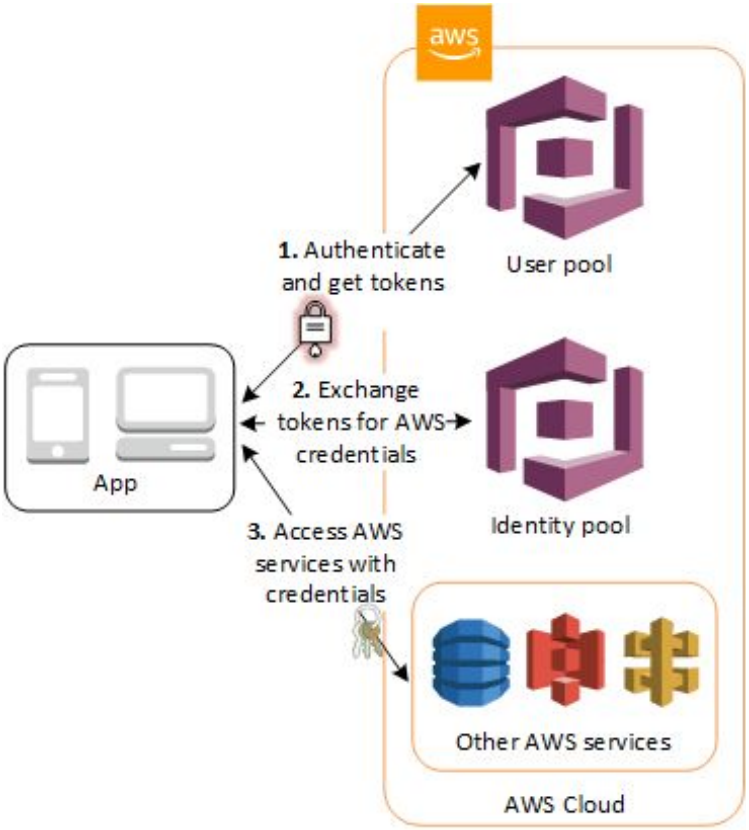
> AWS Cognito lets developers add user sign-up, sign-in, and access control to web and mobile apps quickly and easily. Amazon Cognito scales to millions of users and supports sign-in with social identity providers, such as Facebook, Google, and Amazon, and enterprise identity providers via SAML 2.0.

Amazon Cognito provides authentication, authorization, and user management for your web and mobile applications and has two main components:

- **User pools** are user directories that provide sign-up and sign-in options for your app users
- **Identity pools** enable developers to grant end-users access to AWS services

Identity pools and their configurations were the main subject of this research, since they provide access to potentially sensitive and private information stored in AWS services.

In the figure below it is possible to see how the end-user authenticates against the user pool (1), uses the returned cognito token to obtain AWS credentials (2) and finally consumes other AWS services (3).



AWS Cognito supports both authenticated and unauthenticated roles. Each role can have a different IAM permission set. This concept is easy to understand with an example:

- **CoolCatPics** mobile application wants authenticated users to be able to upload their cat pictures to the application S3 bucket
- Unauthenticated users should only be able to read from the application S3 bucket.
- The developer uses AWS Cognito user pool to authenticate users with Facebook, Google, or other OAuth providers.
- The developer configures the authenticated cognito role to have write permissions on S3, and the unauthenticated cognito role to only have read permissions on the same bucket.
- The developer uses the AWS Android SDK to interact with AWS Cognito, authenticate the user (when needed) and retrieve the AWS credentials to consume the S3 API.

Mobile, web and desktop

AWS Cognito was created for mobile application development, but it is also possible to use it to create web or desktop applications. When used in web applications the end-user browser uses JavaScript to directly consume the AWS service API (*S3 in the previous example*).

The most common libraries used to consume the AWS Cognito service are:

- AWS Android SDK
- AWS iOS SDK
- AWS JavaScript SDK

Other libraries, such as **boto3** (Python) can also be used to interact with AWS Cognito, authenticate users, obtain AWS credentials from identity pools, and consume other AWS services.

Getting AWS credentials

The following Python code can be used to obtain AWS credentials from an identity pool:

```
def get_pool_credentials(identity_pool):
    client = boto3.client('cognito-identity')

    _id = client.get_id(IdentityPoolId=identity_pool)
    _id = _id['IdentityId']

    credentials = client.get_credentials_for_identity(IdentityId=_id)

    access_key = credentials['Credentials']['AccessKeyId']
    secret_key = credentials['Credentials']['SecretKey']
    session_token = credentials['Credentials']['SessionToken']
```

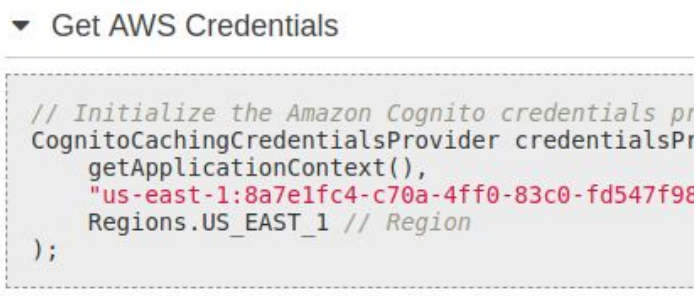
```
    identity_id = credentials['IdentityId']
    return access_key, secret_key, session_token, identity_id
```

The requirements for this function code to work are:

- The **identity_pool** needs to be a valid identity pool identifier
- The identity pool needs to have an unauthenticated role (79.2% of the analyzed identity pools allowed unauthenticated roles)
- The identity pool needs to be correctly configured (found 3.3% of identity pools that for unknown reasons had an invalid configuration and failed to return credentials)

Identity pools use randomly generated unique identifiers

When a new identity pool is generated, the AWS Cognito service shows a message similar to the following to the application developer:



The important information in the previous image is shown in red, where the identity pool is assigned a randomly generated unique identifier (UUID4). This identifier is required to interact with the identity pool (as seen in the code above), thus it will be hard-coded in the mobile, web and desktop applications that use AWS Cognito.

Getting access to a large set of identity pool identifiers was one of the most interesting challenges that had to be solved during this research. This step was a prerequisite to be able to get AWS credentials, enumerate the associated permissions and identify potential misconfigurations.

Identity pool ID sources

Every application that interacts with the AWS Cognito API requires the identity pool identifier. Web and mobile applications will have the identifier stored in their source code, multiple sources were used to extract these identifiers.

Search engines

Initial Google and Yandex searches quickly showed that identifying identity pool IDs was going to be more complicated than expected. Searching for code snippets which are commonly used by the AWS JavaScript SDK to interact with AWS Cognito only returned forum postings and blog posts, but no real usages of those libraries.

Search engines do not index the contents of **<script>** tags, which makes it impossible to search for strings that are found in them (CognitoIdentityCredentials for example):

```
<script>
    window.AWS.config.region = 'us-east-1';
    window.AWS.config.credentials = new window.AWS.CognitoIdentityCredentials({
      IdentityPoolId: 'us-east-1:d917fd38-4...'
    });
```

Even with this limitation, search engines helped identify the first identity pool identifiers. These were used to assert that other tools used during the research were working as expected.

Common Crawl

Common crawl builds and maintains an open repository of web crawl data that can be accessed and analyzed by anyone. Common crawl spiders the internet and stores each HTTP request-response in WARC format. The latest crawl, which was performed between May 19th and 27th 2019 contains **2.65 billion web pages or 220 TiB of uncompressed content**.

During this research, and In order to process this information, cc-lambda was created. The tool uses AWS Lambda to parallelize the process of searching through common crawl data and can:

- Use 1000 concurrent AWS Lambda functions
- Download warc archive, decompress, search using Python regular expression engine
- Store matches in S3

An example run of the **cc-lambda** tool follows:

```
$ python cc-lambda.py
Overall progress: 1.55%
Going to process 250 WARC paths
Got futures from map(), waiting for results...

crawl-data/CC-MAIN-2019-09/.../CC-MAIN-20190215183319-20190215205319-00000.warc.gz
  - Time (seconds): 191
  - Processed pages: 44969
  - Ignored pages: 93005
  - Matches: {'aws_re_matcher': 9, 'cognito_matcher': 4}
```

The **cc-lambda** tool yield 5.3% of the total identified identity pools. Even with this very low number of results the **cc-lambda** tool is very promising for other lines of research which require to analyze the common crawl in any way.

Google Play

Identifying and extracting an identity pool ID from an APK file is simple. The identity pool format is well known: "**{aws-region}:{uuid4}**" and the APK file can be decompressed and later decompiled using automated tools.

The challenge was the huge amount of applications stored in Google Play: **2.6M and growing**. Downloading all applications and inspecting them was the initial approach, but Google Play implements several protections against this process, and the resources and time required to download, decompress, grep and store the results were huge.

It was possible to find multiple paid services that give the customer the possibility to filter the Google Play store using different parameters. One of those services allows customers to filter by the libraries contained by the APK, filtering by AWS Android SDK yield 13000 application names (a huge reduction from 2.6M).

Custom scripts were developed to download the APKs from alternative Google Play Store sites such as apkpure and apkmirror. These scripts received the application name as input and downloaded the APK file to disk, decompressed, inspected, and stored the results to a JSON file.

Google Play crawling yield **83.2% of the total identified identity pools**.

Shodan, ZoomEye and more...

Other sources, such as Shodan, ZoomEye, GitHub, etc. were used to identify more identity pools. These methods didn't yield a considerable number of results when compared with the ones obtained from Google Play.

Permission enumeration

The list of identity pool identifiers obtained in the previous step was used as input for a script that obtained AWS credentials for the Cognito unauthenticated role and then performed permission enumeration.

The enumerate-iam tool was created for this purpose. Other tools, such as Pacu, enumerated only a small subset of the existing API calls, or performed the task in a very slow manner (eg. no threading). The **enumerate-iam** tool was designed following these principles:

- Use IAM to get the role's permissions. In most cases this will fail because the role itself has no permission for the IAM API.
- Only try to run API calls which start with **Get, List or Describe**. These are guaranteed to perform no changes in the target AWS account. By trying anything else the enumeration process might change (or even break) the target AWS account.
- Never send API calls that disclose user-data such as reading S3 bucket contents, DynamoDB table contents, etc.
- Use threads to perform all checks in the fastest possible way
- Enumerate all read-only permissions, for all services

Defining insecure

According to the privilege of least principle each AWS Cognito role should have the smallest set of AWS permissions required to perform the user actions. Going back to the previous **CoolCatPics** application example: the unauthenticated user can only read from S3 and the authenticated user can read from and write to S3.

Tagging an AWS Cognito unauthenticated role permission set as insecure is challenging. One application can follow the privilege of least principle and allow **S3:Put*,** while the same permission can be considered as insecure in a different application: it is all about context.

A detailed analysis of the AWS Cognito use-cases and the API calls allowed me to identify a set of permissions which are, in the great majority of cases, a strong indicator of excessive permissions being granted to the unauthenticated role. These permissions are:

- dynamodb.list_backups()
- dynamodb.list_tables()
- lambda.list_functions()
- s3.list_buckets()

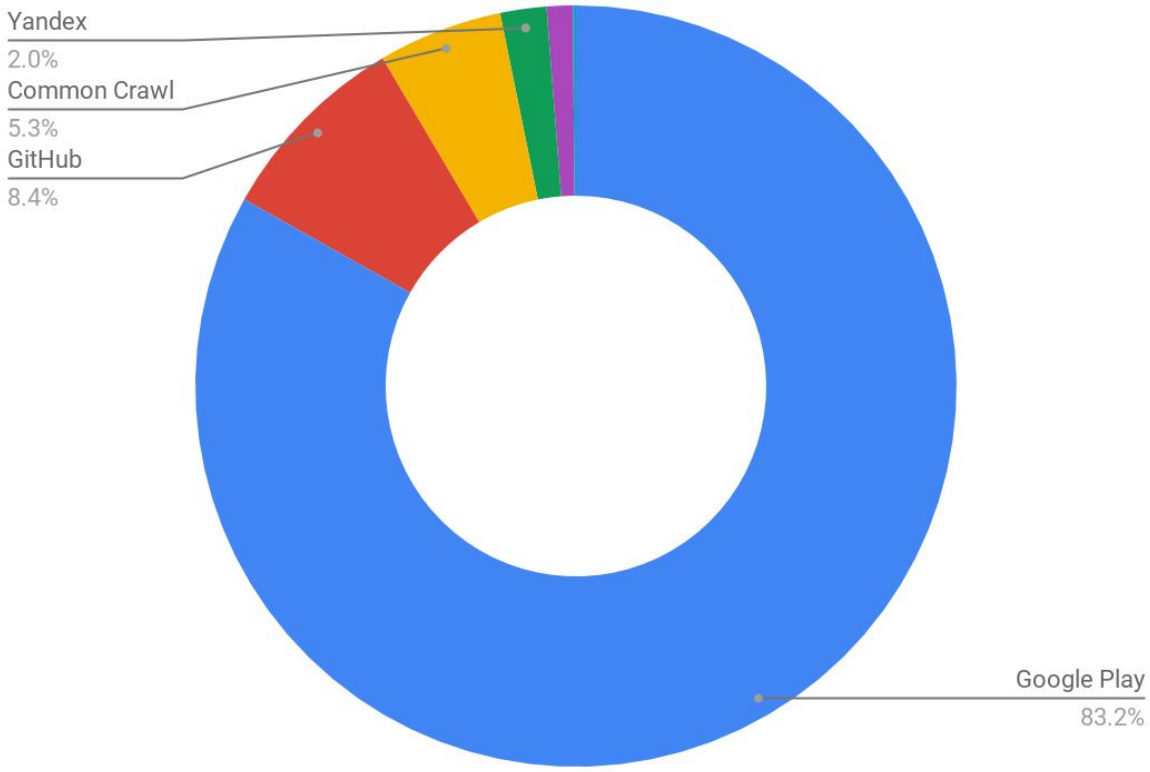
For example, the great majority of the analyzed Android applications that consume AWS Lambda already know the names of the lambda function to run, and there are no dynamically generated AWS lambda functions that would require **lambda.list_functions()** to be allowed.

The same could be said for **s3.list_buckets():** the application already knows the bucket name(s) to use to read and write information, there is no need for the application to list all buckets in the AWS account.

Results

Sources

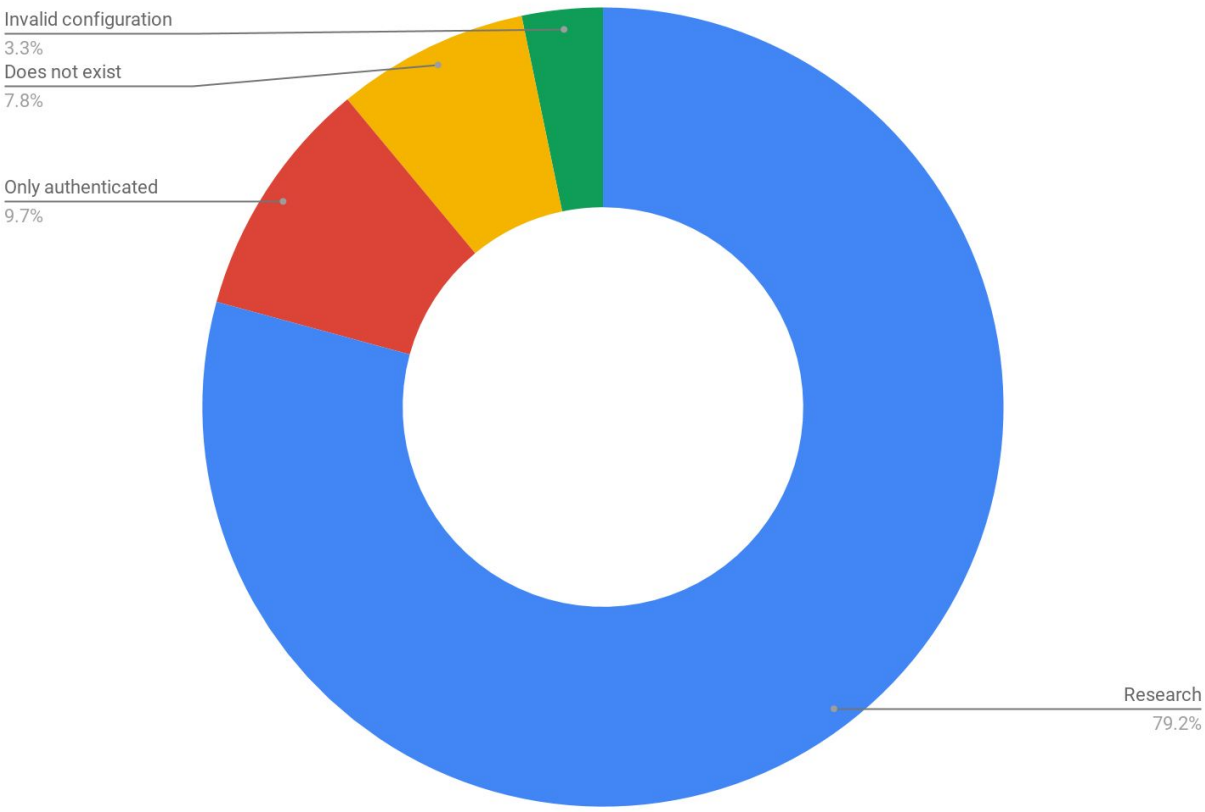
A total of 2504 identity pool identifiers were found in different sources. The following graph shows the distribution of those findings:



Usable identity pools

Identity pool identifiers found in the previously mentioned sources can be in one of the following states:

- **Invalid configuration:** the identify pool has an invalid configuration. The identity pool can't be used until the developer fixes this situation.
- **Does not exist**: the identify pool does not exist anymore. This is most likely an old application in the Google Play store that is deprecated, was not removed from the store but all the AWS resources it uses were removed (to reduce costs).
- **Only authenticated:** the identity pool only allows authenticated roles. This research focuses on the unauthenticated roles associated with AWS Cognito.
- **Research-ready**: identity pool with unauthenticated role and no configuration issues.



Insecure configurations

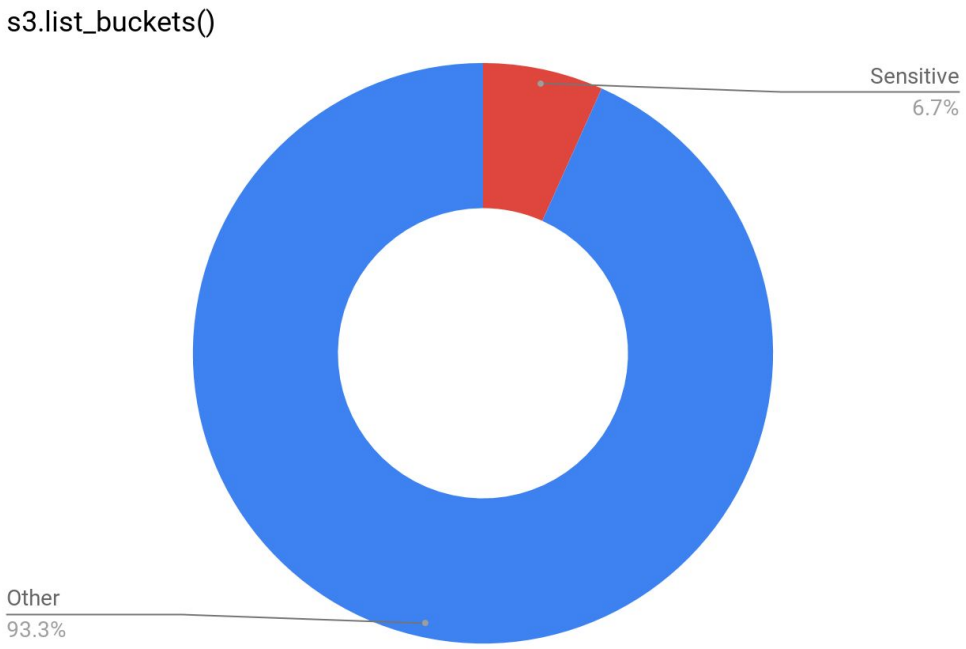
The following table shows the number of unauthenticated AWS Cognito roles which allowed the API calls identified as insecure in the previous section:

API Call	Count	% over total
dynamodb.list_backups()	96	3.83%
dynamodb.list_tables()	101	4.03%
lambda.list_functions()	98	3.91%
s3.list_buckets()	548	**21.88%**

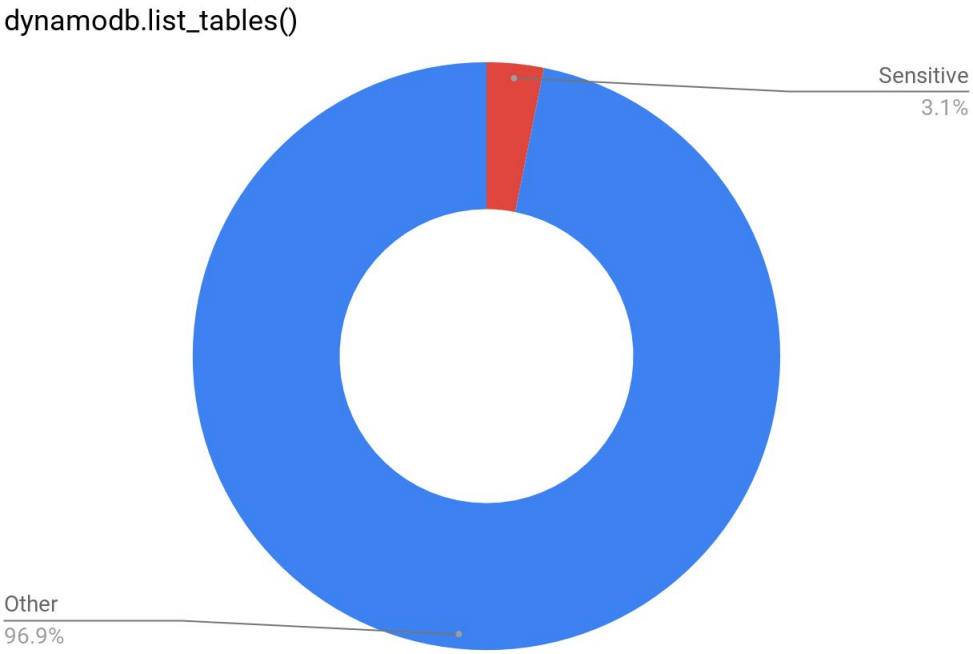
Based on the fact that S3 is the most commonly used AWS service, and using the previous definition of insecure configuration, it is possible to state that **more than 1 in 5 AWS Cognito configurations are insecure.**

Exposed sensitive information

Using the unauthenticated role it was possible to find **906 S3 buckets which contained sensitive information**. Buckets were tagged as sensitive if they contained the word secret, backup, key, source, etc. in their names.



The same analysis was performed on DynamoDB database and table names, identifying at least 37 (3.1%) that potentially hold sensitive information.



It is important to notice that in none of these cases the information stored in the potentially sensitive data stores was inspected. This decision was made to protect the end-user data from exposure.

Credentials in AWS Lambda environment variables

lambda.list_functions(), one of the API calls used to enumerate permissions, returns information about all the AWS lambda functions in the AWS account. This information includes the function name and environment variables used to run it.

Developers commonly use environment variables to store database credentials, API keys, encryption secrets, etc.

During this research it was possible to identify 1572 lambda functions, which exposed **at least 78 environment variables containing**:

- API keys for third-party services
- AWS access keys
- Database credentials
- Passwords

Information that could be used by an attacker to quickly escalate privileges in the AWS account and gain access to sensitive information.

Root cause

Insecure by default documentation

There are multiple AWS Cognito tutorials in the official AWS documentation, many of them lack any recommendations on how to securely assign permissions to authenticated or unauthenticated roles.

For example, the "Uploading Photos to Amazon S3 from a Browser" tutorial allows the unauthenticated AWS Cognito role to perform **S3:* on the application bucket.** Any application user will be able to read, modify and remove all files from the application.

After contacting AWS-security the documentation was updated to include:

> This security posture is useful in this example to keep it focused on the primary goals of the example. In many live situations, however, tighter security, such as using authenticated users and object ownership, is highly advisable.

The AWS-security team started an internal review of all the AWS Cognito documentation, which should improve the examples used by developers to kickstart their applications.

Restrictions on Cognito roles

AWS Cognito imposes **no restrictions** on the permissions a developer can set on AWS Cognito authenticated roles. Given that in most applications that use AWS Cognito the customer can create an application user, this gives the developer too much room for making mistakes.

AWS Cognito allows only 26 services to be associated with the unauthenticated role. For example, it is **impossible to use an IAM role with EC2:* for unauthenticated access**. This restriction is a step in the right direction, but the developer is still able to assign all permissions for services such as DynamoDB, S3, IoT, Lambda, SimpleDB, SES, SNS and SQS to an unauthenticated role.

AWS Cognito allows the developer to shoot himself in the foot: The existing restrictions are not enough, and there are no warnings indicating that a high-risk action is about to be performed. For example, when the developer assigns **S3:*** to an unauthenticated role the UI shows no warning.

Shared responsibility model

All the vulnerabilities identified during this research fall on the client's side of AWS' shared responsibility model. No specific weakness on AWS Cognito was exploited and needs to be patched.

With that said, there is certainly a trend in the results that show that a big percentage of AWS Cognito identity pools are insecurely configured. When only a few developers use a tool in an insecure way, then

it is most likely the developer's responsibility, when a high percentage (one in five) of the developers uses a tool in an insecure way then it is most likely because one or more design issues.

AWS needs to review decisions made during the AWS Cognito design phase. Changes in the user interface, documentation and configuration of the S3 service that prevent public S3 buckets is a good example of how AWS can review design decisions. The same strategy should be applied to AWS Cognito.

Secure coding with AWS Cognito

These are the most important tips for developers using AWS Cognito:

- Always follow the least privilege principle when configuring the IAM roles associated with AWS Cognito authenticated and unauthenticated roles.
- **Remember object level permissions**. This was not even discussed in this research, but keep this in mind: Not all users should be able to read all objects in the S3 bucket, and not all users should be able to read all rows in a DynamoDB table.

Future work

There are multiple ways in which this research could be extended and improved. The following are just some thoughts and ideas that could help the reader continue with this effort:

- **Sources:** The main source of identity pools for this research was Google Play (more than 80%). It would be interesting to extend this research to iOS applications and compare the security score associated with AWS Cognito configurations for Android vs. iOS.
- **Other cloud providers:** There might be other services like AWS Cognito in Azure, GCP or Alibaba.
- **Authenticated role analysis**: It would be possible to implement scripts that perform Facebook, Google, etc. authentication against AWS Cognito user pools and then request AWS credentials from the identity pool. The permissions for those credentials can be enumerated and compared with the ones from the unauthenticated role.
- **Privilege escalation analysis**: Try to answer the question: "In how many AWS Cognito configurations can I perform privilege escalation to gain access to more resources and services?". This is a potentially risky task to perform, since many AWS IAM privilege escalation techniques require the researcher to perform changes on the AWS account.