Serverless Application Security

Workshop Labs

Prerequisites

Install Claudia.js

Claudia.js is a command line utility and can be installed using npm:

```
$ npm install claudia -g
```

You can verify the installation by running the following command:

```
$ claudia --version
5.4.2
```

Configure AWS access keys

In order to use Claudia.js to deploy your serverless functions, you need to configure AWS access keys.

In order to do this, go to the AWS Console and use the **Services** menu to locate the **IAM** service. Go to the **Users tab** and click the **Add User** button. Select a username (I will use `claudia_dev` in all demos) and select the **Programmatic access** (this will allow Claudia.js to use AWS APIs to deploy your serverless functions).

Next step is to assign appropriate permissions to the user. In the **Set permissions** screen, select the **Attach existing policies directly** tab. Then use the filter to add the following policies:

- *IAMFullAccess*
- *AWSLambdaFullAccess*
- *AmazonAPIGatewayAdministrator*
- *AmazonDynamoDBFullAccess*
- *AmazonAPIGatewayPushToCloudWatchLogs*
- *SecretsManagerReadWrite*

Those are quite powerful permissions but are only needed to deploy and configure our serverless application. A different set of permissions will be used to **run** the application.

Finish the user creation process by clicking **Next: Tags** and then **Next: Review** buttons. Finally click the **Create User** button. At this point you will be presented with the **Access key ID** and **Secret access key** values. Write them both to the AWS credentials file in your home directory:

```
$ cat ~/.aws/credentials
[default]
aws_access_key_id=<KEY ID>
aws_secret_access_key=<SECRET KEY>
```

Now Claudia.js is ready to deploy serverless functions.

Install AWS SDK

The last prerequisite step is to install the AWS CLI tools. The best way to do this is to follow instructions at https://aws.amazon.com/cli/.

Verify that the installation works by running the following command:

```
$ aws --version
aws-cli/1.14.52 Python/3.6.4 Darwin/18.5.0 botocore/1.9.5
```

What about testing?

Our sample application is a REST API and we will need to call it. Our recommended tool to do it is Postman. You can download it directly from https://www.getpostman.com/.

If you like another HTTP testing tool such as curl, it will work just fine.

Lab 1: Setup

The goal of the first lab exercise is to deploy the sample serverless application and learn about setting IAM permissions for AWS Lambda functions.

Get the source code

The sample code for this course is freely available on GitHub. You can download it or clone the repository using the following command:

```
$ git clone https://github.com/MarcinHoppe/serverless-appsec
```

Deploy the function

Let's have some fun and deploy our serverless application. You can do this by running the following commands:

```
$ cd tasks-api
$ npm install
$ claudia create --region eu-central-1 --api-module api
```

Please note that a new file (`claudia.json`) file has been created in the root project directory. Claudia.js uses this file to connect your code to a particular AWS Lambda (and related resources) that is just created.

How to update and delete the app

If you modify the code and need to upload it to AWS Lambda, use the following command:

```
$ claudia update
```

Claudia.js will reuse the information stored in the `claudia.json` file and it will not require passing details such as the selected region.

Claudia.js makes it easy to clean the resources it created:

```
$ claudia destroy
```

Inspect the AWS resources

Claudia.js deployment process is quite simple but it does a few things that make the new AWS Lambda immediately ready to accept REST API calls.

Go to the AWS Console and use the **Services** menu to locate the **Lambda** service (it's under the **Compute** group). You can see that the list contains **serverless-appsec** function. Click its name to inspect the details.

One notable attribute of the Lambda function is an IAM role. Click it to see the full definition of the role in the **IAM** service.

Use the **Services** menu once again to locate the **API Gateway** service (it's under the **Networking & Content Delivery** group). Inspect the different resources that were defined by Claudia.js and how they map to the Lambda function we have just deployed.

Deployment looks pretty impressive but we need one more thing to make our serverless application fully working.

Create DynamoDB

Our serverless application requires a database to store tasks. Use the **Services** menu to locate the **DynamoDB** service (it's under the **Database** group). Click the **Create table** button and use *Tasks* as the table name and *taskId* as the primary key. Leave the rest of the settings in their default state and create the table.

Alternatively, you can use to following command to create the table:

```
$ aws dynamodb create-table \
  --table-name Tasks \
  --attribute-definitions AttributeName=taskId,AttributeType=S \
  --key-schema AttributeName=taskId,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1 \
  --region eu-central-1 \
  --query TableDescription.TableArn \
  --output text
```

As a result, you should see the Amazon Resource Name (ARN) of the created DynamoDB table:

```
arn:aws:dynamodb:eu-central-1:246566863362:table/Tasks
```

Store it somewhere, because we will need it in the next step.

Set IAM policy

Now we are almost ready to store and read data from the created table. Our Lambda function does not have the necessary permissions to execute any operations on the DynamoDB table and we will use the AWS CLI to attach a policy to the **tasks-api-executor** role.

Create a new file to hold the policy called `dynamodb.json`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:Scan",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
            "Resource": "<DynamoDB Table ARN>"
        }
    ]
}
```

Remember to replace the `<DynamoDB Table ARN>` with ARN of the table created in the previous step.

Now we can attach this policy to the AWS Lambda execution role:

```
$ aws iam put-role-policy \
  --role-name tasks-api-executor \
  --policy-name TasksApiDynamoDb \
  --policy-document file://./dynamodb.json
```

Now we are ready to test our function.

Test the function

Our serverless API allows us to add tasks, list all tasks, check details of a single task, delete a task and mark a single task as done. The following sections list HTTP requests that can be executed using Postman or curl.

Remember to replace `api-id.execute-api.eu-central-1.amazonaws.com` with your API Gateway URL!

Create a new task

```
POST /latest/tasks HTTP/1.1
Host: api-id.execute-api.eu-central-1.amazonaws.com
Content-Type: application/json
cache-control: no-cache
"Wash dishes"

$ curl -X POST \
  https://api-id.execute-api.eu-central-1.amazonaws.com/latest/tasks \
  -H 'Content-Type: application/json' \
  -H 'cache-control: no-cache' \
  -d '"Wash dishes"'
```

Retrieve all tasks

```
GET /latest/tasks HTTP/1.1
Host: api-id.execute-api.eu-central-1.amazonaws.com
cache-control: no-cache

$ curl -X GET \
  https://api-id.execute-api.eu-central-1.amazonaws.com/latest/tasks \
  -H 'cache-control: no-cache'
```

Check a single task

Remember to replace `ID` with IDs generated for your tasks.

```
GET /latest/tasks/ID HTTP/1.1
Host: api-id.execute-api.eu-central-1.amazonaws.com
cache-control: no-cache

$ curl -X GET \
  https://api-id.execute-api.eu-central-1.amazonaws.com/latest/tasks/ID \
  -H 'cache-control: no-cache'
```

Delete a task

Remember to replace `ID` with IDs generated for your tasks.

```
DELETE /latest/tasks/FAAQJ3W8NO-wc7oHGL60eaId93zTSeeA HTTP/1.1
Host: api-id.execute-api.eu-central-1.amazonaws.com
cache-control: no-cache
```

```
$ curl -X DELETE \
  https://api-id.execute-api.eu-central-1.amazonaws.com/latest/tasks/ID \
  -H 'cache-control: no-cache'
```

Mark task as done

Remember to replace `ID` with IDs generated for your tasks.

```
POST /latest/tasks/ID/done HTTP/1.1
Host: api-id.execute-api.eu-central-1.amazonaws.com
cache-control: no-cache
```

```
$ curl -X POST \
  https://api-id.execute-api.eu-central-1.amazonaws.com/latest/tasks/ID/done \
  -H 'cache-control: no-cache'
```

Lab 2: Authentication with Auth0

The goal of the second lab exercise is to learn how to authenticate and authorize users to call your serverless API.

Configure Auth0 (demo)

We will use Auth0 as an identity provider (IdP) that will issue access tokens for our serverless API. Let's configure all Auth0 components needed to make our serverless application work.

Add user database

Log in to Auth0 **Management Dashboard** and select **Connections** and then **Database** from the left hand side menu. Click the **Create DB Connection** button, use *Tasks-Users* as a name. Leave other settings with their default values and click the **Create** button.

Add API

The next step is to configure our serverless API. Select **APIs** from the left hand side menu and click **Create API** button. Use *Tasks-API* as the API name and *https://example.com/tasks* as the logical API identifier (we will need to use it later to verify the JWT access token). Leave RS256 as the signing algorithm and click the **Create** button.

Hint: in order to obtain access tokens in Postman, make sure that *https://example.com/tasks* is also set as a **Default Audience** (available in **API Authorization Settings** of your Auth0 tenant).

Add application

The last set is to set up an application that will use our serverless tasks API. In a real world deployment this would likely be a Web or a mobile application, but we will use Postman to simulate the client.

Select **Applications** from the left hand side menu and click the **Create Application** menu. Use *Tasks-App* as the name, select **Regular Web Applications** as application type and then click the **Create** button. Go to the **Settings** tab and add *https://app.getpostman.com/oauth2/callback* to the **Allowed Callback URLs** field.

Note the **Domain**, **Client ID**, and **Client Secret** parameters. In the lab we can use the following values:

Domain	infoshare2019.eu.auth0.com

Client ID	FZmXuaG7yBapNZMOfp3HcQacakIAL1oD
Client Secret	rwsQ2ot9AM4o4iezEchwPrxaKoxgua06AViBUwxLa49P-RH26cDDivbbHU6OEgNm

We will need those values in just a moment to configure Postman.

As a last step, go to the **Connections** tab, enable the *Tasks-Users* database connection and disable the *google-oauth2* social connection.

Configure API Gateway Lambda authorizer

Now we are ready to delegate verification of access tokens for the our serverless API to a custom Lambda function called automatically by the API Gateway.

Deploy authorizer Lambda function

Run the following commands from the project root directory:

```
$ cd authorizer
$ npm install
$ claudia create \
  --region eu-central-1 \
  --name authorize \
  --handler authorizer.authorize
```

From now on you can use `claudia update` and `claudia destroy` commands as you would for any Lambda function.

Inspect authorizer source code

The custom authorizer is in the `authorizer.js` file. Inspect the main `authorize` function that verified the access token and generates a policy document that allows the Lambda function to execute.

Apply the authorizer to the API

Now that the authorizer is used to verify access tokens, it needs to be configured for relevant API endpoints. We can do it using Claudia.js. Configure the authorizer in the `tasks-api/api.js` file:

```
const Api = require('claudia-api-builder');
const api = new Api();

api.registerAuthorizer('auth0', { lambdaName: 'authorize' });
```

Now the last step is to configure each API route to use this authorizer:

```
api.get('/tasks', () => {
    return getAllTasks();
}, { customAuthorizer: 'auth0' });

...

api.delete('/tasks/{id}', (req) => {
    return deleteTask(req.pathParams.id);
}, { customAuthorizer: 'auth0' });
```

Inspect the AWS resources

Claudia.js created several AWS resources for us.

As a first step, go to **Lambda** service to see that in addition to the *tasks-api* function there is also an *authorizer* function.

Next, go to the **API Gateway** service and inspect the *tasks-api* API. Click on any of the routes and notice that the **Authorization** is now set to **CUSTOM**. Click on the **Authorizers** section and notice there is a new *auth0* Lambda authorizer configured to call the *authorize* Lambda function.

Test the function

Our serverless API now requires an access token from Auth0 to call it. We will configure Postman to obtain such a token and attach it to all HTTP requests.

Obtain the access token

Create a new HTTP request in Postman and click the **Authorization** tab. Select **OAuth 2.0** from the **TYPE** drop down box. Then click the **Get New Access Token** button on the right hand side. Use *Auth0 token* as a token name and use the **Authorization Code** as the grant type. Then fill the following fields with values obtained from our Auth0 configuration:

Callback URL	https://app.getpostman.com/oauth2/callback
Auth URL	https://infoshare2019.eu.auth0.com/authorize
Access Token URL	https://infoshare2019.eu.auth0.com/oauth/token
Client ID	FZmXuaG7yBapNZMOfp3HcQacakIAL1oD

Client Secret	rwsQ2ot9AM4o4iezEchwPrxaKoxgua06AViBUwxLa49P-RH26cDDivbbHU6OEgNm

The click the **Request Token** button. You will be presented with Auth0 login box where you can sign up for a new account and login. After you sign in, click the **Use Token** button.

Now you are all set to use this access token in Postman.

Call the API with the access token

Use HTTP requests from Lab 1 to exercise the API with the access token sent in the **Authorization** HTTP header.

As a bonus, try to call the API *without* the access token and see how it behaves!

Bonus: verify claims in the JWT token

As a bonus, you can implement individual claims within the JWT token in the `validAccessToken` function:

```
function validAccessToken(token) {
    console.log('validAccessToken', token);
    return true;
}
```

You can look up token properties in CloudWatch logs. The most important claims to verify are `iss` (issuer) and `aud` (audience). Token expiration (`exp`) would be pretty important to check, too!

Lab 3: Secrets

The goal of the third lab exercise is to learn how to store and access secrets and credentials inside a Lambda function.

Store a secret in AWS Secrets Manager

Some secrets can be generated by AWS services, but we will generate an encryption key outside of AWS and import it into AWS Secrets Manager.

Go to the **Secrets Manager** service in the AWS Console. Click the **Store a new secret** button and select **Other type of secrets** as a secret type. Click on the **Plaintext** tab and paste a real SendGrid API key that will be provided to you during the workshop. Click the **Next** button and use *sendgrid-api-key* as the secret name. Click the **Next** button once again and leave the **Disable automatic rotation** as the default option. Click the **Next** button to review settings for the new secret and then click **Store** to complete the process.

You can also create the secret using AWS CLI by running the following command from the root directory:

```
$ aws secretsmanager create-secret \
  --region eu-central-1 \
  --name sendgrid-api-key
  --secret-string `<SENDGRID API KEY>`

{
    "ARN": "<KEY ARN>",
    "Name": "sendgrid-api-key",
    "VersionId": "<KEY VERSION>"
}
```

Set IAM policy

At this point our serverless API does not have necessary permissions to read the secret to encrypt and decrypt data.

Let's create an IAM policy file `secretsmgr.json` similar to the one we created for DynamoDB:

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Effect": "Allow",
            "Resource": "<Secret ARN>"
        }
    ]
}
```

Remember to replace the `<Secret ARN>` with ARN of the secret created in the previous step.

Now we can attach this policy to the AWS Lambda execution role:

```
$ aws iam put-role-policy \
  --role-name tasks-api-executor \
  --policy-name TasksApiSecretsMgr \
  --policy-document file://./secretsmgr.json
```

Now we are ready to use the secret in our function.

Modify the function to use the secret

Now we are all set up to use the SendGrid API key to send a confirmation email when the task has been completed. The first step is to import the appropriate module in the `completeTask.js` file:

```
const AWS = require('aws-sdk');
const db = new AWS.DynamoDB.DocumentClient();
const mail = require('../utils/email');
```

Then we can send the email after the DynamoDB record has been updated in the `completeTask` function:

```
return db.update(params).promise()
        .then((res) => {
            console.log('Task updated!', res);
            return mail.send(
                <FROM ADDRESS>,
                <TO ADDRESS>,
                'Task completed',
                `Task ${id} completed`);
        })
```

```
    .catch((err) => {
        console.log('Error updating task');
        throw err;
    });
```

Remember to adjust `<FROM ADDRESS>` and `<TO ADDRESS>` to e-mail addresses you can access during the lab!

Test the function

Add a new task using a request to `POST /tasks` endpoint and then complete this task with a request to `POST /tasks/<ID>/done` endpoint.

The task completion confirmation e-mail may end up in spam folder. Make sure you check it!

Lab 4: API Gateway and WAF

The goal of the fourth and final lab exercise is to learn how to protect your serverless API using AWS Web Application Firewall (WAF).

Configure WAF for API Gateway

Go to the **WAF & Shield** service in the AWS Console and click the **Go to AWS WAF** button. Then click the **Configure web ACL** button and click **Next** to configure a new WAF rule (taking the time to review the WAF concepts is not a bad idea!). Use *Tasks-ACL* as the name and select the same region that you used to set up the API Gateway. Then select **API Gateway** as a resource and select the *tasks-api* and click **Next**.

Now we are going to configure a rule that will stop attacks trying to flood our APIs with large request payloads. Click **Create condition** next to the **Size constraint conditions** box. Use *LargePayload* as condition name. Then use the **Filter settings** box to filter all requests with body length greater than 100 bytes. Click **Create** and then **Next**. Now in the **Create rules** screen, add a rule that drop all requests that match the *LargePayload* condition. This should be an easy task by now!

Now select **Allow all requests that don't match any rules** as a **Default action** and finalize setting up the rule. Now we are ready to test!

Test blocking malicious requests

Now you can use Postman to send a request to `POST /tasks` endpoint. Try to create a task with name shorter than 100 characters and then try to attack your API by sending a large payload.

Did your attack succeed?

Bonus task

As a bonus, you can configure a rate-based rule to stop excessive number of requests from flooding your API. You can install a package called artillery to simulate an attack:

```
$ npm install -g artillery
```

and use it launch a significant number of requests:

```
$ artillery quick --count 50 -n 100 <API Gateway URL>/tasks
```