Lab 5: Securing Serverless Applications

2 hours Free ★★★★★ Rate Lab

aws training and certification

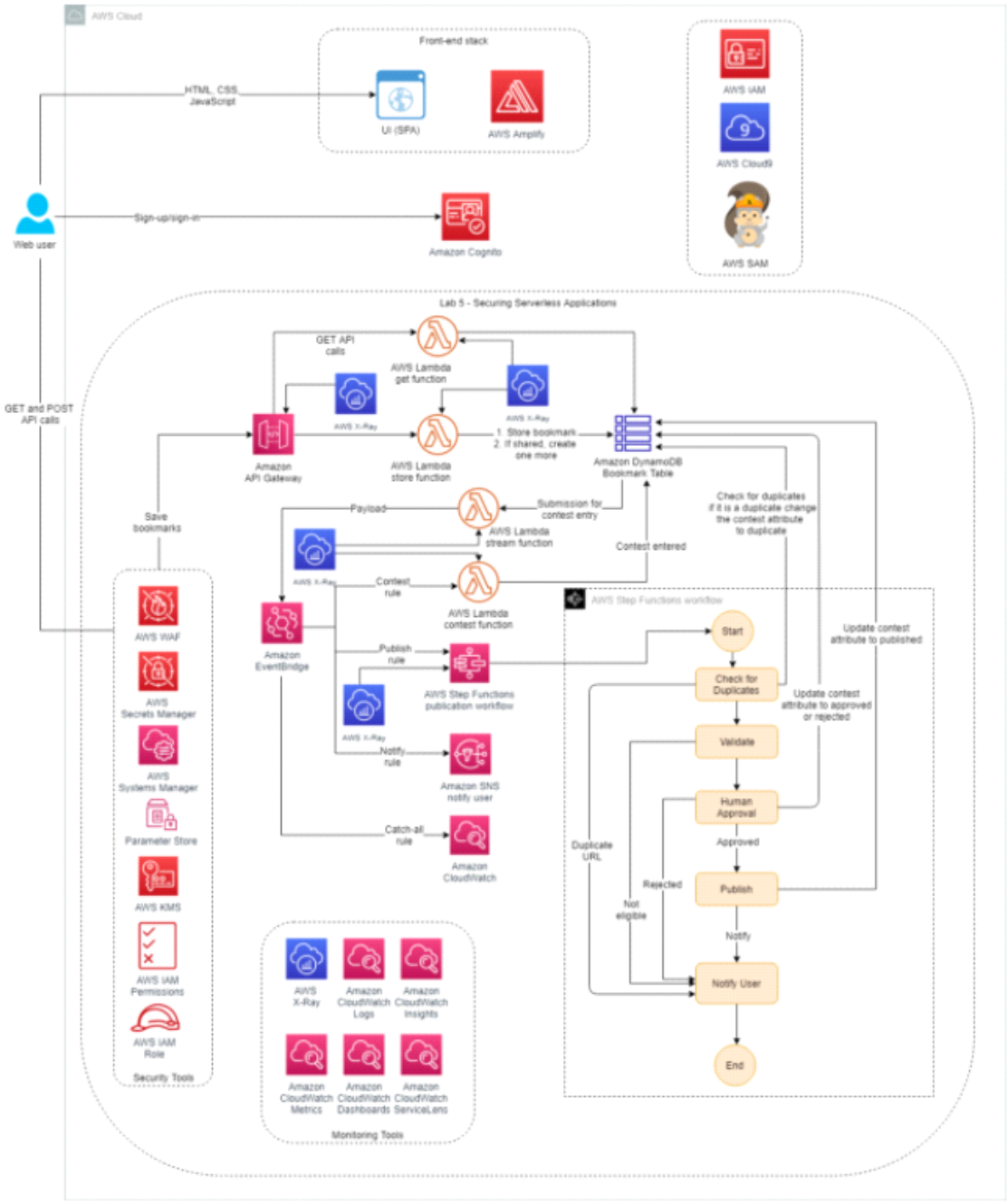
Corrections, feedback, or other questions? Contact us at *AWS Training and Certification*.

Overview

Before you'll be able to make this application available outside of your development team, you need to review security best practices for securing access and protecting resources and data. You have successfully completed coding your application with several features and have also taken care of observability and monitoring aspects of the bookmark application. In this lab, you look into security aspects to ensure the protection of your resources and data and to avoid application outages.

The following architecture diagram shows the components that have been and will be deployed:

The following architecture diagram shows the components that have been and will be deployed:



This lab uses the following services:

- AWS Amplify
- AWS Serverless Application Model (AWS SAM)
- Amazon Cognito
- AWS Cloud9
- Amazon DynamoDB
- Amazon EventBridge
- Amazon Simple Notification Service (Amazon SNS)

- Amazon DynamoDB
- Amazon EventBridge
- Amazon Simple Notification Service (Amazon SNS)
- AWS Step Functions
- AWS Lambda
- Amazon CloudWatch
- Amazon API Gateway
- AWS WAF
- AWS Key Management Service (AWS KMS)
- AWS Systems Manager Parameter Store
- AWS Secrets Manager

Objectives

After completing this lab, you will be able to:

- Secure your application with AWS WAF web ACLs
- Secure access to your API with an API Gateway resource policy
- Secure your Lambda functions and other backend services with AWS KMS, Systems Manager Parameter Store, and Secrets Manager

Prerequisites

This lab requires:

- Access to a notebook computer with Wi-Fi and Microsoft Windows, macOS, or Linux (Ubuntu, SUSE, or Red Hat)
- For Microsoft Windows users, administrator access to the computer
- An internet browser such as Chrome, Firefox, or Internet Explorer 9 (previous versions of Internet Explorer are not supported)
- A text editor

⚠ **Note** The lab environment is not accessible using an iPad or tablet device, but you can use these devices to access the lab guide.

Duration

This lab requires approximately **90 minutes** to complete.

Start Lab

1. At the top of your screen, launch your lab by choosing **Start Lab**

   This starts the process of provisioning your lab resources. An estimated amount of time to provision your lab resources is displayed. You must wait for your resources to be provisioned before continuing.

   ℹ If you are prompted for a token, use the one distributed to you (or credits you have purchased).

2. Open your lab by choosing **Open Console**

   This opens an AWS Management Console sign-in page.

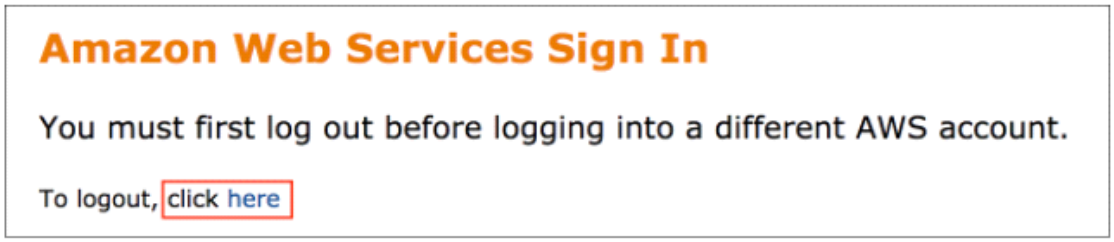
3. On the sign-in page, configure:

   - **IAM user name:** `awsstudent`
   - **Password:** Paste the value of **Password** from the left side of the lab page
   - Choose **Sign In**

⚠ **Do not change the Region unless instructed.**

Common Login Errors

Error: You must first log out



If you see the message, **You must first log out before logging into a different AWS account:**

- Choose **click here**
- Close your browser tab to return to your initial lab window

- Choose **click here**
- Close your browser tab to return to your initial lab window
- Choose Open Console again

Task 1: Understanding AWS WAF and securing the application with web ACLs

AWS WAF is a web application firewall that helps protect your web applications or APIs against common web exploits that may affect availability, compromise security, or consume excessive resources. AWS WAF gives you control over how traffic reaches your applications by enabling you to create security rules that block common attack patterns, such as SQL injection or cross-site scripting, and rules that filter out specific traffic patterns you define.

You can get started quickly using AWS Managed Rules for AWS WAF, a pre-configured set of rules managed by AWS or AWS Marketplace sellers. AWS Managed Rules for AWS WAF address issues like the OWASP Top 10 security risks. These rules are regularly updated as new issues emerge. AWS WAF includes a full-featured API that you can use to automate the creation, deployment, and maintenance of security rules.

First, you launch the application by running a pre-programmed script via **AWS Cloud9**.

4. In the **AWS Management Console**, choose Services and select **Cloud9**.

5. In the left navigation pane, choose the ☰ icon to expand the menu, and choose **Your environments**.

   If the menu is already expanded, move onto the next step.

6. For the **BookmarkAppDevEnv** environment, choose Open IDE

   Within a few seconds, the AWS Cloud9 environment launches.

7. In the AWS Cloud9 terminal, run the following command to download the application

Within a few seconds, the AWS Cloud9 environment launches.

7. In the AWS Cloud9 terminal, run the following command to download the application code and run the startup script:

```
wget https://us-west-2-tcprod.s3-us-west-2.amazonaws.com/courses/ILT-TF-200-SVDVSS/v1.0.2/lab-5-Security/scripts/app-code.zip
unzip app-code.zip
cd app-code
chmod +x startupscript.sh
./startupscript.sh
```

The script takes a couple of minutes to run.

What is the script doing?

- This script is modifying the **samconfig.toml** file within the backend portion of the app code.
- It is replacing values such as AWS Region, stack name, and role Amazon Resource Name (ARN), among others.
- It then updates the **aws-exports.js** file with the Amazon Cognito metadata that was launched in the lab AWS CloudFormation template.
- It then runs **npm build**, deploys the bookmark app, and uploads the **app.zip** file to the **samserverless** Amazon Simple Storage Service (Amazon S3) bucket.

Note Leave this page open.

Task 1.1: Securing with AWS WAF web ACLs

A web access control list (web ACL) has a capacity of 1,500. You can add hundreds of rules and rule groups to a web ACL. The total number that you can add is based on the complexity and capacity of each rule.

A rate-based rule tracks the rate of requests for each originating IP address and triggers the rule action on IPs with rates that exceed a limit. You set the limit as the number of requests per a 5-minute time span. You can use this type of rule to put a temporary block on requests from an IP address that's sending excessive requests.

number of requests per a 5-minute time span. You can use this type of rule to put a temporary block on requests from an IP address that's sending excessive requests. By default, AWS WAF aggregates requests based on the IP address from the web request origin, but you can configure the rule to use an IP address from an HTTP header, such as X-Forwarded-For, instead.

When the rule action triggers, AWS WAF applies the action to additional requests from the IP address until the request rate falls below the limit. It can take a minute or two for the action change to go into effect.

In this task, you create a web ACL to secure the API Gateway resources using AWS WAF.

Create a web ACL

8. In the AWS Management Console, choose **Services** and open **WAF & Shield** in a new browser tab.

9. Choose **Create web ACL**

10. In the **Web ACL details** section, configure the following details:

    - **Region:** Choose the Region from the dropdown list. The Region should be the one displayed in the left column of the lab instructions.

    Note If the Region is not set to the value displayed in the left column of the lab instructions, you need to change the Region before filling in the other fields. Changing the Region clears the other field values if you have entered any information before selecting the Region. If you are not sure about the Region, choose the Global Region dropdown list in the AWS Management Console.

    - **Name:** Enter `BookmarkACL`
    - **Description:** Enter `Block actions from the API Gateway`
    - **CloudWatch metric name:** Enter `BookmarkACL` (pre-populated with the text entered for the **Name** value)
    - **Resource Type:** Choose **Regional resources (Application Load Balancer, API Gateway, AWS AppSync)**

11. Choose **Next**

12. In the **Rules** section, configure the following details:

11. Choose **Next**

12. In the **Rules** section, configure the following details:

    - Choose **Add rules** and select **Add my own rules and rule groups** from the dropdown list.
    - Leave the default **Rule type** as **Rule builder**.

13. In the **Rule builder** section, configure the following details:

    - **Name:** Enter `100ratebasedrule`
    - **Type:** Select **Rate-based rule** from the dropdown list

14. In the **Request rate details** section, configure the following details:

    - **Rate limit:** Enter `100`
    - **IP address to use for rate limiting:** Leave the default value as **Source IP address**
    - **Criteria to count request towards rate limit:** Leave the default value as **Consider all requests**

15. In the **Then** section, leave the default value of **Action** as **Block**.

16. Choose **Add rule**

17. On the **Add rules and rule groups** page, choose **Next**

18. On the **Set rule priority** page, choose **Next**

19. On the **Configure metrics** page, choose **Next**

20. On the **Review and create web ACL** page, scroll to the bottom and choose **Create web ACL**

    The web ACL has been created, and an ID has been generated for the web ACL.

    Note Leave this page open.

Attach the web ACL to API Gateway

21. In the AWS Management Console, choose **Services** and open **API Gateway** in a new browser tab.

new browser tab.

22. Choose the **Bookmark App**.

23. In the left navigation pane, choose **Stages**.

24. In the **Stages** pane, choose the ▸ dev stage.

25. Copy and paste the **Invoke URL** value into a text editor to use later.

    Note The next step is needed in order to select the **Web ACL** because it is grayed out.

26. Choose **Save changes**

27. In the **Web Application Firewall (WAF)** section, from the **Web ACL** dropdown list, choose **BookmarkACL**.

28. Choose **Save changes**

    The web ACL has been assigned to the **dev** stage of the bookmark application.

    Note Leave this page open.

Test the web ACL using Artillery

First, you load the bookmark data using the Artillery tool and then invoke a load test.

29. In the AWS Cloud9 console, choose the **>** arrow next to the **app-code** folder to expand it.

30. Choose the **>** arrow next to the **test** folder to expand it.

31. Open the **simple-post.yaml** file.

32. In the **simple-post.yaml** file, find *(Replace with API Gateway Invoke URL)*, and replace it with the API Gateway **Invoke URL** you copied previously.

33. At the top of the page, choose **File > Save** to save the file.

34. In the AWS Cloud9 terminal, run the following code to install Artillery, install Faker, and run the script.

34. In the AWS Cloud9 terminal, run the following code to install Artillery, install Faker, and run the script.

```
cd test
npm install artillery -g
npm install faker
artillery run simple-post.yaml
```

This script runs for 30 seconds, adding data through the API and then invoking the createBookmark Lambda function.

35. In the command below, replace *(Invoke URL)* with the **Invoke URL** you copied earlier. In the AWS Cloud9 terminal, run the adjusted command to test the BookmarkACL.

```
artillery quick -n 20 --count 100 (Invoke URL)/bookmarks
```

Note This command creates a load test of 100 users with 20 requests from each user because the goal is to reach 100 requests per minute. The 200 status code for each request indicates that the request has been successful.

In the next steps, you run a curl command to retrieve bookmark details in order to test the web ACL.

36. In the AWS Management Console, choose **Services** and open **DynamoDB** in a new tab.

37. In the left navigation pane, choose **Tables**.

38. Choose the table with the **bookmarksTable** name in it.

39. Choose the **Items** tab.

40. From the **id** column in the table, choose one of the IDs.

41. From the pop-up window, copy the value in the **id** field, and paste it into a text editor.

42. In the curl command below, replace *(Invoke URL)* with the **Invoke URL** value, and replace *(id)* with the **id** value:

```
curl (Invoke URL)/bookmarks/(id)
```

```
curl (Invoke URL)/bookmarks/(id)
```

Note You use this adjusted curl command throughout the rest of this lab, so save it to a text editor to use later.

43. In the AWS Cloud9 terminal, run the adjusted curl command to retrieve the bookmark data. The following message is displayed in the console.

```
{"message": "Forbidden"}
```

This message is displayed because the web ACL you created is blocking the request to the bookmark application. Specifying conditions in a web ACL allows you to protect your application resources.

API Gateway rejects any further calls after the first 100 requests made in 1 minute. When AWS WAF sees continuous requests from the same source IP address, it blocks all future calls based on the rule. After a few minutes, AWS WAF releases the restrictions and automatically lets the calls in.

You can check the requests that are blocked by the web ACL in the AWS WAF console.

44. To check these requests, navigate to the AWS WAF console, and choose **BookmarkACL**.

45. In the **Sampled requests** pane, you see all the requests by default.

46. To see the blocked requests, choose **100ratebasedrule** from the dropdown list.

Now, you see the requests in the **BLOCK** state. As AWS WAF releases the restrictions after a few minutes, you invoke the API Gateway endpoint to see if the request is still being blocked.

Note It might take up to 5 minutes for AWS WAF to release the restrictions.

47. In the AWS Cloud9 terminal, run the curl command you updated earlier to test the release:

```
curl (Invoke URL)/bookmarks/(id)
```

This command displays the bookmark details for the provided **id**.

This command displays the bookmark details for the provided **id**.

Note If you do not see the bookmark details, wait until the restrictions are released, and run the above curl command again.

Task 1.2: Securing with AWS WAF using an IP Address

In this section, you create an IP set that contains an IP address from which you want to block any requests to your application.

48. Navigate to the AWS WAF page, and in the left navigation pane, choose **IP sets**.

49. Choose **Create IP set**

50. In the **IP set details** section, configure the following information:

    - **IP set name:** Enter `IPToBlock`
    - **Description:** Enter `Block this IP address`
    - **Region:** Leave the default Region
    - **IP version:** Leave the default value
    - **IP addresses:** In the AWS Cloud9 terminal, run the following command to get the IP address of the server:

```
curl https://checkip.amazonaws.com/
```

    Copy the IP address displayed in the terminal, and paste it into the **IP addresses** field on the **Create IP set** page. Append `/32` to the end of the IP address.

    Note Without the CIDR block of `/32`, an error will be thrown if the **Create IP set** is chosen.

51. On the AWS WAF page, choose **Create IP set**

    An IP set has been successfully created. Now, you need to attach the IP set to a web ACL.

52. In the left navigation pane, choose **Web ACLs**.

53. Choose

52. In the left navigation pane, choose **Web ACLs**.

53. Choose **Create web ACL**

54. In the **Web ACL Details** section, configure the following details:

    - **Region:** Choose the Region from the dropdown list. The Region should be the one displayed in the left column of the lab instructions.

    Note If the Region is not set to the value displayed in the left column of the lab instructions, you need to change the Region before filling in the other fields. Changing the Region clears the other field values if you have entered any before selecting the Region. If you are not sure about the Region, choose the Global Region dropdown list in the AWS Management Console.

    - **Name:** Enter `IPsetbasedACL`
    - **Description:** Enter `Blocks actions from the specified IP address`
    - **CloudWatch metric name:** Enter `IPsetbasedACL` (pre-populated with the text entered for the **Name** value)
    - **Resource Type:** Choose **Regional resources (Application Load Balancer, API Gateway, AWS AppSync)**

55. Choose **Next**

56. In the **Rules** section, configure the following details:

    - Choose **Add rules** and select **Add my own rules and rule groups** from the dropdown list.

57. In the **Rule type** section, choose **IP set**.

58. In the **Rule** section, configure the following information:

    - **Name:** Enter `IPbasedrule`

59. In the **IP set** section, configure the information below:

    - **IP set:** In the dropdown list, choose **IPToBlock**
    - **IP address to use as the originating address:** Leave the default value as **Source IP address**
    - **Action:** Leave the default value as **Block**

60. Choose **Add rule**

60. Choose **Add rule**

61. On the **Add rules and rule groups** page, choose **Next**

62. On the **Set rule priority** page, choose **Next**

63. On the **Configure metrics** page, choose **Next**

64. On the **Review and create web ACL** page, scroll to the bottom and choose **Create web ACL**

    The web ACL has been created, and an ID has been generated for the web ACL.

    Note Leave this page open.

Attach the web ACL to API Gateway

65. Navigate to the API Gateway page you left open earlier in this task.

66. Refresh this page so that the **IPsetbasedACL** web ACL created earlier appears in the **Web ACL** dropdown list.

67. In the **Web Application Firewall (WAF)** section, from the **Web ACL** dropdown list, select **IPsetbasedACL**.

68. Choose **Save changes**

    The web ACL has been assigned to the **dev** stage of the bookmark application.

    Note Leave this page open.

Test the web ACL using Artillery

69. In the AWS Cloud9 terminal, run the curl command you updated earlier:

```
curl (Invoke URL)/bookmarks/(id)
```

70. The following command is displayed on the terminal because the web ACL is

70. The following command is displayed on the terminal because the web ACL is blocking the requests from your IP address:

```
{"message": "Forbidden"}
```

Before you proceed to the next task, you must remove the above web ACL because it blocks requests to the application from your IP address.

71. Navigate to the API Gateway page you left open earlier in this task.

72. Make sure that you are in the **dev** stage and the **Settings** tab for this stage. In the **Web Application Firewall (WAF)** section, from the **Web ACL** dropdown list, select **None**.

73. Choose **Save changes**

This step disassociates the web ACL from the **dev** stage of the bookmark application.

Task 2: Securing the application with API Gateway resource policies

API Gateway resource policies are JSON policy documents that you attach to an API to control whether a specified principal (typically an AWS Identity and Access Management [IAM] user or role) can invoke the API. You can use API Gateway resource policies to allow your API to be securely invoked by the following:

- Users from a specified AWS account
- Specified source IP address ranges or CIDR blocks
- Specified virtual private clouds (VPCs) or VPC endpoints (in any account)

You can attach a resource policy to an API by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS SDKs.

API Gateway resource policies are different from IAM policies. IAM policies are

AWS Command Line Interface (AWS CLI), or AWS SDKs.

API Gateway resource policies are different from IAM policies. IAM policies are attached to IAM entities (users, groups, or roles) and define what actions those entities are capable of doing on which resources. API Gateway resource policies are attached to resources. For a more detailed discussion of the differences between identity-based (IAM) policies and resource policies, see Identity-Based Policies and Resource-Based Policies.

You can use API Gateway resource policies together with IAM policies.

In this task, you learn how to add certain IP addresses or a range of IP addresses to an allow list to access your API Gateway resources. You create a resource policy for the bookmark API that denies access to any IP address that isn't specifically allowed.

74. Choose the following URL to find the IP address of your system:

    https://checkip.amazonaws.com/

    Note The IP address of your system is displayed in a new browser tab.

75. Copy and paste the IP address into a text editor. You use it in the **Resource Policy** you define in the next few steps.

76. Navigate to the API Gateway page you left open earlier.

77. In the left navigation pane, choose **Resource Policy**.

78. Copy and paste the following policy into the text editor:

```
{
  "Version": "2012-10-17",
  "Statement": [{
      "Effect": "Allow",
      "Principal": "*",
      "Action": "execute-api:Invoke",
      "Resource": "execute-api:/*/*/*"
    },
    {
      "Effect": "Deny",
      "Principal": "*",
      "Action": "execute-api:Invoke",
      "Resource": "execute-api:/*/*/*",
      "Condition": {
        "NotIpAddress": {
          "aws:SourceIp": ["(sourceIpOrCIDRBlock)"]
        }
```

```
        "NotIpAddress": {
          "aws:SourceIp": ["(sourceIpOrCIDRBlock)"]
        }
      }
    }
  ]
}
```

79. In the last line of the policy, replace *(sourceIpOrCIDRBlock)* with the IP address you noted earlier.

80. Choose **Save**

    Now, you need to deploy the changes.

81. In the left navigation pane, choose **Resources**.

82. In the **Resources** pane, choose **Actions** and select **Deploy API**.

83. In the pop-up window, for **Deployment Stage** in the dropdown list, choose **dev**.

84. Choose **Deploy**

    This step deploys the resource policy you just created. Now, you test whether the resource policy is working or not.

85. In the AWS Cloud9 terminal, run the curl command you adjusted earlier to test the resource policy.

    ```
    curl (Invoke URL)/bookmarks/(id)
    ```

    In the console, an error message similar to the following message is displayed:

    ```
    {"Message":"User: anonymous is not authorized to perform: execute-
    api:Invoke on resource: arn:aws:execute-api:(AWS:Region):*******1234:
    (BookmarkAppID)/dev/GET/bookmarks/(bookmarkId) with an explicit deny"}
    ```

    This message is displayed because you created a resource policy to deny all the requests from your system's IP address.

    Before you proceed to the next task, you should remove the above resource policy because it blocks the application from being invoked.

because it blocks the application from being invoked.

86. In the API Gateway console in the left navigation pane, choose **Resource Policy**.

87. Select the entire policy in the text editor, and press **Delete**.

88. Choose **Save**

    Now, you need to deploy the changes.

89. In the left navigation pane, choose **Resources**.

90. In the **Resources** pane, choose **Actions** and select **Deploy API**.

91. In the pop-up window, for **Deployment Stage** in the dropdown list, choose **dev**.

92. Choose **Deploy**

    This step deploys the changes you just made.

93. In the AWS Cloud9 terminal, run the curl command you adjusted earlier to test if the policy was removed successfully.

    ```
    curl (Invoke URL)/bookmarks/(id)
    ```

    The console now displays the bookmark details for the **id** provided in the above command returned by API Gateway.

    Note Leave this page open.

Task 3: Securing an AWS Lambda function

A highly recommended best practice is to never store your secrets or passwords in plain text or hard code them as part of the function code. They should always be encrypted to secure them from attacks. The following are a few widely used tools to manage your secrets:

encrypted to secure them from attacks. The following are a few widely used tools to manage your secrets:

- AWS KMS
- Parameter Store
- Secrets Manager

AWS KMS makes it easy for you to create and manage cryptographic keys and control their use across a wide range of AWS services and in your applications. AWS KMS is a secure and resilient service that uses hardware security modules that have been validated under FIPS 140-2, or are in the process of being validated, to protect your keys. AWS KMS is integrated with AWS CloudTrail to provide you with logs of all key use in order to help meet your regulatory and compliance needs.

Parameter Store provides secure, hierarchical storage for configuration data management and secrets management. You can store data such as passwords, database strings, Amazon Machine Image (AMI) IDs, and license codes as parameter values. You can store values as plain text or encrypted data. You can reference Systems Manager parameters in your scripts, commands, Systems Manager documents, and configuration and automation workflows by using the unique name that you specified when you created the parameter.

Secrets Manager helps you protect secrets needed to access your applications, services, and IT resources. The service enables you to easily rotate, manage, and retrieve database credentials, API keys, and other secrets throughout their lifecycle. Users and applications retrieve secrets with a call to Secrets Manager APIs, eliminating the need to hard code sensitive information in plain text. Secrets Manager offers secret rotation with built-in integration for Amazon Relational Database Service (Amazon RDS), Amazon Redshift, and Amazon DocumentDB. Also, the service is extensible to other types of secrets, including API keys and OAuth tokens. In addition, Secrets Manager enables you to control access to secrets using fine-grained permissions and to audit secret rotation centrally for resources in the AWS Cloud, third-party services, and on premises.

In this task, you learn how to secure secrets using AWS KMS, Parameter Store, and Secrets Manager and retrieve the secrets in your Lambda code.

Task 3.1: Securing environment variables using AWS KMS

94. In the AWS Management Console, choose **Services** and open **Key Management Service** in a new tab.

94. In the AWS Management Console, choose **Services** and open **Key Management Service** in a new tab.

95. In the left navigation pane, choose **Customer managed keys**.

96. Choose **Create key**

97. In the **Configure key** section, leave the default option for **Key type** as **Symmetric**.

98. Choose **Next**

99. In the **Add labels** section, configure the following details:

    - **Alias:** Enter `LambdaSecrets`
    - **Description:** Enter `Creating a Lambda secrets key`

100. Choose **Next**

101. In the **Key administrators** section, select the check box to the left of **awsstudent** user.

102. Choose **Next**

103. In the **This account** section, select the check box to the left of **awsstudent** user.

104. Choose **Next**

     Review the policy in the **Key policy** section.

105. Choose **Finish**

106. Copy and paste the **Key ID** value into a text editor to use in the next step.

107. In the AWS CLI command below, replace *(Key ID)* with the **Key ID** from the previous step. In the AWS Cloud9 terminal, run the adjusted command.

```
aws kms encrypt --plaintext "Key Management Service Secrets" \
                --query CiphertextBlob \
                --output text \
                --key-id (Key ID)
```

     You are encrypting the string **Key Management Service Secrets** using the AWS KMS key ID.

You are encrypting the string **Key Management Service Secrets** using the AWS KMS key ID.

108. Copy and paste the base64 encoded output from the console into a text editor.

    The resulting encoded output is base64 encoded and is provided as an environment variable to your Lambda function. The Lambda function decrypts the data to get the plaintext in order to actually use it.

Create a Lambda function and configure it in API Gateway to test the AWS KMS

In this section, you create a new Lambda function to test the AWS KMS secrets.

109. In the AWS Management Console, choose **Services** and then choose **Lambda**.

110. Choose **Create function**

111. Choose **Author from scratch**, and configure the following information:

    - **Function name:** Enter `sam-bookmark-app-secrets-function`
    - **Runtime:** Choose **Node.js 14.x**

112. Expand ▸ **Change default execution role**, and configure the following information:

    - **Execution role:** Choose **Use an existing role**
    - **Existing role:** Choose **xxxx-SamDeploymentRole-xxxx**

113. Choose **Create function**

114. In the **Code source** section, select **index.js**, open the context(right-click) menu and choose **Open**..

115. Delete the existing code snippet and paste the following code into the code box:

```
const aws = require('aws-sdk');

const kmsSecret = process.env.KMS_SECRET;

let decodedSecret;
let DecodedKMSSecret;
```

```
let decodedSecret;
let DecodedKMSSecret;

const kms = new aws.KMS();
const ssm = new aws.SSM();
const sm = new aws.SecretsManager();

exports.handler = async message => {
    console.log(message);
    let secretType = message.pathParameters.id
    console.log("Secret Type:", secretType);

    if(secretType == 'kms')
        decodedSecret = await decodeKMSSecret();
    else if (secretType == 'ssm')
        decodedSecret = await decodeSSMSecret();
    else if (secretType == 'sm') {
        var password = await decodeSMSecret(userId);
        decodedSecret = "Password is: " + password;
    }
    else
        decodedSecret = "Provide a valid secret type (kms, ssm, or sm
(secrets manager))";

    console.log(decodedSecret);
    const response = {
        statusCode: 200,
        headers: {},
        body: JSON.stringify('Plain text secret(s): ' + decodedSecret)
    };
    return response;
};

async function decodeKMSSecret() {
    if (DecodedKMSSecret) {
        return DecodedKMSSecret;
    }
    const params = {
      CiphertextBlob: Buffer.from(kmsSecret, 'base64')
    };
    const data = await kms.decrypt(params).promise();
    DecodedKMSSecret = data.Plaintext.toString('utf-8');
    return DecodedKMSSecret;
}
```

116. Choose **Deploy**

    You should see a message that says **Changes deployed**

117. Choose the **Configuration** tab to configure the environment variables.

118. In the left navigation pane, choose **Environment variables**.

118. In the left navigation pane, choose **Environment variables**.

119. In the **Environment variables** section, choose Edit

120. In the **Edit environment variables** page, choose Add environment variable and configure the following details:

    - **Key:** Enter `KMS_SECRET`
    - **Value:** Enter the base64 encoded output that you pasted into a text editor earlier

121. Choose Save

    Note This Lambda function shows how to use the AWS KMS SDK to read secrets. Review the function code after it is deployed.

    Note To read the secrets, an IAM permission, **kms:Decrypt**, is needed for the Lambda function. For the purposes of this lab, the permission has already been added to the **SamDeploymentRole**, which is assigned to the Lambda function during the pre-build lab process.

    To test this Lambda function, configure it in API Gateway.

122. Navigate to the API Gateway page you left open at the end of task 2.

123. In the left navigation pane, choose **Resources**.

124. In the **Resources** pane, choose the **/** endpoint, and then choose Actions

125. In the dropdown list, choose **Create Resource**.

126. In the right navigation pane, enter `secrets` for the **Resource Name**.

127. Choose Create Resource

    Once the resource creation is complete, the **secrets** resource appears in the **Resources** pane.

128. In the **Resources** pane, choose the **/secrets** endpoint, and then choose Actions

129. In the dropdown list, choose **Create Resource**.

130. In the right navigation pane, enter `{id}` for the **Resource Name**.

130. In the right navigation pane, enter `{id}` for the **Resource Name**.

131. In the **Resource Path**, the value is set to **-id-** by default when you enter the **Resource Name**. Replace it with `{id}`

132. Choose **Create Resource**

    Once the resource creation is complete, the **{id}** resource appears under the **secrets** resource in the **Resources** pane.

133. In the **Resources** pane, choose the **/secrets/{id}** endpoint, and then choose **Actions**

134. In the dropdown list, choose **Create Method**.

135. In the dropdown menu under the **/secrets/{id}** endpoint, choose the **GET** method, and then choose ✔.

    Note If you see a warning message on top of the pop-up window indicating **Invalid model identifier specified: Empty**, choose the **x** on the window.

136. In the right navigation pane, select the check box for **Use Lambda Proxy integration**.

137. For **Lambda Function**, enter `sam-bookmark-app-secrets-function`

    This is the new Lambda function name that you just created.

138. Choose **Save**

    A window pops up with a message that reads **Add Permission to Lambda Function**.

139. Choose **OK**

    The Lambda function has been integrated into the new API endpoint. Now, the new endpoint should be deployed in order to test the function.

140. In the **Resources** pane, choose **Actions**

141. In the dropdown list, choose **Deploy API**.

142. In the **Deploy API** pop-up window, from the **Deployment stage** dropdown list, choose **dev**.

142. In the **Deploy API** pop up window, from the **Deployment stage** dropdown list, choose **dev**.

143. Choose **Deploy**

    This step has successfully deployed the new endpoint to the **dev** stage.

144. In the **Stages** pane, expand the ▸ **dev** stage.

145. Under the **/secrets/{id}** endpoint, choose **GET**.

146. Copy and paste the **Invoke URL** value into a text editor.

147. Replace *{id}* with `kms` in this **Invoke URL**.

148. Copy the updated **Invoke URL**, paste it into a new browser window, and press **Enter**.

    The browser displays the following message from the Lambda function code:

    ```
    "Plain text secret(s): Key Management Service Secrets"
    ```

    The AWS KMS secret has been successfully decoded and is displaying the actual password.

    If you enter a value other than **kms**, you will see the following message:

    ```
    "Plain text secret(s): Provide a valid secret type (kms, ssm or sm
    (secrets manager))"
    ```

Task 3.2: Storing and accessing passwords using Systems Manager Parameter Store

The Parameter Store is part of Systems Manager. It is used to store not only encrypted secrets but almost any data. IAM permissions can control who is able to access and change the data in a very fine-grained way. For each record in the Parameter Store, a history is stored, which makes it possible to know exactly when a change has occurred.

149. In the AWS Cloud9 terminal, run the following command to store secrets in the Parameter Store:

149. In the AWS Cloud9 terminal, run the following command to store secrets in the Parameter Store:

```
aws ssm put-parameter --name /db/secret --value 'Hello, Parameter Store!'
--type SecureString
```

You are storing a secret with the name **/db/secret** that has a value of **Hello, Parameter Store!**

The following output is displayed:

```
{
    "Version": 1,
    "Tier": "Standard"
}
```

View the Parameter Store in the AWS Management Console

150. In the AWS Management Console, choose **Services** and then choose **Systems Manager**.

151. In the left navigation pane, choose **Parameter Store**.

The **/db/secret** secret you created earlier is displayed here.

152. To view the details of the secret, choose **/db/secret**.

153. To view the value of the secret, choose **Show**.

Test the secret using a Lambda function

Update the **sam-bookmark-app-secrets-function** function code to test the new parameter.

154. In the AWS Management Console, choose **Services** and then choose **Lambda**.

155. Enter `sam-bookmark-app-secrets-function` into the box, and select this function.

156. In the **Code source** section, add the following to the **index.js** file:

156. In the **Code source** section, add the following to the **index.js** file:

- Add the following constant after the kmsSecret constant (line 4):

```
const ssmSecret = process.env.SSM_SECRET;
```

- Add the following code snippet to the end of the existing code:

```
async function decodeSSMSecret() {
    const params = {
        Name: ssmSecret,
        WithDecryption: true
    };
    const result = await ssm.getParameter(params).promise();
    return result.Parameter.Value
}
```

The code should look like the following after you add the previous two snippets:

```
const aws = require('aws-sdk');

const kmsSecret = process.env.KMS_SECRET;
const ssmSecret = process.env.SSM_SECRET;

let decodedSecret;
let DecodedKMSSecret;

const kms = new aws.KMS();
const ssm = new aws.SSM();
const sm = new aws.SecretsManager();

exports.handler = async message => {
    console.log(message);
    let secretType = message.pathParameters.id
    console.log("Secret Type:", secretType);

    if(secretType == 'kms')
        decodedSecret = await decodeKMSSecret();
    else if (secretType == 'ssm')
        decodedSecret = await decodeSSMSecret();
    else if (secretType == 'sm') {
        var password = await decodeSMSecret(userId);
        decodedSecret = "Password is: " + password;
    }
    else
        decodedSecret = "Provide a valid secret type (kms, ssm, or sm
(secrets manager))";
```

```
    else
        decodedSecret = "Provide a valid secret type (kms, ssm, or sm
(secrets manager))";

    console.log(decodedSecret);
    const response = {
        statusCode: 200,
        headers: {},
        body: JSON.stringify('Plain text secret(s): ' + decodedSecret)
    };
    return response;
};

async function decodeKMSSecret() {
    if (DecodedKMSSecret) {
        return DecodedKMSSecret;
    }
    const params = {
      CiphertextBlob: Buffer.from(kmsSecret, 'base64')
    };
    const data = await kms.decrypt(params).promise();
    DecodedKMSSecret = data.Plaintext.toString('utf-8');
    return DecodedKMSSecret;
}

async function decodeSSMSecret() {
    const params = {
        Name: ssmSecret,
        WithDecryption: true
    };
    const result = await ssm.getParameter(params).promise();
    return result.Parameter.Value
}
```

157. Choose **Deploy**

    You should see a message that says **Changes deployed**

158. Choose the **Configuration** tab to configure the environment variables.

159. In the left navigation pane, choose **Environment variables**.

160. In the **Environment variables** section, choose **Edit**

161. In the **Edit environment variables** page, choose **Add environment variable** and configure the following details:

    - **Key:** Enter `SSM_SECRET`
    - **Value:** Enter `/db/secret`

- **Value:** Enter `/db/secret`

162. Choose **Save**

The Lambda function has been updated to read the Parameter Store and display the password.

Note There is an IAM permission, **ssm:GetParameter**, needed for the Lambda function to read Systems Manager. This permission has been added to the **SamDeploymentRole** in the pre-build process of the lab.

163. To test the above changes, go to the text editor with the **Invoke URL** that you saved from task 3.1. In the **Invoke URL**, replace *{id}* with `ssm`

164. Copy and paste the updated **Invoke URL** into a new browser tab, and press **Enter**.

You should see the following text in the browser.

```
"Plain text secret(s): Hello, Parameter Store!"
```

Task 3.3: Storing a secret using Secrets Manager

Secrets Manager helps you meet your security and compliance requirements by enabling you to rotate secrets safely without the need for code deployments. You can store and retrieve secrets using the Secrets Manager console, AWS SDK, AWS CLI, or CloudFormation. To retrieve secrets, you simply replace plaintext secrets in your applications with code to pull in those secrets programmatically using the Secrets Manager APIs.

165. In the AWS Cloud9 terminal, run the following command to create a secret:

```
aws secretsmanager create-secret --name dbUserId --secret-string
"secretsmanagerpassword"
```

This is similar to storing database credentials. Here the UserId is **dbUserId** with a password of **secretsmanagerpassword**.

The following output is displayed:

The following output is displayed:

```
{
    "ARN": "arn:aws:secretsmanager:us-west-2:
{AWS::AccountId}:secret:dbUserId-xxxxxx",
    "Name": "dbUserId",
    "VersionId": "xxxxx"
}
```

View the secret in Secrets Manager from the AWS Management Console

166. In the AWS Management Console, choose **Services** and then choose **Secrets Manager**.

    The **dbUserId** secret you created earlier is displayed here.

167. To view the details of the secret, choose **dbUserId**.

168. In the **Secret value** section, choose **Retrieve secret value** to view the value of the secret.

Test the secret using a Lambda function

The **sam-bookmark-app-secrets-function** function code should be updated to test the secret.

169. In the AWS Management Console, choose **Services** and then choose **Lambda**.

170. Enter `sam-bookmark-app-secrets-function` into the box, and select this function.

171. In the **Code source** section, add the following to the **index.js** file:

    - Add the following constant after the ssmsecret constant (line 5):

```
const userId =  process.env.SM_USER_ID;
```

    - Add the following code snippet to the end of the function code:

- Add the following code snippet to the end of the function code:

```
async function decodeSMSecret(smkey) {
    console.log("SM Key:", smkey);
    const params = {
        SecretId: smkey
    };
    const result = await sm.getSecretValue(params).promise();
    return result.SecretString;
}
```

The code should look like the following code after you add the two previous snippets:

```
const aws = require('aws-sdk');

const kmsSecret = process.env.KMS_SECRET;
const ssmSecret = process.env.SSM_SECRET;
const userId =  process.env.SM_USER_ID;

let decodedSecret;
let DecodedKMSSecret;

const kms = new aws.KMS();
const ssm = new aws.SSM();
const sm = new aws.SecretsManager();

exports.handler = async message => {
    console.log(message);
    let secretType = message.pathParameters.id
    console.log("Secret Type:", secretType);

    if(secretType == 'kms')
        decodedSecret = await decodeKMSSecret();
    else if (secretType == 'ssm')
        decodedSecret = await decodeSSMSecret();
    else if (secretType == 'sm') {
        var password = await decodeSMSecret(userId);
        decodedSecret = "Password is: " + password;
    }
    else
        decodedSecret = "Provide a valid secret type (kms, ssm, or sm
(secrets manager))";

    console.log(decodedSecret);
    const response = {
        statusCode: 200,
        headers: {},
        body: JSON.stringify('Plain text secret(s): ' + decodedSecret)
    };
    return response;
};
```

```
    };
    return response;
};

async function decodeKMSSecret() {
```