Protecting S3 Files

In this lesson, you will learn how to protect your files through encryption when you store them in S3.

WE'LL COVER THE FOLLOWING

- Encrypting files

Encrypting files

Users can now upload files, but they may rightly be concerned about security and privacy. You can encrypt the file contents to protect them. In a typical three-tier server application, an application server could receive user data and then encrypt it before saving it to S3. With a direct upload, you can't control what is sent to S3, because you've removed the gatekeeper. You could try encrypting this on the client device before sending it, but then you couldn't use just a simple browser form. Plus, you'd have to somehow send your encryption keys to client devices, which can create a security nightmare. Because encryption was such a common need, AWS implemented it as part of the platform. With serverless architectures, most gatekeeper roles are passed onto the platform, not to a Lambda function.

You can just flip a switch and all newly created files on S3 will be encrypted at rest, regardless of where they come from. With CloudFormation, that switch is behind the `BucketEncryption` property of the `AWS::S3::Bucket` resource. You can change the template resource definition for the bucket to look like this:

```
Resources:
  UploadS3Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketEncryption:
        ServerSideEncryptionConfiguration:
          - ServerSideEncryptionByDefault:
              SSEAlgorithm: AES256
```

Line 19 to Line 26 of code/ch8/template.yaml

Line 19 to Line 26 of code/ch8/template.yaml

Once activated, server-side encryption works without any changes to the client code. As objects are uploaded, S3 will encrypt them before storage, and decrypt them before sending them back to the client code. In this case, you're letting S3 create an encryption key for your account and use it without any special configuration. When S3 manages the encryption keys, there is no special cost for automatically encrypting or decrypting the content. You can, of course, set your own encryption keys if you need a higher level of security. For more information, check out the section *Protecting Data Using Server-Side Encryption* in the AWS S3 documentation.

To finish things off and deploy this version of the functions, you'll need to rewrite the template significantly. You can leave the web API as it was in the previous chapter, but you'll have to reconfigure the functions. The function that displays the web form now needs full access to the bucket because it will need to generate upload signatures. You also need to pass the bucket name and allowed upload size as environment variables. Modify the function template to look like this:

```
ShowFormFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: user-form/
    Handler: show-form.lambdaHandler
    Runtime: nodejs12.x
    Events:
      ShowForm:
        Type: Api
        Properties:
          Path: /
          Method: get
          RestApiId: !Ref WebApi
    Environment:
      Variables:
        UPLOAD_S3_BUCKET: !Ref UploadS3Bucket
        UPLOAD_LIMIT_IN_MB: !Ref UploadLimitInMb
    Policies:
      - S3FullAccessPolicy:
          BucketName: !Ref UploadS3Bucket
```

Line 31 to Line 50 of code/ch8/template.yaml

The function that showed a thank-you note now does a lot more, so it is renamed to reflect its new responsibilities. You changed the JavaScript file name for the function, so you'll need to upload the `Handler` property

accordingly. Also, you need this function to respond to the `/confirm` URL, so update the `Path` property in the event.

Finally, because it only needs to sign a download URL, there's no need to keep full bucket access for this function. You can reduce it to read-only access and tighten up security. SAM has a convenient template policy for this as well, called `S3ReadPolicy`. You can change the function template according to the next listing:

```
ConfirmUploadFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: user-form/
    Handler: confirm-upload.lambdaHandler
    Runtime: nodejs12.x
    Events:
      ConfirmForm:
        Type: Api
        Properties:
          Path: /confirm
          Method: get
          RestApiId: !Ref WebApi
    Environment:
      Variables:
        UPLOAD_S3_BUCKET: !Ref UploadS3Bucket
    Policies:
      - S3ReadPolicy:
          BucketName: !Ref UploadS3Bucket
```

Line 51 to Line 69 of code/ch8/template.yaml

You haven't added any important new resources in this chapter, so the `Outputs` section of the file will remain the same as before. Now you can build, package, and deploy the stack.

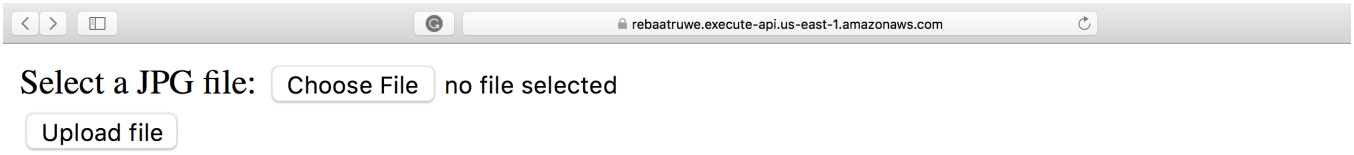
Environment Variables

Key:	Value:
AWS_ACCESS_KEY_ID	Not Specified...
AWS_SECRET_ACCE...	Not Specified...
BUCKET_NAME	Not Specified...
AWS_REGION	Not Specified...

```
{
  "body": "{\"message\": \"hello world\"}",
```

```
  "resource": "/{proxy+}",
  "path": "/path/to/resource",
  "httpMethod": "POST",
  "isBase64Encoded": false,
  "queryStringParameters": {
    "foo": "bar"
  },
  "pathParameters": {
    "proxy": "/path/to/resource"
  },
  "stageVariables": {
    "baz": "qux"
  },
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate, sdch",
    "Accept-Language": "en-US,en;q=0.8",
    "Cache-Control": "max-age=0",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-Country": "US",
    "Host": "1234567890.execute-api.us-east-1.amazonaws.com",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Custom User Agent String",
    "Via": "1.1 08f323deadbeefa7af34d5feb414ce27.cloudfront.net (CloudFront)",
    "X-Amz-Cf-Id": "cDehVQoZnx43VYQb9j2-nvCh-9z396Uhbp027Y2JvkCPNLmGJHqlaA==",
    "X-Forwarded-For": "127.0.0.1, 127.0.0.2",
    "X-Forwarded-Port": "443",
    "X-Forwarded-Proto": "https"
  },
  "requestContext": {
    "accountId": "123456789012",
    "resourceId": "123456",
    "stage": "prod",
    "requestId": "c6af9ac6-7b61-11e6-9a41-93e8deadbeef",
    "requestTime": "09/Apr/2015:12:34:56 +0000",
    "requestTimeEpoch": 1428582896000,
    "identity": {
      "cognitoIdentityPoolId": null,
      "accountId": null,
      "cognitoIdentityId": null,
      "caller": null,
      "accessKey": null,
      "sourceIp": "127.0.0.1",
      "cognitoAuthenticationType": null,
      "cognitoAuthenticationProvider": null,
      "userArn": null,
      "userAgent": "Custom User Agent String",
      "user": null
    },
    "path": "/prod/path/to/resource",
    "resourcePath": "/{proxy+}",
    "httpMethod": "POST",
    "apiId": "1234567890",
    "protocol": "HTTP/1.1"
  }
}
```

When the application is deployed, you can open the API URL in a browser. You should see a form with a button to select a file.



The display form function shows an HTML form with the upload policy in hidden fields, ready to process a file upload.

Select an image file from your local disk using the *Choose File* button then click the *Upload file* button to send it to S3. If you monitor the network requests in your browser, you'll see a connection directly to S3 to transfer the file.

Once the upload completes, your browser will redirect you to the confirmation page. The download link on that page allows you to immediately check that the upload was processed correctly.

Of course, just letting people upload files and then download them back isn't particularly useful. In Chapter 9, you'll set up a service process to handle uploads. In Chapter 10, you'll extend the process to produce thumbnails from uploaded images.

Hopefully you are having a wonderful learning experience. After some interesting experiments and a quiz, you'll move on to the next chapter. Stay tuned!