Signing Requests

Get ready to learn how to sign requests to an AWS service using security keys!

WE'LL COVER THE FOLLOWING

IAM user keys

To explain how temporary grants work, you first need to understand the role of the security keys you entered when configuring command-line access in Chapter 2.

Each IAM user has two keys:

- an access key
- a secret key

When the SDK makes a request to an AWS service, for example `s3.putObject()`, it sends the access key in the request headers. This allows the service to map the request to an AWS account. The SDK also sends a cryptographic signature based on the request body and the secret key using Amazon's Signature Version 4 Signing Process (SIGV4 algorithm). The receiving service uses the access key to locate the corresponding secret key in the IAM database, and also creates a SIGV4 signature for the request. If the two signatures match, AWS knows that the request was authorised by the user.

The interesting part of this is that some services accept templated signatures, so you can create a grant upfront without knowing all the request parameters.

This allows you to effectively produce temporary grants for users to perform limited operations with your AWS resources. For example, you could sign a request matching an S3 upload to a specific file key, in a specific bucket, up to a specific size, and only valid for a specific period of time. You can then safely send this signature to a client, without the risk of exposing the real secret key. The client code can try to upload a file and include the signature in the authorisation header. S3 will examine the signature, evaluate the request against the access policy, and allow or reject the request.

S3 will evaluate the signature both based on the restrictions in the temporary grant and the permissions assigned to the access key. The grant will be invalid if you sign the operation using a key that does not have access to the S3 bucket. This means that you can safely sign temporary grants from a Lambda function. A function can never pass on privileges that it does not have itself.

`s3.createPresignedPost`

Using a Lambda function to just pre-sign uploads for client devices is a very common usage pattern. The AWS SDK for JavaScript already has a shortcut method for that operation, so you don't have to learn the details of the SIGV4 signature process. The method is `s3.createPresignedPost`. It takes a map of conditions such as the expiration time of the policy, partial or full matching on the uploaded file key, maximum allowed length, and the default access level for the uploaded file. The function returns a list of fields that can be used to construct an HTML form.

There are two ways to use those fields. One is to create an HTML form directly and display it on a web page. Browsers will show a button to select a local file and post it to S3, and S3 will redirect the user to a specified page after a successful upload.

Alternatively, you can use those fields to create a `FormData` browser object and upload the file using a background process, such as browser HTML5 `fetch` or `XMLHttpRequest`. The second approach is a lot more flexible because you could use JavaScript to set the content type or target file name, and it would allow you to create a nicer user experience. But to keep things simple for now, you'll use the first approach and just let the browser show a form.

With a browser form workflow, you can tell S3 to redirect users somewhere after a successful upload. You can set up an API endpoint to handle the

redirects, and then provide the URL to the function signing the credentials.

The redirect comes from S3, not your API, so you'll need to define an absolute URL. You could pass the API ID and stage into a Lambda function as variables to create an absolute URL, but you don't have to. When API Gateway sends a request using the Lambda proxy integration, it will also add a `requestContext` property with some operational information about the request itself.

`requestContext`

Check one of the CloudWatch logs from the previous deployments to see this. You'll see that it contains two interesting fields:

- `event.requestContext.domainName` is the full domain name used by the client to make the request.
- `event.requestContext.stage` is the API Gateway stage that received the request.

With API Gateway Lambda proxy integrations, the `requestContext` field contains all the contextual information about the API Gateway resource. This information does not come from the client devices, but from the API Gateway, so it's more reliable than using request headers.

Using the request context, you can easily construct an absolute URL to some other resource on the same API stage. For example, you'll set up a new function to confirm that the upload was successful on `/confirm`. To get the absolute URL, you need to use something like this:

```
const apiHost = event.requestContext.domainName,
  prefix = event.requestContext.stage,
  redirectUrl = `https://${apiHost}/${prefix}/confirm`,
```

Line 7 to Line 9 of code/ch8/user-form/show-form.js

Upload constraints

Instead of always returning a static web form, you can create a pre-signed policy that allows users to upload a file. To prevent security problems, it helps to constrain the uploads:

- Limit the upload key (file ID) to something based on the Lambda request

ID. This ensures that one user can't overwrite a file uploaded by someone else.

- Limit the policy to just 10 minutes. This should be enough to upload even large files, but will not allow people to indefinitely reuse this same grant.
- Limit the size to just a few megabytes so someone couldn't abuse the upload policy to overload your service and spend a lot of money for storage.
- Limit the access level to `private`, meaning that the file will only be accessible from your account. This prevents anyone from abusing your system for public sharing.

```
  params = {
    Bucket: process.env.UPLOAD_S3_BUCKET,
    Expires: 600,
    Conditions: [
      ['content-length-range', 1, uploadLimitInMB * 1000000],
      ['starts-with', '$key', context.awsRequestId + '/']
    ],
    Fields: {
      success_action_redirect: redirectUrl,
      acl: 'private'
    }
  },
  form = s3.createPresignedPost(params);
form.fields.key = context.awsRequestId + '/${filename}';
```

Line 10 to Line 23 of code/ch8/user-form/show-form.js

Note the `startsWith` condition on line 6 in the previous listing doesn't stop someone from uploading a file with the wrong content type. It just ensures that a user can upload only to a very specific prefix. By using an AWS request ID that is unique across all requests, you can guarantee that multiple users do not overwrite each other's uploads.

The `key` field added to the form on line 14 uses a special placeholder `${filename}`. Although this looks like JavaScript variable substitution, it is a plain string (we're using single quotes, not backticks). S3 will use this variable when processing the upload, and replace it with the actual uploaded file name. This trick allows us to pre-sign the upload form without restricting in advance the file name for user uploads.

With the S3 form signing function, you can set many more constraints to ensure that direct uploads are safe enough for your context. For more

information on all the available options, check out the S3 documentation page *Creating an HTML Form (Using AWS Signature Version 4)*.

Once you have a pre-signed policy, you will need to format it as an HTML form. To make the code easier to read, you'll extract that responsibility into a separate method. Here is a utility function that will format the results of `s3.createPresignedPost`:

```
module.exports = function buildForm(form) {
  const fieldNames = Object.keys(form.fields);
  const fields = fieldNames.map(field =>
    `<input type="hidden" name="${field}" value="${form.fields[field]}"/>`
  ).join('\n');
  return `
    <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      </head>
      <body>
      <form action="${form.url}" method="post" enctype="multipart/form-data">
      ${fields}
      Select a JPG file:
      <input type="file" name="file" /> <br />
      <input type="submit" name="submit" value="Upload file" />
      </form>
    </html>
  `;
};
```

code/ch8/user-form/build-form.js

To complete the form display function, you'll need to set up the AWS S3 object and configure the allowed capacity for uploads. You'll use another CloudFormation parameter for this, so the function just needs to read the value from an environment variable. The full function will look like this:

```
const htmlResponse = require('./html-response');
const buildForm = require('./build-form');
const aws = require('aws-sdk');
const s3 = new aws.S3();
const uploadLimitInMB = parseInt(process.env.UPLOAD_LIMIT_IN_MB);
exports.lambdaHandler = async (event, context) => {
  const apiHost = event.requestContext.domainName,
    prefix = event.requestContext.stage,
    redirectUrl = `https://${apiHost}/${prefix}/confirm`,
    params = {
      Bucket: process.env.UPLOAD_S3_BUCKET,
      Expires: 600,
      Conditions: [
        ['content-length-range', 1, uploadLimitInMB * 1000000],
        ['starts-with', '$key', context.awsRequestId + '/']
      ],
      Fields: {
```

```
        success_action_redirect: redirectUrl,
        acl: 'private'
      }
    },
    form = s3.createPresignedPost(params);
  form.fields.key = context.awsRequestId + '/${filename}';
  return htmlResponse(buildForm(form));
};
```

code/ch8/user-form/show-form.js

Let's add an additional parameter for the maximum upload size to your template. Include the block from the following listing in the `Parameters` section of the template, indented one level (so that `UploadLimitInMb` aligns with `AppStage`, which was added in the previous chapter).

```
UploadLimitInMb:
  Type: Number
  Default: 5
  Description: Maximum upload size in megabytes
  MinValue: 1
  MaxValue: 100
```

Line 13 to Line 18 of code/ch8/template.yaml

In the next lesson, you will learn how to get a signed download request.