Exercise 3: Working With Encryption Context

In this section, you will work with encryption context and explore its use in the Document Bucket and other applications.

Background

The Busy Engineer's Document Bucket has metadata, called `context`, associated with each document. This meatadata is a set of key-value string pairs, associated with the item in DynamoDB, searchable there, and attached to the S3 object as well.

One of the features AWS KMS and the AWS Encryption SDK both provide is called encryption context. At its core, encryption context is this metadata mapping: keys that are associated with context about the object, and values indicating information about what that context is. All the information in the map is non-secret, and is the basis for several feature integrations.

One useful model for thinking about encryption context is as Assertions about the Five Ws [https://en.wikipedia.org/wiki/Five_Ws]: Who, What, Where, When, Why. For example:

- *Who* should have access to this decrypted data?
- *What* data is being decrypted?
- *Where* is the decryption happening?
- *When* is this data being used?
- *Why* is this data being decrypted?

AWS KMS: KMS Operations and Encryption Context

AWS KMS allows you to specify an encryption context on `kms:Encrypt`. If you do so, you must provide the exact same encryption context on `kms:Decrypt`, or the operation will fail. (The match is case-sensitive, and key-value pairs are compared in an order independent way.)

Behind the scenes, KMS is cryptographically binding the encryption context to the key material you are `kms:Encrypt` or `kms:Decrypt`ing as *Additional Authenticated Data (AAD)*. In short, this is non-secret data that must be identical (not tampered-with or incomplete), or decryption fails.

This feature defends against risks from ciphertexts being tampered with, modified, or replaced -- intentionally or unintentionally. It both defends against an attacker replacing one ciphertext with another as well as problems like operational events.

For example, if a bad deployment swaps `us-west-2.cfg` with `eu-central-1.cfg` on your fleets, having `{ fleet: us-west-2 }` asserted in `us-west-2.cfg`'s encryption context will prevent it from accidentally being loaded by `eu-central-1`.

AWS KMS: Policy and Audit Hooks

KMS also makes the encryption context available to use in Key Policies and Grants. This means that you can use assertions you make about your data to control usage of your CMKs. Perhaps your `eu-central-1` fleet should only ever be permitted to access encrypted data for `{ shard: europe }`. You can write CMK policies that require `{ shard: europe }` to be asserted about all cryptographic operations, so that KMS refuses to authorize an attempt to decrypt, say, `{ shard: north-america }`. These options can help you secure your application and defend against both operational and security-related risks.

Additionally, as part of the audit features that KMS provides, it logs the encryption context that was supplied with every operation. You can use this information to audit who was accessing what data and when, to detect anomalous call patterns, or to identify unexpected system states.

What questions would you like to answer with CloudTrail Logs for your KMS operations? encryption context can help.

The AWS Encryption SDK

The AWS Encryption SDK includes the encryption context as a core component. Encryption context *may* be supplied on encrypt -- it is optional, both for the Encryption SDK and for KMS, but strongly recommended.

The Encryption SDK writes the encryption context in the encrypted message format. And on decrypt, the Encryption SDK validates the encryption context with KMS and returns the contents to you for you to make assertions about the contents.

Using the Encryption SDK with KMS, you can use all of KMS' policy and audit features from encryption context, and use the Encryption SDK to make assertions to safeguard your application.

Use in the Document Bucket

So how can encryption context be useful in the Document Bucket?

In this exercise, you will walk through a few examples of how leveraging encryption context can help you secure and audit your application, and even build some convenience features.

The Document Bucket already has the `context` map available for operations. It writes the `context` to the DynamoDB records for objects as well, and generates searchable DynamoDB records for context keys, to let you find documents that have certain attributes.

Now you will plumb that context through the AWS Encryption SDK, so that KMS and the Encryption SDK bind those properties as security assertions about your data. You will also add an assertion facility to ensure that your data is what you expect it to be when you call `retrieve` .

What this means is that with this change, you will be able to use encryption context to defend against these kind of risks:

- DynamoDB record updates that create metadata mismatches between a document and its properties
- Swapping objects in S3 so that the data is no longer what it was expected to be
- Accidentally loading the wrong data blob
- Defending against objects being listed as having a certain context key / property when they actually do not

Also, after this change, the contents of `context` will be available in audit log entries written by KMS, and you can now use that metadata in your Key Policies and Grants.

Remember, encryption context is not secret!

Make the Change

Starting Directory

If you just finished Using Multiple CMKs [../multi-cmk/], you are all set.

If you aren't sure, or want to catch up, jump into the `encryption-context-start` directory for the language of your choice.

Java Typescript Node.JS JavaScript Node.JS Python

```
cd ~/environment/workshop/exercises/java/encryption-context-start
```

Step 1: Set Encryption Context on Encrypt

Java JavaScript Node.JS Typescript Node.JS Python

```
// Edit ./src/main/java/sfw/example/esdkworkshop/Api.java and
find store(...)
    // ENCRYPTION-CONTEXT-START: Set Encryption Context on
Encrypt
    CryptoResult<byte[], KmsMasterKey> encryptedMessage =
        awsEncryptionSdk.encryptData(mkp, data, context);
    DocumentBundle bundle =
DocumentBundle.fromDataAndContext(encryptedMessage.getResult(), context);
// Save your changes
```

What Just Happened

The Document Bucket `context` will now be supplied to the AWS Encryption SDK and AWS KMS as encryption context. If a non-empty key-value pair map is supplied to `store`, those key-value pairs will be used in encryption and decryption operations all the way through to KMS:

- The contents of `context` will appear in KMS audit logs.
- The contents of `context` will be availble to use in KMS Key Policies and Grants to make authorization decisions.
- The contents of `context` will be written to the Encryption SDK message.
- Supplying the exact-match contents of `context` will be required to decrypt any encrypted data keys.
- The contents of `context` will now be available on Decrypt to use in making assertions.

Next you will update `retrieve` to use the encryption context on decrypt.

Step 2: Use Encryption Context on Decrypt

Java JavaScript Node.JS Typescript Node.JS Python

```
// Edit ./src/main/java/sfw/example/esdkworkshop/Api.java and find retrieve(...)
    // ENCRYPTION-CONTEXT-START: Use Encryption Context on Decrypt
    Map<String, String> actualContext = decryptedMessage.getEncryptionContext();
    PointerItem pointer = PointerItem.fromKeyAndContext(key, actualContext);
// Save your changes
```

What Just Happened

Now on decrypt, the validated encryption context from the Encryption SDK Message Format header will be passed back to the application. Any business logic that would benefit from using the encryption context data for making decisions can use the version bound and validated by the Encryption SDK and KMS.

Next you will add a mechanism for the application to test assertions made in encryption context before working with the returned data.

Step 3: Making Assertions

Java JavaScript Node.JS Typescript Node.JS Python

```java
// Edit ./src/main/java/sfw/example/esdkworkshop/Api.java and
find retrieve(...)
    // ENCRYPTION-CONTEXT-START: Making Assertions
    boolean allExpectedContextKeysFound =
actualContext.keySet().containsAll(expectedContextKeys);
    if (!allExpectedContextKeysFound) {
        // Remove all of the keys that were found
        expectedContextKeys.removeAll(actualContext.keySet());
        String error =
        String.format(
            "Expected context keys were not found in the actual
encryption context! "
            + "Missing keys were: %s",
            expectedContextKeys.toString());
      throw new DocumentBucketException(error, new
NoSuchElementException());
    }
    boolean allExpectedContextFound =

actualContext.entrySet().containsAll(expectedContext.entrySet());

    if (!allExpectedContextFound) {
        Set<Map.Entry<String, String>> expectedContextEntries =
expectedContext.entrySet();

expectedContextEntries.removeAll(actualContext.entrySet());
```

```
        String error =
            String.format(
                "Expected context pairs were not found in the
actual encryption context! "
                + "Missing pairs were: %s",
                expectedContextEntries.toString());
        throw new DocumentBucketException(error, new
NoSuchElementException());
    }
// Save your work
```

What Just Happened

`retrieve` will use its "expected context keys" argument to validate that all of those keys (with any associated values) are present in the encryption context. `retrieve` will also use its "expected context" argument to validate that the exact key-value pairs specified in expected context are present in the actual encryption context. If either of those assumptions is invalid, `retrieve` will raise an exception before returning the data. These assertions safeguard against accidentally returning unintended, corrupted, or tampered data to the application.

Now the Document Bucket will use AWS KMS and the AWS Encryption SDK to ensure that the `context` metadata is consistent throughout the lifetime of the objects, resistant to tampering or corruption, and make the validated context available to the application logic to make additional business logic assertions safely.

Checking Your Work

If you want to check your progress, or compare what you've done versus a finished example, check out the code in one of the `-complete` folders to compare.

There is a `-complete` folder for each language.

Java Typescript Node.JS JavaScript Node.JS Python

```
cd ~/environment/workshop/exercises/java/encryption-context-
complete
```

Try it Out

Now that you pass encryption context all the way through to KMS and validate it on return, what assertions do you want to make about your data?

Here's some ideas for things to test:

- Expecting exact match of key-value pairs for keys like `stage`, `shard`, and `source-fleet`
- Expecting a set of keys to be present like `submit-date` and `category`
- Expecting an exact match of a subset of the supplied key-value pairs (e.g. only `stage` and `shard`, not `source-fleet`)
- Doing the same for expected keys with any value
- Adding a constraint of a new key that you didn't supply at encryption time
- Adding a constraint with a different value, like `stage=production`
- Changing capitalization
- Using sorted versus unsorted mappings, such as `java.util.SortedMap<K, V>` in Java or `collections.OrderedDict` in Python

There's a few simple suggestions to get you started in the snippets below.

Java JavaScript Node.JS JavaScript Node.JS CLI Typescript Node.JS Typescript Node.JS CLI Python

```
// Compile your code
mvn compile

// To use the API programmatically, use this target to launch
jshell
mvn jshell:run
/open startup.jsh
import java.util.HashMap;
Api documentBucket = App.initializeDocumentBucket();
HashMap<String, String> context = new HashMap<String, String>();
context.put("shard", "test");
context.put("app", "document-bucket");
context.put("origin", "development");
documentBucket.list();
documentBucket.store("Store me in the Document
Bucket!".getBytes(), context);
for (PointerItem item : documentBucket.list()) {
    DocumentBundle document =
documentBucket.retrieve(item.partitionKey().getS(), context);
    System.out.println(document.toString());
}
// Ctrl+D to exit jshell

// Or, to run logic that you write in App.java, use this target
after compile
mvn exec:java
```

Explore Further

Encryption context can provide different types of features and guardrails in your application logic. Consider these ideas for further exploration:

- **Detecting Drift** - `context` contents are stored on the DynamoDB item. S3 has object metadata that could also use the `context` pairs. How would you use the validated encryption context to validate and guardrail those two data sources? What could that feature add to your application?
- **Meta-operations on Encryption Context** - the encryption context is stored on the open-specification AWS Encryption SDK Message Format [https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/message-format.html]. Would it help your system to write tools to process the metadata -- such as the encryption context -- on the message format?
- **DynamoDB Keys and Indexes** - the Document Bucket adds composite indexes by `context` key. What about adding composite keys by key-value pairs? If you know a particular key should always be present in well-formed encrypted data, perhaps that should also be a Secondary Index

[https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html]?

- **Enforing EC Keys** - If you know that there is a key that should always be present, and that you want to index on in DynamoDB, do you want to enforce that it's always present? You can extend the Cryptographic Materials Manager [https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/concepts.html#crypt-materials-manager] component in the AWS Encryption SDK to enforce this during cryptographic operations.
- **Alarms and Monitoring** - How can you leverage encryption context and CloudWatch Alarms for CloudTrail [https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudwatch-alarms-for-cloudtrail.html] to monitor and protect your application?

Next exercise

That's it! You have officially completed the Busy Engineer's Document Bucket workshop. Proceed to Thank You and Closing [../thank-you-and-closing/] for some parting thoughts and information.