AWS: In-band protocol negotiation and robustness weaknesses in AWS KMS and Encryption SDKs

Low **sirdarckcat** published **GHSA-wqgp-vphw-hphf** on Sep 28, 2020

Package

AWS Encryption SDK (Java, Python, C, Javascript)

Affected versions	Patched versions
< 2.0.0	2.0.0

Description

Authors: Thai "thaidn" Duong

Summary

The following security vulnerabilities was discovered and reported to Amazon, affecting AWS KMS and all versions of AWS Encryption SDKs prior to version 2.0.0:

- **Information leakage**: an attacker can create ciphertexts that would leak the user's AWS account ID, encryption context, user agent, and IP address upon decryption
- **Ciphertext forgery**: an attacker can create ciphertexts that are accepted by other users
- **Robustness**: an attacker can create ciphertexts that decrypt to different plaintexts for different users

The first two bugs are somewhat surprising because they show that the ciphertext format can lead to vulnerabilities. These bugs (and the infamous alg: "None" bugs in JWT) belong to a class of vulnerabilities called **in-band protocol negotiation**. This is the second time we've found in-band protocol negotiation vulnerabilities in AWS cryptography libraries; see this bug in S3 Crypto SDK discovered by my colleague Sophie Schmieg.

In JWT and S3 SDK the culprit is the algorithm field—here it is the key ID. Because the key ID is used to determine which decryption key to use, it can't be meaningfully authenticated despite being under the attacker's control. If the key ID is a URL indicating where to fetch the key, the attacker can replace it with their own URL, and learn side-channel information such as the timing and machines on which the decryption happens (this can also lead to SSRF issues, but that's another topic for another day).

In AWS, the key ID is a unique Amazon Resource Name. If an attacker were to capture a ciphertext from a user and replace its key ID with their own, the victim's AWS account ID, encryption context, user agent, and IP address would be logged to the attacker's AWS account whenever the victim attempted to decrypt the modified ciphertext.

The last bug shows that the non-committing property of AES-GCM (and other AEAD ciphers such as AES-GCM-SIV or (X)ChaCha20Poly1305) is especially problematic in multi-recipient settings. These ciphers have a property that can cause nonidentical plaintexts when decrypting a single ciphertext with two different keys! For example, you can send a single encrypted email to Alice and Bob which, upon decryption, reads "attack" to Alice and "retreat" to Bob. The AWS Encryption SDKs are vulnerable to this attack because they allow a single ciphertext to be generated for multiple recipients, with each decrypting using a different key. I believe this kind of problem is prevalent. I briefly looked at JWE and I think it is vulnerable.

Mitigations

Amazon has fixed these bugs in release 2.0.0 of the SDKs. A new major version was required because, unfortunately, the fix for the last bug requires a breaking change from earlier versions. All users are recommended to upgrade. More details about Amazon's mitigations can be found in their announcement.

We're collaborating with Shay Gueron on a paper regarding fast committing AEADs.

Vulnerabilities

Information Leakage

The Encrypt API in AWS KMS encrypts plaintext into ciphertext by using a customer master key (CMK). The ciphertext format is undocumented, but it contains metadata that specifies the CMK and the encryption algorithm. I reverse-engineered the format and found the location of the CMK. Externally the CMK is identified by its key ARN, but within a ciphertext it is represented by an internal ID, which remained stable during my testing.

When I replaced the internal ID of a CMK in a ciphertext with the internal ID of another CMK, I found that AWS KMS attempted to decrypt the ciphertext with the new CMK. The encryption failed and the failure event—including the AWS Account ID, the user agent and the IP address of the caller—was logged to Cloud Trail in the account that owned the replacement CMK.

This enables the following attack:

- The attacker creates a CMK that has a key policy that allows access from everyone. This requires no prior knowledge about the victim.
- The attacker intercepts a ciphertext from the victim, and replaces its CMK with their CMK.
- Whenever the victim attempts to decrypt the modified ciphertext, the attacker learns the timing of such actions, the victim's AWS Account ID, user agent, encryption context, and IP address.

This attack requires the victim to have an IAM policy that allows them to access the attacker's CMK. I found that this practice was allowed by the AWS Visual Policy Editor, but I don't know whether it is common.

The AWS Encryption SDKs also succumb to this attack. The SDKs implement envelope encryption: encrypting data with a data encryption key (DEK) and then wrapping the DEK with a CMK using the Encrypt API in AWS KMS. The wrapped DEK is stored as part of the final ciphertext (format is defined here). The attacker can mount this attack by replacing the CMK in the wrapped DEK with their own.


```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "AWSAccount",
        "principalId": "<redacted this is the principal ID of the victim>",
        "accountId": "<redacted - this is the AWS account ID of the victim>"
    },
    "eventTime": "2020-06-21T21:05:04Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "us-west-2",
```

```
    "sourceIPAddress": "<redacted - this is the IP address of the victim>",
    "userAgent": "<redacted - this is the user agent of the victim>",
    "errorCode": "InvalidCiphertextException",
    "requestParameters": {
        // The encryption context might include other data from the victim
        "encryptionContext": {
            "aws-crypto-public-key": "AzfNOGOnNYFmpHspKrAm1L6XtRybONkmkhmB/IriKSA7b2NsV4MEPMph9yX2KTPKWw=="
        },
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
    },
    "responseElements": null,
    "requestID": "aeced8e8-75a2-42c3-96ac-d1fa2a1c5ee6",
    "eventID": "780a0a6e-4ad8-43d4-a426-75d05022f870",
    "readOnly": true,
    "resources": [
        {
            "accountId": "<redacted - this is the account ID of the attacker>",
            "type": "AWS::KMS::Key",
            "ARN": <redacted - this is the key ARN of the attacker>
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "<redacted - this is the account ID of the attacker>",
    "sharedEventID": "033e147c-8a36-42f5-9d6c-9e071eb752b7"
}
```

Figure 1: A failure event logged to the attacker's Cloud Trail when the victim attempted to decrypt a modified ciphertext containing the attacker's CMK.

Ciphertext Forgery

The Decrypt API in AWS KMS doesn't require the caller to specify the CMK. This parameter is required only when the ciphertext was encrypted under an asymmetric CMK. Otherwise, AWS KMS uses the metadata that it adds to the ciphertext blob to determine which CMK was used to encrypt the ciphertext.

This leads to the following attack:

- The attacker creates a CMK that has a key policy that allows access from everyone. This requires no prior knowledge about the victim.
- The attacker generates a ciphertext by calling the Encrypt API with their key.
- The attacker intercepts a ciphertext from the victim, and replaces it entirely with their ciphertext.
- The victim successfully decrypts the ciphertext, as if it was encrypted under their own key. The attacker also learns when this happened, the victim's AWS Account ID, user agent, encryption context, and IP address.

Similar to the information leakage attack, this attack also requires the victim to have an IAM policy that allows them to access the attacker's CMK.

The AWS Encryption SDKs also succumb to this attack. They don't specify the CMK when they call the Decrypt API to unwrap the DEK.

Robustness

The AWS Encryption SDKs allow a single ciphertext to be generated for multiple recipients, with each decrypting using a different key. To that end, it wraps the DEK multiple times, each under a different CMK. The wrapped DEKs can be combined to form a single ciphertext which can be sent to multiple recipients who can use their own credentials to decrypt it. It's reasonable to expect that all recipients should decrypt the ciphertext to an identical plaintext. However, because of the use of AES-GMAC and AES-GCM, it's possible to create a ciphertext that decrypts to two valid yet different plaintexts for two different users. In other words, the AWS Encryption SDKs are not robust.

The encryption of a message under two CMKs can be summarized as follows:

- A DEK is randomly generated, and two wrapped DEKs are produced by calling the Encrypt API using the two CMKs
- A per-message AES-GCM key (K) is derived using HKDF from the DEK, a randomly generated message ID, and a fixed algorithm ID.
- A header is formed from the wrapped DEKs, the encryption context, and other metadata. A header authentication tag is computed on the header using AES-GMAC with K and a zero IV.
- The message is encrypted using AES-GCM with K, a non-zero IV, and fixed associated additional data. This produces a message authentication tag.
- The ciphertext consists of the header, the header authentication tag, the encrypted message, and the message authentication tag.

(There's also a self-signed digital signature that is irrelevant to this discussion).

In order to decrypt a ciphertext, the AWS Encryption SDKs loops over the list of wrapped DEKs and returns the first one that it can successfully unwrap. The attacker therefore can wrap a unique DEK for each recipient. Next, the attacker exploits the non-committing property of GMAC to produce two messages that have the same GMAC tag under two different keys. The attacker has to do this twice, one for the header authentication tag and one for the message authentication tag.

```
Given a data blob B of one 128-bit block B_1, a GMAC tag is computed as follows:

B_1 * H^2 + B_len * H + J

where H and J depends on the key and B_len depends on the length of B.

To find a message that can produce the same tag under two different keys, one
can add append to B a new block B_2 whose value can be deduced by solving
an algebraic equation. That is, we want to find B_2 such that:

B_1 * H^3 + B_2 * H^2 + B_len * H + J = B_1 * H'^3 + B_2 * H'^2 + B_len * H' + J'

where H' and J' are the corresponding H and J of the other key.

B_2 is the only unknown value in this equation, thus it can be computed using
finite field arithmetics of GF(2^128):

B_2 = [B_1 * (H^3+H'^3) + B_len * (H + H') + J + J'] * (H^2 + H'^2)^-1.
```

Figure 2: How to find a message that has the same GMAC tag under two different keys.

The overall attack works as follows:

- The attacker generates a random DEK, derives a per-message key K, and encrypts message M with it using AES in counter mode. This generates a ciphertext C.
- The attacker generates another random DEK', derives a per-message key K', and performs trial decryption of C until the decrypted message M' has desirable properties. For example, if the attacker wants the first bit of M' different from that of M, this process should only take a few attempts.
- The attacker finds a block C* such that the GMAC of C' = C || C* under K and K' are identical. Denote this tag C'_tag.
- The attacker wraps DEK and DEK' under two recipients' CMK.
- The attacker forms a header H and adds a block H* to the encryption context such that the new H' has the same authentication tag H'_tag under K and K'.
- The attacker output H', H'_tag, C', C'_tag.

This attack is similar to the one discovered in Facebook Messenger.

Acknowledgement

I'm grateful to Jen Barnason for carefully editing this advisory. I will never publish anything without her approval! I want to thank my friend and coworker Sophie "Queen of Hashing" Schmieg for wonderful discussions and for showing me how the arithmetic in GF(2^128) works. I want to thank Jonathan Bannet for asking the questions that led to this work.

Severity

Low

CVE ID

CVE-2020-8897

Weaknesses

No CWEs

Credits

 **thaidn**