**INTERNATIONAL ORGANIZATION FOR STANDARDIZATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 03 MPEG SYSTEMS**

**ISO/IEC JTC 1/SC 29/WG 03 N1508**

**May 2025, Online**

|  |  |
| --- | --- |
| **Title** | **Exploration on De-CENC related issue** |
| **Source** | **WG 03, MPEG Systems** |
| **Status** | **Approved** |
| **Serial Number** | **25044** |

# Introduction

In September 2024, phrack.org published an article titled “[MPEG-CENC: Defective by Specification](https://phrack.org/issues/71/6.html#article)” which outlines an attack, called De-CENC, for compromising the Secure Video Path to enable the capture of decrypted, compressed content without breaching DRM protections. In response to this attack, MPEG is exploring approaches to mitigate it. This document provides a detailed description of the attack, guiding principles and potential solutions, which could ultimately be used to produce a new amendment to the Common Encryption specification. No timeline is yet defined for this work.

# Attack description

Overview of the attack

The core principle exploited by the deCENC attack is to inject new bitstream elements around and into an encrypted “original” bitstream, creating a new “meta bitstream” that wraps the original bitstream. This wrapping prevents or bypasses the decoding of the original bitstream, while still allowing decryption by the Content Decryption Module (CDM). The result is a side channel which enables infering the binary data of the decrypted (wrapped) bitstream when decoding the meta bitstream.

One way to achieve this is to create a meta bitstream that displays the binary data of the wrapped bitstream as a visual pattern on screen, which can then be digitized. This is demonstrated by deCENC using I\_PCM macro blocks in the meta bitstream, embedding the wrapped bitstream as payload data. The I\_PCM macro blocks are specifically crafted to render their payload bytes as grayscale dots when decoded. Other attack variants may construct a meta bitstream which leaks information more subtly, but also at a lower rate than what is achievable with codec constructs like I\_PCM, which can print information verbatim on screen.

The attack can either target directly the decrypted original bitstream, or aim to extract the AES CTR mode keystream of ‘cenc’ content by creating a decryption oracle, and decrypt the original content in a second step. This is possible because in AES CTR mode the ciphertext gets decrypted by XORing it with the keystream, which means the keystream can be derived by XORing the decryption output with the ciphertext.

When aiming at extraction of the AES CTR mode keystream, the original bitstream can be replaced with random values. This approach avoids that valid bitstream elements in the wrapped bitstream interfere with the injected bitstream, and simplifies working around luma clamping in various video formats by generating two variants of the random input (with toggled MSB) for the same keystream block, ensuring one output stays within the clamped luma range.

In contrast, AES CBC mode doesn't have a keystream, so the original encrypted bitstream of ‘cbcs’ encrypted content needs to be directly attacked. To work around the aforementioned decoding challenges, another way of altering the decryption output in a controlled manner is required. In AES CBC mode the IV (or the previous block’s cipher text) is used to XOR the output after decryption, which can be leveraged to modify the decrypted bitstream by using a random IV (thereby randomizing the decryption output). The process can be reversed after rendering the decrypted data on screen, ultimately applying the original IV to arrive at the correct decrypted output.

This means that while the primary principle of the attack doesn’t interfere with decryption itself, it requires modification of IVs and/or encrypted payloads to work around decoding and rendering issues of the injected meta bitstream, when leaking the decrypted wrapped bitstream.

One specificity of the deCENC attack is to leverage CENC subsample encryption to inject the new bitstream into expanded clear ranges. But other attack vectors are available as well, e.g. sample groups with unencrypted samples, which are used to implement the “clear lead” concept and certain ad-insertion use cases.

Applicability to:

* Any codec
* Any cipher mode: CTR, CBC
* Any DRM system
* Any protection scheme defined in MPEG-CENC

Focus on some parts of the attack and some extensions:

* use of subsample encryption
* use of I\_PCM block
* use of clear lead

# Guiding principles

This section outlines proposed guiding principles for designing a solution. The solution should:

* Protect any encoded video stream, in any format, no matter what coding tool is used, including tools like I\_PCM.
* Consider backwards and non-backwards approaches
* Be optimized for the content pipeline
* Minimize the additional load on end-user devices
* Consider legitimate use cases for the manipulation of encrypted data (e.g. HEVC tile composition, …)

# Considerations for a solution

This section lists elements of a solution with their pros and cons. They include use of existing ISOBMFF structures as well as potential new structure definitions, discuss compatible approaches as well as non-backwards compatible approaches.

## Authenticated Encryption applied to CENC

* + 1. Approach 1: Replace the current encryption algorithms (AES-CTR, AES-CTR) with an Authenticated Encryption algorithm. The unencrypted data can be hashed using SHA1 or SHA256 and then provided to the algorithm as Additional Associated Data (AAD). Algorithms: AES-GCM, AES-GCM-SIV, ChaCha20-Poly1305.
       1. Like AES-CTR, AES-GCM is not secure if the nonce (IV) is reused, where AES-GCM-SIV does not have the issue. Thus the nonce (IV) needs to be unique for AES-GCM. The AES-GCM runs much faster than AES-GCM-SIV.
       2. ChaCha20-Poly1305 is faster than AES-GCM in pure software implementations, but AES-GCM can take advantage of hardware acceleration (AES specific instruction), which is widely available.
       3. Winner: AES-GCM, but need to make sure that the nonce (IV) is unique.
    2. Approach 2: Keep the existing encryption algorithms (AES-CTR for cenc, AES-CBC for cbcs), and perform authentication after encryption. Algorithms: HMAC-SHA1, HMAC-SHA256, Poly1305.
       1. HMAC-SHA1 has better performance than HMAC-SHA256, but HMAC-SHA256 has better collision resistance.
       2. Poly1305 is fastest, but it is not as widely available as HMAC. Note that Poly1305 requires a unique nonce for every message.
    3. Compared to approach 1, approach 2 is backward compatible and is more flexible.
       1. Allows the encrypted content to be processed and the authentication tag to be re-generated without the need to decrypt the content first.
       2. The authentication key can be different to the encryption key, and can be shared with different entities.
       3. Compatible with the existing encryption schemes and the existing implementations, and also easier to implement on top of the existing implementations.
       4. Allows license providers to choose whether to enforce the authentication on the server side on a per content basis.
    4. Rough performance comparison measured on a Google Cloud VM, assuming average sample size of 26KB.

| **Algorithms** | **MB/s** |
| --- | --- |
| *AES-CTR* | *752* |
| *AES-CBC* | *924* |
| AES-GCM | 414 |
| AES-GCM-SIV | 96 |
| ChaCha20-Poly1305 | 263 |
| AES-CBC + HMAC-SHA256 | 581 |
| AES-CTR + HMAC-SHA256 | 508 |
| AES-CBC + Poly1305 | 647 |
| AES-CTR + Poly1305 | 558 |
| *AES-CBC 1:9* | *9240* |
| AES-GCM 1:9 sha256 | 1227 |
| AES-GCM 1:9 poly1305 | 1521 |
| AES-CBC 1:9 + HMAC-SHA256 | 1342 |
| AES-CBC 1:9 + Poly1305 | 1754 |

## Protecting initialization vectors

* + 1. As mentioned in the attack section, the IV can be manipulated to craft attack controlled bitstreams if not protected, so the IV must be either authenticated or included in the derivation of the authentication key, e.g. if Poly1305 is chosen as the authentication algorithm.
    2. How can the IV be authenticated:
       1. AES-GCM: aad = SHA256(IV + …), output..auth\_tag = AES-GCM(...)
       2. HMAC: auth\_tag = HMAC(IV + …)
       3. Poly1305: poly1305\_key = HMAC(IV), auth\_tag = Poly1305(...)

## Protecting Sample Auxiliary Information or Sample Data

* + 1. Protecting Sample Auxiliary Information alone (IV + number of bytes in subsamples) is not sufficient
       1. There is a high possibility that attackers can craft I\_PCM bitstreams matching one of the authenticated ranges.
    2. How about protecting Sample Auxiliary Information (IV + number of bytes in subsamples) and all the encrypted bytes?
       1. It is still possible to craft I\_PCM bitstreams by manipulating the ClearData alone with some range values.
       2. There are other primitives that can be used to leak data.
       3. Need further investigations to understand the risk.
    3. Protecting all the data including the unencrypted data in ClearData and SkipByteBlock
       1. Most secure but with the worst performance regression, >30% as shown above.
       2. There are two variants: (1) Generate one authentication tag per sample; (2) Generate one authentication tag per subsample + one optional authentication tag for the Sample Auxiliary Information.
          1. The first option is more secure but the second option is more flexible.

## Current support in HW/SW

* + 1. HMAC-SHA256 is available in all the current HW / SW.
    2. AES-GCM should be available in most SW - need confirmation.
    3. ChaCha20-Poly1305 was added to MbedTLS, one of the most widely used crypto libraries for Trusted Applications, in 2018. It should be available to use in most TEE OS now - need confirmation.

## Box descriptions

* + 1. Add the authentication tag to CencSampleAuxiliaryDataFormat and the the per sample section of SampleEncryptionBox

aligned(8) class CencSampleAuxiliaryDataFormat

{

if (aux\_info\_type\_parameter==0) {

unsigned int(Per\_Sample\_IV\_Size\*8) InitializationVector;

if (sample\_info\_size > Per\_Sample\_IV\_Size ) {

unsigned int(16) subsample\_count;

{

unsigned int(16) BytesOfClearData;

unsigned int(32) BytesOfProtectedData;

} [subsample\_count ]

**unsigned int(256) auth\_tag;**

}

}...

}

* + - 1. Issues: not backward compatible
         1. Existing implementations, which check the version or flag field, will fail to parse the new format.
         2. Other existing implementations will assume the old format and return with incorrect data (e.g. Chrome), which will result in decoding failures.
    1. Add the authentication tag to the end of encrypted sample in mdat
       1. Pros
          1. No additional mp4 boxes or fields that need to be parsed.
          2. No Decryptor API changes needed to pass in the authentication tags, although the Decryptor still needs a way to know whether the authentication tags are present in the bitstream, but it can be implicit.
       2. Issues: not backward compatible
          1. Existing Decryptor implementations that check **Sample Size == SUM(Subsample Sizes)** will fail.
    2. Add a new box to carry authentication tags

aligned(8) class SampleAuthBox extends FullBox('saut', version, flags)

{

unsigned int(32) sample\_count;

{

unsigned int(256) auth\_tag;

}[ sample\_count ]

}

* + - 1. Pros: backward compatible
         1. Allows contents to be re-packaged without breaking existing clients, if the server chooses not to enforce the authentication on these clients.
         2. The content packaging and the server enforcements are two separate processes.
         3. Content providers can choose to enforce the authentication only when needed on the server side.
    1. Add HMAC to the end of SampleEncryptionBox

aligned(8) class SampleEncryptionBox extends FullBox('senc', version, flags)

{

unsigned int(32) sample\_count;

// IV and subsample info.

{

…

}[ sample\_count ]

// HMACs.

{

unsigned int(256) auth\_tag;

}[ sample\_count ]

}

* + - 1. Pros: backward compatible. See above.