Research Project

Exploration and Implementation of a mini OTT (Over-The-Top) using Encrypted Media Extensions (EME)

Master 1 of Cybersecurity / ISTIC

Written by

Bassirou BADIANE
Ali BA FAQAS
Alan PRADO
Theo LUDEAU

2023/2024

Supervisor

Mr Mohamed SABT

Contents

Introduction

The utilization of the Internet for the provision of streaming video services has experienced a significant surge in recent times. This growth has brought about new challenges in terms of copyright protection and preventing unauthorized distribution, sharing, and modification of digital content. While copyright laws exist to address these issues, monitoring the vast expanse of the internet to prevent illegal activity poses a considerable challenge. Digital Rights Management (DRM) addresses this by putting barriers in place to prevent digital content from being stolen. DRM is the application of technology to regulate and oversee access to copyrighted content. In order to enable the use of HTML5 video across all major web browsers, a standard DRM API named Extended Media Extension (EME) was created by W3C in 2017. EME is an API that allows the client to play encrypted audio and video content without using additional plugins. We begin our project with an in-depth exploration of the EME API, a standard interface for handling encrypted media in web applications. Following this, we implement a mini OTT platform using the Shaka Player for media playback, a JavaScript library for adaptive video streaming. We then delve into the process of packaging a video with the Shaka Packager and subsequently decrypting it with our OTT platform. Finally, we host the video and its Media Presentation Description (MPD) on an HTTPS server and view it using the EME API. This project provides a hands-on approach to understanding and implementing modern video streaming technologies.

1 Short Overview of our mini OTT

The DRM process of our OTT platform can be divided into two distinct components: content encryption in the packager and content decryption by the client. The figure below shows the DRM workflow, both encryption and decryption.

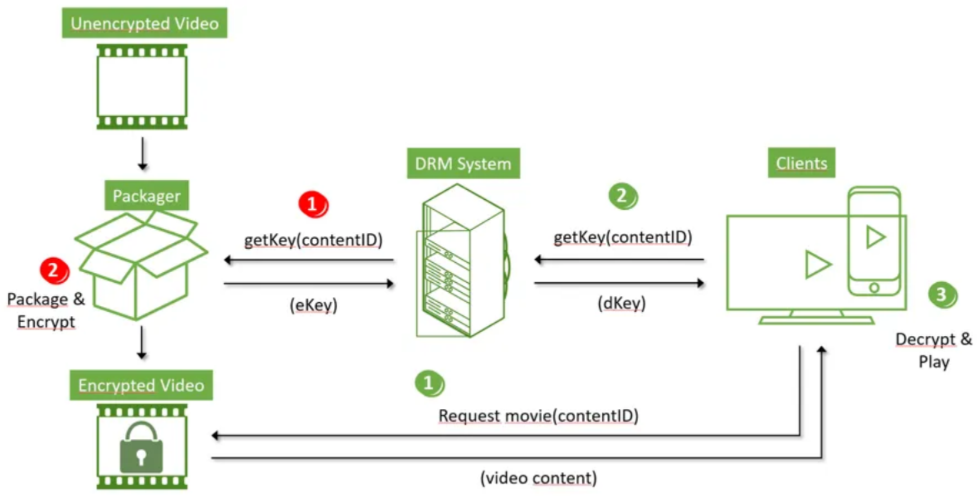


Figure 1: OTT video content distribution chain

Each subsection below provides an overview of primary aspects of the DRM workflow shown by the previous figure.

1.1 Encryption in the Packager

In the process of encrypting content, the packager initiates a request to the DRM system for an encryption key. The DRM system then generates and assigns an encryption key to the specific content ID. Subsequently, the packager utilizes the encryption key to encrypt and repackage the content, ensuring its secure transmission and protection.

1.2 Decryption in the Client

In the playback process, the client initiates a request for the desired content by providing the content URL. Upon receiving the content, the client must decrypt it before it can be played back. To accomplish this, the client sends a request to the DRM system for the decryption key associated with the specific content ID. Once the client obtains the decryption key, it proceeds to decrypt the content and initiate playback.

1.3 DRM System

The DRM system acts as a safeguard for content providers by implementing encryption techniques to protect their intellectual property. It facilitates content encryption, key management, and user authentication processes.
It's worth to note that DRM system works in conjunction with EME API in order to add an additional layer of security, preventing unauthorized access and distribution of digital content.

2 EME external components

The Encrypted Media Extensions (EME) offers an API that allows web applications to engage with content protection systems, facilitating the playback of encrypted audio and video. EME is engineered to ensure that the same application and encrypted files can operate in any browser, irrespective of the specific protection system in place.
EME implementations use external components to work correctly, the figure below shows the external components of EME and the workflow of the protocol.

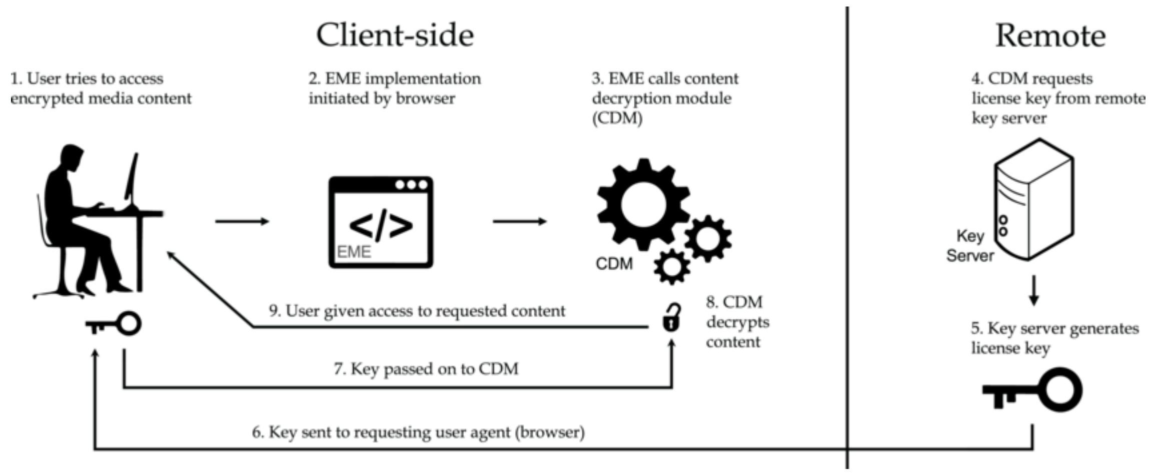

Client-side
Remote
1. User tries to access encrypted media content
2. EME implementation initiated by browser
3. EME calls content decryption module (CDM)
4. CDM requests license key from remote key server
EME
CDM
Key Server
5. Key server generates license key
6. Key sent to requesting user agent (browser)
7. Key passed on to CDM
8. CDM decrypts content
9. User given access to requested content

Figure 2: EME's external components

Among external components that EME API uses, we can list:

2.1 User agent

The implementation of EME resides in the user-agent, such as a web browser, and it is invoked via JavaScript to establish communication with a Content Decryption Module (CDM) and remote servers, utilizing non-transparent messages.

2.2 Content Decryption Module (CDM)

CDM is a system, either software or hardware-based, that resides on the client-side, facilitates the playback of encrypted media. Similar to Key Systems, EME does not specify any Content Decryption Modules (CDMs), but it does offer an interface that allows applications to interact with the available CDMs. There are several CDMs, and we used Google's, specifically Widevine, in this project.

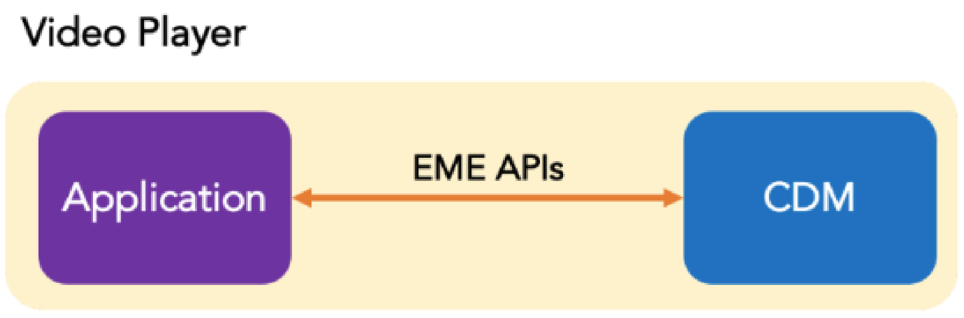


Figure 3: Communication between CDM and browser via EME API

2.3 Content Delivery Network (CDN)

The Content Delivery Network operates as a remote server that supplies encrypted media to the user's device as needed. It also provides essential data for license requests and configurations related to Digital Rights Management (DRM).

2.4 License Server

The license Server interacts with a Content Decryption Module (CDM) to supply keys that decrypt media. The task of negotiating with the license server falls under the purview of the application.

3 Understanding EME API

In this part we will discuss the main methods of EME API.

3.1 CDM selection and configuration negotation

Everything starts from the method *requestMediaKeySystemAccess*, attached to the browser's *navigator* object, is essential for decrypting video content. It initiates the process by querying the available configurations for a specified CDM.

```
requestMediaKeySystemAccess(keySystem, supportedConfigurations)
```

The method requires a *keySystem* string as its initial argument. This string specifies the desired CDM for the session. For example, to select Widevine as the CDM, the *keySystem* string would be *com.widevine.alpha*. Subsequently, the method *requestMediaKeySystemAccess* takes a second argument, which is a configuration object. This object outlines potential settings for decrypting video or audio content. Then the method return a promise to either fulfill with a compatible CDM configuration, if available, or to reject if the requested configuration is not supported on the device in question.

3.2 Creating the media keys and sessions

Having completed the initial step, we've identified a compatible configuration that meets our requirements.

3.2.1 Creating the MediaKeys

To start, we must generate an instance of *MediaKeys*, which serves as a collection of decryption keys that a linked HTMLMediaElement can utilize.

```
createMediaKeys(event.initDataType, event.initData,keySystemAccess);
```

It generates a promise that resolves with the newly created MediaKey. Once we have created the media key, we can associate it with the VideoElement in the following manner:

```
video.setMediaKeys(mediaKeys);
```

By taking this step, we inform the video element to utilize the specified instance of *MediaKeys* for decrypting the media content.

A *MediaKey* represents a CDM instance that contain the *createSession* method which permits to create a session before getting the keys.

3.2.2 Creating session using the MediaKeys

The invocation of the *createSession* method is essential for progressing with the decryption process, enabling the video to become playable within the web browser.

```
const keysSession = mediaKeys.createSession(config[0]['sessionTypes']);
```

The *createSession* method accepts a parameter that determines whether the deciphering session is temporary or persistent. Notably, the server may issue licenses with persistent keys, which the CDM can retain for future use. These keys are securely stored in the device's local storage.
The *createSession* method returns an interface that is utilized to create a challenge, which is subsequently dispatched to the license server.

3.3 Requesting license from the server

The journey of license request begins after creating a session, the interface returned by *createSession* method contains the *generateRequest* method used to generate a challenge that will be sent to the license server.

3.3.1 Asking the CDM to generate a challenge

A challenge consists of unprocessed binary data that encapsulates instructions for content decryption.

```
keysSession.generateRequest(initDataType, initData);
```

The method requires two arguments: the initial specifies the encryption scheme, typically cenc for Common Encryption. The second argument is the initialization data, which refers to the format-specific information that a CDM utilizes to create a license request. To acquire the initialization data, we must monitor the onencrypted event of the video element. This event is triggered, providing the necessary *initDataType* and *initData*, as soon as the first encrypted video segment is loaded into the buffer. This challenge must be conveyed to the license server, which, in response, provides a license embedded with the requisite keys for video content decryption.

3.3.2 Pushing the license to the CDM

After invoking the *generateRequest* method, it is necessary to await the generation of the challenge by the CDM. If everything goes well and the server gives back the license, we need to give the license to the CDM by using the update method keysSession.

```
keySession.update(utf8resp)
```

If the CDM accepts the license, then the video will be all set to go. The browser will start showing the video right on the web page.

4 OTT implementation

The primary goal of our project is to construct our own OTT. However, before we could reach this stage, there were several essential steps that we needed to undertake.

4.1 Media Content Encryption

The initial phase involved encrypting a video using Shaka Packager.
Shaka Packager, developed by Google, is an open-source tool designed for preparing and packaging videos for streaming. It is widely used to create adaptive video streams, providing a smooth streaming experience by automatically adjusting video quality based on device capabilities and available bandwidth.

After installing the packager-linux-x64 version of Shaka Packager from the GitHub repository, we explored the documentation to understand the command options required for encrypting our videos with Widevine, a DRM system. We identified the –enable_widevine_encryption option as essential for this process.

Utilizing information from the documentation, we structured our encryption command, including parameters such as –key_server_url to specify the Widevine key server URL and –content_id to define the content identifier. We also utilized test information provided in the documentation for parameters like –signer, –aes_signing_key, and –aes_signing_iv.

```
./packager-linux-x64 \
in=video.mp4,stream=audio,output=audio.mp4 \ in=video.mp4,stream=video,output=video.mp4 \
 --enable_widevine_encryption \
 --key_server_url https://license.uat.widevine.com/cenc/getcontentkey/widevine_test \
 --content_id 00cdf9d16a11ff027ab6875971023924fc40bd39267827d3e6fc4c277ac76091 \
 --signer widevine_test \
--aes_signing_key 1ae8ccd0e7985cc0b6203a55855a1034afc252980e970ca90e5202689f947ab9 \
--aes_signing_iv d58ce954203b7c9a9a9d467f59839249 \
--mpd_output output.mpd
```

Upon successfully executing the command, we obtained three resulting files: an encrypted video in .mp4 format, an encrypted audio file in .mp4 format, and an MPD (Media Presentation Description) file containing all information about our content.

To verify that our video was properly encrypted, we used a verification command that provided detailed information about the process.

```
./packager-linux-x64 input=video.mp4 --dump_stream_info
I0512 08:57:14.461000    8469 demuxer.cc:94] Demuxer::Run() on file 'video.mp4'.
I0512 08:57:14.469127    8469 demuxer.cc:160] Initialize Demuxer for file 'video.mp4'.
W0512 08:57:14.627034    8469 avc_decoder_configuration_record.cc:94]
Insufficient bits in bitstream for given AVC profile

File "video.mp4":
Found 1 stream(s).
Stream [0] type: Video
 codec_string: avc1.64001f
 time_scale: 90000
 duration: 684006 (7.6 seconds)
 is_encrypted: true
 codec: H264
 width: 640
 height: 360
 pixel_aspect_ratio: 1:1
 trick_play_factor: 0
 nalu_length_size: 4

Packaging completed successfully.
```

After confirming that our video was correctly encrypted, it was crucial to ensure its viewability.

4.2 OTT Utilization via Shaka Player

To ensure that our previously encrypted video is viewable, we used Shaka Player. Shaka Player is an open-source library developed by Google for streaming video playback on the web. Designed to support the latest streaming standards such as MPEG-DASH and HLS, Shaka Player provides a smooth and adaptive playback experience, automatically adjusting video quality based on available bandwidth and device capabilities. By leveraging standard web technologies like HTML5 and JavaScript, Shaka Player allows developers to create custom and integrated video players in their web applications, while also offering advanced features such as DRM support for video content security.

One solution was to use a website already using this library, entering our MPD URL in the provided field to view our video. However, we wanted to create our own test page to ensure that our encrypted videos were viewable via DRM. Hence, we imported the precompiled library into our HTML file and then utilized this tool.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/shaka-player/3.0.0/
shaka-player.compiled.js"></script>
```

Therefore, we declared a Shaka player by specifying the ID of our HTML video element:

```
const player = new shaka.Player(videoPlayer);
```

Next, it was important to configure this player correctly by indicating that we wanted to use Widevine DRM servers with a specific license server:

```
    try {
    player.configure({
        drm: {
            servers: {
                'com.widevine.alpha': licenseServerUrl
            }
        }
    });
}
```

Finally, all that remained was to load the MPD into the player:

```
await player.load(mpdUrl);
videoPlayer.play();
console.log('The␣MPD␣was␣successfully␣loaded.');
```

At this stage, we were able to view our encrypted video using the Shaka Player library. We then needed to succeed in viewing the video with our own player using the EME API.

4.3 OTT Implementation with EME

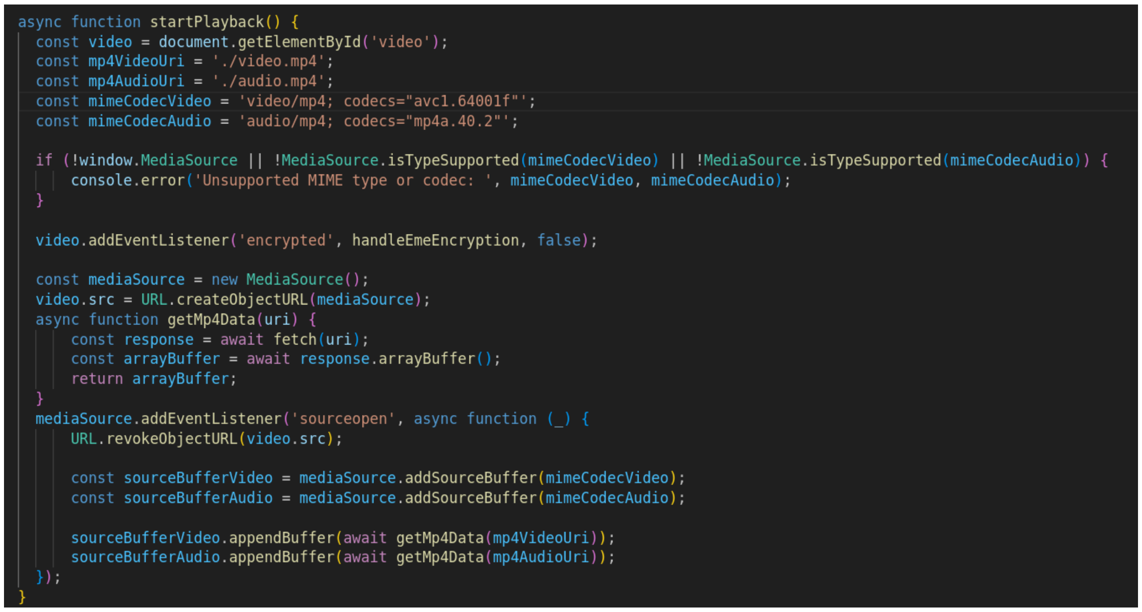
In this part of the project, we have implemented a media streaming solution using the MediaSource API and the Encrypted Media Extensions (EME) API. These APIs provide us with the ability to handle media streams directly in the browser and manage DRM-protected media content.

4.3.1 MediaSource API

We chose the MediaSource API for several reasons:

- Dynamic and Adaptive Streaming: The MediaSource API allows us to adjust the quality of the content in real-time based on network conditions.
- Segmented Downloading: This feature improves performance and allows for playback of long videos without needing to download the entire file upfront.
- Encrypted Content Handling: The MediaSource API provides a way to handle encrypted content through the EME API, which is crucial for DRM-protected media.

The figure below shows the implementation of the playback function using media source API.



```
async function startPlayback() {
  const video = document.getElementById('video');
  const mp4VideoUri = './video.mp4';
  const mp4AudioUri = './audio.mp4';
  const mimeCodecVideo = 'video/mp4; codecs="avc1.64001f"';
  const mimeCodecAudio = 'audio/mp4; codecs="mp4a.40.2"';

  if (!window.MediaSource || !MediaSource.isTypeSupported(mimeCodecVideo) || !MediaSource.isTypeSupported(mimeCodecAudio)) {
    console.error('Unsupported MIME type or codec: ', mimeCodecVideo, mimeCodecAudio);
  }

  video.addEventListener('encrypted', handleEmeEncryption, false);

  const mediaSource = new MediaSource();
  video.src = URL.createObjectURL(mediaSource);
  async function getMp4Data(uri) {
    const response = await fetch(uri);
    const arrayBuffer = await response.arrayBuffer();
    return arrayBuffer;
  }
  mediaSource.addEventListener('sourceopen', async function (_) {
    URL.revokeObjectURL(video.src);

    const sourceBufferVideo = mediaSource.addSourceBuffer(mimeCodecVideo);
    const sourceBufferAudio = mediaSource.addSourceBuffer(mimeCodecAudio);

    sourceBufferVideo.appendBuffer(await getMp4Data(mp4VideoUri));
    sourceBufferAudio.appendBuffer(await getMp4Data(mp4AudioUri));
  });
}
```

Figure 4: Media playback function with media source

The *startPlayback* function orchestrates the playback of a video in a web browser, leveraging the MediaSource API for stream manipulation and ensuring compatibility with the browser's supported MIME types. It attaches an event listener to handle potential encryption via the EME API and initializes a MediaSource object to represent the video source.

4.3.2 Encrypted Media Extensions (EME) API

The EME API is a standard interface for handling DRM-protected media in browsers. We used EME to handle encrypted video content in our project. Here's a brief overview of our implementation:

1. Media Key System Access: We started by requesting access to a Media Key System, 'com.widevine.alpha', which is Google's Widevine DRM system.
2. Creating Media Keys: Once we had access to the Media Key System, we created MediaKeys. These are used to handle the decryption of media data.

3. Creating a Media Key Session: With the MediaKeys created, we then created a MediaKeySession. This represents a context for message exchange with the DRM system.
4. Generating a License Request: We then called generateRequest() on the MediaKeySession to generate a license request. This was sent to the DRM server, which responded with a license (or a challenge for more information).
5. Handling Messages: We added an event listener for 'message' events on the MediaKeySession. These messages can be requests for a license, or they can be messages from the DRM system. We handled these messages by sending them to the license server or by updating the session with new keys.
6. Updating the Session: Once we had a license from the server, we called update() on the MediaKeySession to provide the license to the CDM. This allowed the CDM to decrypt the media.

5 Results & Discussions

The main goal of this project has been achieved. However, some improvements are still needed to fully complete it.

5.1 Achievements

Before we could implement this OTT, we had to encrypt a video using a utility called shaka-packager. Then, we tested the proper functioning of the encryption by exploring shaka-player, an open-source tool that allows the streaming of multimedia content.

5.1.1 Encrypting a video with Shaka Packager

In order to encrypt a video, we use the media packaging tool which takes a video as input, encrypt it, and output it as an encrypted video.mp4. The encryption is enabled for Widevine, with the necessary details provided for the key server and signing.

After encrypting the video, we obtained a Manifest File (MPD) file containing specific information related to the protection of multimedia content, as well as two encrypted files: output.mp4 and video.mp4.

5.1.2 Playing the encrypted video using Shaka player

To ensure that a player could read our video. Two options were available to us: The first option was to visit a site using shaka-player, an open-source JavaScript library developed by Google. By testing this approach, we found that the player was able to play our video by simply providing our MPD file. As for the second option, we decided to integrate the shaka-player library into our website ourselves.

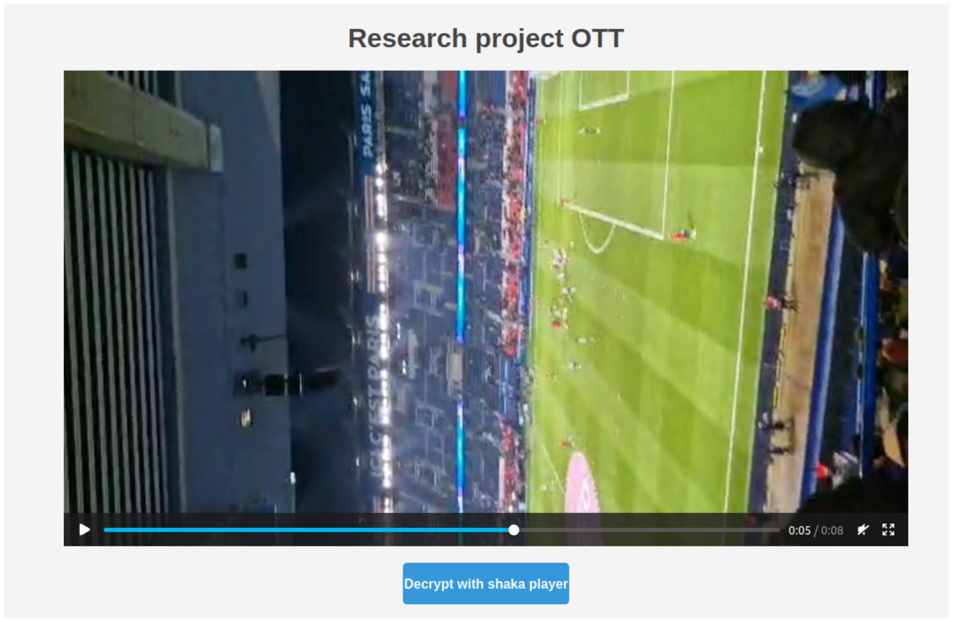


Figure 5: Playing encrypted video with shaka player

After importing the library, configuring the player, and loading the MPD, we were able to play the encrypted video on our own website.

5.1.3 Streaming Encrypted Content via MPD File Provision

In this segment of the project, we have engineered a video player capable of loading and playing encrypted video content. This is achieved through the utilization of a Manifest MPD, which ensures that the video stream is both protected and compliant with the necessary DRM standards.

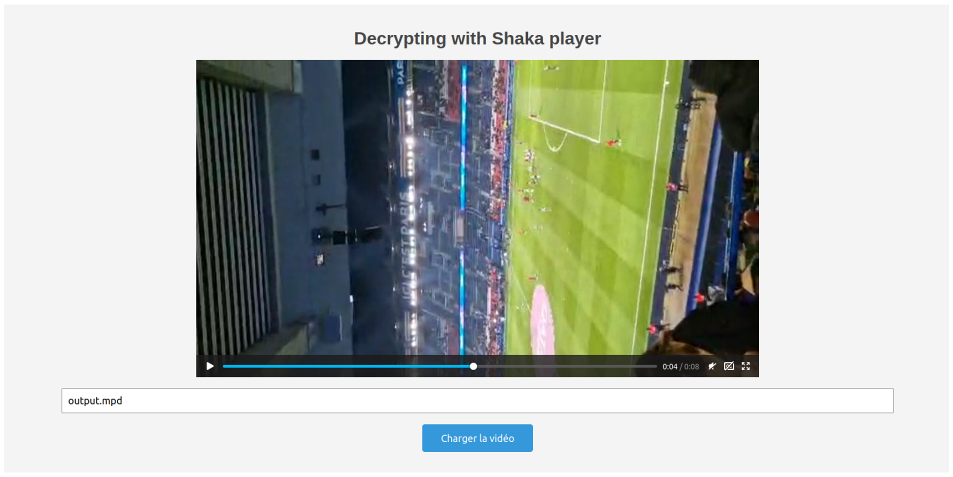


Figure 6: Playing encrypted video of a mpd file

We put the MPD file of the video in the URL and click on the load button to view the video.We have generated two MPD files: the first, *output.mpd*, was created without audio and loads correctly; however, the second, *vision.mpd*, encounters a playback issue where the video freezes after five seconds, although the audio continues to load without any problems. We attempted to manipulate the options of the Shaka Packager command to enable video playback with sound, but it was unsuccessful.

5.2 Areas for Improvement

There are still a few tasks remaining to improve our project. First, we need to enhance our MPD parser to facilitate video playback. Next, we must address the persistent video sound issues. The problem related to the video's sound caused playback to stop. To address this, we attempted to encrypt the video without separating the sound, which allowed us to play the video in its entirety, but unfortunately without sound. We then wondered if this error stemmed from incorrect syntax in the encryption command or an incompatibility with our video's format. We've explored various configurations of the Shaka Packager command to make the sound work, yet none have yielded the desired outcome. We also need to test the remove() function to ensure we understand its operation correctly. We had already studied the behavior of remove but we only had a temporary license.

Link To our OTT

Link to the github project

Conclusion

In conclusion, the expansion of the streaming industry has led to significant investments by major platforms in content production and protective encryption technologies like EME to prevent unauthorized distribution. Our project aligns with these industry standards by creating a mini OTT platform using the EME API for video protection. Our journey began with educating ourselves on DRM and the EME API through research papers and hands-on experiments. Following that, we encrypted a video using shaka-packager and successfully integrated the Shaka Player library into our website to decrypt the video. However, we faced some challenges that we were unable to overcome; we encountered persistent audio issues in the video, despite our attempts to resolve them. Overall, this project has enhanced our understanding of the challenges associated with protecting video content on the web and has acquainted us with the latest tools and technologies in the field.

References

"Your DRM Can Watch You Too: Exploring the Privacy Implications of Browsers (mis)Implementations of Widevine EME" by Gwendal Patat, Mohamed Sabt and Pierre-Alain Fouque.
Encrypted Media Extensions documentation
Encrypted Media Extensions API and Watching Protected Video Content on the Web - Paul Rosset
Shaka packager documentation
Securing OTT Content — DRM by Eyevinn Technology
What is EME? - Sam Dutton