**DRM example: MPEG-DASH server and video player**

In this example we will set up a basic HTTPS server that will serve an adaptive and DRM-protected video stream using MPEG-DASH nd common encryption (CENC). Additionally, we will set up a basic player that relies on W3C standards Encrypted Media Extensions and Google’s Shaka Player.

The complementary reading for Unit 3 contains documents describing the stardard technologies that we will be using, including MPEG-DASH, CENC, MSE and EME.

**Required software**

* ffmpeg [download link](https://www.ffmpeg.org/download.html)
* mp4box from GPAC [download link](https://gpac.wp.imt.fr/downloads/)
* openssl [download link for Windows](https://slproweb.com/products/Win32OpenSSL.html). Linux users can get it through *apt*.
* Python v3.5+ [download link](https://www.python.org/downloads/)
* NodeJS v8+ [download link](https://nodejs.org/en/download/)
* Git v1.9+ [download link](https://git-scm.com/downloads)
* Java Runtime Environment v8+ [download link](https://java.com/en/download/)

**Instructions**

1. Create a new folder called **mpegdash**, we will use this as our working directory.
2. Download the following video by right-clicking on this [link](https://archive.org/download/ElephantsDream/ed_hd.mp4) and selecting *Save link as…* and saving it into our working directory.
3. Download the following font file by right-clicking on this [link](https://www.fontsquirrel.com/fonts/download/source-sans-pro) and selecting *Save link as…* and saving it into our working directory. From the zip file, extract the font file **SourceSansPro-Regular.otf**.

**Generating the video and audio streams**

1. Having in our working directory the files **ed\_hd.mp4** and **SourceSansPro-Regular.otf**, we will generate three video streams at different bitrates using **ffmpeg** from the command line. The video streams will include a tag with the running time and the bitrate.
2. 500k bitrate stream

ffmpeg -i ed\_hd.mp4 -vf "scale=640:-1,drawtext=fontfile=SourceSansPro-Regular.otf:text="500Kb":fontcolor=white:fontsize=24:box=1:boxcolor=black:x=20:y=80,drawtext=fontsize=21:fontfile=SourceSansPro-Regular.otf:timecode='00\:00\:00\:00':rate=24:fontsize=32:fontcolor='white':box=1:boxcolor=black:x=20:y=50" -r 24 -g 50 -codec:v libx264 -s 1280x720 -b:v 500k output\_500k.mp4

1. 1000k bitrate stream

ffmpeg -i ed\_hd.mp4 -vf "scale=640:-1,drawtext=fontfile=SourceSansPro-Regular.otf:text="1000Kb":fontcolor=white:fontsize=24:box=1:boxcolor=black:x=20:y=80,drawtext=fontsize=21:fontfile=SourceSansPro-Regular.otf:timecode='00\:00\:00\:00':rate=24:fontsize=32:fontcolor='white':box=1:boxcolor=black:x=20:y=50" -r 24 -g 50 -codec:v libx264 -s 1280x720 -b:v 1000k output\_1000k.mp4

1. 2000k bitrate stream

ffmpeg -i ed\_hd.mp4 -vf "scale=640:-1,drawtext=fontfile=SourceSansPro-Regular.otf:text="2000Kb":fontcolor=white:fontsize=24:box=1:boxcolor=black:x=20:y=80,drawtext=fontsize=21:fontfile=SourceSansPro-Regular.otf:timecode='00\:00\:00\:00':rate=24:fontsize=32:fontcolor='white':box=1:boxcolor=black:x=20:y=50" -r 24 -g 50 -codec:v libx264 -s 1280x720 -b:v 2000k output\_2000k.mp4

1. Next, we will strip out the audio from the video streams, and generate a standalone audio stream with **ffmpeg**.

ffmpeg -i output\_2000k.mp4 -an -c copy output\_2000k\_v.mp4

ffmpeg -i output\_1000k.mp4 -an -c copy output\_1000k\_v.mp4

ffmpeg -i output\_500k.mp4 -an -c copy output\_500k\_v.mp4

ffmpeg -i output\_2000k.mp4 -map 0:1 -c copy audio\_only.m4a

At this point, our working directory should contain the following files:

1. ed\_hd.mp4
2. SourceSansPro-Regular.otf
3. output\_500k.mp4
4. output\_1000k.mp4
5. output\_2000k.mp4
6. output\_500k\_v.mp4
7. output\_1000k\_v.mp4
8. output\_2000k\_v.mp4
9. audio\_only.m4a

Files 1 through 5 are no longer necessary and can be deleted if you want.

**Encrypting the media streams with MP4Box**

Next, we will encrypt our media streams using Common Encryption in AES128 CTR mode. Check your notes from the *Cryptography* course to remember the Block ciphers’ modes of operation, also here’s the [wiki page](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Counter_(CTR)) on them.

1. Download and install GPAC’s **MP4Box** from their website [GPAC MP4Box](https://gpac.wp.imt.fr/downloads/gpac-nightly-builds/).
2. Create a new file and name it **drm.xml**. Then copy the following text into it:

<GPACDRM type="CENC AES-CTR">

<DRMInfo type="pssh" version="1">

<BS ID128="1077efecc0b24d02ace33c1e52e2fb4b"/>

<BS bits="32" value="1"/>

<BS ID128="cd7eb9ff88f34caeb06185b00024e4c2"/>

</DRMInfo>

<CrypTrack IV\_size="8" first\_IV="0x603f39bfdb3d9799" isEncrypted="1" saiSavedBox="senc" trackID="1">

<key KID="0xcd7eb9ff88f34caeb06185b00024e4c2" value="0x63cb5f7184dd4b689a5c5ff11ee6a328"/>

</CrypTrack>

</GPACDRM>

Details on the structure of this file and the meaning of each value are given in the following [link](https://github.com/gpac/gpac/wiki/Common-Encryption).  
For now, remember that our media streams will be encrypted using AES128 in CBR mode, with an initialization vector 603f39bfdb3d9799 of 8 bytes. The key id is cd7eb9ff88f34caeb06185b00024e4c2 and its value is 63cb5f7184dd4b689a5c5ff11ee6a328.

1. Finally, we use **MP4Box** to encrypt our media streams:

mp4box -crypt drm.xml output\_500k\_v.mp4 -out output\_500k\_v\_enc.mp4

mp4box -crypt drm.xml output\_1000k\_v.mp4 -out output\_1000k\_v\_enc.mp4

mp4box -crypt drm.xml output\_2000k\_v.mp4 -out output\_2000k\_v\_enc.mp4

mp4box -crypt drm.xml audio\_only.m4a -out audio\_only\_enc.m4a

**Packaging the video and audio in DASH format**

The last step is to package the media streams in the MPEG-DASH format, using again **MP4Box**\*. This [link](https://github.com/gpac/gpac/wiki/DASH-Introduction) contains more information about the DASH format and the **MP4Box** command.

MP4Box -dash 5000 -segment-name "output/outputseg-%s" -url-template -bs-switching no -out output.mpd -rap audio\_only\_enc.m4a output\_2000k\_v\_enc.mp4 output\_1000k\_v\_enc.mp4 output\_500k\_v\_enc.mp4

This command will create a folder named **output** in our working directory, which contain segments of the streams. The **output.mpd** manifest file contains information on these segments and how to locate them in our working directory.

**Building our media server**

We will now build a webserver to serve our media content. Since Chrome 50, in order to use EME/MSE with CENC (Common Encryption), all encrypted video and audio fragments need to be served over HTTPS. Our server will therefore have to be set up to use SSL.

1. First, generate the server certificates using **openssl** in your working directory:

openssl req -new -x509 -keyout server.pem -out server.pem -days 365 -nodes

You will have to fill out some information requested by **openssl**. After this, the **server.pem** file will be in our working directory.

1. Next, we will build a basic webserver using Python’s HTTPServer. Create a new file in your working directory and name it [**server.py**](http://server.py/). Copy the following code into the file:

from http.server import HTTPServer, SimpleHTTPRequestHandler

import ssl

class CORSRequestHandler (SimpleHTTPRequestHandler):

def end\_headers (self):

self.send\_header('Access-Control-Allow-Origin', '\*')

SimpleHTTPRequestHandler.end\_headers(self)

httpd = HTTPServer(('localhost', 4443), CORSRequestHandler)

httpd.socket = ssl.wrap\_socket(httpd.socket, certfile='./server.pem', server\_side=True)

httpd.serve\_forever()

**Download and build Google’s Shaka Player**

Now we will download and compile Google’s Shaka player. The Shaka Player is an open-source JavaScript library for adaptive media. It plays adaptive media formats (such as DASH and HLS) in a browser, without using plugins or Flash. Instead, Shaka Player uses the open web standards Media Source Extensions and Encrypted Media Extensions.

1. In your working directory, clone the Shaka Player Github repository:

git clone https://github.com/google/shaka-player.git

cd shaka-player

1. Compile the player:

python build/all.py

You can find additional documentation for building and testing the Shaka Player, as well as documentation for building websites that include the player in [here](https://shaka-player-demo.appspot.com/docs/api/tutorial-welcome.html).

**Building the website**

1. Create a new file in your working directory and call it **index.html**, then copy the following code into it:

<!DOCTYPE html>

<html>

<head>

<!-- Shaka Player compiled library: -->

<script src="./shaka-player/dist/shaka-player.compiled.js"></script>

<!-- Your application source: -->

<script>

var manifestUri = './output.mpd';

function initApp() {

// Install built-in polyfills to patch browser incompatibilities.

shaka.polyfill.installAll();

// Check to see if the browser supports the basic APIs Shaka needs.

if (shaka.Player.isBrowserSupported()) {

// Everything looks good!

initPlayer();

} else {

// This browser does not have the minimum set of APIs we need.

console.error('Browser not supported!');

}

}

function initPlayer() {

// Create a Player instance.

var video = document.getElementById('video');

var player = new shaka.Player(video);

player.configure({

drm:{

clearKeys:{

'cd7eb9ff88f34caeb06185b00024e4c2': '63cb5f7184dd4b689a5c5ff11ee6a328'

}

},

abr: {

enabled: false

}

});

// Attach player to the window to make it easy to access in the JS console.

window.player = player;

// Listen for error events.

player.addEventListener('error', onErrorEvent);

// Try to load a manifest.

// This is an asynchronous process.

player.load(manifestUri).then(function() {

// This runs if the asynchronous load is successful.

console.log('The video has now been loaded!');

initSelector(player);

}).catch(onError); // onError is executed if the asynchronous load fails.

console.log(player.getConfiguration())

}

function initSelector(player) {

var select = document.querySelector('.variantSelector');

options = player.getVariantTracks().map(variant => "<option>" + (variant['videoBandwidth'] / 1024) + "</option>").join('\n');

console.log(options)

select.innerHTML = options;

select.addEventListener('change', (event) => {

player.selectVariantTrack(player.getVariantTracks()[event.target.selectedIndex], true, 3)

})

}

function onErrorEvent(event) {

// Extract the shaka.util.Error object from the event.

onError(event.detail);

}

function onError(error) {

// Log the error.

console.error('Error code', error.code, 'object', error);

}

document.addEventListener('DOMContentLoaded', initApp);

</script>

</head>

<body>

<div>

<label> Select the bitrate: </label>

<select class='variantSelector' name="variantSelector"></select>

<label> Kb </label>

</div>

<video id="video"

height="720"

width="1280"

controls autoplay></video>

</body>

</html>

Note how the variable manifestURI points to the DASH manifest we created in previous steps, and that the key id and the key value are included in player.configure(...).

1. You can also create a second file and call it **indexTest.html**, with the following code:

<!DOCTYPE html>

<html>

<head>

<!-- Shaka Player compiled library: -->

<script src="./shaka-player/dist/shaka-player.compiled.js"></script>

<!-- Your application source: -->

<script>

var player;

function initApp() {

// Install built-in polyfills to patch browser incompatibilities.

shaka.polyfill.installAll();

// Check to see if the browser supports the basic APIs Shaka needs.

if (shaka.Player.isBrowserSupported()) {

// Everything looks good!

player = initPlayer();

} else {

// This browser does not have the minimum set of APIs we need.

console.error('Browser not supported!');

}

var button = document.getElementById('loadButton')

button.addEventListener("click", (event) => {

manifest = document.getElementById('manifest').value

kid = document.getElementById('kid').value

key = document.getElementById('key').value

console.log(manifest)

console.log(kid)

console.log(key)

player = initPlayer()

loadVideo(player, manifest, kid, key)

});

}

function initPlayer() {

// Create a Player instance.

var video = document.getElementById('video');

var player = new shaka.Player(video);

// Attach player to the window to make it easy to access in the JS console.

window.player = player;

// Listen for error events.

player.addEventListener('error', onErrorEvent);

return player

}

function loadVideo(player, manifest, kid, key) {

player.configure("drm.clearKeys." + kid, key)

player.configure('abr.enabled', false)

// Try to load a manifest.

// This is an asynchronous process.

player.load(manifest).then(function() {

// This runs if the asynchronous load is successful.

console.log('The video has now been loaded!');

initSelector(player);

}).catch(onError); // onError is executed if the asynchronous load fails.

console.log(player.getConfiguration())

}

function initSelector(player) {

var select = document.querySelector('.variantSelector');

options = player.getVariantTracks().map(variant => "<option>" + (variant['videoBandwidth'] / 1024) + "</option>").join('\n');

console.log(options)

select.innerHTML = options;

select.addEventListener('change', (event) => {

player.selectVariantTrack(player.getVariantTracks()[event.target.selectedIndex], true, 3)

})

}

function onErrorEvent(event) {

// Extract the shaka.util.Error object from the event.

onError(event.detail);

}

function onError(error) {

// Log the error.

console.error('Error code', error.code, 'object', error);

}

document.addEventListener('DOMContentLoaded', initApp);

</script>

</head>

<body>

<div>

<label> Manifest: </label>

<input type="text" id="manifest" name="manifest">

<label> KID: </label>

<input type="text" id="kid" name="kid">

<label> KEY: </label>

<input type="text" id="key" name="key">

<button type="button" id="loadButton" name="loadButton">Load</button>

</div>

<div>

<label> Select the bitrate: </label>

<select class='variantSelector' name="variantSelector"></select>

<label> Kb </label>

</div>

<video id="video"

height="720"

width="1280"

controls autoplay></video>

</body>

</html>

This second page will allow us to manually introduce the manifest address and the keys to decrypt the video stream.

**Testing it all**

By now, your working directory should *at least* include the following files and directories:

* output/
* shaka-player/
* index.html
* indexTest.html
* output.mpd
* server.pem
* [server.py](http://server.py/)

If using Chrome, you might need to load chrome://flags/#allow-insecure-localhost in the URL bar, click on enable and restart your browser. Since we generated a *self-signed* server certificate, the browser will prevent you from accessing the video site. This will allow insecure connections if they come from your own computer. Once you are done with this tutorial, I recommend you to disable this flag.

1. Start the webserver with the following command:

python server.py

1. In your browser, go to https://localhost:4443. There you should see the video player loading the video and play it. Above the video player. There is a bitrate selector. If you change the bitrate, the player should change the video stream after a couple of seconds.
2. If you go to https://localhost:4443/indexTest.html, you should also see the video player but with no video loaded. In the Manifest text box, enter output.mpd and click on the Load button. The player should now load your video stream, but will be unable to play it since it cannot decrypt it. Enter cd7eb9ff88f34caeb06185b00024e4c2 in the KID field and 63cb5f7184dd4b689a5c5ff11ee6a328 in the KEY field and press Load again. Now, the video should play normally.

**Observations**

We have used the **ClearKey** method for distributing the keys, that is, the key is directly shipped with the content. Other mechanisms allow for keys to be downloaded from key servers after some form of authentication.

Google offers a complete solution for DRM-protected media streaming called **Widevine**. This solution is based on the same technologies that we have used in this demo, but offers additional resources, such as license servers to fetch the decryption keys. Among others, Netflix uses Widevine to protect their video streaming services.