What is EME?

By Sam Dutton

Sam is a Developer Advocate

Encrypted Media Extensions provides an API that enables web applications to interact with content protection systems, to allow playback of encrypted audio and video.

EME is designed to enable the same app and encrypted files to be used in any browser, regardless of the underlying protection system. The former is made possible by the standardized APIs and flow while the latter is made possible by the concept of Common Encryption.

EME is an extension to the HTMLMediaElement specification — hence the name. Being an 'extension' means that browser support for EME is optional: if a browser does not support encrypted media, it will not be able to play encrypted media, but EME is not required for HTML spec compliance. From the EME spec:

> This proposal extends HTMLMediaElement providing APIs to control playback of protected content.
>
> The API supports use cases ranging from simple clear key decryption to high value video (given an appropriate user agent implementation). License/key exchange is controlled by the application, facilitating the development of robust playback applications supporting a range of content decryption and protection technologies.
>
> This specification does not define a content protection or Digital Rights Management system. Rather, it defines a common API that may be used to discover, select and interact with such systems as well as with simpler content encryption systems. Implementation of Digital Rights Management is not required for compliance with this specification: only the Clear Key system is required to be implemented as a common baseline.
>
> The common API supports a simple set of content encryption capabilities, leaving application functions such as authentication and authorization to page authors. This is achieved by requiring content protection system-specific messaging to be mediated by the page rather than assuming out-of-band communication between the encryption system and a license or other server.

EME implementations use the following external components:

- **Key System:** A content protection (DRM) mechanism. EME doesn't define Key Systems themselves, apart from Clear Key (more about that below).
- **Content Decryption Module (CDM):** A client-side software or hardware mechanism that enables playback of encrypted media. As with Key Systems, EME doesn't define any CDMs, but provides an interface for applications to interact with CDMs that are available.
- **License (Key) server:** Interacts with a CDM to provide keys to decrypt media. Negotiation with the license server is the responsibility of the application.
- **Packaging service:** Encodes and encrypts media for distribution/consumption.

Note that an application using EME interacts with a license server to get keys to enable decryption, but user identity and authentication are not part of EME. Retrieval of keys to enable media playback happens after (optionally) authenticating a user. Services such as Netflix must authenticate users within their web application: when a user signs into the application, the application determines the user's identity and privileges.

How does EME work?

Here's how the components of EME interact, corresponding to the code example below:

If multiple formats or codecs are available, MediaSource.isTypeSupported() or HTMLMediaElement.canPlayType() can both be used to select the right one. However, the CDM may only support a subset of what the browser supports for unencrypted content. It's best to negotiate a MediaKeys configuration before selecting a format and codec. If the application waits for the encrypted event but then MediaKeys shows it can't handle the chosen format/codec, it may be too late to switch without interrupting playback.

The recommended flow is to negotiate MediaKeys first, using MediaKeysSystemAccess.getConfiguration() to find out the negotiated configuration.

If there is only one format/codec to choose from, then there's no need for getConfiguration(). However, it's still preferable to set up MediaKeys first. The only reason to wait for the encrypted event is if there is no way of knowing whether the content is encrypted or not, but in practice that's unlikely.

1. A web application attempts to play audio or video that has one or more encrypted streams.
2. The browser recognizes that the media is encrypted (see box below for how that happens) and fires an encrypted event with metadata (initData) obtained from the

media about the encryption.

3. The application handles the encrypted event:
    a. If no MediaKeys object has been associated with the media element, first select an available Key System by using navigator.requestMediaKeySystemAccess() to check what Key Systems are available, then create a MediaKeys object for an available Key System via a MediaKeySystemAccess object. Note that initialization of the MediaKeys object should happen before the first encrypted event. Getting a license server URL is done by the app independently of selecting an available key system. A MediaKeys object represents all the keys available to decrypt the media for an audio or video element. It represents a CDM instance and provides access to the CDM, specifically for creating key sessions, which are used to obtain keys from a license server.
    b. Once the MediaKeys object has been created, assign it to the media element: setMediaKeys() associates the MediaKeys object with an HTMLMediaElement, so that its keys can be used during playback, i.e. during decoding.
4. The app creates a MediaKeySession by calling createSession() on the MediaKeys. This creates a MediaKeySession, which represents the lifetime of a license and its key(s).
5. The app generates a license request by passing the media data obtained in the encrypted handler to the CDM, by calling generateRequest() on the MediaKeySession.
6. The CDM fires a message event: a request to acquire a key from a license server.
7. The MediaKeySession object receives the message event and the application sends a message to the license server (via XHR, for example).
8. The application receives a response from the license server and passes the data to the CDM using the update() method of the MediaKeySession.
9. The CDM decrypts the media using the keys in the license. A valid key may be used, from any session within the MediaKeys associated with the media element. The CDM will access the key and policy, indexed by Key ID.

Media playback resumes.

How does the browser know that media is encrypted?

This information is in the metadata of the media container file, which will be in a format such as ISO BMFF or WebM. For ISO BMFF this means header metadata, called the protection scheme information box. WebM uses the Matroska ContentEncryption element, with some WebM-specific additions. Guidelines are provided for each container in an EME-specific registry.

Note that there may be multiple messages between the CDM and the license server, and all communication in this process is opaque to the browser and application: messages are only understood by the CDM and license server, although the app layer can see what type of message the CDM is sending. The license request contains proof of the CDM's validity (and trust relationship) as well as a key to use when encrypting the content key(s) in the resulting license.

But what do CDMs actually do?

An EME implementation does not in itself provide a way to decrypt media: it simply provides an API for a web application to interact with Content Decryption Modules.

What CDMs actually do is not defined by the EME spec, and a CDM may handle decoding (decompression) of media as well as decryption. From least to most robust, there are several potential options for CDM functionality:

- Decryption only, enabling playback using the normal media pipeline, for example via a <video> element.
- Decryption and decoding, passing video frames to the browser for rendering.
- Decryption and decoding, rendering directly in the hardware (for example, the GPU).

There are multiple ways to make a CDM available to a web app:

- Bundle a CDM with the browser.
- Distribute a CDM separately.
- Build a CDM into the operating system.
- Include a CDM in firmware.
- Embed a CDM in hardware.

How a CDM is made available is not defined by the EME spec, but in all cases the browser is responsible for vetting and exposing the CDM.

EME doesn't mandate a particular Key System; among current desktop and mobile browsers, Chrome supports Widevine and IE11 supports PlayReady.

Getting a key from a license server

In typical commercial use, content will be encrypted and encoded using a packaging service or tool. Once the encrypted media is made available online, a web client can obtain a key (contained within a license) from a license server and use the key to enable decryption and playback of the content.

The following code (adapted from the spec examples) shows how an application can select an appropriate key system and obtain a key from a license server.

```
var video = document.querySelector('video');

var config = [{initDataTypes: ['webm'],
  videoCapabilities: [{contentType: 'video/webm; codecs="vp09.00.10.08"'}]}];

if (!video.mediaKeys) {
  navigator.requestMediaKeySystemAccess('org.w3.clearkey',
      config).then(
    function(keySystemAccess) {
      var promise = keySystemAccess.createMediaKeys();
      promise.catch(
        console.error.bind(console, 'Unable to create MediaKeys')
      );
      promise.then(
        function(createdMediaKeys) {
          return video.setMediaKeys(createdMediaKeys);
        }
      ).catch(
        console.error.bind(console, 'Unable to set MediaKeys')
      );
      promise.then(
        function(createdMediaKeys) {
          var initData = new Uint8Array([...]);
          var keySession = createdMediaKeys.createSession();
          keySession.addEventListener('message', handleMessage,
              false);
          return keySession.generateRequest('webm', initData);
        }
      ).catch(
        console.error.bind(console,
          'Unable to create or initialize key session')
      );
    }
  );
}

function handleMessage(event) {
  var keySession = event.target;
  var license = new Uint8Array([...]);
```

```
  keySession.update(license).catch(
    console.error.bind(console, 'update() failed')
  );
}
```

Common encryption

Common Encryption solutions allow content providers to encrypt and package their content once per container/codec and use it with a variety of Key Systems, CDMs and clients: that is, any CDM that supports Common Encryption. For example, a video packaged using Playready could be played back in a browser using a Widevine CDM obtaining a key from a Widevine license server.

This is in contrast to legacy solutions that would only work with a complete vertical stack, including a single client that often also included an application runtime.

Common Encryption (CENC) is an ISO standard defining a protection scheme for ISO BMFF; a similar concept applies to WebM.

Clear Key

Although EME does not define DRM functionality, the spec currently mandates that all browsers supporting EME must implement Clear Key. Using this system, media can be encrypted with a key and then played back simply by providing that key. Clear Key can be built into the browser: it does not require the use of a separate decryption module.

While not likely to be used for many types of commercial content, Clear Key is fully interoperable across all browsers that support EME. It is also handy for testing EME implementations, and applications using EME, without the need to request a content key from a license server. There is a simple Clear Key example at simpl.info/ck. Below is a walkthrough of the code, which parallels the steps described above, though without license server interaction.

```
// Define a key: hardcoded in this example
// – this corresponds to the key used for encryption
var KEY = new Uint8Array([
  0xeb, 0xdd, 0x62, 0xf1, 0x68, 0x14, 0xd2, 0x7b,
  0x68, 0xef, 0x12, 0x2a, 0xfc, 0xe4, 0xae, 0x3c
]);
```

```
var config = [{
  initDataTypes: ['webm'],
  videoCapabilities: [{
    contentType: 'video/webm; codecs="vp8"'
  }]
}];

var video = document.querySelector('video');
video.addEventListener('encrypted', handleEncrypted, false);

navigator.requestMediaKeySystemAccess('org.w3.clearkey', config).then(
  function(keySystemAccess) {
    return keySystemAccess.createMediaKeys();
  }
).then(
  function(createdMediaKeys) {
    return video.setMediaKeys(createdMediaKeys);
  }
).catch(
  function(error) {
    console.error('Failed to set up MediaKeys', error);
  }
);

function handleEncrypted(event) {
  var session = video.mediaKeys.createSession();
  session.addEventListener('message', handleMessage, false);
  session.generateRequest(event.initDataType, event.initData).catch(
    function(error) {
      console.error('Failed to generate a license request', error);
    }
  );
}

function handleMessage(event) {
  // If you had a license server, you would make an asynchronous XMLHttpRequest
  // with event.message as the body.  The response from the server, as a
  // Uint8Array, would then be passed to session.update().
  // Instead, we will generate the license synchronously on the client, using
  // the hard-coded KEY at the top.
  var license = generateLicense(event.message);

  var session = event.target;
  session.update(license).catch(
    function(error) {
      console.error('Failed to update the session', error);
    }
  );
```

```
}

// Convert Uint8Array into base64 using base64url alphabet, without padding.
function toBase64(u8arr) {
  return btoa(String.fromCharCode.apply(null, u8arr)).
      replace(/\+/g, '-').replace(/\//g, '_').replace(/=*$/, '');
}

// This takes the place of a license server.
// kids is an array of base64-encoded key IDs
// keys is an array of base64-encoded keys
function generateLicense(message) {
  // Parse the clearkey license request.
  var request = JSON.parse(new TextDecoder().decode(message));
  // We only know one key, so there should only be one key ID.
  // A real license server could easily serve multiple keys.
  console.assert(request.kids.length === 1);

  var keyObj = {
    kty: 'oct',
    alg: 'A128KW',
    kid: request.kids[0],
    k: toBase64(KEY)
  };
  return new TextEncoder().encode(JSON.stringify({
    keys: [keyObj]
  }));
}
```

To test this code, you need an encrypted video to play. Encrypting a video for use with Clear Key can be done for WebM as per the webm_crypt instructions. Commercial services are also available (for ISO BMFF/MP4 at least) and other solutions are being developed.

Related technology #1: Media Source Extensions (MSE)

The HTMLMediaElement is a creature of simple beauty.

We can load, decode and play media simply by providing a src URL:

```
<video src='foo.webm'></video>
```

The Media Source API is an extension to HTMLMediaElement enabling more fine- grained control over the source of media, by allowing JavaScript to build streams for playback from

'chunks' of video. This in turn enables techniques such as adaptive streaming and time shifting.

Why is MSE important to EME? Because in addition to distributing protected content, commercial content providers must be able to adapt content delivery to network conditions and other requirements. Netflix, for example, dynamically changes stream bitrate as network conditions change. EME works with playback of media streams provided by an MSE implementation, just as it would with media provided via a src attribute.

How to chunk and play back media encoded at different bitrates? See the DASH section below.

You can see MSE in action at simpl.info/mse; for the purposes of this example, a WebM video is split into five chunks using the File APIs. In a production application, chunks of video would be retrieved via Ajax.

First a SourceBuffer is created:

```
var sourceBuffer = mediaSource.addSourceBuffer('video/webm; codecs="vorbis,
```

The entire movie is then 'streamed' to a video element by appending each chunk using the appendBuffer() method:

```
reader.onload = function (e) {
  sourceBuffer.appendBuffer(new Uint8Array(e.target.result));
  if (i === NUM_CHUNKS - 1) {
    mediaSource.endOfStream();
  } else {
    if (video.paused) {
      // start playing after first chunk is appended
      video.play();
    }
    readChunk_(++i);
  }
};
```

Find out more about MSE in the MSE primer.

Related technology #2: Dynamic Adaptive Streaming over HTTP (DASH)

Multi-device, multi-platform, mobile — whatever you call it, the web is often experienced under conditions of changeable connectivity. Dynamic, adaptive delivery is crucial for coping with bandwidth constraints and variability in the multi-device world.

DASH (aka MPEG-DASH) is designed to enable the best possible media delivery in a flaky world, for both streaming and download. Several other technologies do something similar — such as Apple's HTTP Live Streaming (HLS) and Microsoft's Smooth Streaming — but DASH is the only method of adaptive bitrate streaming via HTTP that is based on an open standard. DASH is already in use by sites such as YouTube.

What does this have to do with EME and MSE? MSE-based DASH implementations can parse a manifest, download segments of video at an appropriate bitrate, and feed them to a video element when it gets hungry — using existing HTTP infrastructure.

In other words, DASH enables commercial content providers to do adaptive streaming of protected content.

DASH does what it says on the tin:

- **Dynamic:** responds to changing conditions.
- **Adaptive:** adapts to provide an appropriate audio or video bitrate.
- **Streaming:** allows for streaming as well as download.
- **HTTP:** enables content delivery with the advantage of HTTP, without the disadvantages of a traditional streaming server.

The BBC has begun providing test streams using DASH:

> The media is encoded a number of times at different bitrates. Each encoding is called a Representation. These are split into a number of Media Segments. The client plays a programme by requesting segments, in order, from a representation over HTTP. Representations can be grouped into Adaptation Sets of representations containing equivalent content. If the client wishes to change bitrate it can pick an alternative from the current adaption set and start requesting segments from that representation. Content is encoded in such a way to make this switching easy for the client to do. In addition to a number of media segments, a representation generally also has an Initialization Segment. This can be thought of as a header, containing information about the encoding, frame sizes, etc. A client needs to obtain this for a given representation before consuming media segments from that representation.

To summarize:

1. Media is encoded at different bitrates.

2. The different bitrate files are made available from an HTTP server.
3. A client web app chooses which bitrate to retrieve and play back with DASH.

As part of the video segmentation process, an XML manifest known as a Media Presentation Description (MPD) is built programmatically. This describes Adaptation Sets and Representations, with durations and URLs. An MPD looks like this:

```
<MPD xmlns="urn:mpeg:DASH:schema:MPD:2011" mediaPresentationDuration="PT0H3
type="static">
  <Period duration="PT0H3M1.63S" start="PT0S">
    <AdaptationSet>
      <ContentComponent contentType="video" id="1" />
      <Representation bandwidth="4190760" codecs="avc1.640028" height="1080" id="
        <BaseURL>car-20120827-89.mp4</BaseURL>
        <SegmentBase indexRange="674-1149">
          <Initialization range="0-673" />
        </SegmentBase>
      </Representation>
      <Representation bandwidth="2073921" codecs="avc1.4d401f" height="720" id="2
        <BaseURL>car-20120827-88.mp4</BaseURL>
        <SegmentBase indexRange="708-1183">
          <Initialization range="0-707" />
        </SegmentBase>
      </Representation>

      …

    </AdaptationSet>
  </Period>
</MPD>
```

(This XML is taken from the .mpd file used for the YouTube DASH demo player.)

According to the DASH spec, an MPD file could in theory be used as the src for a video. However, to give more flexibility to web developers, browser vendors have chosen instead to leave DASH support up to JavaScript libraries using MSE such as dash.js. Implementing DASH in JavaScript allows the adaptation algorithm to evolve without requiring browser updates. Using MSE also allows experimentation with alternative manifest formats and delivery mechanisms without requiring browser changes. Google's Shaka Player implements a DASH client with EME support.

Mozilla Developer Network has instructions on how to use WebM tools and FFmpeg to segment video and build an MPD.

Conclusion

Use of the web to deliver paid-for video and audio is growing at a huge rate. It seems that every new device, whether it's a tablet, game console, connected TV, or set-top box, is able to stream media from the major content providers over HTTP. Over 85% of mobile and desktop browsers now support <video> and <audio>, and Cisco estimates that video will be 80 to 90 percent of global consumer internet traffic by 2017. In this context, browser support for protected content distribution is likely to be become increasingly significant, as browser vendors curtail support for APIs that most media plugins rely on.

Further reading

Specs and standards

EME spec: latest Editor's Draft Common Encryption (CENC) Media Source Extensions: latest Editor's Draft DASH standard (yes, it's a PDF) Overview of the DASH standard

Articles

DTG Webinar (partially obsolete) What is EME?, by Henri Sivonen Media Source Extensions primer MPEG-DASH Test Streams: BBC R&D blog post

Demos

Clear Key demo: simpl.info/ck Media Source Extensions (MSE) demo Google's Shaka Player implements a DASH client with EME support

---

Last updated July 2, 2018.