Hacked in Translation – "Director's Cut" – Full Technical Details (http://blog.checkpoint.com/2017/07/08/hacked-translation-directors-cut-full-technical-details/)

by Omri Herscovici, Omer Gull and Yannay Livneh posted 2017/07/08

(http://blog.checkpoint.com/2017/07/08/hacked-translation-directors-cut-full-technical-details/)

Background

Recently, Check Point researchers revealed (http://blog.checkpoint.com/2017/05/23/hacked-in-translation/) a brand new attack vector – a by subtitles. As discussed in the previous post and in our demo (https://www.youtube.com/watch?v=vYT_EGty_6A), we showed how attacke can use subtitles files to take over users' machines, without being detected.

The attack vector entailed a number of vulnerabilities found in prominent streaming platforms, including VLC, Kodi (XBMC), PopcornTime strem.io.

The potential damage the attacker could inflict is endless, ranging anywhere from stealing sensitive information, installing ransomware, r Denial of Service attacks, and much more.

After our original publication appeared, the vulnerabilities were fixed, which allows us to tell the full tale and share the technical details o attack.

PopcornTime

Developed as an open source project in just a couple of weeks, the multi-platform "Netflix for pirates" integrated the deadly combination bit Torrent client, a video player, and endless scraping capabilities under a very friendly graphical user interface.

(http://blog.checkpoint.com/wp-content/uploads/2017/07/figure-1.jpg)

Figure 1 – PopcornTime GUI

Gaining massive popularity and plenty of attention from mainstream media ([1] (http://www.pcmag.com/article2/0,2817,2454833,00.asp), [(http://www.cbc.ca/news/technology/popcorn-time-is-like-netflix-for-pirates-dan-misener-1.2567929)) for its ease-of-use and vast movie collection, the program was abruptly taken down due to pressure from the Motion Picture Association Of America (https://torrentfreak.com/hollywood-tries-crush-popcorn-time-141219/).

After its discontinuation, the PopcornTime application was forked by various different groups to maintain the program and develop new features. Members of the original PopcornTime project announced that they would endorse the popcorntime.io (that meanwhile turned in popcorntime.sh) project as the successor to the original discontinued Popcorn Time.

The webkit powered interface is packed with movie information and metadata. It presents trailers, plot summaries, cast information, cove photos, IMDB ratings and much more.

Subtitles in PopcornTime

To make the user's life even easier, subtitles are fetched automatically. Can this behavior be exploited? (Hint: Yes)

Behind the scenes, PopcornTime uses open-subtitles (https://www.opensubtitles.org) as their sole subtitle provider. With over 4,000,000 entries and a very convenient API (http://trac.opensubtitles.org/projects/opensubtitles), it is an extremely popular repository.

This API not only allows for easy search and download of subtitles, but it also has a recommendation algorithm to help you find the right fi your movie and release.

Attack Surface

As mentioned earlier, PopcornTime is webkit based, NW.js to be exact.

Previously known as node-webkit, the NW.js platform lets the developer use web technologies such as HTML5, CSS3 and WebGL in his na applications.

Moreover, the Node.js API and 3rd party modules can be directly called from the DOM.

Essentially, an NW.js application is a web page for any matter, all code is written in JavaScript or HTML and styled with CSS. Like any web it may be vulnerable to an XSS attack. In this case, due to the fact that it is running on a node js engine, XSS allows the usage of the server capabilities. In other words, XSS is actually RCE.

Ready… Set… Go!

Our journey begins as soon as the user starts playing a movie.

PopcornTime issues a query using the previously mentioned API and downloads the recommended subtitle (we will dive deeper into that process later on, as it turns out to be a key step in our striving for world domination).

Next, PopcornTime tries to transcode the file into the .srt format:

```
//transcode .ass, .ssa, .txt to SRT
var convert2srt = function (file, ext, callback) {
    var readline = require('readline'),
        counter = null,
        lastBeginTime,

        //input
        orig = /([^\\]+)$/.exec(file)[1],
        origPath = file.substr(0, file.indexOf(orig)),

        //output
        srt = orig.replace(ext, '.srt'),
        srtPath = Settings.tmpLocation,
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-2.png)

Figure 2 – /src/app/vendor/videojshooks.js (https://github.com/popcorn-official/popcorn-desktop/blob/development/src/app/vendor/videojshooks.js#L137)

After various decoding and parsing functions, the created element (a single subtitle) is appended to the display at the right time, using the "cues" array:

```
// Add cue HTML to display
vjs.TextTrack.prototype.updateDisplay = function(){
  var cues = this.activeCues_,
      html = '',
      i=0,j=cues.length;

  for (;i<j;i++) {
    html += '<span class="vjs-tt-cue">'+cues[i].text+'</span>';
  }

  this.el_.innerHTML = html;
};
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-3.png)

Figures 3 – updateDisplay function()

This enables us to add any html object to the view.

Obviously, a complete control over any HTML element is dangerous by itself. However, when dealing with node based applications, it is important to understand that XSS equals RCE.

System commands can be easily executed using modules such as child_process (https://nodejs.org/api/child_process.html).

Once our unsanitized JavaScript is loaded to the display, code execution is just a few lines away.

A basic SRT file looks something like this:

1

00:00:01,000 –> 00:00:05,000

Hello World

Instead of the "Hello World" text, we can use an HTML tag – the image tag.

We try to load an inexistent image and provide it with the onerror (https://www.w3schools.com/jsref/event_onerror.asp) attribute.

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-4.png)

Figure 4 – malicious.srt – example

```
var exec = require("child_process").exec;
exec("calc.exe", function(error, stdout, stderr){});
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-5.png)

Figure 5 – evil.js (Command execution)

As seen in Figure 4, we use the onerror attribute JavaScript capabilities to remove the revealing icon of the broken image and append our malicious remote payload to the page. Needless to say, evil.js (Figure 5) will pop the traditional calc.exe.

OpenSubtitles – The Watering Hole

So we can execute code on PopcornTime.

Client-side vulnerabilities are valuable, but they tend to rely on some user interaction.

For successful exploitation to occur, a link has to be clicked, a pdf must be read, or a site needs to be hacked.

In the case of subtitles, the user needs to load the malicious subtitles. Can we somehow omit this step?

We all know that subtitles are carelessly fetched from open communities around the internet and treated as harmless text files. So after v proved these files can be dangerous, we took a step back and looked at the bigger picture.

With over 4,000,000 entries and an average of 5,000,000 daily downloads, OpenSubtitles is the largest online community for subtitles.

Their extensive API is also widely integrated into many other video players.

They even offer a smart search capability which is a chained function that returns the best matching subtitles based on the information yo provide.

The question remains: Can we manipulate this API to eliminate any user interaction and make sure a malicious subtitle stored on OpenSubtitles is the one automatically downloaded?

API Drill Down

When a user starts playing a movie, a *SearchSubtitles* request is immediately sent, resulting in an XML containing all the subtitle objects t match our criteria (IMDBid).

```
<?xml version="1.0"?>
<methodCall>
  <methodName>SearchSubtitles</methodName>
  <params>
    <param>
      <value>
        <string>UNTcwPbO17BkdC16oOyTTWv3hX5</string>
      </value>
    </param>
    <param>
      <value>
        <array>
          <data>
            <value>
              <struct>
                <member>
                  <name>imdbid</name>
                  <value>
                    <string>2294629</string>
                  </value>
                </member>
                <member>
                  <name>sublanguageid</name>
                  <value>
                    <string>all</string>
                  </value>
                </member>
              </struct>
            </value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodCall>
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure.png)

```
<struct>
  <member>
    <name>MatchedBy</name>
    <value>
      <string>imdbid</string>
    </value>
  </member>
  <member>
    <name>IDSubtitleFile</name>
    <value>
      <string>1954993323</string>
    </value>
  </member>
  <member>
    <name>SubFileName</name>
    <value>
      <string>Frozen (2013).srt</string>
    </value>
  </member>
  <member>
    <name>SubSize</name>
    <value>
      <string>80504</string>
    </value>
  </member>
  <member>
    <name>SubHash</name>
    <value>
      <string>2887f6e8a64e52bd29dcd1cd998a0b7e</string>
    </value>
  </member>
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-7-1.png)

Figure 6 – API SearchSubtitles request **Figure 7 – API SearchSubtitles response**

In figure 6, we see the search criteria is "imdbid", and the response in figure 7 contains all subtitles matched by imdbid.

Now comes the interesting part, as the API has an algorithm that ranks subtitles based on their filename, IMDBid, uploader rank, etc.

Skimming through the documentation, we discovered this ranking scheme:

```
matched by 'hash' and uploaded by:
    + admin|trusted     12
    + platinum|gold     11
    + user|anon         8

matched by tag and uploaded by:
    + admin|trusted     11
    + platinum|gold     10
    + user|anon         7

matched by imdb and uploaded by:
    + admin|trusted     9
    + platinum|gold     8
    + user|anon         5

matched by other and uploaded by:
    + admin|trusted     4
    + platinum|gold     3
    + user|anon         0

bonus of fps matching if:
    + nothing matches   2
    + imdb matches      0.5
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-8.png)

Figure 8 – API's ranking method documentation

In figure 8, we see how many points are added to the subtitles ranking, based on the matching criteria, such as: tag, IMDBid, uploading us etc.

According to the chart, assuming we (as "user|anon") upload our malicious subtitles to OpenSubtitles, our subtitles will only get 5 points.

But here we learned a valuable lesson: reading the documentation is not enough, as the source code revealed an undocumented behavior

The *matchTags* function:

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-9.png)

Figure 9 – opensubtitles-api ranking algorithm

The request sent by PopcornTime specified only IMDBid (as seen in figure 6), which means that the condition of MatchedBy === 'tag' will fo be false.

This calls the function *matchTags*():

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-10.png)

Figure 10 – matchTags function

The matchTags function breaks down the filename of the movie and the subtitle to tags.

A tag is basically an isolated word or number found in the file name, and these are usually separated by dots (".") and dashes ("-").

The amount of shared tags between the movie file name and the subtitles file name is then divided by the number of movie tags, and mult by a maxScore of 7, which is the maxScore that can be assigned in case of full compatibility between the two filenames.

For example, if the movie file name is "Trolls.2016.BDRip.x264-[YTS.AG].mp4", the tags are the following list:

[Trolls, 2016, BDRip, x264, YTS, AG, mp4]

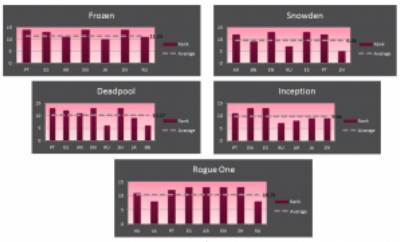
As the name of the movie file name that the application (e.g PopcornTime) is downloading can easily be discovered (by using a sniffer), we make sure our subtitle file has exactly the same name, but ending with the "srt" extension – rewarding the subtitles rank with an extra 7 p (!).

Quick Recap

Putting it all together, we can confidently achieve a score of 12. The match of IMDBid is trivial(+5), and knowing the specific release used b torrent sites and PopcornTime is as easy as opening a packet sniffer. So we can make the malicious subtitles result in full compatibility(+7

This is a fairly good score but we are still not satisfied.

These are the recommended subtitle scores for some of the most popular content available on-line: Snowden, Deadpool, Inception, Rogue A Star Wars Story and Frozen:

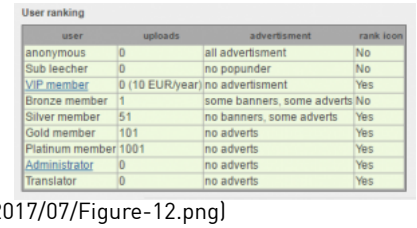


(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-11.png)

Figure 11 – Movies subtitles rank

These graphs (figure 11) show the score for the 7 most popular languages in the world, and display their average and highest score. Skim automatically through a bunch of popular subtitles, we noticed that the highest score a subtitle got is 14, while the average is around 10.

Reviewing the scoring system once more, we realized we can move up in the ranks quite easily.



User ranking

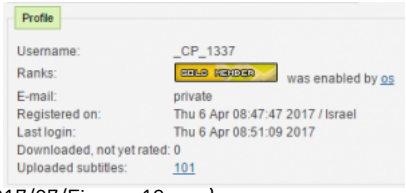
user	uploads	advertisment	rank icon
anonymous	0	all advertisment	No
Sub leecher	0	no popunder	No
VIP member	0 (10 EUR/year)	no advertisment	Yes
Bronze member	1	some banners, some adverts	No
Silver member	51	no banners, some adverts	Yes
Gold member	101	no adverts	Yes
Platinum member	1001	no adverts	Yes
Administrator	0	no adverts	Yes
Translator	0	no adverts	Yes

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-12.png)

Figure 12 – User tanking criteria

Apparently all it takes is 101 subtitle uploads to be a gold member.

So we signed up to OpenSubtitles, and 4 minutes and 40 lines of Python later, we were golden.



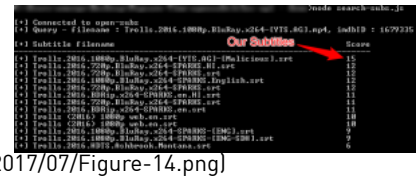
Profile

Username:	_CP_1337
Ranks:	GOLD MEMBER was enabled by os
E-mail:	private
Registered on:	Thu 6 Apr 08:47:47 2017 / Israel
Last login:	Thu 6 Apr 08:51:09 2017
Downloaded, not yet rated:	0
Uploaded subtitles:	101

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-13.png)

Figure 13 – Our new user rank

We wrote a small script that shows all available subtitles for a given movie. In the following image, you can see that our subtitles had the highest score of 15 (!):



(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-14.png)

Figure 14 – Our malicious subtitle is ranked #1

What this basically means is, given any movie, we can force the player to load our crafted malicious subtitles and exploit the machine.

KODI

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-15.png)

KODI, formerly known as XBMC, is an award winning open-source, cross-platform media player and an entertainment hub. Available in al major platforms (Windows, Linux, Mac, iOS and Android), 72 languages, and used by over 40 million people, it is probably the most commc used Media Center software around. KODI is also a popular combination with Smart TVs and Raspberry-Pis making it interesting from the attackers' perspective.

Subtitles in KODI

Like many other KODI features, subtitles are managed by Python plugins.

The most common subtitle plugin is Open-Subtitles, and as we are already familiar with their API, let's dive right in to the subtitles downlo process.

The plugin searches for subtitles using the following function:

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-16.png)

Figure 15 – Search Function

searchsubtitles() retrieves a list of subtitles, including their metadata, from OpenSubtitles.

A for loop iterates over these subtitles and adds them using addDirectoryItem() to the GUI as shown below:

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-16-1.png)

Figure 16 – Subtitles Menu (Indonesian subtitles for "Snowden" movie)

As you can in figure 15, the string sent to addDirectoryItem() is:

plugin://%s/?action=download&link=%s&ID=%s&filename=%s&format=%s

As Open-Subtitles is, well, open, an attacker has control over the filename parameter received under the value of SubFileName as seen b

(http://blog.checkpoint.com/wp-content/uploads/2017/07/figure-17.png)

Figure 17 – API response

Given the fact that the filename is completely controlled by an attacker, it is also possible to overwrite the previous parameters such as lir ID by uploading a file named:

Subtitles.srt&link=<controlled>&ID=<controlled>

Which results in the following string:

plugin://%s/?action=download&link=%s&ID=%s&filename=*Subtitles.srt&link=<controlled>&ID=<controlled>***&format=%s**

This overwrite is possible due to the use of a basic split function when parsing the string.

Both of these tampered parameters are crucial for the function that runs after the user selects one of the options available in the subtitle (as seen in figure 16).

Once the user chooses an item from the subtitles menu, it is sent to Download():

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-18.png)

Figure 18 – Download Function

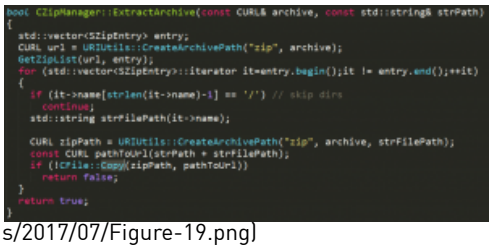
Now that we control all the parameters passed to it, we can abuse its functionality.

By providing an invalid **id** (like "-1"), we reach the "if not result" branch. This branch is supposed to download "raw" archives in case the O Subtitles API fails to fetch the necessary file.

With the **url** parameter at our disposal, we can make it download any zip file that we wish (such as http://attacker.com/evil.zip).

Downloading an arbitrary zip archive from the internet is careless, but chaining this behavior with another vulnerability found in KODI's bu extraction makes it lethal.

Auditing ExtractArchive(), we noticed it concatenates the **strPath**(extraction destination path) to **strFilePath**(the file path inside the archive yielded by the iterator).



(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-19.png)

Figure 19 – ExtractArchive() function

Constructing a zip containing folders named ".." recursively allowed us to control the extraction destination path (CVE-2017-8314).

Using this directory traversal weakness, we overwrote KODI's own subtitle plugin.

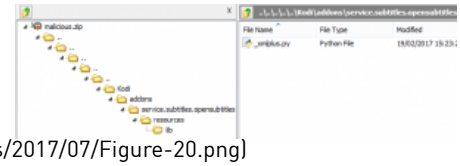

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-20.png)

Figure 20 – Malicious ZIP file structure

Overwriting the plugin means that KODI will soon execute our file. Our malicious Python code can be an exact duplicate of the original plu with the addition of any desired malicious behavior.

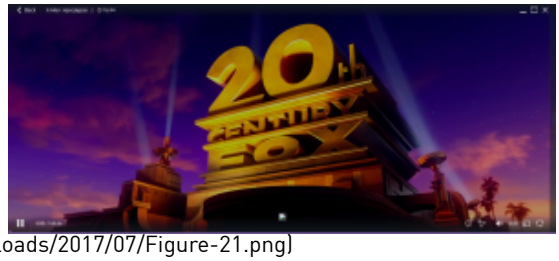
Stremio

PopcornTime definitely marked the rise of streaming apps, but when it was abruptly shut down by the MPAA, users were left looking for alternatives.

Stremio, a semi-open source content aggregator, offered just that. Like PopcornTime, it is designed with ease of use in mind and has a sir user interface. Interestingly enough, Stremio shares a few characteristics with PopcornTime under the hood as well. Most importantly for is a web-kit based application that uses Opensubtitle.org as its subtitle provider.

Strem.IO also adds the subtitles content to the webkit interface, so we assumed XSS would be a good direction here as well.

However, trying the same technique that worked on PopcornTime failed:



(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-21.png)

Figure 21 – Stremio with broken image in the subtitles

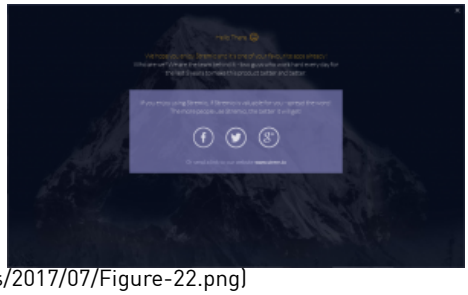
We can see the broken image at the bottom (Figure 21), but no code was executed.

Apparently, our JavaScript has been sanitized. It was time to dig a little deeper.

Stremio code is archived as an ASAR (https://github.com/electron/asar) file, a simple TAR like format that concatenates all files together without the compression. Extracting the source code and prettifying it, we realized that any text added to the screen is passed through An Sanitize (https://docs.angularjs.org/api/ngSanitize/service/$sanitize).

The sanitize service will parse an HTML and only allow safe and white-listed markup and attributes to survive, thus sterilizing a string so contains no scripting expressions or dangerous attributes. Being forced to use only static HTML tags with no scripting capabilities really li our options. The situation called for a creative solution.

If you ever used Stremio, you are probably familiar with their "Support us" pop up banner.



(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-22.png)

Figure 22 – Stremio "support us" image

Using the HTML <img> tag, we were able to present an exact copy of that banner right in the middle of the screen. Wrapping it with an <a h tag meant that clicking the close button redirects this web-kit instance to our unsanitized page:

1

00:00:01,000 –> 00:01:00,000

<img src="http://attacker.com/support.jpg">

That page is exactly the same as the evil.js in the PopcornTime attack, which utilized the nodejs capabilities to execute code on the victim' machine.

VLC – The Obvious Target

Introduction

Once we realized the disastrous potential of subtitles as an attack vector, our next target was obvious. With over 180,000,000 users, VLC is of the most popular media players out there.

This open-source, portable, cross-platform media player\streamer is available for almost any platform imaginable: Windows, OS X, Linux, Windows Phone, Android, Tizen and iOS. It is practically everywhere.

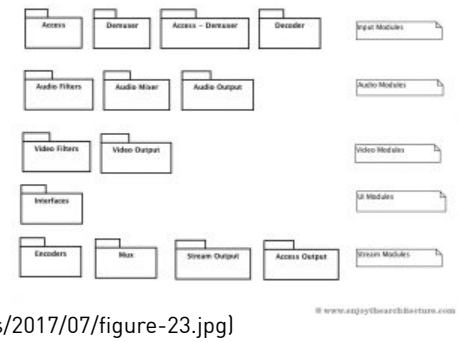
Described by its own authors as a "very popular, but quite large and complex piece of software", we were confident subtitles-related vulnerabilities exist here as well.

Design

VLC is, in fact, a complete multimedia framework (like DirectShow (https://msdn.microsoft.com/en-us/library/windows/desktop/dd375454(v=vs.85).aspx) or GStreamer (https://gstreamer.freedesktop.org/)) where you can load and plug-in modules dynamically.

The core framework does the "wiring" and the media processing, from input (files, network streams) to output (audio or video, on a screen network). It uses modules to do most of the work at every stage (various demuxers (https://wiki.videolan.org/Documentation:Hacker%27s_Guide/Demux/), decoders (https://wiki.videolan.org/Documentation:Hacker%27s_Guide/Decoder/), filters and outputs)

Below is a chart that represents the principal module capabilities implemented in VLC:



(http://blog.checkpoint.com/wp-content/uploads/2017/07/figure-23.jpg)

Figure 23 – VLC modules

Subtitles

Maybe this would be a good time to take a short break from VLC and discuss the complete chaos that is the world of subtitles formats.

During our research we encountered more than 25 (!) subtitle formats. Some are binary, some are textual, and only a few are well docume

It is common knowledge that SRT supports a limited set of HTML tags and attributes, but we were quite surprised to learn about other ex functionalities offered by various formats. SAMI (https://msdn.microsoft.com/en-us/library/ms971327.aspx) subtitles, for example, allows embedded images. SSA (https://en.wikipedia.org/wiki/SubStation_Alpha#Principal_sections_in_an_SSA_subtitle) supports definition of multiple themes\styles and then refers to them from each subtitle. ASS (https://en.wikipedia.org/wiki/SubStation_Alpha#Advanced_SubStation_Alpha) even allows binary font embedding. The list goes on and o

Usually there are no libraries to parse all these formats, which leaves the task to each and every developer. Inevitably, things go wrong.

Back to VLC

Textual subtitles are parsed by VLC in its demuxer called subtitle.c (https://github.com/videolan/vlc/blob/2.2.0-git/modules/demux/subtitl

Below are all the formats it supports and their parsing functions.

```
) sub_read_subtitle_function [] =
{
    { "microdvd",    SUB_TYPE_MICRODVD,    "MicroDVD",     ParseMicroDvd },
    { "subrip",      SUB_TYPE_SUBRIP,      "SubRIP",       ParseSubRip },
    { "subviewer",   SUB_TYPE_SUBVIEWER,   "SubViewer",    ParseSubViewer },
    { "ssa1",        SUB_TYPE_SSA1,        "SSA-1",        ParseSSA },
    { "ssa2-4",      SUB_TYPE_SSA2_4,      "SSA-2/3/4",    ParseSSA },
    { "ass",         SUB_TYPE_ASS,         "SSA/ASS",      ParseSSA },
    { "vplayer",     SUB_TYPE_VPLAYER,     "VPlayer",      ParseVplayer },
    { "sami",        SUB_TYPE_SAMI,        "SAMI",         ParseSami },
    { "dvdsubtitle",SUB_TYPE_DVDSUBTITLE, "DVDSubtitle", ParseDVDSubtitle },
    { "mpl2",        SUB_TYPE_MPL2,        "MPL2",         ParseMPL2 },
    { "aqt",         SUB_TYPE_AQT,         "AQTitle",      ParseAQT },
    { "pjs",         SUB_TYPE_PJS,         "PhoenixSub",   ParsePJS },
    { "mpsub",       SUB_TYPE_MPSUB,       "MPSub",        ParseMPSub },
    { "jacosub",     SUB_TYPE_JACOSUB,     "JacoSub",      ParseJSS },
    { "psb",         SUB_TYPE_PSB,         "PowerDivx",    ParsePSB },
    { "realtext",    SUB_TYPE_RT,          "RealText",     ParseRealText },
    { "dks",         SUB_TYPE_DKS,         "DKS",          ParseDKS },
    { "subviewer1", SUB_TYPE_SUBVIEW1,    "Subviewer 1", ParseSubViewer1 },
    { "text/vtt",    SUB_TYPE_VTT,         "WebVTT",       ParseVTT },
    { NULL,          SUB_TYPE_UNKNOWN,     "Unknown",      NULL }
};
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/figure-24.png)

Figure 24 – subtitle.c array of parsing functions

The demuxers' only job is to parse the different timing conventions of each of the formats and send every subtitle to its decoder. Other tha and ASS that are decoded by the open-source library libass (https://github.com/libass/libass), all these formats are sent to VLC's own dec subsdec.c (https://github.com/videolan/vlc/blob/2.2.0-git/modules/codec/subsdec.c).

subsdec.c parses the text field of every subtitle and creates two version of it. The first is a plain text version with all tags, attributes and st stripped off.

This is used in case later rendering fails.

The second, more feature-rich version is referred to as the HTMLsubtitle. HTML subtitles contain all the fancy styling attributes such as fo alignment etc.

After they are decoded, subtitles are sent to the final stage of rendering. Text rendering is mostly done using the freetype library (https://www.freetype.org/developer.html).

That pretty much sums up the life span of a subtitle from load to display.

Bug Hunting

Going over the VLC subtitle related code, we immediately noticed a lot of parsing is done using raw pointers instead of built-in string funct This is generally a bad idea.

For example, while consuming the possible attributes of a font tag, such as family, size or color, VLC fails to validate the end of the string i some places. The decoder will continue reading from the buffer until a '>' is met, skipping any Null terminator. (CVE-2017-8310)

```
657             else if( !strncasecmp( psz_subtitle, "<font ", 6 ))
658             {
659                 const char *psz_attribs[] = { "face=", "family=", "size=",
660                         "color=", "outline-color=", "shadow-color=",
661                         "outline-level=", "shadow-level=", "back-color=",
662                         "alpha=", NULL };
663
664                 HtmlCopy( &psz_html, &psz_subtitle, "<font " );
665                 HtmlPut( &psz_tag, "f" );
666
667                 while( *psz_subtitle != '>' )
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/figure-25.png)

Figure 25 – subsdec.c CVE-2017-8310

Fuzzing

While auditing the code manually, we also started fuzzing VLC for subtitles related vulnerabilities.

Our weapon of choice was the brilliant AFL (http://lcamtuf.coredump.cx/afl/). This security-oriented fuzzer employs compile-time instrumentation and genetic algorithms to discover new internal states and trigger edge cases in the targeted binary. AFL has already fou countless bugs (http://lcamtuf.coredump.cx/afl/#bugs), and given the right corpus, it is capable of providing very interesting test cases in fairly short time.

For our corpus, we downloaded and rewrote several subtitle files with different functionalities in various formats.

To avoid the rendering and display of the video (our fuzzing server did not have any graphical interface), we used the transcode functionali convert a short movie containing nothing but black screen from one codec to another.

This is the command we used to run AFL:

./afl-fuzz –t 600000 –m 2048 –i input/ -o output/ –S "fuzzer$(date +%s)" -x subtitles.dict — ~/sources/vlc-2.2-afl/bin/vlc-static –q –I dumm subfile

@@ -sout='#transcode{vcodec="x264",soverlay="true"}:standard{access="file",mux="avi",dst="/dev/null"}' ./input.mp4 vlc://quit

The Victim

It didn't take AFL long to find a vulnerable function: ParseJSS. JSS, which stands for JACO Sub Scripts files. JACOsub is a very flexible for allowing for timing manipulations (like shifts), inclusion of external JACOsub files, clock pauses and many other tricks that can be found i full specification (http://unicorn.us.com/jacosub/jscripts.html).

JACO script relies heavily on directives. A directive is a series of character codes strung together. They determine a subtitle's position, fon style, color, and so forth. Directives affect only the single subtitle to which they are prepended.

The crash found by AFL was due to an out-of-bound read while trying to skip unsupported directives (a functionality which is not fully implemented yet) – CVE-2017-8313.

```
1807    /* Parse the directives */
1808    if( isalpha( (unsigned char)*psz_text ) || *psz_text == '[' )
1809    {
1810        while( *psz_text != ' ' )
1811        { psz_text++ ;};
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/figure-26.png)

Figure 26 – Subtitle.c (CVE-2017-8313)

In case a directive is written without any following spaces, this while loop will skip the Null byte terminating *psz_text* over-running the buf Here, and throughout the code, *psz_text* is a pointer to a Null terminated string allocated on the heap.

This drew our attention to the ParseJSS function and we soon manually found another two out-of-bound read issues in the parsing of othe directives. This time, it was the parsing of shift and time directives (cases 'S' and 'T' respectively). This happens due to the fact that the *sh* be greater than the *psz_text* length (CVE-2017-8312).

```
1726            case 'S':
1727                shift = isalpha( (unsigned char)psz_text[2] ) ? 6 : 2 ;
1728
1729                if( sscanf( &psz_text[shift], "%d", &h ) )
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-27-1.png)

```
1763            case 'T':
1764                shift = isalpha( (unsigned char)psz_text[2] ) ? 6 : 2 ;
1765
1766                sscanf( &psz_text[shift], "%d", &p_sys->jss.i_time_resolution );
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-27-2.png)

Figure 27-28 – Subtitle.c (CVE-2017-8312)

The aforementioned VLC vulnerabilities, while enabling attackers to crash the program, weren't sufficient for us. We were after code exec and for that we needed a vulnerability which enables an attacker to write some data. We continued reading the ParseJSS function and loo other directives.

The C[olor] and F[ont] directives granted us some more powerful primitives. Due to a faulty double increment, we were able to skip the delimiting Null byte and write outside the buffer. This heap based overflow allowed us to ultimately execute arbitrary code (CVE-2017-831

```
1865            if( ( toupper((unsigned char)*(psz_text + 1 ) ) == 'C' ) ||
1866                ( toupper((unsigned char)*(psz_text + 1 ) ) == 'F' ) )
1867            {
1868                psz_text++; psz_text++;
1869                break;
1870            }
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/figure-28.png)

Figure 29 – Subtitle.c (CVE-2017-8311)

In another case, VLC **INTENTIONALLY SKIPS THE NULL BYTE** (line 1883)

```
1882        else if( *(psz_text + 1 ) == '\r' || *(psz_text + 1 ) == '\n' ||
1883                 *(psz_text + 1 ) == '\0' )
1884        {
1885            psz_text++;
```

(http://blog.checkpoint.com/wp-content/uploads/2017/07/figure-30.png)

Figure 30 – Subtitle.c (Also CVE-2017-8311)

This behavior resulted in a heap buffer overflow as well.

Exploitation

VLC supports many platforms – OSs and hardware architectures. Each platform may have some different characteristics and heap implementation details that affect the exploitation. From pointer sizes to caching, everything matters.

In our PoC, we decided to exploit Ubuntu 16.04 x86_64. As a modern and popular platform demonstrates, the PoC is applicable to the real world. Having an open-source implementation of the heap lets us explain and understand in great detail the bits of the exploitation proces

There are a (very) few general purpose heap exploitation techniques for GLibC-malloc that survived through the years. However, the condi in which this vulnerability happens prevent us from using any of these methods.

Our only option is to use the vulnerability as a write primitive to overwrite some application specific data. This overwritten data, in turn, wi either lead to stronger primitives (write what where) or complete control over code execution.

VLC is a highly threaded application, and due the implementation of the heap, it means that every thread has its own heap arena. This limi number of objects we may overwrite – only objects that are allocated in the thread that handles subtitles. Also, it's much more likely we ca overflow an object that is allocated in the vicinity of the code used to trigger the vulnerability (or used for Feng Shui; more on that later).

The code running since the creation of our thread and the vulnerable function is not too long. We manually started looking for objects that useful. We came up with *demux_sys_t* and *variable_t*. Also, by automatically tracking every allocated object on the heap, we also found *link_map*, *es_out_id_t* and some *Qt* objects which had virtual tables in them. By process of elimination, we eventually picked *variable_t (http://git.videolan.org/?p=vlc/vlc-2.2.git;a=blob;f=src/misc/variables.h;h=8c203b686749226e31d9b918942d428cf84b688e;hb=6259d80d343c4307b29603b2d88c081c57d6856* object to be the victim.

(http://blog.checkpoint.com/wp-content/uploads/2017/07/figure-31.png)

Figure 31 – structure of variable_t

This object is used for holding variable types of data within the VLC application, including the module's configuration values and commanc options. There are plenty of them all over, which increases our chances of manipulating the heap to have a free slot before one of them. TI *variable_t* struct has a *p_ops* field which holds a pointer to function pointers that operate the value of the variable. Controlling this field er an attacker to gain control over the program.Other objects were either not exploitable or posed too many restrictions.

Now that we have a victim object, we must ensure we can allocate a JACOSubScript (JSS) subtitle before it. This process of manipulating t heap a predictable and useful state is called Heap Feng Shui (a.k.a. Heap-Fu or Grooming).Fortunately, we were quite lucky this time. By r chance, we happen to have a hole right before a victim object, the variable_t for `"sub-fps"`.

(http://blog.checkpoint.com/wp-content/uploads/2017/07/Figure-32.png)

Figure 32 – Memory layout before variable_t

Even though we didn't have to use any other heap shaping primitive, we did find a very promising and interesting code flow which can be o great aid, in case a more subtle design is required. When opening a subtitles file, VLC doesn't know which module to use for parsing the n file. VLC's architecture is very modular, and when parsing a file, it looks at all its modules (libraries), loads them and checks whether they how to parse the given stream (in this case, file). The vulnerable code resides in the subtitle module, but it's not the first module loaded. T modules earlier, the VobSub module is loaded and checks whether the subtitles are of VobSub format. We can trick this module to think o is actually a VobSub file by putting the VobSub magic constant in the first line. Then, this module starts parsing the file, making various allocations and de-allocations. This code runs before allocating the victim object. So this nice VobSub/JSS polyglot can be used for Feng S

The vulnerability enabled us to linearly override data after an allocated subtitle string. This posed a major problem, the variable_t struct's field is psz_name which is assumed to be a pointer to string. This pointer is dereferenced a few times in the life-cycle of VLC. As the Parse function copies strings, we can't write NULL bytes, which are the top two bytes of a valid pointer. Therefore, we can't write valid pointers a must not overflow naively into the variable_t struct. To overcome this problem, we abused the heap's metadata. We used a complex series allocation-overflow-deallocation sequence and overwrote chunks' size metadata ("The poisoned NULL byte, 2014 edition (https://googleprojectzero.blogspot.co.il/2014/08/the-poisoned-nul-byte-2014-edition.html)" style). This enabled us to overwrite the p_op in the variable_t structure without overrding the psz_name field.

Now, we find ourselves facing the eternal question, what should we write? The *p_ops* field is used in the *Destroy* function, when closing V The code invokes the *pf_free* function in the array pointed to by this field and passes the value as a parameter. So we need to put a pointer pointer to our first gadget (actually, 16 bytes before). Our main problem here is ASLR. We don't know where anything is. Welcome to the h world of *scriptless* exploitation.

One way to overcome this problem is partial overwrite. The original pointer points to the *float_ops* static array in the libvlccore library. We partially overwrite this value and make it point somewhere else within this library.

Another viable option is to point to the main binary which, in ubuntu, is not randomized. We found some very interesting gadgets in the ma binary. For example, a gadget that invokes *dlysm* and then invokes the result with another register as first argument (in code: dlsym(-1, $r ($rbx)).

A third way to overcome this problem is to make a partial copy. As our vulnerability copies from beyond a chunk boundary, we can manipul the heap to write a heap pointer in the chunk and then partially copy it.

While these options seem very promising, we didn't follow this road. Scriptless exploitation poses many challenges, and it is too much to investigate for the sake of a demo. Instead, we disabled the ASLR and pointed to our heap. The address of the arena changed a little, most depending on the threading behavior, but it was statistically fine to assume it will be in a certain address. Our next question is, where with arena should we point to? VLC reads the subtitles file line by line and copies each line to chunk on the heap. The low-level line reading mechanism poses a synthetic limit on the line's size of 204800 bytes.

We put our data in the longest allowed line and found out where it is statistically. We built a ROP-chain based libvlccore and put a nice lon in the beginning. Then, we roughly pointed the *p_ops* field to our sled and launched VLC with our subtitles file. Lo and behold, a *gnome-calculator* popped up.

Summary

We showed that by using various vulnerabilities, we could exploit the most popular streaming platforms and take over the victims' machin The vulnerabilities types ranged from simple XSS, through logical bugs, up to memory corruptions.

Being extremely widespread, these media players (and we believe others as well), provide a very vast attack service, potentially affecting hundreds of millions of users.

The main lesson learned is that even overlooked areas, however benign they may seem, can be taken advantage of by attackers looking fo way into your system. We will continue to look for and understand how these breaches can be exploited, and protect users against attacke

©1994-2017 Check Point Software Technologies Ltd. All rights reserved.
Copyright (https://www.checkpoint.com/copyright/) | Privacy Policy (https://www.checkpoint.com/privacy/)