How They Bypass YouTube Video Download Throttling

Posted Aug 14, 2023

By vedard

8 min read

Have you ever tried to download videos from YouTube? I mean manually without relying on software like youtube-dl, yt-dlp or one of “these” websites. It’s much more complicated than you might think.

Youtube generates revenue from user ad views, and it’s logical for the platform to implement restrictions to prevent people from downloading videos or even watching them on an unofficial client like YouTube Vanced. In this article, I will explain the technical details of these security mechanisms and how it’s possible to bypass them.

A google search for: youtube downloader

Extracting the URL

The first step is to find the actual URL containing the video file. For this, we can communicate with the YouTube API. Specifically, the `/youtubei/v1/player` endpoint allows us to retrieve all the details of a video, such as the title, description, thumbnails, and most importantly, the formats. It is within these formats that we can locate the URL of the file based on the desired quality (SD, HD, UHD, etc.).

Here's an example for the video with the ID `aqz-KE-bpKQ`, where we will get the URL for one of the format. Note that the other variables contained within the `context` object are preconditions validated by the API. The accepted values were found by observing the requests sent by the web browser.

Shell

```shell
echo -n '{"videoId":"aqz-KE-bpKQ","context":{"client":{"clientName":"WEB","clientVersion":
  http post 'https://www.youtube.com/youtubei/v1/player' |
  jq -r '.streamingData.adaptiveFormats[0].url'

https://rr1---sn-8qu-t0aee.googlevideo.com/videoplayback?expire=1691828966&ei=hu7WZOCJHI7T
```

However, attempting to download from this URL leads to really slow download:

Shell

```shell
http --print=b --download 'https://rr1---sn-8qu-t0aee.googlevideo.com/videoplayback?expire

Downloading to videoplayback.webm
[ ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ ]   0% ( 0.0/1.5 GB ) 6:23:45 66.7 kB/s
```

The speed is always limited to around 40-70kB/s. Unfortunately for this 10-minute video, it would take approximately 6 and a half hours to download the entire video. Clearly, the video is not downloaded at this speed when using a web browser. So what's different?

Here's the complete URL broken down. It's rather complicated, but there's a specific parameter that interests us.

YAML

```yaml
Protocol: https
Hostname: rr1---sn-8qu-t0aee.googlevideo.com
Path name: /videoplayback
Query Parameters:
  expire: 1691829310
  ei: 3u_WZJT7Cbag_9EPn7mi0A8
  ip: 203.0.113.30
```

```
    id: o-ABGboQn9qMKsUdClvQHd6cHm6l1dWkRw4WNj3V7wBgY1
    itag: 315
    aitags: 133,134,135,136,160,242,243,244,247,278,298,299,302,303,308,315,394,395,396,397,
    source: youtube
    requiressl: yes
    mh: aP
    mm: 31,29
    mn: sn-8qu-t0aee,sn-t0a7ln7d
    ms: au,rdu
    mv: m
    mvi: 1
    pcm2cms: yes
    pl: 18
    initcwndbps: 1422500
    spc: UWF9fzkQbIbHWdKe8-ahg0uWbE_UrbUM0U6LbQfFxg
    vprv: 1
    svpuc: 1
    mime: video/webm
    ns: dn5MLRkBtM4BWwzNNOhVxHIP
    gir: yes
    clen: 1536155487
    dur: 634.566
    lmt: 1662347928284893
    mt: 1691807356
    fvip: 3
    keepalive: yes
    fexp: 24007246,24363392
    c: WEB
    txp: 553C434
    n: mAq3ayrWqdeV_7wbIgP
    sparams: expire,ei,ip,id,aitags,source,requiressl,spc,vprv,svpuc,mime,ns,gir,clen,dur,lm
    sig: AOq0QJ8wRgIhAOx29gNeoiOLRe1GhEfE52PAiXW64ZEWX7nNdAiJE6ezAiEA0Plw6Yn0kmSFFZHO2JZPZyM
    lsparams: mh,mm,mn,ms,mv,mvi,pcm2cms,pl,initcwndbps
    lsig: AG3C_xAwRQIgZVOkDl4rGPGnlK6IGCAXpzxk-cB5RRFmXDesEqOWTRoCIQCzIdPKE6C6_JQVpH6OKMF3wc
```

Since mid-2021, YouTube has included the query parameter `n` in the majority of file URLs. This parameter needs to be transformed using a JavaScript algorithm located in the file `base.js`, which is distributed with the web page. YouTube utilizes this parameter as a challenge to verify that the download originates from an "official" client. If the challenge is not resolved and `n` is not transformed correctly, YouTube will silently apply throttling to the video download.

The JavaScript algorithm is obfuscated and changes frequently, so it's not practical to attempt reverse engineering to understand it. The solution is simply to download the JavaScript file, extract the algorithm code, and execute it by passing the `n` parameter to it. The following code accomplishes this.

JavaScript

```javascript
import axios from 'axios';
import vm from 'vm'

const videoId = 'aqz-KE-bpKQ';

/**
 * From the Youtube API, retrieve metadata about the video (title, video format and audio
 */
async function retrieveMetadata(videoId) {
    const response = await axios.post('https://www.youtube.com/youtubei/v1/player', {
        "videoId": videoId,
        "context": {
            "client": { "clientName": "WEB", "clientVersion": "2.20230810.05.00" }
        }
    });

    const formats = response.data.streamingData.adaptiveFormats;

    return [
        response.data.videoDetails.title,
        formats.filter(w => w.mimeType.startsWith("video/webm"))[0],
        formats.filter(w => w.mimeType.startsWith("audio/webm"))[0],
    ];
}

/**
 * From the Youtube Web Page, retrieve the challenge algorithm for the n query parameter
 */
async function retrieveChallenge(video_id){

    /**
     * Find the URL of the javascript file for the current player version
     */
    async function retrieve_player_url(video_id) {
        let response = await axios.get('https://www.youtube.com/embed/' + video_id);
        let player_hash = /\/s\/player\/(\w+)\/player_ias.vflset\/\w+\/base.js/.exec(respo
        return `https://www.youtube.com/s/player/${player_hash}/player_ias.vflset/en_US/ba
    }

    const player_url = await retrieve_player_url(video_id);

    const response = await axios.get(player_url);
    let challenge_name = /\.get\("n"\)\)&&\(b=([a-zA-Z0-9$]+)(?:\[(\d+)\])?\([a-zA-Z0-9]\)
    challenge_name = new RegExp(`var ${challenge_name}\\s*=\\s*\\[(.+?)\\]\\s*[,;]`).exec(

    const challenge = new RegExp(`${challenge_name}\\s*=\\s*function\\s*\\(([\\w$]+)\\)\\s

    return challenge;
}

/**
 * Solve the challenge and replace the n query parameter from the url
 */
function solveChallenge(challenge, formatUrl) {
    const url = new URL(formatUrl);

    const n = url.searchParams.get("n");
    const n_transformed = vm.runInNewContext(`((a) => {${challenge}})('${n}')`);

    url.searchParams.set("n", n_transformed);
    return url.toString();
}


const [title, video, audio] = await retrieveMetadata(videoId);
const challenge = await retrieveChallenge(videoId);

video.url = solveChallenge(challenge, video.url);
audio.url = solveChallenge(challenge, audio.url);

console.log(video.url);
```

Downloading the media files

With this new URL containing the correctly transformed `n` parameter, the next step is to download the video. However, YouTube still enforces a throttling rule. This rule imposes a variable download speed limit based on the size and length of the video, aiming to provide a download time that's approximately half the duration of the video. This aligns with the streaming nature of videos. It would be a massive waste of bandwidth for YouTube to always provide the media file as quickly as possible.

Shell

```
http --print=b --download 'https://rr1---sn-8qu-t0aee.googlevideo.com/videoplayback?expire

Downloading to videoplayback.webm
[ ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ ]   4% ( 0.1/1.5 GB ) 0:06:07 4.0 MB/s
```

To bypass this limitation, we can break the download into several smaller parts using the HTTP `Range` header. This header allows you to specify which part of the file you want to download with each request (eg: `Range bytes=2000-3000`). The following code implements this logic.

JavaScript

```javascript
/**
 * Download a media file by breaking it into several 10MB segments
 */
async function download(url, length, file){
    const MEGABYTE = 1024 * 1024;

    await fs.promises.rm(file, { force: true });

    let downloadedBytes = 0;

    while (downloadedBytes < length) {
        let nextSegment = downloadedBytes +  10 * MEGABYTE;
        if (nextSegment > length) nextSegment = length;

        // Download segment
        const start = Date.now();
        let response = await axios.get(url, { headers: { "Range": `bytes=${downloadedBytes

        // Write segment
        await fs.promises.writeFile(file, response.data, {flag: 'a'});
        const end = Date.now();

        // Print download stats
        const progress = (nextSegment / length * 100).toFixed(2);
        const total = (length / MEGABYTE).toFixed(2);
        const speed = ((nextSegment - downloadedBytes) / (end - start) * 1000 / MEGABYTE).
        console.log(`${progress}% of ${total}MB at ${speed}MB/s`);

        downloadedBytes = nextSegment + 1;
    }
}
```

This works because the throttling rule takes some time to apply, and the small segments are downloaded very rapidly, always utilizing a new connection.

Shell

```shell
node index.js

```

```
0.68% of 1464.99MB at 46.73MB/s
1.37% of 1464.99MB at 60.98MB/s
2.05% of 1464.99MB at 71.94MB/s
2.73% of 1464.99MB at 70.42MB/s
3.41% of 1464.99MB at 68.49MB/s
4.10% of 1464.99MB at 68.97MB/s
4.78% of 1464.99MB at 74.07MB/s
5.46% of 1464.99MB at 81.97MB/s
6.14% of 1464.99MB at 104.17MB/s
```

We are now able to download videos much faster. During my tests, certain download were close to fully utilizing a 1 Gb/s connection. However, the average speeds typically ranged between 50-70 MB/s or 400-560 Mb/s, which is still pretty fast.

Post-processing

YouTube distributes the video and audio channels in two separate files. This approach helps save space, as an HD or UHD video can reuse the same audio file. Additionally, some videos now offer different audio channels based on the language. Therefore, the final step is to combine these two channels into a single file, and for that, we can simply use `ffmpeg`.

</> JavaScript

```javascript
/**
 * Using ffmpeg, combien the audio and video file into one
 */
async function combineChannels(destinationFile, videoFile, audioFile)
{
    await fs.promises.rm(destinationFile, { force: true });
    child_process.spawnSync('ffmpeg', [
        "-y",
        "-i", videoFile,
        "-i", audioFile,
        "-c", "copy",
        "-map", "0:v:0",
        "-map", "1:a:0",
        destinationFile
    ]);

    await fs.promises.rm(videoFile, { force: true });
    await fs.promises.rm(audioFile, { force: true });
}
```

Finally, for those interested, the full code can be downloaded here.

Conclusion

Many projects currently use these techniques to circumvent the limitations put in place by YouTube in order to prevent video downloads. The most popular one is yt-dlp (a fork of youtube-dl) programmed in Python, but it includes its own custom JavaScript interpreter to transform the `n` parameter.

yt-dlp

https://github.com/ytdl-org/youtube-dl/blob/master/youtube_dl/extractor/youtube.py

VLC media player

https://github.com/videolan/vlc/blob/master/share/lua/playlist/youtube.lua

NewPipe

https://github.com/Theta-Dev/NewPipeExtractor/blob/dev/extractor/src/main/java/org/schabi/newpipe/extractor/services/youtube/YoutubeJavaScriptExtractor.java

node-ytdl-core

https://github.com/fent/node-ytdl-core/blob/master/lib/sig.js

History

youtube nsig vlc youtube-dl yt-dlp

Share:

Further Reading

Aug 6, 2023

How the Nintendo Wii Security Was Defeated

This is the story and the technical details of how the hacker group named Fail0verflow (formerly known as Team Twiizer) discovered and exploited numerous vulnerabilities to defeat the security mech...

Nov 3, 2022

HF Doom

Writeup for the “HF Doom” challenge created by @MaxWhite for the Hackfest CTF 2022. 03 - The Hidden Function This challenge involves finding a hidden function within a WebAssembly version of the ...

Nov 3, 2022

File Type Detective 2.0

Writeup for the “File Type Detective 2.0” challenge created by @dax for the Hackfest CTF 2022. Undercover For this challenge, only one file named unknown is provided. $ file unknown unknown: dat...

OLDER
How the Nintendo Wii Security Was Defeated

NEWER
-