Welcome to my

```
::'########::'##::::::::'######:::'######:::
:: ##.... ##: ##::::::::'##... ##:'##... ##::
:: ##:::: ##: ##::::::::##:::..:::##:::..:::
:: ########:: ##::::::::##:::::::::##:'####:
:: ##.... ##: ##::::::::##:::::::::##::: ##::
:: ##:::: ##: ##::::::::##::: ##: ##::: ##::
:: ########:: ########:. ######::. ######:::
::........::........::.......::::......::::
```

CTF writeups, programming, and miscellaneous stuff.

Blog Index

The Quest for Netflix on Asahi Linux

By David Buchanan, 9th of March 2023

"do not violate the DMCA challenge 2023"

Note: If you're here because you just want to watch Netflix on Asahi, install **this**, and then scroll down to the "Netflix-Specific Annoyances" section.

About 6 months ago, the macOS install on my 1-year-old macbook decided to soft-brick itself.

Rather than wiping and reinstalling, I took the opportunity to switch to Asahi Linux, and I haven't looked back.

Something that bugged me was that I couldn't use the official Spotify app anymore, and similarly, that I couldn't watch Netflix.

Truth be told, I don't care very much about Netflix - the UX offered by BitTorrent is superior. On the other hand, I'm quite invested in the Spotify ecosystem, and while there are 3rd party open-source clients that run great on Asahi, I do prefer the official interface (which is a controversial take, I gather).

While Spotify does not yet offer a native client for aarch64 Linux, their web app is perfectly usable—or at least, it would be, if it didn't show this wonderful error message:

> Playback of protected content is not enabled.

The root cause of this error is that the **Widevine DRM** module is not installed (which is also why Netflix doesn't work).

Thus begins the "do not violate the DMCA challenge 2023". The goal of this challenge is to figure out how to watch Netflix on Asahi Linux *without* bypassing or otherwise breaking DRM.

You may notice that this article is significantly longer than my **280-character publication** on doing the latter, from 2019.

Installing Widevine

Obviously, the solution to all our problems is to install and use Widevine. Unfortunately, one does not simply *Install Widevine* on a platform that isn't officially supported by Google.

The only officially supported way to use Widevine on Linux is using Chrome on an x86_64 CPU.

The astute reader may ask, roughly in this order:

- How come it also works in Chromium and Firefox, on x86_64 Linux?
- How come it also works on Android, which is aarch64 Linux under the hood?
- How come it also works on Raspberry Pi, which is aarch64 Linux?

Wow, you ask excellent questions! I will address them, in order:

Widevine in Firefox on x86_64 Linux?

Webpages access DRM modules through the **Encrypted Media Extensions API**, which is a W3C standard. This specifies how webpages talk to the browser about doing DRM Things™.

In the instance of Chrome, the browser doesn't implement the DRM itself, but delegates it to a native library referred to as a CDM (Content Decryption Module).

In the case of Widevine-in-Chrome-on-Linux, this CDM takes the form of a dynamic library called `libwidevinecdm.so`. This library is an opaque proprietary blob that we are forbidden to look inside of (at least, that's how they'd prefer it to be).

Graciously, as part of the Chromium project, Google provides the **C++ headers** required to interface with The Blob. This interface allows other projects like Firefox to implement support for Widevine, via the EME API, using the exact same `libwidevinecdm.so` blob as Chrome does. This is convenient, but it doesn't help us on aarch64 Linux - there is no corresponding

`libwidevinecdm.so` for us to borrow from Chrome, because Widevine-in-Chrome-on-Linux-on-aarch64 is not an officially supported platform.

Widevine on Android?

The short answer is: the DRM works differently on Android. Because of the differences between the Android platform and desktop Linux, the Android implementations of Widevine will not be useful to us, unless you're planning on losing the Do Not Violate The DMCA Challenge (this is left as an exercise to the reader).

Widevine on Raspberry Pi?

Earlier, I said that "Widevine-in-Chrome-on-Linux-on-aarch64" is not an officially supported platform.

I lied.

Chromebooks exist, many have aarch64 CPUs, they run Chrome on Linux (more or less), and they officially support Widevine. At some point, somebody noticed that there's a perfectly cromulent `libwidevinecdm.so` available inside Google's Chromebook recovery images, and wrote **tools** to download and extract it from the publicly available images. As far as I can tell, the Raspberry Pi folks use scripts like that to obtain the CDM, and helpfully package it in a .deb file for easy installation on a Pi.

However, there's another catch: Although Chromebooks have aarch64 CPUs and kernels, they run an armv7l userspace. This means that the CDM blob is also armv7l. This is fine on Pis because they can also run an armv7l userspace (I think it's even the default still?). Unfortunately, Apple Silicon *cannot* run an armv7l userspace natively, the hardware simply does not support it.

Wait! I lied again!

Or rather, that was all true at the time when I first investigated Widevine-on-Asahi, several months ago. A few weeks ago, Google decided to enter the 21st century and started shipping aarch64 userspaces on certain Chromebook models. This means that "Widevine-in-Chrome-on-Linux-on-aarch64" *does* exist. The ChromeOS blob extraction process works as before, and the Pi Foundation conveniently **packages it as a .deb** for Pi users.

This *is* a viable path forward. Onwards!

Widevine on Arch Linux ARM

The next hurdle is that ChromeOS is not quite the same thing as desktop Linux. ChromeOS's glibc has some weird patches that make it incompatible with the glibc you'll find on a standard Arch Linux ARM box. If you try to load `libwidevinecdm.so`, you'll be hit with a hard-to-debug segfault somewhere in glibc.

The **glibc-widevine** AUR package neatly solves this problem, by patching glibc to work around ChromeOS's idiosyncrasies. As far as I can tell, Raspbian's glibc has similar compatibility patches out-of-the-box.

Since it's not an officially supported platform, the upstream Chromium sources are not configured to support Widevine on aarch64 by default. However, re-enabling that support at build-time is trivial, and thankfully that's what happens in the Arch Linux Arm Chromium package due to a forward-thinking **patch**.

If you haven't looked at the position of the scroll bar on this article, you might think we're almost there! If you have a non-Apple-Silicon aarch64 device (a Linux'd Nintendo Switch?), you can probably just install the **widevine-aarch64** AUR package (which grabs the CDM blob from the Pi repos and sets things up), and you'll have a fully functioning Widevine install.

To recap, the chain of events so far is:

- Google publishes a ChromeOS image with an aarch64 userspace.
- Pi Foundation (presumably) uses a script to extract the Widevine blob from this image and packages it for Raspbian as a .deb.
- Glibc is patched to work around ChromeOS's weirdness.
- Chromium build config is patched to enabled Widevine support on ARM.
- Chrom{e,ium} and Firefox can now use Widevine DRM (yay?)
- Netflix and Spotify work (yay!)

All these steps had already been figured out by various helpful people before I came along. But, this isn't the end of the story for Asahi users...

Widevine on Asahi-flavoured Arch Linux ARM

We're on the home stretch now, right?

Right???

Not quite, there is one last showstopper for Asahi users, and it's a big one: Asahi Linux is built to use **16K page sizes**. The Widevine blobs available to us only support 4K pages. While it is *possible* to run a 4K kernel on Apple Silicon, it's a bit of a bodge right now and it's not something I'm prepared to do on my daily-driver machine, and I assume most other people don't want to either.

This incompatibility is such a significant hurdle that Asahi contributor `chadmed` told me not to bother:

```
07:58 <chadmed> Retr0id: dont bother its LOAD sections are 0x1000 aligned
                (i already tried)
07:58 <Retr0id> I'm trying harder :P
07:59 <chadmed> :D
```

I have a strange affliction whereby I can only solve a problem if somebody else implies that I can't, so this was *exactly* what I needed to hear.

Earlier, I described the CDM as an "opaque proprietary blob that we are forbidden to look inside of". If we did look, this would happen:

live reenactment

How can we overcome this page alignment issue, without being obliterated by the full force of viewing obfuscated code?

We will not gaze directly into the abyss today, but we *will* tiptoe around the edge - both in an effort to preserve our sanity, and to make sure we don't get nerd-sniped into losing the Do Not Violate The DMCA Challenge.

Before we continue, I need to explain what the actual problem is. Before *that*, I need to introduce the Problem Factory:

ELF Loading

ELF stands for Executable and Linkable Format, and it's the de facto format for storing executable code on Linux, and our CDM library (`libwidevinecdm.so`) is no exception.

In general, an ELF is loaded by a loader (wow!), which could be the Linux kernel itself, or a runtime loader like **`ld.so`** . In either case, the ELF headers tell the loader how to load the code (and data) into memory, and prepare it for execution.

`libwidevinecdm.so` is always loaded dynamically at runtime (which is typical for anything plugin-like). Specifically, browsers load it using **`dlopen()`** , which is provided by glibc.

When `dlopen()` is invoked, there are three main phases that occur:

- Loading
- Linking and Relocation
- Execution

The ELF contains a table of Program Headers, each of which describes a Segment of the program. These headers are parsed by the loader, and each `LOAD` segment (a specific segment type) contains metadata to tell the loader how it should be loaded into memory, and which permissions the memory for the Segment should have (read/write/execute). It's the alignment of these `LOAD` Segments that's causing issues with Asahi's 16K pages (more on this later!)

Libraries get loaded at a random base address. The purpose of the linker is to adjust the code and data to account for this arbitrary new location, resolving references within the library, but also potentially references to other libraries (e.g. where one library calls a function from another). The dynamic linker (part of glibc, in this case) parses a bunch of ELF structures to make this happen, including but not limited to the Section Header Table, and the Dynamic Section. These headers (of various types) enumerate Relocations, which tell

the linker specifically where and how the code/data should be adjusted, to make everything work.

Side note: Earlier, chadmed meant to say "LOAD segments", not "LOAD sections". In short, Segments relate to how code is mapped into memory, and Sections relate to how it is linked and relocated. It's an easy mistake to make, and honestly I use them interchangeably when I'm not writing articles like this one. Whoever designed ELF really ought to have picked better names for things. Anyway, moving on...

Although a shared library is not an "executable" per se, it still has an `INIT_ARRAY`, which is an array of function pointers that are called sequentially by the loader, after the ELF has been linked. The library uses these `INIT_ARRAY` entries to do whatever startup initialisation tasks it requires.

After that, the whole loading process is complete, and the "host" program can start calling functions from the newly loaded library.

The Problem

During this process, the loader uses the `mmap()` system call to map subsections of the ELF file into memory at a particular address, as directed by the `LOAD` segments. If we **RTFM** (for mmap), we find out that:

> *offset* must be a multiple of the page size

(`offset` being the offset into the file to start from). And:

> *addr* must be suitably aligned: for most architectures a multiple of the page size is sufficient

(`addr` being the memory address to map it at.)

As a consequence of these limitations, the linker imposes a related **restriction** on `LOAD` segments:

```
case PT_LOAD:
    /* A load command tells us to map in part of the file.
       We record the load commands and process them all later.  */
    if (__glibc_unlikely (((ph->p_vaddr - ph->p_offset)
         & (GLRO(dl_pagesize) - 1)) != 0))
      {
        errstring
    = N_("ELF load command address/offset not page-aligned");
        goto lose;
      }
```

To avoid becoming a `goto loser`, we must ensure that `(vaddr - offset) % pagesize == 0`, where `vaddr` is the Virtual (memory) Address to map a given segment at, and `offset` is the offset within the ELF file where the data to be mapped resides.

Here's a listing of the ELF Program Headers in my copy of `libwidevinecdm.so`:

```
Type            Offset     VAddr      FileSize   MemSize    Align      Prot
PT_PHDR         0x00000040 0x00000040 0x00000230 0x00000230 0x00000008 r--

PT_LOAD         0x00000000 0x00000000 0x00904290 0x00904290 0x00001000 r-x
PT_LOAD         0x00904290 0x00905290 0x00007500 0x00007500 0x00001000 rw-
PT_LOAD         0x0090b790 0x0090d790 0x00000df0 0x00c36698 0x00001000 rw-

PT_TLS          0x00904290 0x00905290 0x00000018 0x00000018 0x00000008 r--
PT_DYNAMIC      0x00909618 0x0090a618 0x00000220 0x00000220 0x00000008 rw-
PT_GNU_RELRO    0x00904290 0x00905290 0x00007500 0x00007d70 0x00000001 r--
PT_GNU_EH_FRAME 0x00524a24 0x00524a24 0x000010fc 0x000010fc 0x00000004 r--
PT_GNU_STACK    0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 rw-
PT_NOTE         0x00000270 0x00000270 0x00000024 0x00000024 0x00000004 r--
```

The three `PT_LOAD` segments are the ones to focus on (I added some whitespace to help).

When `pagesize == 0x1000` (4KB), the constraint holds true, for all load segments in `libwidevinecdm.so`. However, when we increase `pagesize` to `0x4000` (16KB) as it is under Asahi Linux, then it no longer holds true (in the second and third `PT_LOAD` segments).

You may notice that none of the individual fields have any particular alignment properties. This is fine, because there isn't a 1:1 correspondence between segments and calls to `mmap()` - the loader has logic to figure out the actual mappings required.

The Solution

We can't do anything that affects the relative positions of the segments in memory - this would make the CDM very angry, because it would break the relative offsets used to reference data in one segment from another. Furthermore, CDMs generally get pissed off if you make *any* changes to their code that are detectable at runtime. Soothing the CDM's rage could run afoul of the DMCA (we may not "circumvent" any "**technological protection measures**"), and so we must avoid enraging it in the first place.

As a reminder, we need some way to uphold the `(vaddr - offset) % pagesize == 0` constraint. Changing `vaddr` is not an option (because it would affect the runtime memory layout), but we *can* adjust `offset` - i.e. by moving the data around in the ELF file itself.

The first `PT_LOAD` segment is already perfectly aligned, but if we look at the second one we find that 0x00905290 - 0x00904290 = 0x1000, which is no good.

We can fix this by adding 0x1000 bytes of padding into the ELF file, in between where the first and second segments are stored. We adjust the `offset` field accordingly, giving 0x00904290 - 0x00904290 == 0. We do a similar thing to fix the third and final segment.

This is slightly easier said than done, because once you start inserting padding bytes into the ELF file, lots of other offsets in the ELF headers become invalid and need to be adjusted (since anything after the padding-insertion point has moved).

Importantly, these changes only affect the layout of the code *in the ELF file*. Once the code is loaded into memory, the layout should be identical to that of an unpatched binary loaded on a 4K-page system. The CDM runtime only ever gets to inspect the loaded version of itself, so it remains unbothered by any load-time shenanigans.

Permission Granularity

Each page of memory has its own permissions (read/write/execute).

On a system with 4K pages, each 4KB range can have its own permissions. The binary was compiled with this in mind.

However, we've just loaded the same layout onto 16K pages, so we have less granularity

than was anticipated. This causes two interrelated issues:

The `.text` section (which needs to be executable) and the `.data` section (which needs to be writable) end up partially overlapping the same 16KB page. This single page needs to have permissions to handle both usecases, and so we must make its permissions "RWX". We can do this, and it *works*, but it's a bit unfortunate because RWX pages are a significant security risk. I don't see any way around this right now. (Edit: I thought of something! Watch this space...)

Similarly, the "RELRO" security mitigation (read-only relocations) tries to make the GOT read-only after relocations have been applied. Unfortunately, at 16K granularity, this also overlaps with the end of the `.text` section, so we must disable the RELRO mitigation. Again, this is a solution, but it's not very good for security.

To elaborate on the security concerns: If someone is trying to pwn you through a malicious webpage by leveraging a browser vuln, they might start by constructing an "arbitrary read" primitive, followed by an "arbitrary write" primitive. The next thing they might want to do is execute native code. If RWX mappings exist, they can use their arbwrite to write malicious code into that mapping. Overwriting a GOT entry might be a convenient way to redirect execution into this new code (due to lack of RELRO). At that point, they'd either try to escape the browser sandbox, or directly exploit the kernel over any un-sandboxed attack surface.

So, while we're not introducing any new vulnerabilities, we are significantly weakening existing security mitigations. If someone was trying to exploit you, they'd have a much easier time (although they'd still need at least one browser vuln). Realistically, you aren't *that* worried about this in practice, and neither am I. If this does bother you, maybe use a dedicated browser for Netflix'ing?

The CDM *could* detect these page permission inconsistencies and get mad at us, but it has no real incentive to, and in practice it doesn't seem to mind.

Applying the ELF Patches

At first, I tried using **LIEF** to perform these modifications. I'm not sure whether I was just using it wrong, or if I was running into LIEF bugs, but I wasn't able to make this work. It kept on giving me broken ELFs that segfaulted at link-time.

In a caffeine-fueled trance, I whipped out `hexedit` and performed the necessary adjustments by hand (with some bonus Python code to insert the padding bytes). Much to my surprise, my hand-edited ELF actually worked!

Unfortunately, I don't think I'm allowed to distribute my patched ELF, but I didn't want to have to repeat that process for each Widevine update anyway, so I wrote a **Python script** that automates the whole process.

After running the script, we have a functioning `libwidevinecdm.so` that's loadable by Firefox or Chrome on Asahi Linux!

The Finishing Touches

Due to the aforementioned ChromeOS glibc weirdness, one of the hacks required to make it work on other distros involves `LD_PRELOAD` -ing a small library that contains the functions `__aarch64_ldadd4_acq_rel` and `__aarch64_swp4_acq_rel`. The reason for this requirement is not terribly interesting, but I don't like having to `LD_PRELOAD` things unnecessarily, so I devised a way to include these functions within the `libwidevinecdm.so` binary itself.

Recall that we added 0x1000 bytes of padding between the first and second segments. These bytes are still loaded into memory, and they'll end up inside an executable memory page, so we can actually insert a small amount of code. I was worried that doing this would bother the CDM, but it was fine in practice.

So, I put the code for these two functions (all 44 bytes of it!) inside the empty space. Normally, the program expects to be able to call these library functions via the Global Offset Table (GOT), which is effectively a table of function pointers, populated at link-time. The ELF contains a Relocation entry that says "please point this GOT entry at the `__aarch64_ldadd4_acq_rel` symbol" - a symbol that isn't included in any of the libraries normally available on Arch Linux. I patched the Relocation to instead say "please point this GOT entry at [offset of my inserted code]" (and of course, something similar for `__aarch64_swp4_acq_rel`).

The end result is that rather than trying to resolve nonexistent symbols, the linker simply redirects them to the code we added.

The code to do this is also included in my Python script linked above.

After talking to the maintainer of the `widevine-aarch64` AUR package, we added my script to the package so that Asahi users (or anyone else with 16K pages) can have working Widevine out-of-the-box. Furthermore, users on 4K aarch64 platforms can still benefit from my `LD_PRELOAD` avoidance patch (The patcher script does not apply the RWX or no-RELRO hacks in this instance, avoiding any unnecessary security holes).

So now all you have to do is install the `widevine-aarch64` package, and you have a working Widevine installation on Asahi Linux ready to go, without `LD_PRELOAD` hacks!

Netflix-Specific Annoyances

Even after all that, Netflix will refuse to work at all unless you have a ChromeOS useragent. I used this one:

```
Mozilla/5.0 (X11; CrOS aarch64 15236.80.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.
```

Most other services don't require this, Netflix is just being silly here (e.g. Spotify's web player Just Works).

The Widevine version we've installed is known as "L3", the least-secure tier of Widevine you can use. Going higher requires hardware support (and yes, Apple Silicon has relevant hardware for doing "stronger" DRM, but the software/firmware plumbing is not here to make use of it under Linux, which is perhaps for the best).

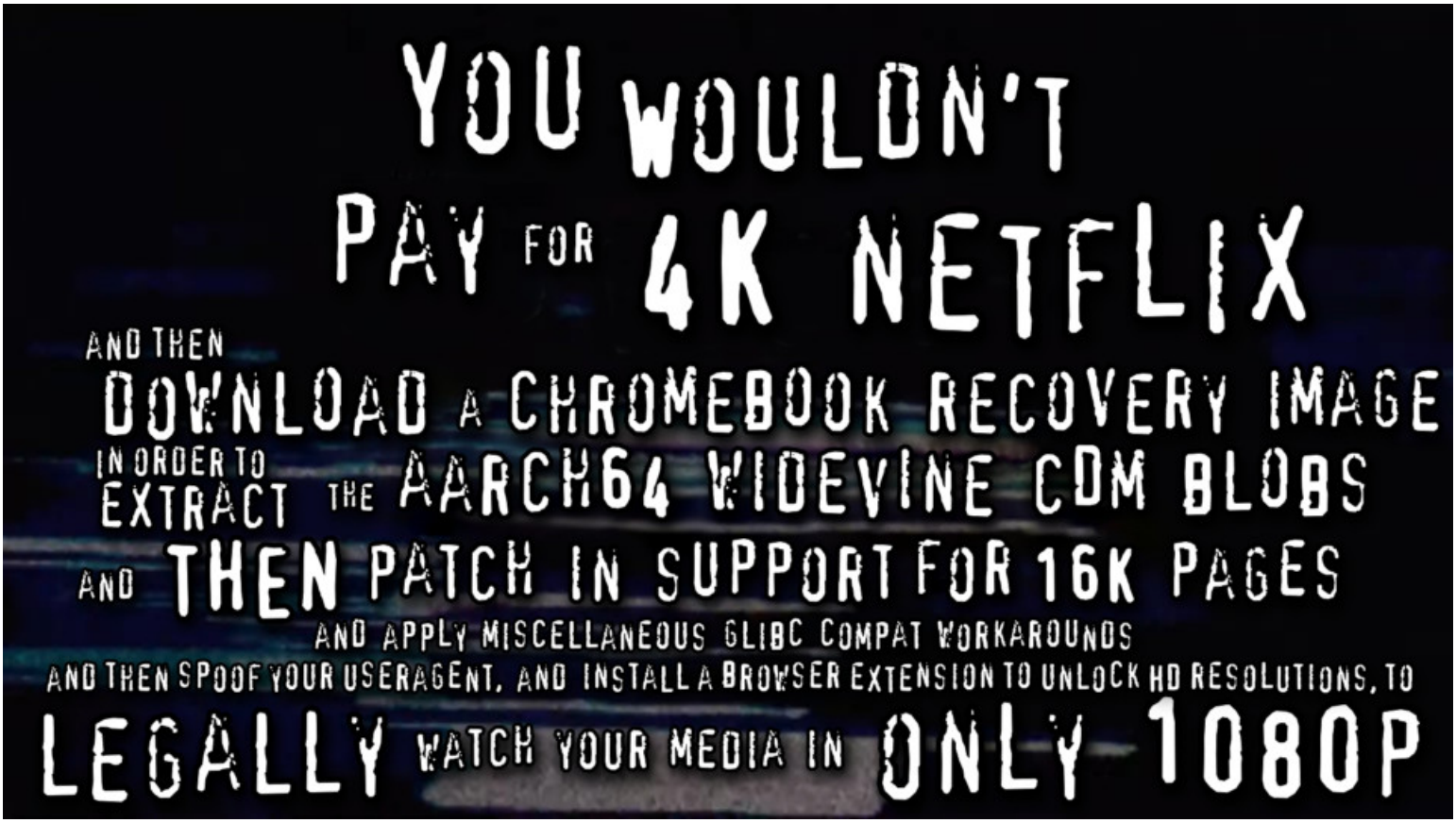
Most streaming platforms will limit you to only "HD" content on L3 (as opposed to 4K on L1). On Netflix, this upper limit is 1080p (although it might depend on the specific content you're trying to watch?), *but* you are further limited to a mere 720p by default. For some reason, you can only get 1080p if your client asks nicely for it (at the protocol level), and there are **browser extensions** that do this for you automatically.

I don't know why it's like this; my best guess is that there are some devices out there that have performance issues with 1080p playback through Widevine L3 (which necessitates software video decoding), and so they disabled it across the board to reduce the customer support burden.

Dear Netflix: please add a secret menu item to override this behaviour somewhere.

Summary

To summarise the above in meme format:



*You wouldn't make your own memes at **youwouldntsteala.website***

Disclaimer: This meme does not constitute legal advice. That said, I don't think this process violates the letter or spirit of the DMCA, nor any other relevant legislation, to the best of my knowledge.

Conclusion

I didn't expect my first Widevine-related blog post to be about using the DRM as intended.

Nevertheless, the most technically interesting method of streaming media on aarch64 Linux right now is to wrangle DRM into doing its job.

This should be alarming to anyone with a stake in content "protection". Look how many

hoops I had to jump through just to legally watch Netflix as a paying customer!

It would have been orders of magnitude more convenient for me to use a torrent client, but that would be a boring article. Making the DRM work should not be the "interesting" path!

Dear Google: There are simple steps you can take to improve this situation. Adding aarch64 Ubuntu LTS (or something similarly boring) to your matrix of supported platforms really shouldn't be that hard, given that you already support x86_64 Linux, and aarch64 ChromeOS. This means distro maintainers can spend less time wrangling glibc compatibility issues and perhaps make it easier for Spotify to release a native client for aarch64 Linux!

Addendum: The EME API is a good thing! (kinda)

A common objection to the mere existence of the EME API goes something along the lines of:

> DRM??!?!?! In *my* W3C standards??? I do not like DRM so this is Very Bad.

I also do not like DRM, and I would love to live in a world where it doesn't exist (and does not need to exist). Unfortunately, it does exist. Worse, market forces will ensure that it is crammed into every available DRM-shaped hole. If a DRM-shaped hole does not exist, one will be created. I would rather have my DRM-shaped holes emplaced by a coordinated standardisation process, than by forcing DRM vendors to do it ad-hoc (and inevitably, as opaquely as possible).

My attitude may be slightly defeatist, but if the EME API didn't exist, we probably wouldn't be able to watch Netflix on Asahi Linux at all, at least, not without violating the DMCA.
