Giving Rust a chance for in-kernel codecs

Daniel Almeida
Consultant Software Engineer
Collabora

COLLABORA

This talk is about codec hardware accelerators and their inherent safety issues

COLLABORA

It is also about how we can fix this problem

Hardware codec accelerators

- Specialized hardware to speed up decoding / encoding
- They are usually faster and generate less heat
- Their use frees up the main CPU, but..
- We now need drivers and an API to communicate with userland

Hardware codec accelerators

- With a CPU implementation, everything is in userspace
- With a hardware accelerator, there's a userspace component
- And also a kernel component, which means a highly privileged execution context

COLLABORA

Let’s look inside a video bitstream


Metadata
Tile/Slice Data

Bitstream metadata

- Controls the decoding process,
- A change in one parameter changes how the hardware interprets the rest of the bitstream
- **Is parsed from untrusted input**
- Interpreted and fed to the device by the kernel

Current validation process

- Userspace programs *may* introduce their own checks
- Kernel has an extremely ad-hoc validation strategy
- If something breaks, we hope it's before the kernel gets involved
- Otherwise, we hope that the device simply hangs

COLLABORA

Note: if the device hangs, you will have to reboot the machine

The Most Dangerous Codec in the World:
Finding and Exploiting Vulnerabilities in H.264 Decoders

Willy R. Vasquez
The University of Texas at Austin

Stephen Checkoway
Oberlin College

Hovav Shacham
The University of Texas at Austin

Abstract

Modern video encoding standards such as H.264 are a marvel of hidden complexity. But with hidden complexity comes hidden security risk. Decoding video in practice means interacting with dedicated hardware accelerators and the proprietary, privileged software components used to drive them. The video decoder ecosystem is obscure, opaque, diverse, highly privileged, largely untested, and highly exposed—a dangerous combination.

We introduce and evaluate H26FORGE, domain-specific infrastructure for analyzing, generating, and manipulating syntactically correct but semantically spec-non-compliant video files. Using H26FORGE, we uncover insecurity in depth across the video decoder ecosystem, including kernel memory corruption bugs in iOS, memory corruption bugs in Firefox and VLC for Windows, and video accelerator and application processor kernel memory bugs in multiple Android devices.

In the second case study, we played a larger corpus of random H26FORGE-generated videos on a variety of Windows software and Android systems from many dated but still relevant vendors. In all, we identified a memory corruption vulnerability in Firefox video playback; a use-after-free in hardware-accelerated VLC video playback; and insecurity in depth across the hardware decoder ecosystem, including disclosure of uninitialized memory and of prior decoder state; accelerator memory corruption; and kernel driver memory corruption and crashes.

COLLABORA

Last year’s proposal:

COLLABORA

Maybe let’s write a codec driver in Rust?

Why Rust?

- Sized arrays
- Runtime bound checks using get()
- Iterators instead of dangerous for-loops
- References (which are always valid) instead of pointers
- Ownership, lifetimes, destructors, etc...

Brief recap

- A driver would need a layer of bindings, i.e.: abstractions
- This layer of bindings did not please the maintainers
- Therefore, this approach was abandoned

Feedback from last year

- Who maintains what?
- This will slow down development in C
- This may break C code
- The community is overwhelmed

COLLABORA

What if we could write Rust code without bindings?

COLLABORA

We can do so by converting a few *functions* at a time

COLLABORA

This would sidestep most of the issues raised last year!

```
1 #[no_mangle]
2 pub extern "C" fn call_me_from_c()
```

- Generate machine code that can be called from C
- Make it so the linker can find it
- Can be used as an entry point to call other Rust code

```
1 #[no_mangle]
2 pub extern "C" fn call_me_from_c()
```

- We don’t want this: _RNvNtCs1234_7mycrate3foo3bar
  - So no generics,
  - No closures
  - No namespacing
  - No methods, etc.

```
1 #[no_mangle]
2 pub extern "C" fn call_me_from_c()
```

- We need this to be callable from C, hence “extern C”
- Rustc will give us the machine code for the symbol.
- That’s it really, the linker will happily comply.

```
1 #[no_mangle]
2 pub extern "C" fn call_me_from_c()
```

- The **public API** is then rewritten as per above
- But we need a way to expose the new API to C somehow.
- Because…

```
/* src/foo.h */
void call_me_from_c();
```

- This works.
- But it is not a good idea.
- It can quickly get out of sync.
- Nasty bugs can creep if we are not careful.

COLLABORA

No worries, there’s a tool

Cbindgen

- Cbindgen can automatically generate a C header
  - Keeps things in sync
  - Ensure proper type layout and ABI
- Avoids link errors and/or subtle bugs
- Maintained by Mozilla

Cbindgen

- If a function takes arguments, *cbindgen* will generate equivalent C structs
- This works because of **#[repr(C)]**

Summary

- Convert self-contained components into Rust

- Ask *rustc* to generate the machine code
- Annotate the public API so that it's callable from C

- Automatically generate a header file using cbindgen
- #include the header in C code

COLLABORA

#include the header, that’s it.

Potential targets

- This type of conversion works best when:
  - There is a self-contained component
  - That exposes a small public API
- For *video4linux*, this means:
  - Codec libraries
  - Codec parsers

Codec libraries

- Codec algorithms that run on the CPU
- Results are fed back to the hardware
- Abstracted so drivers can rely on a single implementation
- Very self-contained

Rewriting the VP9 library

- Two drivers were converted
- There is a testing tool
- **We got the exact same score when running the tool**
- Relatively pain-free process

COLLABORA

New proposals:

Proposals

- Merge the code
- Gate it behind a KCONFIG
- Users get the C implementation by default
- Run the Rust implementation on a CI
- Eventually deprecate the C implementation

Compared to last year

- Overall, a less ambitious approach
- Less inconvenience to maintainers
- **The Rust code can be used by future Rust drivers, if any**

COLLABORA

Fixing the metadata handling should be good enough for now

Feedback

- Provide examples of actual crashes that are fixed by Rust
- Measure any performance impacts
- Enable the Rust support in media-ci
- Use media-ci to continuously test the Rust code
- Merge the code in *staging/media*

Performance

- There should be no overhead in using this approach
- This means that the #[no_mangle], extern “C” stuff is free
- The added checks are not free, of course
- Programming the HW is *by far* not the bottleneck

COLLABORA

Hopefully we can use this approach for stateless encoders once they are introduced

COLLABORA

Thoughts?

Thank you!