30 SEP 2026 • LACH STUFF

wine vs champagne

What would happen if you add extra CO2 into wine, and what it does with PE libraries.

There are many articles covering PE manual mapping on Windows.[1] Is there anything left to cover? What about manual mapping PE... On Linux?

What even is manual mapping? That's simple: the operating system provides the means to load arbitrary dynamic library files, `LoadLibrary{,A,W}` on Windows and `dlopen` on Linux/BSD. Manual mapping is implementing such stuff manually.

The DLL I could not load

My use case was VivePro2-Linux-Driver - SteamVR driver for HTC Vive Pro 2 VR headset, for which Vive had implemented lens math inside of LibLensDistortion.dll, which obviously can't be loaded into a Linux process.

LibLensDistortion implements just a couple of functions: loading the JSON file from the Vive Pro 2 headset, and then doing the basic optical transforms: `distortUV` per eye per color channel, `getIntrinsic`, `getGrowForUndistort` etc. The JSON file is baked into the headset at the factory, and there are many possible configurations in the wild.

How is it implemented? It uses OpenCV matrices and some proprietary math, because apparently optics is hard. Of course, that means it links to the 80MB opencv_world346.dll.



Why not reverse-engineer the math and rewrite it in Rust?

Well, Beyley already did port one of the calibration modes to the Monado VR compositor. However, other configurations are possible, and I wanted my implementation to be correct: my own headset already came back from repair with an unsupported config, and only HTC knows what else they might write to this JSON.

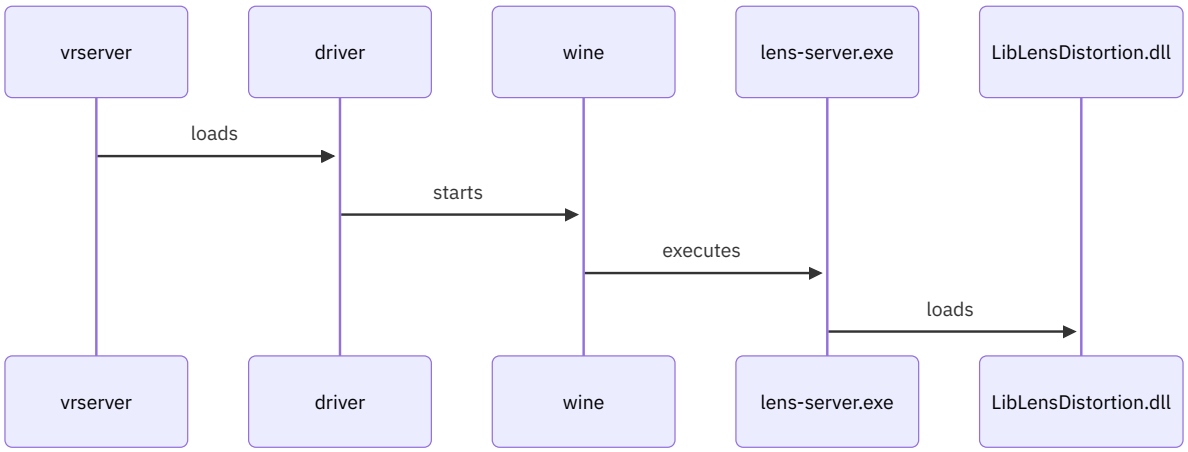
Thus I think the most correct way to handle this is to treat the JSON the headset provides as a mostly opaque blob, and let the official library do all the work.

How it was done before

To run Windows DLLs on Linux we had a solution ages ago - Wine. Wine works fine when you want to run a game, but what if you have some math library which only supports Windows? One solution here is writing a stdio server, with `_setmode` to handle arbitrary data without dumb Windows conventions (CR/LF and some other stuff that ends up corrupting the arbitrary byte stream), compiling it under MinGW, and then at runtime starting it in Wine, waiting until Wine sets up the prefix, and finally you can communicate.

That's what I did. The old version of VivePro2-Linux-Driver had a lens-server.exe binary, which I started in Wine just to load LibLensDistortion and wrap its calls into postcard IPC:

1. Here, for example: https://mahemium.blog/dll-manual-mapping, https://rootkiddie.com/blog/post/notes/windows-manual-mapping/notes



Obviously, it isn't that easy. Wine needs to be installed on the host system. What could go wrong? Everything.

There are many variants of Wine (`wine`, `wine64`, `winewow`, `winewow64`): 32-bit only, 64-bit only, the classic WoW64 build with both halves installed side by side, and the new WoW64 mode, where 32-bit Windows code runs inside a 64-bit process. The binaries mostly work, the prefix is the problem: it is created for one architecture (win32 or win64), it expects the wineserver of exactly the same Wine version, and every Wine upgrade rewrites it on the next start. Somehow users had their default prefix incompatible with the system Wine binary (e.g. the prefix was created by `wine`, but the system is using `wine64`).

Where is the prefix even coming from? My driver didn't set `WINEPREFIX`, so it used `~/.wine` - whatever the user, winetricks or some other Wine build created there years ago.

Other problem - the first Wine start runs `wineboot` to initialize the prefix (and so does the first start after every Wine upgrade), and this process is somehow very long sometimes, resulting in the SteamVR watchdog killing the process, making the prefix even more broken than before.

My next solution was Steam's Proton. SteamVR is started from Steam, so maybe we can find Proton there, if the user has ever started any Windows game? Again, it somehow broke: I was using the latest Proton version I could find, and the prefix broke for multiple users. For some users it wasn't initializing the prefix correctly, without copying the necessary libraries, why?! (https://github.com/CertainLach/VivePro2-Linux-Driver/pull/50)

Then there were problems with `start.exe` somehow starting the lens-server process in a separate window, only for some users, instead of giving me its stdio.

In the end I was never happy with this solution. This process is complicated, slow, somehow less reliable due to weird Wine behaviors... There has to be a better way.

Existing solutions

What are the existing solutions? Well, there are multiple: Wine, which was already mentioned, can start .exe files perfectly fine, but can't work as a library, and https://github.com/taviso/loadlibrary, which is intended to load antivirus engines, but is too far from being usable for general libraries. I wanted to use loadlibrary for it first, but it was far too primitive. The most basic thing it was lacking is the ability to load librar**ies**: out of the box it only loads a single librar**y**. The second reason is that it only supports 32-bit x86, while LibLensDistortion is x86_64.

Btw, champagne now has a drop-in replacement for loadlibrary's mpclient as a usage example.[2].

 What about Winelib?

Someone already asked me about that, and the reason is that it is mostly intended for recompiling existing apps from Windows to Linux. Of course, I can't recompile LibLensDistortion (if I could, I wouldn't need Winelib in the first place), and even if I wrote lens-server using Winelib... Winelib does not support using LoadLibrary from a regular glibc process, and that conflicts with my requirement to load this library directly into the SteamVR vrserver process, together with my driver.

That's why I decided to do my own implementation.

Loading: the part everyone already wrote about

Parsing the headers, copying sections to their virtual addresses, applying base relocations, setting page protections - all of that works on Linux exactly as it does on Windows, and the article from the footnote at the start explains it better than I would. I use https://github.com/CasualX/pelite for parsing, and the whole mapper is a couple hundred lines. Only two things were different.

First, there is no `VirtualAlloc`, and plain anonymous `mmap` was not enough either: my kernel refuses to add `PROT_EXEC` to an anonymous mapping, which is exactly what a loader does after writing relocations. So the mapping is not anonymous:

```
let map = shm_open(
   shm_name.as_str(),
   OFlag::O_RDWR | OFlag::O_CREAT | OFlag::O_EXCL, Mode::S_IRWXU,
)?;
shm_unlink(shm_name.as_str())?;
ftruncate(map.as_raw_fd(), len as i64);
MmapMut::map_mut(map.as_raw_fd())?
```

Create a POSIX shared memory object, unlink it right away so nothing is left behind in `/dev/shm`, size it, map it. For the kernel this is now a file mapping, and file mappings are allowed to become executable.

Second, the module list. On Windows, the usual reason to manual map is to *hide* a module: it never appears in the PEB loader lists, so `GetModuleHandle`, `EnumProcessModules` and every anticheat walking those lists can't see it. Champagne does the opposite. There is no other loader in the process, so the list I maintain is the only source of truth: the next DLL finds its imports there, `GetModuleHandle` and `GetProcAddress` search it, and the unwinder uses it to find which module an instruction pointer belongs to. So every image gets a real `LDR_DATA_TABLE_ENTRY`, linked into a real `PEB_LDR_DATA`, before its imports are even resolved:

```
let mut entry = OwnedLdrData::new(
   full_name,
   base_name,
   image.as_ptr().cast(),
   image.len(),
   ep,
);
get_peb().lock().add_entry(entry.unchecked_get_pinned());
```

With `full_name` set to `C:\libs\` plus the file name, because a Windows DLL deserves a Windows path.

2. mpengine requirements are not documented anywhere, so I had llm debug the missing part for a couple of hours, but the end result is that champagne now has everything required to load and run microsoft defender antivirus engine



So it's a manual mapper that carefully registers everything in the PEB... isn't that just a loader?

Pretty much. If you manual map on a system with no loader, you have to become one.

Implementing kernel32 on demand

Windows DLLs don't talk to the kernel directly, they go through `kernel32.dll` and friends, and there are no such DLLs here (I think it should be possible to load real kernel32.dll, but I would want to avoid loading half of the Windows implementation just for some math library). Every import is resolved in three steps:

```
// A real module always wins
let real = linker.find_entry(&dllstr).map(|dll| (dll.exported_fn_raw(namestr), dll));
*new = match real {
   Some((Ok(fun), _)) => fun as u64,
   _ => match default(&dllstr, namestr) {
      Some(v) => v as u64,
      None => make_stub(format!("function was not defined: {dllstr}:{namestr}")) as usize as u64,
   },
};
```

1. If a DLL with that name is already mapped and exports the function, use it. This is how opencv gets the real `ucrtbase.dll` and `msvcp140.dll`.
2. Otherwise, look it up in the builtin table: functions I wrote in Rust.
3. Otherwise, generate a stub that logs the function name and aborts.

The third step is what makes this whole thing possible. `opencv_world346.dll` imports 531 functions from 26 DLLs, including `user32.dll`, `gdi32.dll`, `d3d11.dll`, Media Foundation and `comdlg32.dll` - the file open dialog. The lens library never calls any of them. If every missing import was a load error, I would have to implement the Windows GUI before seeing a single `distortUV` call. With stubs, the workflow is: run it, get `stub was called: function was not defined: kernel32.dll:SomethingNew`, implement that one function, run again. About 110 builtins later, the whole lens stack runs.

The stubs themselves are generated at runtime with Cranelift: a tiny function that passes a pointer to the message into a Rust function, which logs it and aborts.



You pulled in an entire compiler backend to print a string?

Yes, and what can you do with that?

Seriously though, I wanted to synthesize kernel32.dll and some others here in the future, but so far it was not required for anything.

The builtins are plain Rust functions with an attribute:

```
#[winfn]
fn GetCommandLineA() -> *const c_char {
   // Must be writable: plenty of argv parsers tokenize it in place.
   static CMD: OnceLock<Box<[u8]>> = OnceLock::new();
   CMD.get_or_init(|| Box::from(*b"dllloader\0"))
      .as_ptr()
      .cast()
}
```

`#[winfn]` does two things: makes the function `extern "win64"`, and registers it in a global table through the https://github.com/dtolnay/inventory crate, so a new builtin can live in any module without being listed

anywhere. `#[alias(LoadLibraryW)]` registers the same function under extra names.

 What is `extern "win64"` doing there?

Windows and Linux disagree on how to call a function on x86_64. Windows passes the first four arguments in `rcx`, `rdx`, `r8`, `r9` and wants 32 bytes of shadow space on the stack. System V (Linux) uses `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, and considers `rsi`, `rdi` and `xmm6-xmm15` scratch registers, while on Windows the callee has to preserve them. This is usually where you write assembly thunks. Rust just supports both conventions: builtins are `extern "win64"`, so the DLL calls them directly, and DLL exports are typed as `unsafe extern "win64" fn(...)`, so my code calls into the DLL directly too. No thunks at all.

One more wrinkle: modern MSVC binaries don't import the CRT from `ucrtbase.dll`, they import it from API sets like `api-ms-win-crt-stdio-l1-1-0.dll` - virtual DLL names that Windows redirects to real ones. I cheat: every name that starts with `api-ms-win-` becomes `ucrtbase.dll`. That's right for the `crt` sets and wrong for the `core` ones, but the builtin table is looked up by function name only, ignoring the DLL name, so those land in the right place anyway.

Pretending to be Windows: TEB and PEB

For better compatibility with Windows libraries, champagne uses real PEB/TEB structures on Linux, as references to those are stored in the gs register on x86_64 Windows, and on Linux this register is free to use. This way, on Windows, it should behave just as a normal manual mapper, operating on the real PEB/TEB instead of faking them (the version published on GitHub doesn't do that yet, TBD).

 Wait, you can just write to gs?

Since Linux 5.9, yes: on CPUs with the FSGSBASE extension, userspace may use `rdgsbase`/`wrgsbase` directly. glibc keeps its own thread-locals in `fs` and leaves `gs` alone, so it is free to point at a Windows TEB:

```
pub fn enter(&self) -> EnteredVirtualTib {
   let prevbase: *const ();
   let gs: *const TibNtrnl = self.0.get();
   unsafe {
      asm!(
         "rdgsbase {prevbase}",
         "wrgsbase {gs}",
         prevbase = out(reg) prevbase,
         gs = in(reg) gs,
      );
   };
   EnteredVirtualTib { prevbase }
}
```

The returned guard writes the old value back on drop, and calls into the DLL happen while it is alive.

Why bother with the real layout instead of intercepting API calls? Because compiled code doesn't ask. For a `__declspec(thread)` variable MSVC emits something like this:

```
mov ecx, dword ptr [_tls_index]
mov rax, qword ptr gs:[0x58]; TEB->ThreadLocalStoragePointer
mov rax, qword ptr [rax + rcx*8]
```

There is no function call to hook, just a read at a fixed offset from gs. So the TEB has to be real bytes at real offsets, and the offsets are pinned at compile time:

```
const _: () = assert!(offset_of!(TibNtrnl, this) == 0x30);
const _: () = assert!(offset_of!(TibNtrnl, peb) == 0x60);
const _: () = assert!(offset_of!(TibNtrnl, tls_slots) == 0x1480);
```

Same for the PEB: loader data at `0x18`, loader lock at `0x110`, and so on. My builtins use them too: `GetLastError` reads `LastErrorValue` from the TEB, not some Rust thread-local, so state lives where Windows code expects it.

Everything Windows keeps in the kernel - the handle table, thread ids, TLS templates - hangs off the PEB as well, in the `SubSystemData` field, guarded by a magic value (`DLLLOADR`) in case the guest overwrites it. This way all state is reachable from `gs`, and in theory one Linux process can hold several independent Windows "processes". Handles and thread ids are handed out in multiples of four, like on Windows, because some code uses the low two bits of a handle for its own tags.

TLS, DllMain and threads

Windows has two kinds of thread-local storage, and the lens stack needs both.

Static TLS is the `__declspec(thread)` kind from above. A PE with thread-locals has a TLS directory: a template (initial data plus zero-fill size) and an address where the loader writes the module's TLS index. The loader picks the index, writes it there, and gives every thread its own copy of the template at `ThreadLocalStoragePointer[index]`. In champagne the templates are stored in the PEB private data, and each new thread materializes its copies before running any guest code.

The TLS directory also lists callbacks: something like a `DllMain` that runs before `DllMain`. They get `DLL_PROCESS_ATTACH` before the entry point, and then attach/detach for every thread.

Dynamic TLS is `TlsAlloc`/`TlsGetValue`: 64 slots right inside the TEB at `0x1480`, with allocation tracked by a bitmap in the PEB. Slot 0 is reserved from the start, same as ntdll does in `LdrpInitializeProcess`.

`DllMain` itself runs under the loader lock, which is a critical section pointed to by `PEB+0x110`. It has to be reentrant: a `DllMain` may load another library, and a plain mutex would deadlock right there.

`CreateThread` maps to `std::thread::spawn`, plus everything a Windows thread expects to exist before its first instruction:

```
builder.spawn(move || {
   let tib = VirtualTib::for_peb(peb, id);
   let _entered = tib.enter();
   t.wait_while_suspended();

   peb.to_ref().materialize_current_tls();
   notify_modules(DLL_THREAD_ATTACH);

   let start: ThreadStart = unsafe { std::mem::transmute(start) };
   let code = start(parameter as *mut c_void);

   notify_modules(DLL_THREAD_DETACH);
   t.finish(code);
})?;
```

Its own TEB, its own TLS copies, `DLL_THREAD_ATTACH` to every module (TLS callbacks first, then `DllMain`, reversed on detach), and a handle in the private handle table so `WaitForSingleObject` and `GetExitCodeThread` work on it.

When things go wrong: unwinding

On x86_64 Windows there is no frame pointer chain to walk. Instead, every non-leaf function has an entry in the `.pdata` section: a `RUNTIME_FUNCTION` with start and end addresses and a pointer to `UNWIND_INFO`, a small bytecode describing what the prologue did - pushed `rbx`, allocated `0x28` bytes, set up `rbp` as frame pointer. Unwinding one frame means finding the entry for the current `rip` and undoing those operations on a saved register context.

The CRT needs this even if nobody throws anything. When a `/GS` stack cookie check fails, or a CRT function gets an invalid parameter, it asks `IsProcessorFeaturePresent(PF_FASTFAIL_AVAILABLE)` - mine says no to everything - and takes the fallback path: `RtlCaptureContext`, then `RtlLookupFunctionEntry` and `RtlVirtualUnwind` to step out of itself, then `UnhandledExceptionFilter`. So that's what I implemented: `RtlCaptureContext` as a naked asm function, a lookup that finds the module through the PEB and binary searches its `.pdata`, and an interpreter for the unwind codes.

The reward: my `UnhandledExceptionFilter` prints a real backtrace, with module names from the PEB:

```
WARN champagne_winapi::seh: RaiseException: code=0xe06d7363 (C++ exception) flags=0x1
WARN champagne_winapi::seh:   frame 0: ucrtbase.dll+0x2dd2cf
WARN champagne_winapi::seh:   frame 1: opencv_world_346.dll+0x183730
WARN champagne_winapi::seh:   frame 2: LibLensDistortion.dll+0x183730
WARN champagne_winapi::seh:   frame 2: 0x5b3f1dfd1fab (no unwind info)
```

 What is `0xE06D7363`?

That's every C++ `throw` in MSVC: `0xE0` followed by `msc` in ASCII.

Putting it together

The lens stack is five DLLs, loaded in dependency order into one virtual PEB (slightly simplified):

```
let peb = VirtualPeb::new();
let tib = VirtualTib::new(&peb);
for lib in [
   "ucrtbase.dll",
   "vcruntime140.dll",
   "msvcp140.dll",
   "opencv_world346.dll",
   "LibLensDistortion.dll",
] {
   let _entered = tib.enter();
   let mut m = PeImage::open(format!("{libs}/{lib}"))?;
   m.resolve_imports(override_import, &peb)?;
   let m = m.finish()?;
   unsafe {
      m.init_static_tls()?;
      m.call_ep_if_exists()?;
   }
}
```

`open` maps, relocates and registers the image, `resolve_imports` fills the import table, `finish` applies section protections, and then TLS and `DllMain` run, like a real loader would do. After that, exports are just typed function pointers:

```
let distort_uv:
   unsafe extern "win64"
```

```
    fn(eye: u32, color: u32, u: f32, v: f32, c1: *mut f32, c2: *mut f32) -> u32
= unsafe { m.exported_fn("distortUV")? };
```

Why champagne

Wine is a recursive acronym: **W**ine **I**s **N**ot an **E**mulator. Champagne is wine that spent some time under pressure, and after years of debugging my users' wine prefixes, so did I. It runs Windows code with no Wine anywhere near it, so of course it needs a recursive acronym too:

CHAMPAGNE **H**osts **A**lien **M**icrosoft **P**E **A**longside **G**NU/**N**ative **E**LF.

 That's not what your README says.

The README says "**C**HAMPAGNE: **H**ostile **A**cquisition of **M**icrosoft **P**E, **A**nnexed by **G**NU/**N**ative **E**LF", followed by "Placeholder acronym, TBD, IDK". I still DK, and will invent more recursive acronyms before someone stops me.

Conclusion

In the driver, the "replace wine with champagne" commit deleted three crates: `lens-server`, `lens-protocol` and `lens-client`, together with all the Wine and Proton discovery code. The lens math now runs inside the SteamVR `vrserver` process, next to my driver. Every call is a plain function call instead of an IPC round trip, and users don't need to install anything or have a working prefix.

The PE format turned out to be the easy part - it is well documented, and pelite handles it for me. The actual work is pretending to be just enough of Windows: a TEB behind gs, loader lists in the PEB, two kinds of TLS, a loader lock, and about 110 kernel32 functions, each written after a stub told me exactly which one was missing.

It is far from complete, but if you have a Windows-only library you want to call from a Linux process, give it a try - when something is missing, the stub message will tell you what to implement, or what to send me in issues.

- champagne: https://github.com/CertainLach/champagne
- VivePro2-Linux-Driver: https://github.com/CertainLach/VivePro2-Linux-Driver

---

#rust #reverse-engineering