The Many APIs Of Gaming On Proton

by Arek Hiler

Proton Janitor @ CODE WEAVERS SOFTWARE LIBERATORS

@ivyl@treehouse.systems

ivyl @ libera & oftc

In computing, an **emulator** is hardware or software that enables one computer system (called the host) to behave like another computer system (called the guest). An emulator typically enables the host system to run software or use peripheral devices designed for the guest system. Emulation refers to the ability of a computer program in an electronic device to emulate (or imitate) another program or device.

- Wikipedia

In software engineering, **a compatibility layer** is an interface that allows binaries for a legacy or foreign system to run on a host system. This translates system calls for the foreign system into native system calls for the host system. With some libraries for the foreign system, this will often be sufficient to run foreign binaries on the host system.

- also Wikipedia

How Does Wine Work?

NAME

ld.so, ld-linux.so - dynamic linker/loader

SYNOPSIS

The dynamic linker can be run either indirectly by running some dynamically linked program or shared object (in which case no command-line options to the dynamic linker can be passed and, in the ELF case, the dynamic linker which is stored in the .interp section of the program is executed) or directly by running:

```
/lib/ld-linux.so.*  [OPTIONS] [PROGRAM [ARGUMENTS]]
```

DESCRIPTION

The programs ld.so and ld-linux.so* find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it.

Wine is kinda like that but for PEs

wine foo.exe

- finds and loads **ntdll.so** (more on .so vs .dll later)
- calls **__wine_main()**
- wine → **wine64-preloader** (addr reservation) → wine
- **inits environment** (PEB, TEB, USD and other windowisms) and stacks
- starts **wineserver** (more on that later)
- **loads foo.exe + imports**
- **loads ntdll.dll**
- signal_start_thread() → call_init_thunk() → RtlUserThreadStart() → **entry_point()**

PE <-> Unix

Syscalls

Syscalls On Windows

- A bunch of **Nt*() functions**, e.g. NtDelayExecution(), NtSetEvent(), NtWriteFile(), NtGdiCreateBitmap() that do the syscall for you.

- **No contract** wrt SYSENTER / `INT 0x80` / etc.

“Sycalling” Into The Unix Side

```
0x17000d2f0 mov r10, rcx                | ulong sym.ntdll.dll_NtClose(ulong Handle)
0x17000d2f3 mov eax, 0x15               |
                                        |
                                        | {
0x17000d2f8 test byte [0x7ffe0308], 1   |     if ((usd->SystemCall & 1) == 0) {
0x17000d300 jne 0x17000d305             |
0x17000d302 syscall                     |         return syscall(0x15);
0x17000d304 ret                         |
                                        |     }
0x17000d308 call qword [0x7ffe1000]     |     return __wine_syscall_dispatcher(0x15);
                                        | }
```

```
*0x7ffe0308 == user_shared_data->SystemCall == 1
*0x7ffe1000 == __wine_syscall_dispatcher() (one page after user_shared_data)
```

__wine_syscall_dispatcher()

- **stores context** in amd64_thread_data()->syscall_frame (amd64_thread_data() is TEB->GdiTebBatch)
- **switches thread's stack** from the "user space" one (Windows) to the "kernel" one (UNIX)
- **figures out what to call on the UNIX side** using the syscall number and System Service Descriptor Table (see syscalls[] and ntdll_init_syscalls())
- **calls the UNIX function** it has found and takes the result
- **restores the context / Windows stack**
- **returns** to the syscall thunk

Real Syscalls

- Some apps try to be sneaky and either **hardcode the syscall** numbers or **extract them from the thunks**.

- Proton installs a filter using **seccomp-bpf** that captures all syscalls in the **PE address range**.

- We get a **sigsys** and our **handler dispatches the syscall**.

Non-Syscall Unix Bits

- **The Unix Part lives in a .so** and exports a table of calls.
- The loader knows how to **load it along the .dll**.
- It has a table of the calls accessible via **NtQueryVirtualMemory() with magic MemoryWineUnixFuncs**. Done once in DllMain().
- **__wine_unix_call()** - stripped down and optimized __wine_syscall_dispatcher() that can be called directly.

wineserver

Responsibilities of Wineserver

- **“Kernel-like”**. Single-threaded. Accessible by the Unix side.
- **Keeping track of processes / threads**, querying their information, suspending / resuming.
- Win32 clipboard, atoms, file descriptor management, window tracking.
- **HANDLE management.**
- Async IO, sockets.
- **Registry access.**
- Change notifications.
- **Shared memory.**

The Unix-backed APIs
40 DLLs

Synchronization

Multiple implementations backing Nt*Event(), Nt*Semaphore(), NtWaitFor*() syscalls.

- **serversync** – wineserver is the mediator, each operation = IPC
- **esync** – eventfd-based, requires high nofile, not everything is or can be implemented accurately, e.g. waiting on all
- **fsync** – futex-based, futex_waitv, no dependency on nofile, similar limitations to esync
- **winesync/fastsync/ntsync** – upcoming Linux kernel driver

Audio – MMDevice API Backends

Multiple Unix backends:

- **winepulse.drv** – PulseAudio, including PipeWire via pipewire-pulse
- **winealsa.drv** - ALSA ¯_(ツ)_/¯
- **wineoss.drv** – OSS, BSD folks rejoice!
- **winecoreaudio.drv** – for OSX users

Implements/stubs things like ISpatialAudio and other things you can query from MMDevice.

3D - Vulkan – winevulkan.dll

- **make_vulkan** written in Python
- parses **vk.xml** and generates both Unix side thunks and PE thunks that calls them
- if conversion cannot be handled automatically there are handwoven wrappers
- **extension mapping**, e.g. vkCreateWin32SurfaceKHR → vkCreateXlibSurfaceKHR (more on that later)

3D - OpenGL – openg32.dll

- **make_opengl** written in Perl
- parses **gl.xml** and **winegl.xml** (unofficial extensions) and generates both Unix side thunks and PE thunks that calls them
- if conversion cannot be handled automatically there are handwoven wrappers

Windows, Events & Basic Graphics

Multiple Unix backends: winex11.drv, winewayland.drv, wineandroid.drv, winemac.drv
exposed on the PE side mostly by win32u.dll

Provides implementation behind:

- **Graphics Device Interface** - Nt*Gdi*() - bitmap creation, device context, blitting, drawing
- **keyboard and mouse handling** – NtUser*Keyboard*(), NtUser*Cursor*()
- **window creation and management** – NtUserSetFocus(), NtUserCreateWindowEx()
- **display management** – populating registry, NtUserChangeDisplaySettings()
- **platform specific WSI** – OpenGL’s wgl*(), Vulkan’s vkCreateWin32SurfaceKHR and related

winex11.drv

- Mostly **pain and misery**.
- Mapping windowing behavior between two APIs that are each **almost 40 years old**.
- A lot of things are up to Window Managers.
- There’s a humorous talk from XDC 2023 on this.

winegstreamer.dll

- **Wraps native decoders / encoders** and expose them to the PE side.
- Uses **GStreamer**.
- Base for our **Quartz** and **Media Foundation** implementations.

Controllers – winebus.sys

- **Multiple Unix backends:** SDL, udev (hidraw/evdev), iohid.
- **Exposes everything as HID devices** – creates a faux descriptor and reports when we don’t have hidraw access.
- There’s special, **hidden HID device for XInput** consumption.

Networking

- **ntdll.dll** – NtDeviceIoControlFile() for sockets on top of glibc and socket creation mediated via wineserver
- **ws2_32.dll** – uses ioctls, and unixlib for getaddr/gethost
- **dnsapi.dll** – DNS stuff, queries built on top of libresolve
- **bcrypt.dll** – provider of cryptographic functions (hashes, symmetric and asymetric encryption) built on top of gnutls

The PE APIs
643 DLLs

msvcrt.dll / ucrtbase.dll

- **C / C++ Runtime Library**
- fwrite(), malloc()
- **cos(), atan2()**
- std::

Audio

Everything implemented on top of mmdevapi.

XAudio – uses **FAudio**

DSound - DirectSound

Controllers

- **DInput –** based purely on top of HID. Should work on Windows. I don’t think anyone has tried.
- **XInput –** uses special devices we expose. Won’t work on Windows.
- **Windows Gaming Input –** build on top of DInput & HID, should work on Windows.

wined3d

- **DirectX 1-11** on top of OpenGL (and Vulkan)
- **tried and tested**
- people are using it on Windows for better compatibility with older games

DXVK – DirectX 9-11 Over Vulkan

- **Independent from Wine.**
- **With Proton since the beginning.**
- **Just a PE**, runs on Windows. People do use it.
- **DXVK-Native** for people who want DirectX on Linux.

vkd3d-proton

- Originally **forked** from WineHQ’s vkd3d.
- Depends on DXVK’s DXGI.
- **Solid DirectX 12 implementation** that’s running modern **AAA games**.

vkd3d

- **The original project.**
- Slowly catches up on DirectX 12 over Vulkan front.
- Used in Proton for the **libvkd3d-shader** part to provide **d3dcompiler*.dll**.

DXVK-NVAPI

- Provides implementation of **Nvidia's NVAPI**.
- Included with the Windows drivers.
- Game integration **assumes DLLs to be present** if GPU vendor == NVIDIA.
- So we had to **fake all Nvidia GPUs to be AMD RX480**.
- Integrates with DXVK and vkd3d-proton.

AMDAGS & ATIADLXX

- **AMD’s counterparts of NVAPI.**
- Implemented in Proton’s Wine.
- **Many many versions of the API.**
- Native DLLs ship with the game and like to break things.

That’s it! Now you have a good idea how Wine / Proton works.

CODE WEAVERS®

SOFTWARE LIBERATORS