Fake It ‘til We Make It:

The Art of Windows User Space Emulation

Who am I?

Maurice Heumann

- DRM Developer @ WIBU-SYSTEMS
    - DRM company in Karlsruhe, Germany

- Reversed & Bypassed many DRMs:
    - Steam CEG → many (older) Steam games
    - Arxan → Call of Duty, GTA V, Fortnite (old), ...
    - Denuvo → Hogwarts Legacy, ...
    - ...

- Twitter: momo5502

Agenda

- What is Windows User Space Emulation?
- What are the Applications?
- Existing Solutions
- Implementation
- Demo
- Final Words

Windows User Space Emulation

What’s that?

What’s that?

Emulation

- code runs on virtual CPU
- hardware is simulated
- full control of executed code

Windows User Space Emulation

- process runs in emulator
- OS + kernel is simulated

What’s that?

Emulator offers hooking points

- memory access hook
  - read, write, execute
- instruction execution hook
  - syscall, cpuid, rdtsc
- new code path execution hook

What are the applications?

DRM Analysis

- modern DRMs are too strong
  - obfuscation/anti tampering/anti debugging/...
  - static/dynamic analysis often impossible

- hooking points allow easy analysis
- external communication can be intercepted
- execution flow can be traced

→ Emulation was key for Denuvo analysis

Vulnerability Analysis

- blackbox fuzzing within the emulator
- input can be randomized
- emulator state can be saved/restored
- coverage feedback through hooks
- execution is predictable and repeatable

Malware Analysis

Similar to DRM Analysis

- hooking points allow easy analysis
- external communication can be intercepted
- execution flow can be traced

→ seems widely adopted already

Mobile Gaming?

- applications & games on mobile
- a lot of work needed
- performance might be too bad for gaming?
- only suited for old games?

→ still a dream I have 😄

Existing Solutions

Existing Solutions

- Qiling, Speakeasy, Dumpulator, ...
  - are written in Python

- DRM analysis requires a lot of hooks
  - e.g. hooking every memory read
  - can be extremely slow in Python

→ I need speed: C++

Existing Solutions

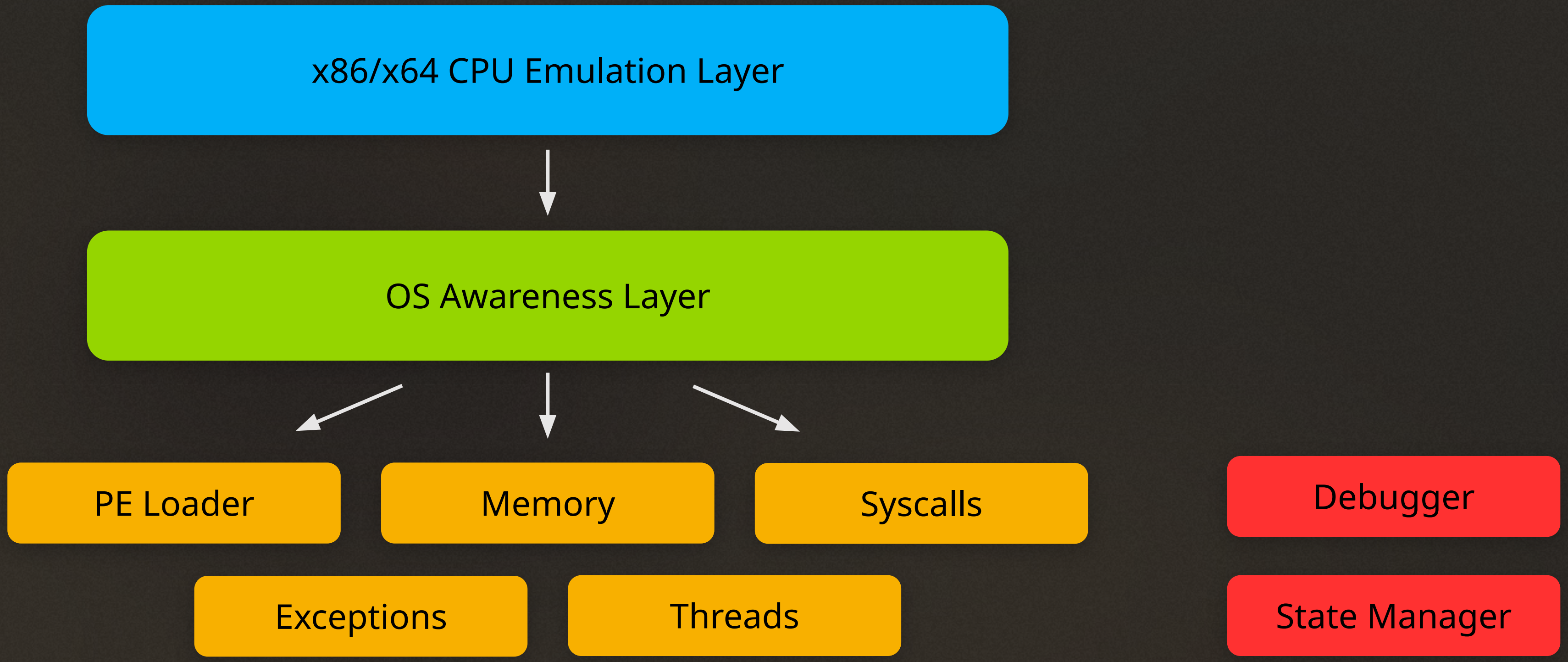
- Binee, Unicorn PE, ...
    - emulate on API level

- every API function from every DLL is reimplemented
- incomplete → there are so many APIs
- error prone → every reimplementation can contain bugs

→ I want to reuse all DLLs on my system

Architecture

Architecture



x86/x64 CPU Emulation Layer

OS Awareness Layer

PE Loader

Memory

Syscalls

Exceptions

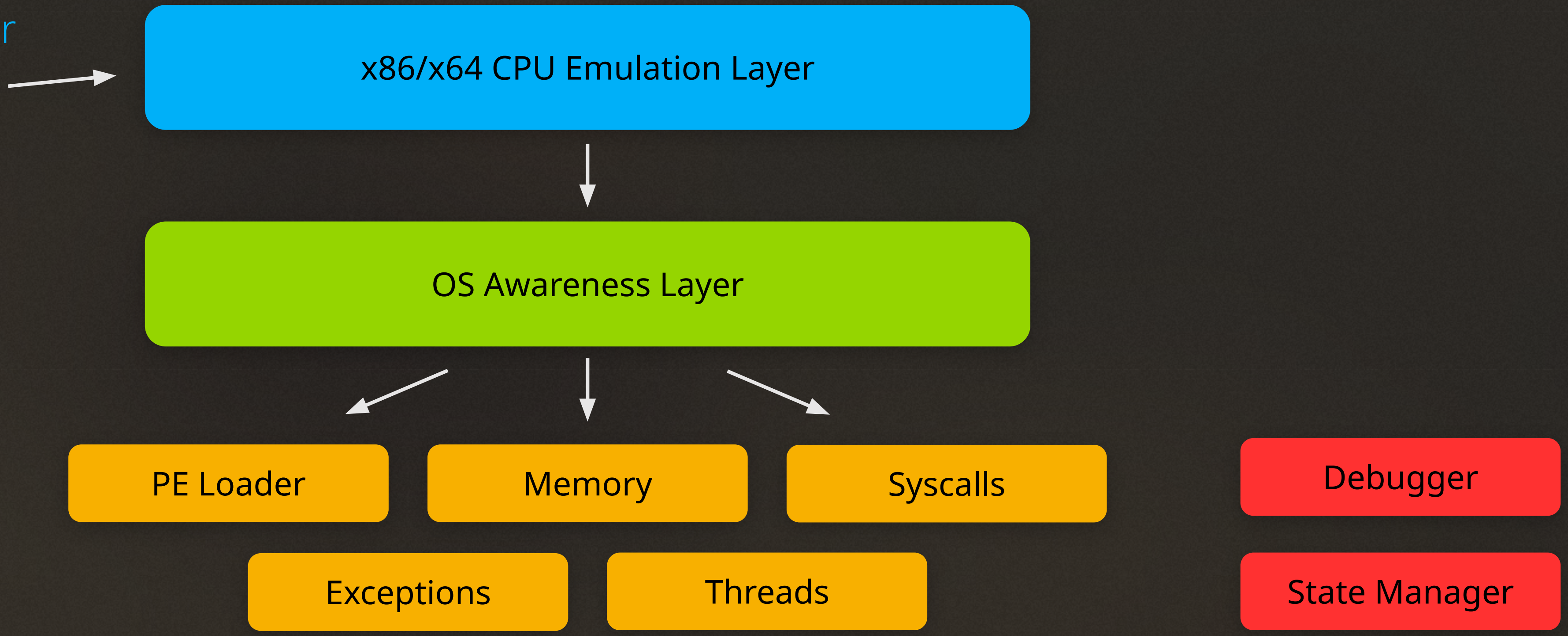
Threads

Debugger

State Manager

Architecture

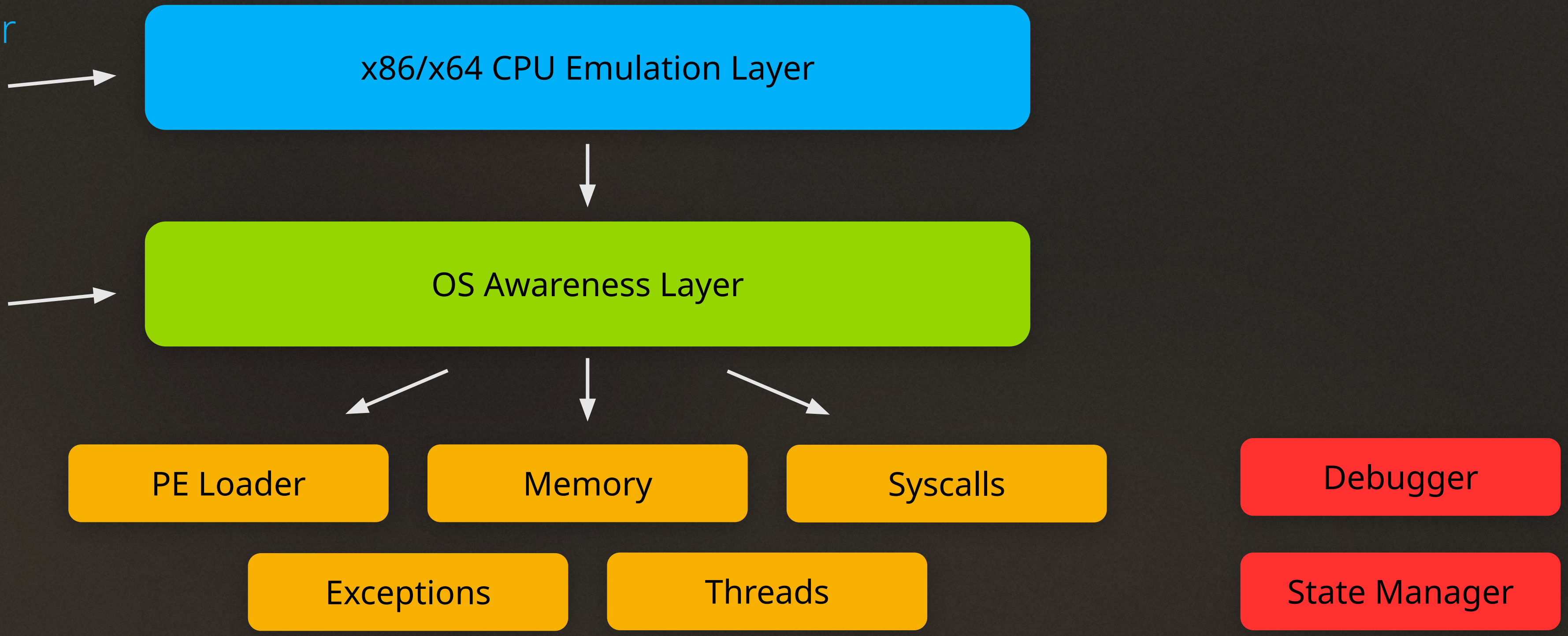
- 3rd Party: Unicorn Emulator
- based on QEMU
- reasonably fast → TCG



Architecture

- 3rd Party: Unicorn Emulator
- based on QEMU
- reasonably fast → TCG

- own implementation
- essentially a virtual kernel
- emulates on syscall level



PE Loading

PE Loading

- Portable Executables (EXE & DLLs)
  - are mapped by the kernel
  - NtMapViewOfSection syscall
- Kernel maps:
  - Headers
  - Sections with permissions
  - Relocations

→ Imports are resolved by NTDLL

→ DllMain is called by NTDLL

PE Loading

- Executable and NTDLL are always mapped at process start
- other DLLs are mapped on demand via NTDLL

Memory Manager

Memory Manager

- Unicorn supports basic memory
  - with permissions: read/write/execute
- Windows supports more types
  - reserved
  - committed
  - guard pages
  - …

→ must be implemented ontop of Unicorn

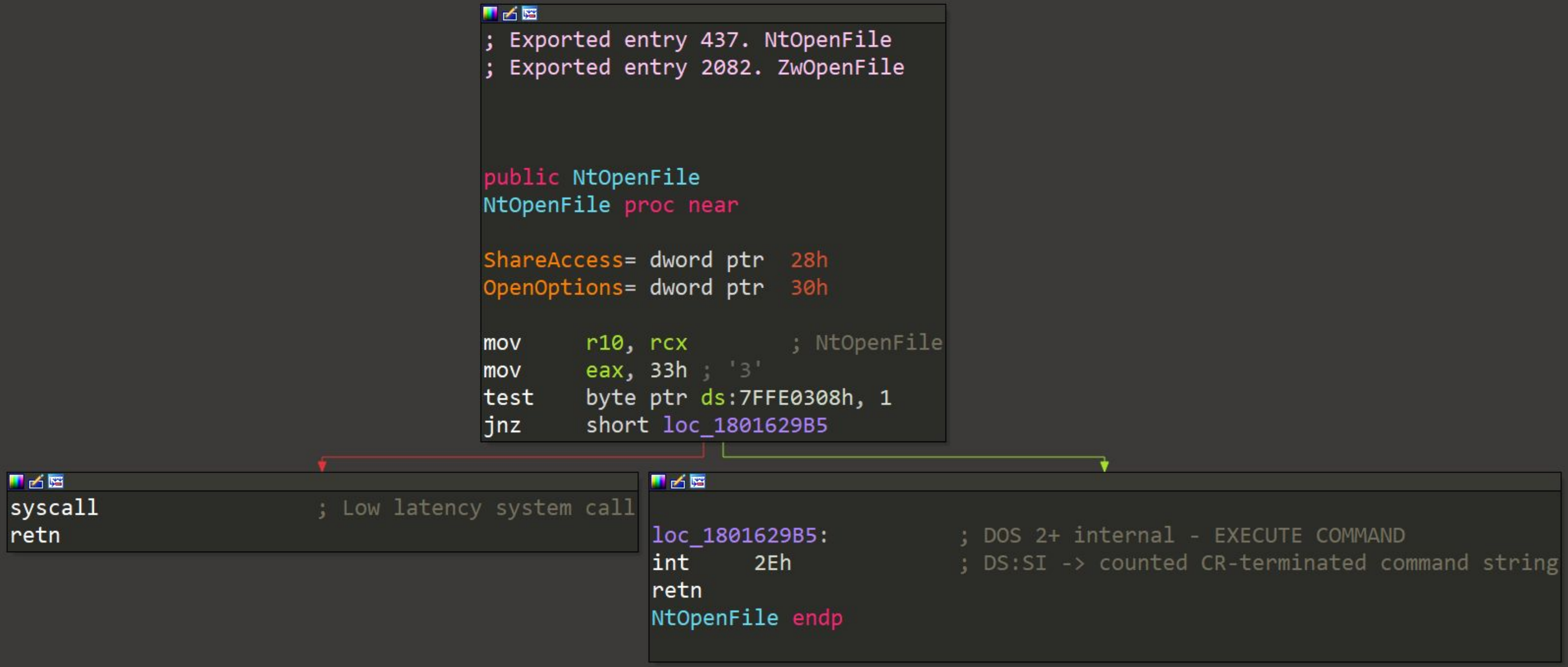
Memory Manager

Mapped memory

- TEB → Thread Environment Block
  - thread specific storage
- PEB → Process Environment Block
  - process specific storage
- KUSER_SHARED_DATA
  - quick access to important data → faster than syscalls
  - e.g. current time, processor info, OS version, ...
  - always mapped at 0x7FFE0000

Syscalls

Syscalls

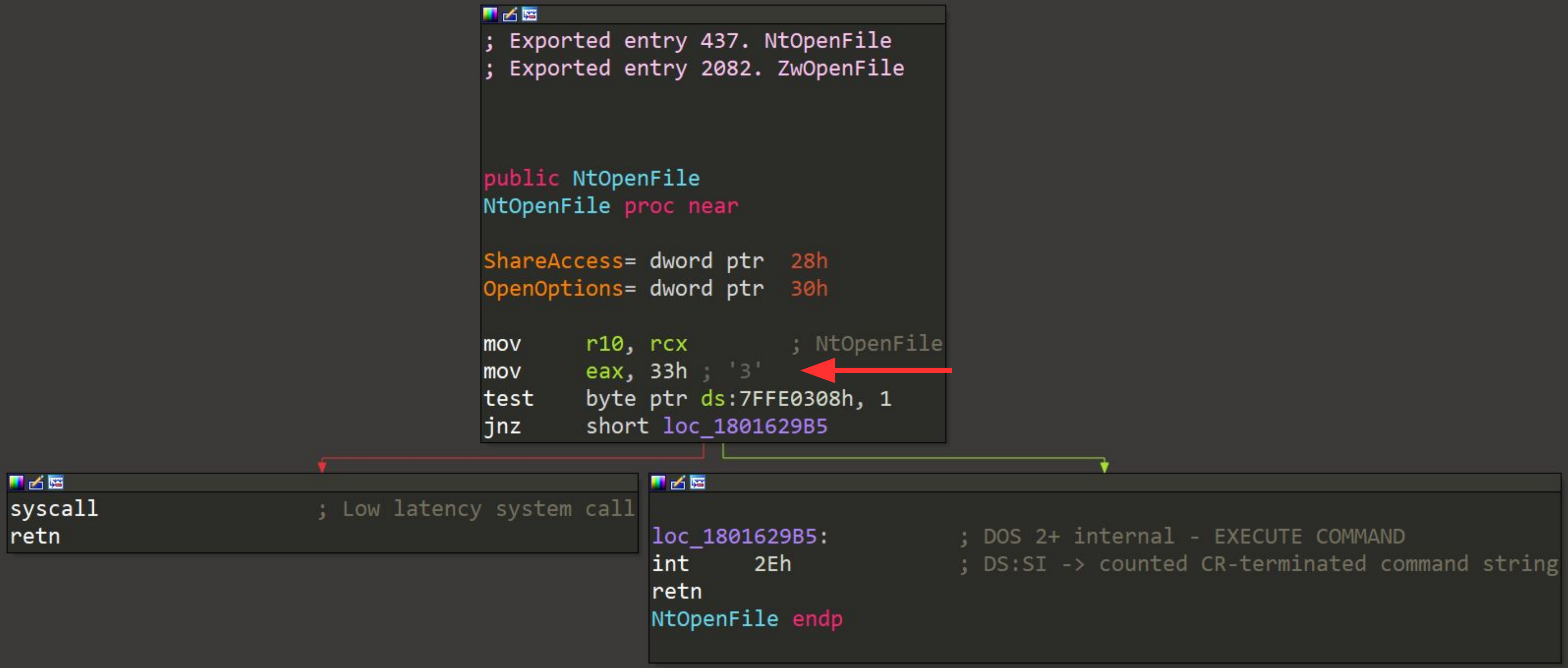


```
; Exported entry 437. NtOpenFile
; Exported entry 2082. ZwOpenFile



public NtOpenFile
NtOpenFile proc near

ShareAccess= dword ptr  28h
OpenOptions= dword ptr  30h

mov     r10, rcx        ; NtOpenFile
mov     eax, 33h ; '3'
test    byte ptr ds:7FFE0308h, 1
jnz     short loc_1801629B5
```

```
syscall                    ; Low latency system call
retn
```

```
loc_1801629B5:           ; DOS 2+ internal - EXECUTE COMMAND
int     2Eh              ; DS:SI -> counted CR-terminated command string
retn
NtOpenFile endp
```

Syscalls

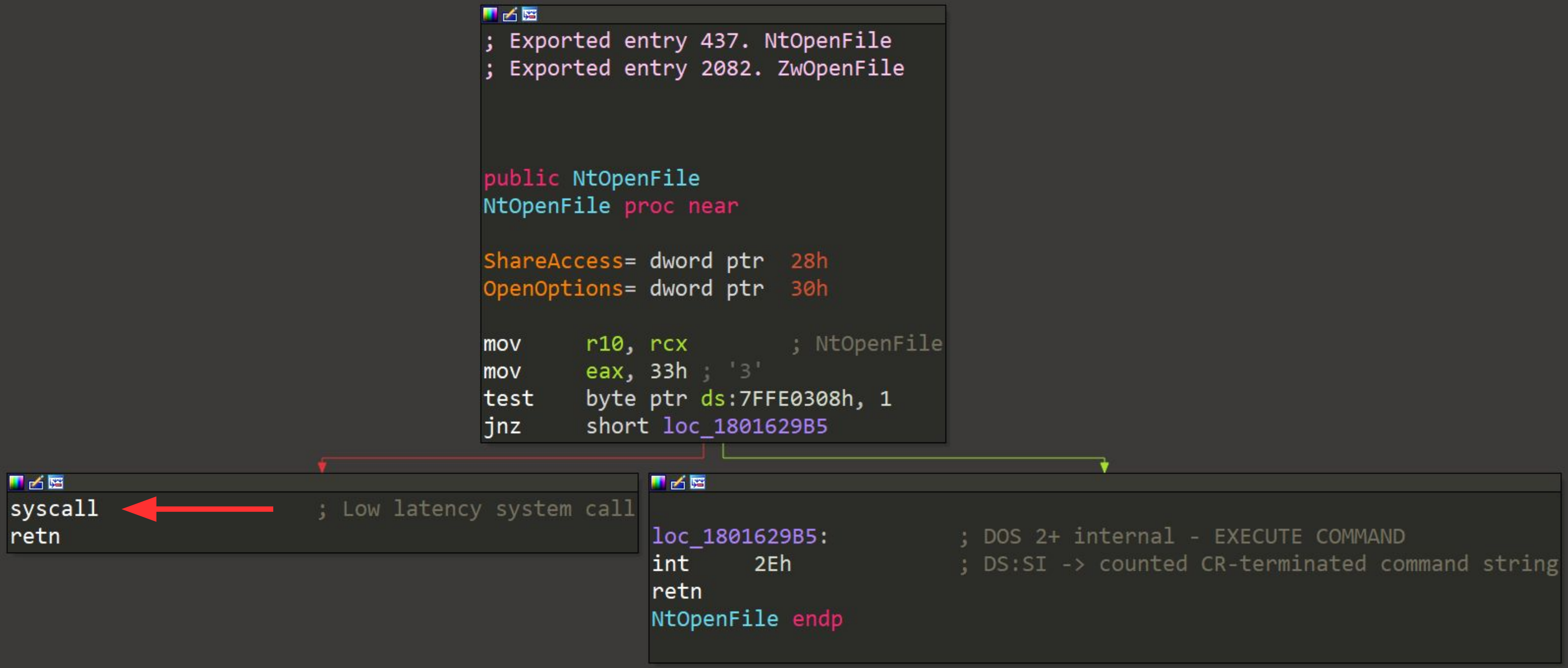


```
; Exported entry 437. NtOpenFile
; Exported entry 2082. ZwOpenFile



public NtOpenFile
NtOpenFile proc near

ShareAccess= dword ptr  28h
OpenOptions= dword ptr  30h

mov     r10, rcx          ; NtOpenFile
mov     eax, 33h ; '3'
test    byte ptr ds:7FFE0308h, 1
jnz     short loc_1801629B5
```

```
syscall                  ; Low latency system call
retn
```

```
loc_1801629B5:              ; DOS 2+ internal - EXECUTE COMMAND
int     2Eh                 ; DS:SI -> counted CR-terminated command string
retn
NtOpenFile endp
```

Syscalls



```
; Exported entry 437. NtOpenFile
; Exported entry 2082. ZwOpenFile



public NtOpenFile
NtOpenFile proc near

ShareAccess= dword ptr  28h
OpenOptions= dword ptr  30h

mov     r10, rcx        ; NtOpenFile
mov     eax, 33h ; '3'
test    byte ptr ds:7FFE0308h, 1
jnz     short loc_1801629B5
```

```
syscall                 ; Low latency system call
retn
```

```
loc_1801629B5:          ; DOS 2+ internal - EXECUTE COMMAND
int     2Eh             ; DS:SI -> counted CR-terminated command string
retn
NtOpenFile endp
```

Syscalls

- 409 regular syscalls → ntdll.dll
- 1474 UI syscalls → win32u.dll

→ Emulator can use syscall instruction hook

Syscall IDs

- e.g. NtOpenFile → 0x33
- IDs can vary between Windows versions

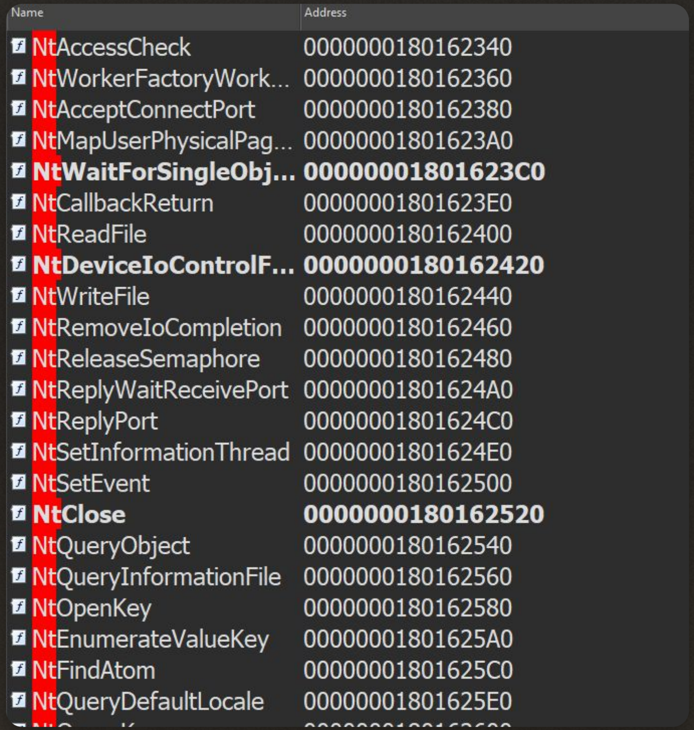
→ How to find Syscall IDs?

Syscalls

- Filter NTDLL exports starting with Nt
- Sort exports by address

→ Order matches Syscall IDs

- ○ NtAccessCheck → 0
- ○ NtWorkerFactoryWork… → 1
- ○ NtAcceptConnectPort → 2
- ○ …



Name	Address
NtAccessCheck	0000000180162340
NtWorkerFactoryWork...	0000000180162360
NtAcceptConnectPort	0000000180162380
NtMapUserPhysicalPag...	00000001801623A0
NtWaitForSingleObj...	**00000001801623C0**
NtCallbackReturn	00000001801623E0
NtReadFile	0000000180162400
NtDeviceIoControlF...	**0000000180162420**
NtWriteFile	0000000180162440
NtRemoveIoCompletion	0000000180162460
NtReleaseSemaphore	0000000180162480
NtReplyWaitReceivePort	00000001801624A0
NtReplyPort	00000001801624C0
NtSetInformationThread	00000001801624E0
NtSetEvent	0000000180162500
NtClose	**0000000180162520**
NtQueryObject	0000000180162540
NtQueryInformationFile	0000000180162560
NtOpenKey	0000000180162580
NtEnumerateValueKey	00000001801625A0
NtFindAtom	00000001801625C0
NtQueryDefaultLocale	00000001801625E0

Syscalls

→ Syscalls now need to be implemented 1 by 1…

- I/O
- Registry
- RPC
- Events
- …

Exception Handling

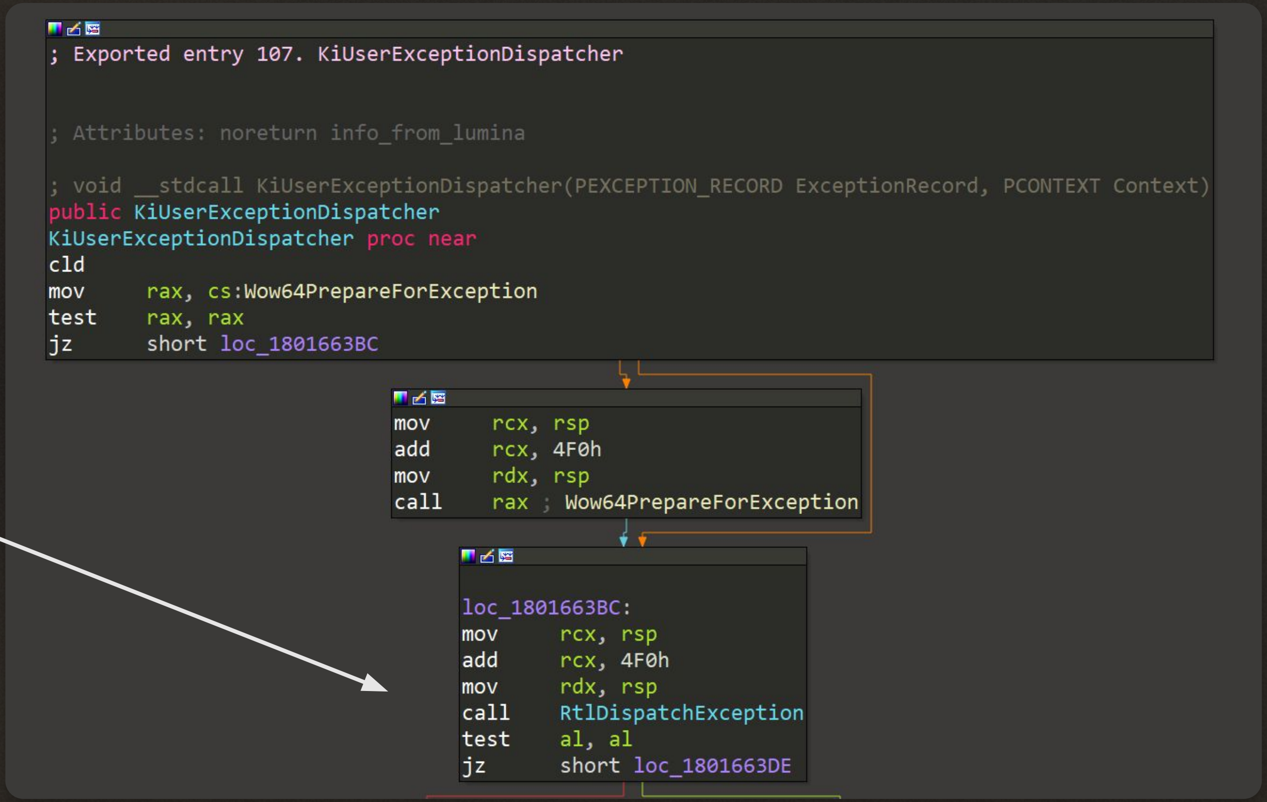
Exception Handling

- critical exceptions are handled by the kernel
  - memory violation
  - invalid instruction
  - breakpoints
  - …
- kernel forwards to the application
  - invokes NTDLL → KiUserExceptionDispatcher

Exception Handling

- receives arguments on stack
    - EXCEPTION_RECORD
    - CONTEXT
    - a few other things
- forwards to

  → RtlDispatchException
- performs unwinding
- calls exception handlers
- ...



```
; Exported entry 107. KiUserExceptionDispatcher


; Attributes: noreturn info_from_lumina

; void __stdcall KiUserExceptionDispatcher(PEXCEPTION_RECORD ExceptionRecord, PCONTEXT Context)
public KiUserExceptionDispatcher
KiUserExceptionDispatcher proc near
cld
mov     rax, cs:Wow64PrepareForException
test    rax, rax
jz      short loc_1801663BC
```

```
mov     rcx, rsp
add     rcx, 4F0h
mov     rdx, rsp
call    rax ; Wow64PrepareForException
```

```

loc_1801663BC:
mov     rcx, rsp
add     rcx, 4F0h
mov     rdx, rsp
call    RtlDispatchException
test    al, al
jz      short loc_1801663DE
```

Exception Handling

Implementation in Emulator

- Unicorn supports hooks for interrupts/violations
- build EXCEPTION_RECORD and CONTEXT on stack
- invoke KiUserExceptionDispatcher from emulator

Threads

Threads

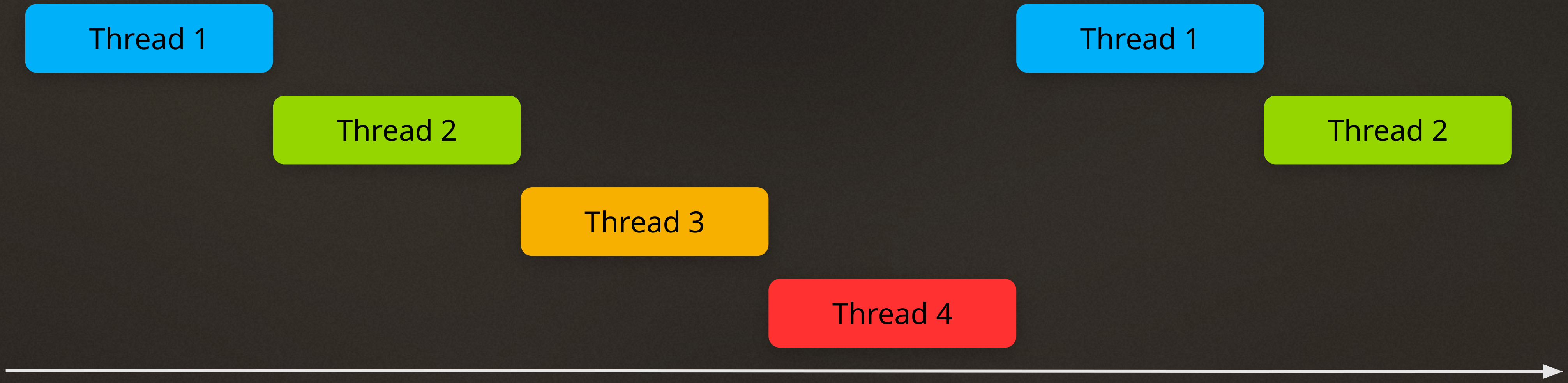
→ Unicorn has no thread awareness

Custom abstraction needed

- Scheduling
- Real Threading

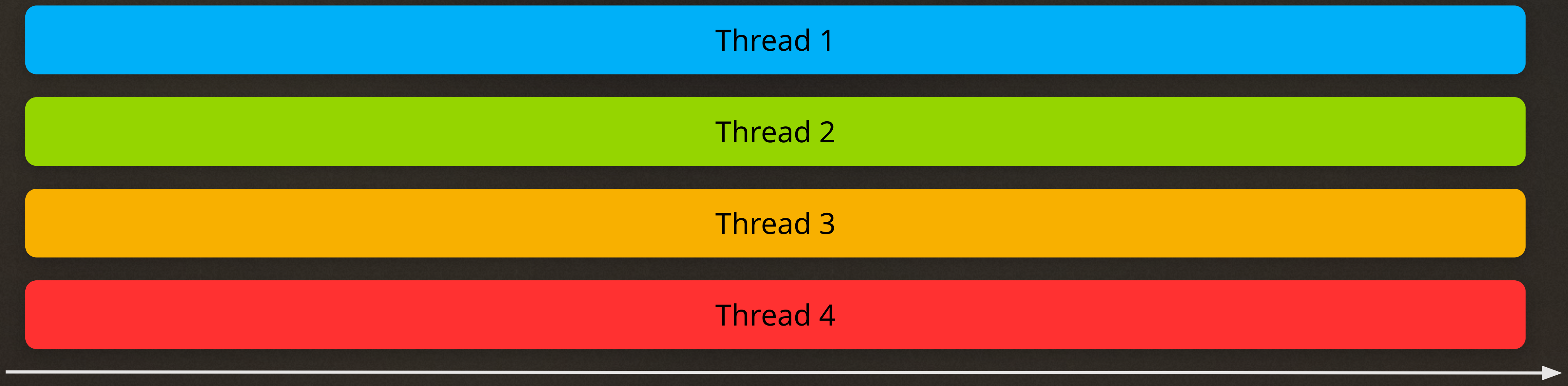
Scheduling

- Round-robin like scheduler
- Threads share emulator
- Context switches after *N* instructions
- Predictable, but slow



Real Threading

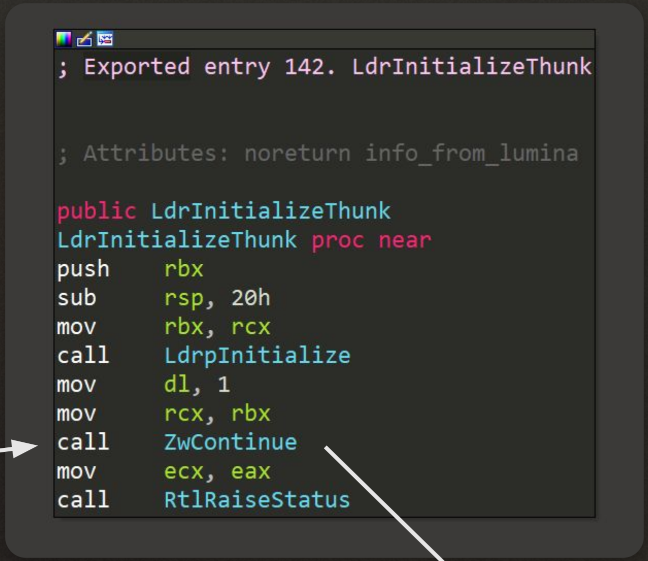
- Multiple threads on the emulator host
- One emulator instance per thread
- Shared memory
- Fast, but unpredictable



Thread Start

LdrInitializeThunk

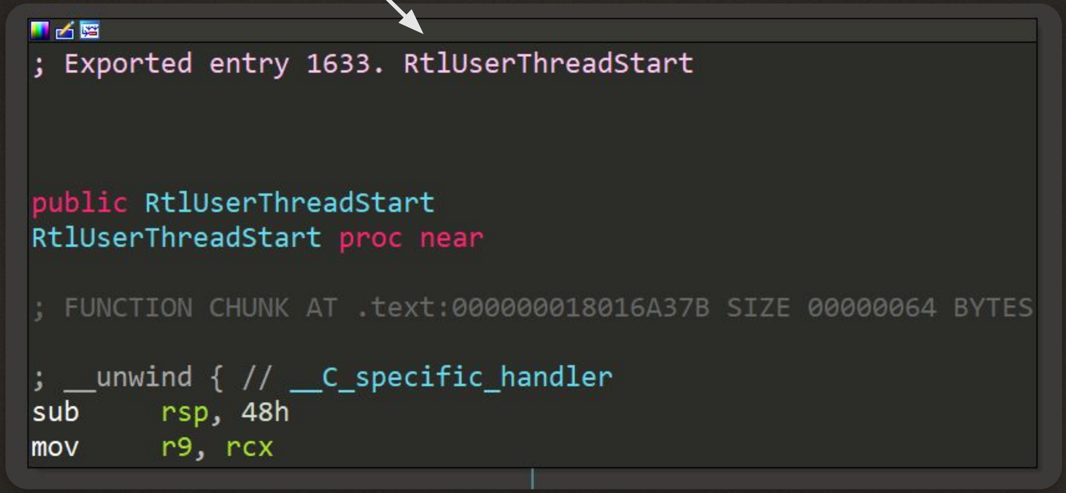
- performs initialization
- runs DllMain & TLS callbacks
- APC call ends with ZwContinue



```
; Exported entry 142. LdrInitializeThunk


; Attributes: noreturn info_from_lumina

public LdrInitializeThunk
LdrInitializeThunk proc near
push    rbx
sub     rsp, 20h
mov     rbx, rcx
call    LdrpInitialize
mov     dl, 1
mov     rcx, rbx
call    ZwContinue
mov     ecx, eax
call    RtlRaiseStatus
```

RtlUserThreadStart

- entry-point of the thread
- runs the thread routine



```
; Exported entry 1633. RtlUserThreadStart



public RtlUserThreadStart
RtlUserThreadStart proc near

; FUNCTION CHUNK AT .text:000000018016A37B SIZE 00000064 BYTES

; __unwind { // __C_specific_handler
sub     rsp, 48h
mov     r9, rcx
```

State Management

State Management

- Emulator should store & restore state
- two variants:
  - serialization
  - snapshots

Serialization

- serializes entire emulation state
  - memory
  - registers
  - mapped modules
  - syscall mapping
  - …
- results in a byte stream

→ can be used e.g. for DRM analysis

Snapshot

- snapshots of volatile emulation state
  - memory → incremental changes
  - registers
  - …
- only works in-process
- extremely fast

→ can be used for fuzzing

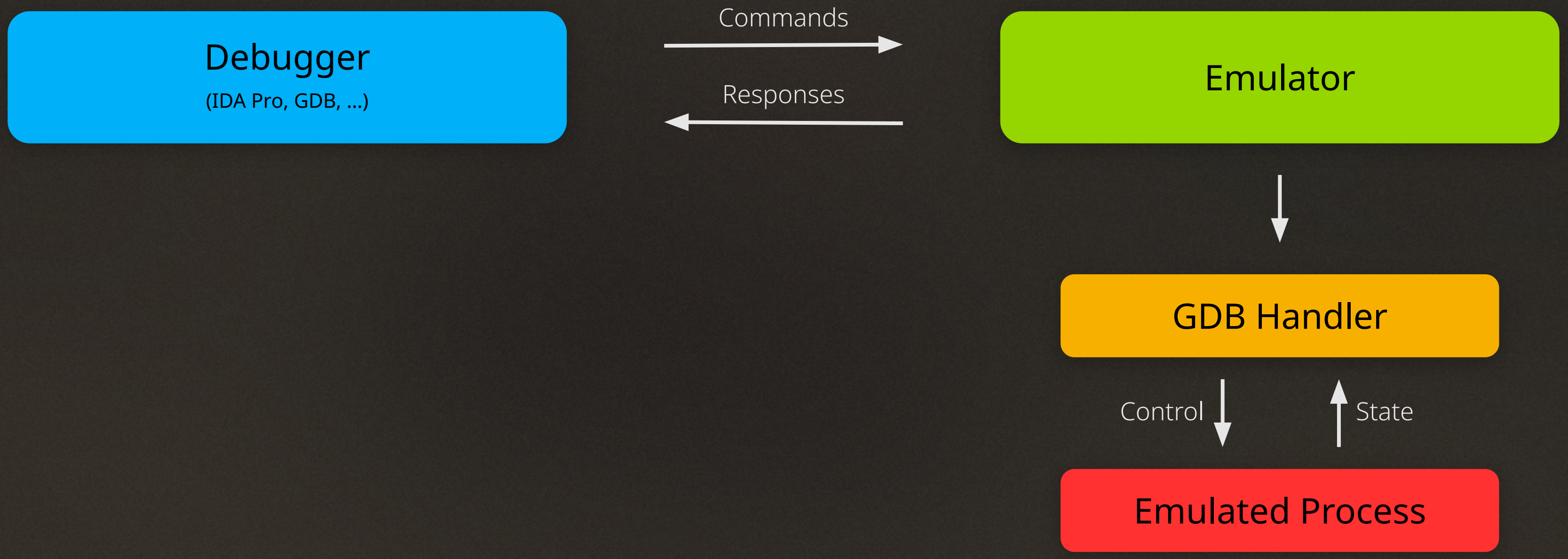
Debugger

Debugger

GDB serial protocol

- simple text protocol over TCP
- registers and memory can be read & written
- supports normal execution and single stepping
- protocol supported by many debuggers
  - GDB
  - LLDB
  - VS Code
  - IDA Pro
  - …

Debugger



Debugger

(IDA Pro, GDB, ...)

Commands

Responses

Emulator

GDB Handler

Control

State

Emulated Process

Demo

Final Words

Final Words

- Emulation is pretty fast → JIT

- Hooks provide analysis interfaces
    - Memory read/write/execute
    - Instruction execution
    - Code coverage

- can be helpful for
    - DRM Analysis
    - Malware Analysis
    - Security Research

Final Words

- still in development
- has no name yet :D
- open source

→ github.com/momo5502/emulator

- a lot of work left todo
  - threading only partially implemented
  - hundreds of syscalls left
  - scripting interfaces → Python / JavaScript / ...
  - eventually replace Unicorn

Thank you

Questions?