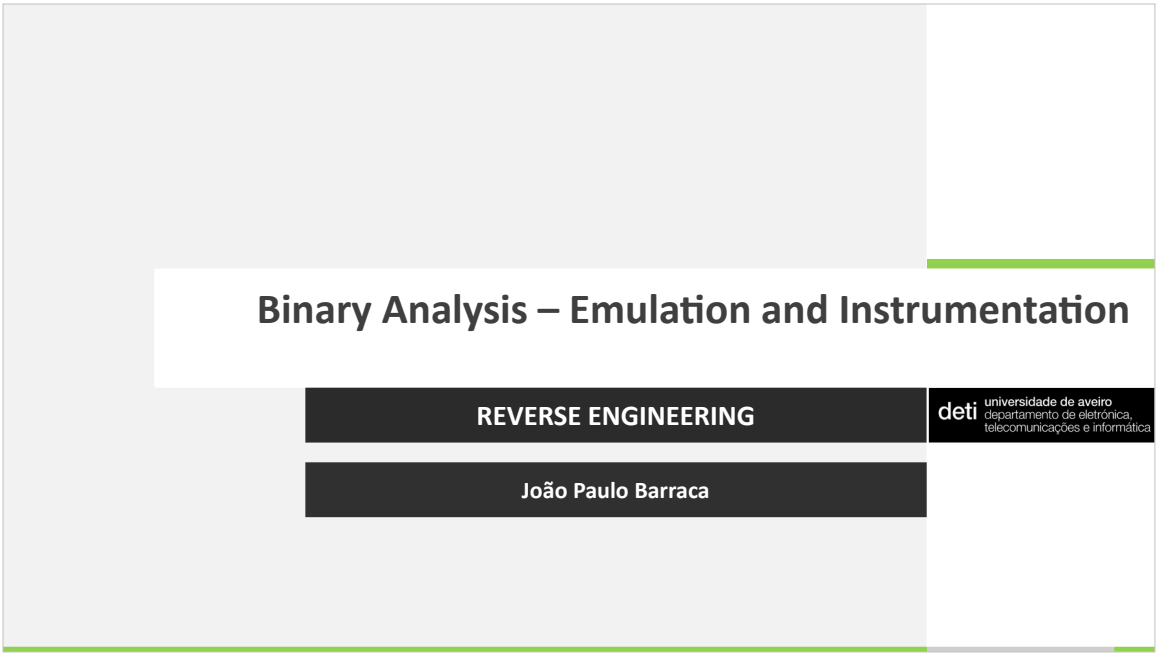


Introduction to the Topic: Welcome to the lecture on Binary Analysis, focusing heavily on Emulation and Instrumentation within the broader context of Reverse Engineering. At the master's level, it is crucial to understand that reverse engineering is not just about translating machine code back to assembly; it is about comprehending the architectural design, algorithmic choices, and the exact runtime behavior of an application without having access to its source code.

The Scope: Emulation and instrumentation are advanced techniques that allow us to observe, manipulate, and track the state of a program precisely as it executes. These techniques are foundational for modern malware analysis, vulnerability research, and security auditing.

Dynamic Binary Analysis

- Allows capturing the dynamic behavior of some code
  - Behavior that depends on external input
  - Data structures and even code revealed during execution time
- Allows runtime validation/evaluation of binary code
  - A program, a firmware, part of a program, a sequence of instructions
  - Under a controlled context
  - On a different (more flexible, or controllable, or safe) environment

Capturing Runtime Reality: Dynamic analysis allows us to capture the dynamic behavior of code that static analysis often misses. This is essential for behaviors heavily dependent on external inputs, environmental variables, or network responses.

Deobfuscation and Unpacking: Many modern binaries, especially malware, employ packers or obfuscation techniques where the actual data structures and malicious code are only decoded and revealed in memory during execution time.

Controlled Evaluation: Dynamic analysis allows for runtime validation and evaluation of diverse binary forms—whether it is an entire program, a piece of firmware, or just a specific sequence of instructions (like a shellcode payload). This evaluation is always done under a controlled, flexible, and safe environment, mitigating the risks of accidental execution of harmful code.

Dynamic Binary Analysis

How

- Load the binary and ***execute*** instructions of the target binary
  - The meaning of “execute” is broader than it may look
- Allow some interaction with the binary while it is running
  - Break the execution at some point
  - Inspect memory and process its content
  - Change memory, either variables or code
  - Execute code in a controlled manner: step by step, in chunks, until a given point

The Concept of "Execution": Dynamic analysis begins with loading and executing the target binary's instructions, but the term "execute" is much broader here. It may mean running the code natively under a debugger, simulating the CPU in software (emulation), or executing it in a highly instrumented virtual machine.

Interactive Control: The true power of dynamic analysis is interactivity. Analysts can set breakpoints to halt execution at critical junctions.

State Manipulation: Once halted, analysts can deeply inspect the memory and process content. More importantly, they can *change* the memory—modifying variables or patching the code itself on the fly. This allows the analyst to force the program down specific execution paths step-by-step or in chunks.

Dynamic Binary Analysis

Approaches

- Analysis of an execution flow can either be **passive** or **active**.
  - Choosing either one or the other has consequences on the soundness, the coverage, etc. of the results
- **Passive analysis**: observation
  - **register values**: return value of functions (**`rax`**), program counter (**`pc`**), stack frame (**`rbp`**, **`rsp`**), etc.
  - **stack inspection**: local variables, input parameters (according to some calling conventions), return address, etc.
  - **heap inspection**: the number of allocated blocks, their content, etc
- **Active analysis**: modification
  - Easily explore paths without finding inputs that actually activate them

Passive vs. Active Analysis: The approach taken dictates the soundness and coverage of the analysis results.

- **Passive Analysis (Observation):** This involves watching the program naturally execute and logging its state. Analysts monitor register values (like return values in `rax`, program counters in `pc`, and stack frames in `rbp`/`rsp`), inspect the stack for local variables, parameters, and return addresses, and monitor the heap to see how memory blocks are allocated and what content they hold.
- **Active Analysis (Modification):** This is a much more aggressive approach. By actively modifying the state (registers, memory, or flags), analysts can easily force the program to explore execution paths without having to reverse-engineer the complex cryptographic inputs or specific network packets that would normally be required to activate them.

Dynamic Binary Analysis

Caveats

- Binary applications **are more powerful and complex**
  - May be written in multiple languages, and have code that runs in a VM
  - May consider code that changes the host system, or is modified in runtime
- Binary analysis of complex applications requires a different toolset
  - The principles will be the same, but the tools will allow fine grained control and isolation
  - Side effects and execution impact may be subtle (remember Meltdown and Spectre)
  - Host systems may be more complex

The Complexity of Modern Binaries: Dynamic analysis is complicated by the fact that modern binary applications are incredibly powerful and complex. A single application might be written in multiple languages and even contain code that runs in an embedded Virtual Machine (like Java or .NET).

Evasive Code: Analysts must account for code that intentionally changes the host system or relies on self-modifying code at runtime to thwart analysis.

Subtle Side Effects: Analyzing such applications requires a toolset that allows fine-grained control and absolute isolation, as side effects and execution impact can be incredibly subtle. A prime example of this subtlety is microarchitectural vulnerabilities like Meltdown and Spectre, which exploit the physical execution context rather than just logical software bugs.

Considerations

(Need for) Stability

- Reversing is significantly more difficult if execution is unstable.
  - Observations are affected by "random" factors, such as multithreaded execution, hardware behavior, user interactions with graphical interface and so on.
  - Applications being reversed should be isolated from external effects are much as possible.
- Determinism in a design results from stable execution of a program run
  - Thus it facilitates debugging and reversing.
  - State may also be deterministically altered for the entire program or for a specific function (fuzzing)
- Logs can be obtained from executions using monitor applications

The Problem of Non-Determinism: Reversing is significantly more difficult if the execution is unstable and non-deterministic. "Random" factors like multithreading, hardware interrupts, Address Space Layout Randomization (ASLR), and asynchronous user interactions can cause the program to behave differently on every run.

Achieving Determinism: To combat this, the application must be isolated from external effects as much as possible. Stable, deterministic execution facilitates accurate debugging and reliable reverse engineering.

Fuzzing: Stability also means the analyst can deterministically alter the state for an entire program or a specific function to perform fuzzing—systematically injecting mutated inputs and using monitor applications to log the execution and identify crashes.

Considerations

(Need for) Save and Replaying

- Reversing may need **tracing** from the current state to the code where a change was produced.
  - It implies moving "back in time".
  - To restore past program state, one must **re-run it** and try to find failure source.
  - This operation may be performed multiple times, **moving backward step-by-step, and then forward**.

- Deterministic replay reconstructs program execution using previously recorded input data.
  - The first program run is used to record these inputs into the log.
  - Then all following runs will reconstruct the same behavior, because the program uses only recorded inputs.
  - Should included all inputs (disk, network)

Time-Travel Debugging: When an analyst discovers a vulnerability or a crash, they often need to trace backward from the current state to the exact instruction that produced the change. This implies moving "back in time".

Deterministic Replay: Because standard execution only goes forward, restoring a past state normally requires restarting the program and hoping to reproduce the failure. This backward-and-forward stepping can be incredibly tedious. Advanced dynamic analysis solves this via deterministic replay, which reconstructs program execution perfectly using a log of previously recorded input data (including disk reads, network packets, and user inputs). Subsequent runs use only this logged data, guaranteeing identical behavior on every replay.

Considerations

(Need for) Safety

- Target binary **may be malicious (… it is always malicious until proven safe)**
- An important aspect of Reversing binaries is malware analysis
  - Malware is way too complex to be analyzed statically
  - But executing the malware may be dangerous
    - **Most important: dangerous in ways unknown to the reverse engineer**
- Solutions must create the adequate isolation boundaries between environments
  - If stability is required, no interactions with the software under analysis
  - Sometimes, isolation must be broken to trigger specific behavior
    - Network connection allowing contact with a C&C address or to download some payload
    - Disk or file presence
    - Whenever possible, such resource should be virtualized

Assuming Malice: A fundamental rule of reverse engineering is that the target binary is always considered malicious until proven safe. Dynamic analysis frequently involves malware that is too heavily obfuscated to be analyzed statically.

Unknown Dangers: Executing this malware natively is highly dangerous, particularly in ways the reverse engineer might not yet understand.

Isolation Boundaries: Therefore, analysts must create strict isolation boundaries between the execution environment and the host machine. While strict stability requires no external interaction, analyzing malware sometimes requires breaking that isolation to observe specific behaviors (e.g., allowing it to contact a Command & Control server or checking for specific files). Whenever possible, these external resources (like the network or disk) should be mocked or virtualized to maintain safety.

Considerations

(Need for) Support of Heterogeneous Architectures

- Dynamic analysis requires the execution of the program under analysis.
- An analyst will mostly run on an Intel x86 64bits computer (a COTS laptop/server)
  - Most embedded devices are ARM, which has several variants
  - Microcontrollers frequently use 8085, AVR or PIC architectures (MIPS)
  - Several specialty SOCs use custom architectures (the list is large…)
  - Several binary formats are popular: ELF, PE, DWARF and then many others from IoT
- Frameworks must be extensible in order to support a wide range of architectures
  - And the related interfaces and customizations
  - While minimizing the need for new tools

The Architecture Gap: Dynamic analysis necessitates executing the program. However, security analysts typically work on standard Intel x86 64-bit hardware, while the targets they are analyzing might not.

Diverse Ecosystems: The modern tech landscape includes embedded devices (predominantly ARM variants), microcontrollers (using 8085, AVR, or MIPS architectures), and specialized System-on-Chips (SOCs) with custom architectures. Furthermore, binaries come in many formats (ELF, PE, DWARF, and proprietary IoT formats).

Extensible Frameworks: Because of this heterogeneity, dynamic analysis frameworks must be highly extensible to support cross-architecture execution via emulation. They must support diverse interfaces and customizations to handle these varying formats without forcing analysts to build entirely new tools for every new device they encounter.

Considerations

(Need for) Support of Peripherals and external entities

- Reversing an application with external interactions may require the existence of the related entities
  - Web sites, servers in fixed/dynamic IP addresses
  - Common physical devices for user input, storage, …
  - Exotic external devices communicating through known or unknown buses
  - Hardware Dongles
- Need to recreate the set of devices/entities required to trigger a specific path
  - Frequently resorts to device emulation with mock software constructs

The Challenge of External Dependencies: Reversing an application that relies on external interactions often requires those related entities to be present for the program to function naturally. This includes web sites, servers with fixed or dynamic IP addresses, and common physical devices for user input or storage.

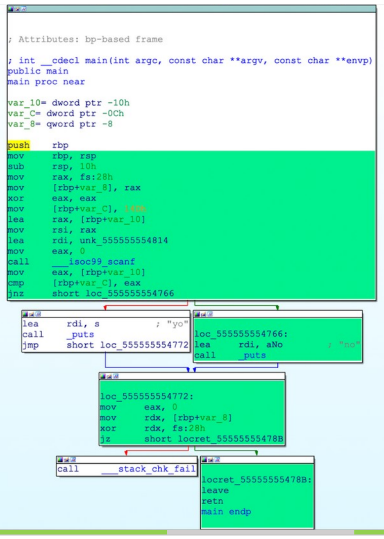
Exotic and Proprietary Hardware: The complexity increases significantly when dealing with exotic external devices communicating through specialized or unknown buses, or legacy protection mechanisms like hardware dongles. If the binary expects a specific cryptographic response from a dongle to decrypt its core logic, analyzing it natively becomes impossible without the hardware.

Mocking and Emulation: Analysts must recreate the set of devices or entities required to trigger a specific execution path. This frequently resorts to device emulation using mock software constructs (e.g., using a local DNS server to sinkhole C2 traffic, or writing Python scripts in emulation frameworks to simulate the memory-mapped I/O of a specific hardware peripheral).

Considerations

(Need for) Context manipulation (instrumentation)

- The main limitation of a dynamic approach is **coverage**.
  - Every path that is not covered by the instrumented executions cannot be analyzed.
  - This limitation can be slightly reduced by performing active instrumentation, and in particular by **forcing conditional branching**



Example of Intel PIN coverage output provided to IDA
https://hex-rays.com/products/ida/support/tutorials/pin/pin_tutorial.html

The Coverage Limitation: The main limitation of any dynamic approach is *coverage*. Unlike static analysis, which can theoretically parse all code paths, dynamic analysis only observes the single path taken during execution. Every path that is not covered by the instrumented execution cannot be analyzed.

Forcing Execution Paths: This severe limitation can be slightly reduced by performing active instrumentation. By manually intervening in the process state, analysts can force conditional branching (e.g., flipping the Zero Flag right before a `jz` instruction) to make the program execute the "unseen" path, even if the required cryptographic input or complex state was never actually met.

Considerations

(Need for) Context manipulation (instrumentation)

- A reversing task will need to observe **structure** and **behavior**
  - The analysis should have **enough coverage to recover the adequate level of detail**
  - But while static analysis aims for wide coverage, dynamic analysis aims for focus
  - What if a specific course of execution is not triggered?
  - **Results of dynamic analysis are dependent on the context of the execution**

- Context manipulation allows setting the adequate state to trigger a specific flow of execution, **increasing the reversing coverage**
  - Achieved by careful manipulation of execution state, registers and memory content
  - Problems:
    - May lead to the recovery of an incorrect design as the found flow may be a decoy!
    - May lead to the recovery of artificial vulnerabilities, that do not really exist

Focus vs. Coverage: A reversing task requires observing both structure and behavior to recover an adequate level of detail. However, a fundamental dichotomy exists: while static analysis aims for wide coverage (seeing everything), dynamic analysis aims for focus (seeing exactly what happens in a specific scenario).

The Context Dependency Problem: Results of dynamic analysis are highly dependent on the context of the execution. What if a specific course of execution (like a malicious payload dropping) is not triggered because the environment isn't right?

Risks of Manipulation: Context manipulation allows setting the adequate state to trigger specific flows (via careful manipulation of execution state, registers, and memory). However, this introduces analytical risks:

- **Decoys:** It may lead to the recovery of an incorrect design, as the forced flow might be a decoy intentionally planted by the malware author.
- **Artificial Vulnerabilities:** Forcing states that are mathematically or logically impossible in natural execution may lead to the discovery of artificial vulnerabilities that do not really exist.

Considerations

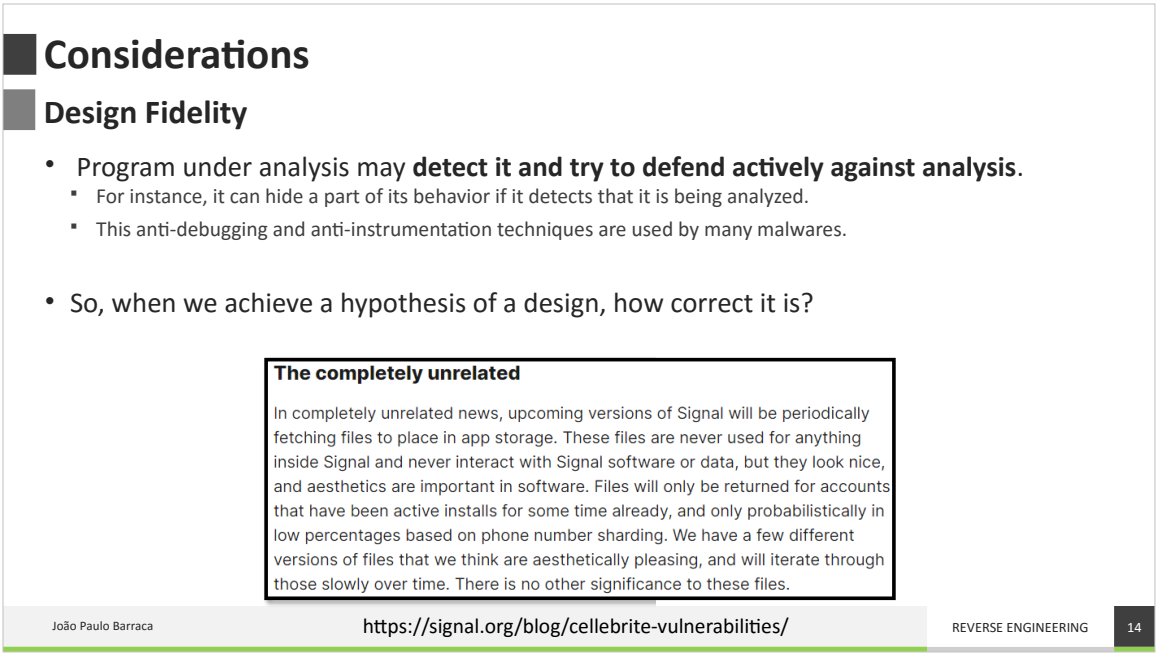
Context manipulation (instrumentation)

- **Live patching**: modifying RAM in a debugger/controlled environment
- **File Patching**: alter binaries files to replace their content
- **Binary Instrumentation**: Real time, automated modification

Live Patching: This involves modifying RAM while the program is in a debugger or controlled environment. It is an immediate, interactive way to change variables or bypass checks temporarily, but it only lasts for that specific execution session.

File Patching: This is a permanent approach where the analyst alters the binary file on disk to replace its content (e.g., overwriting a conditional jump with `NOP` instructions). This ensures the patch persists across reboots but modifies the file's original signature (like its hash).

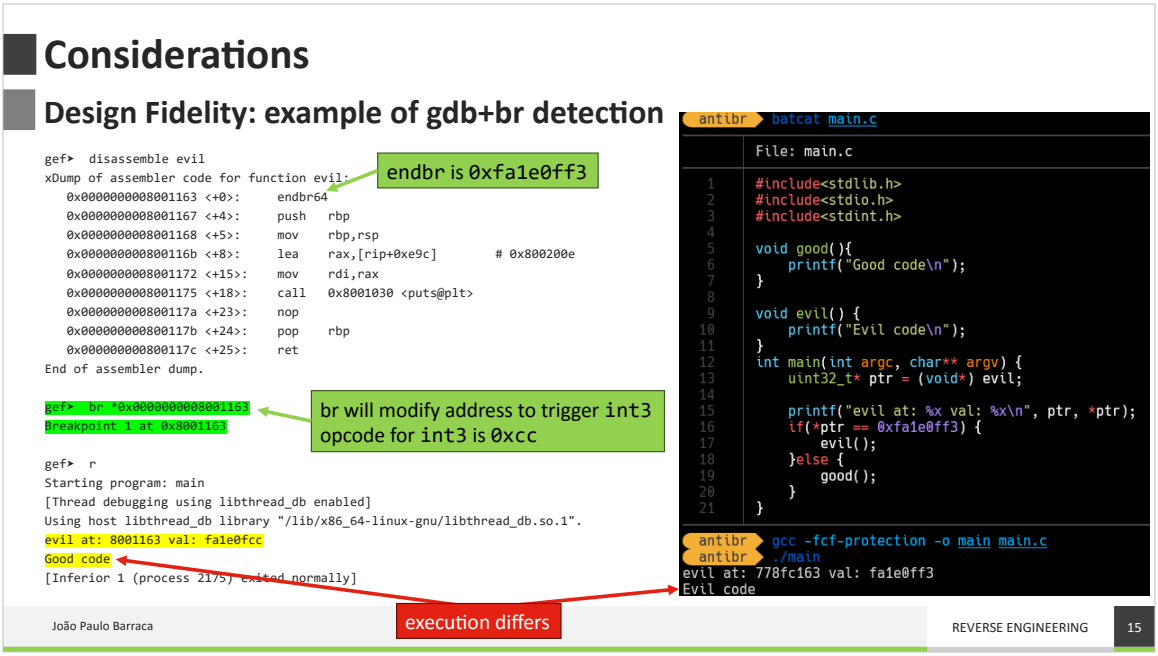
Binary Instrumentation: This is real-time, automated modification. Using frameworks like Intel PIN or DynamoRIO, analysts can script automated changes that are injected into the binary seamlessly as it runs, combining the persistence of file patching with the flexibility of live patching



The Observer Effect in Reversing: A program under analysis may detect that it is being monitored and try to actively defend against the analysis.

Evasive Behaviors: For instance, sophisticated malware can hide a part of its behavior if it detects an analysis environment. These anti-debugging and anti-instrumentation techniques are highly prevalent in modern malicious code.

The Epistemological Question: Therefore, when we achieve a hypothesis of a design through dynamic analysis, we must always ask: *how correct is it?*. If the program altered its execution flow because it detected our debugger, our entire understanding of its primary purpose might be flawed.



Detecting Breakpoints: This slide showcases a concrete example of anti-analysis where a program detects GDB breakpoints.

The `int3` Artifact: When an analyst uses the `br` (break) command in GDB, the debugger temporarily overwrites the instruction at that address with the opcode `0xCC`, which corresponds to the `int3` (interrupt 3) instruction.

The Execution Divergence: The program (in this example, a function named `evil`) is designed to inspect its own memory. If it sees the `0xCC` opcode where a valid instruction (like `endbr64` which is `0xfa1e0ff3`) should be, it knows a debugger is attached. Consequently, the execution diverges, outputting a decoy message or exiting, thwarting the analysis.

Dynamic Binary Analysis of Binaries

Processes

- **Tracing**: Logging external events.
- **Debugging**: Interactive, step-by-step execution control.
- **Sandboxing**: Confined execution for behavioral observation.
- **Emulation**: Software-based simulation of hardware.
- **Instrumentation**: Automated, real-time code modification.

The Spectrum of Techniques: Dynamic analysis is an umbrella term encompassing several distinct processes, each with varying levels of complexity, isolation, and depth.

Tracers

… Already briefly discussed in previous lectures

- **Tracers execute a binary, logging information about function and system calls**
- Binary is executed in the analyst's system
  - That is: In a VM!
- Tracer adds hooks to application or kernel to gain information about execution
  - Access to files, packets sent, registry access
- No confinement or security measures in place
  - Actually, there may be no interaction between the tracer and the application
    - Tracer monitors system through kernel debug interfaces

Functionality: Tracers execute a binary while logging information about function calls and system calls. The binary executes in the analyst's system (ideally within a Virtual Machine).

How They Hook: Tracers add hooks to the application or kernel to gain information about the execution, such as files accessed, packets sent, or registry keys modified.

Kernel Interfaces: Notably, there may be no direct interaction between the tracer and the application itself; instead, the tracer monitors the system silently through kernel debug interfaces. However, it is crucial to remember that tracing inherently offers *no confinement or security measures*.

Tracers

… Already briefly discussed in previous lectures

- **Limitations:**
  - No isolation, no capability to analyze malicious or harmful code
  - Can only inspect interactions between the application and the external environment
  - Host environment must be compatible with the target binary
    - No possibility of analyzing windows binaries on linux, vice-versa, embedded systems on windows, etc…

- **Linux**: `ltrace, strace (ptrace), bpftrace, wireshark, valgrind, cachegrind, callgrind, helgrind`
- **Windows**: `process monitor, wireshark`

Security Risks: Tracing natively provides no isolation, meaning there is no safe capability to analyze malicious or harmful code directly on a host machine.

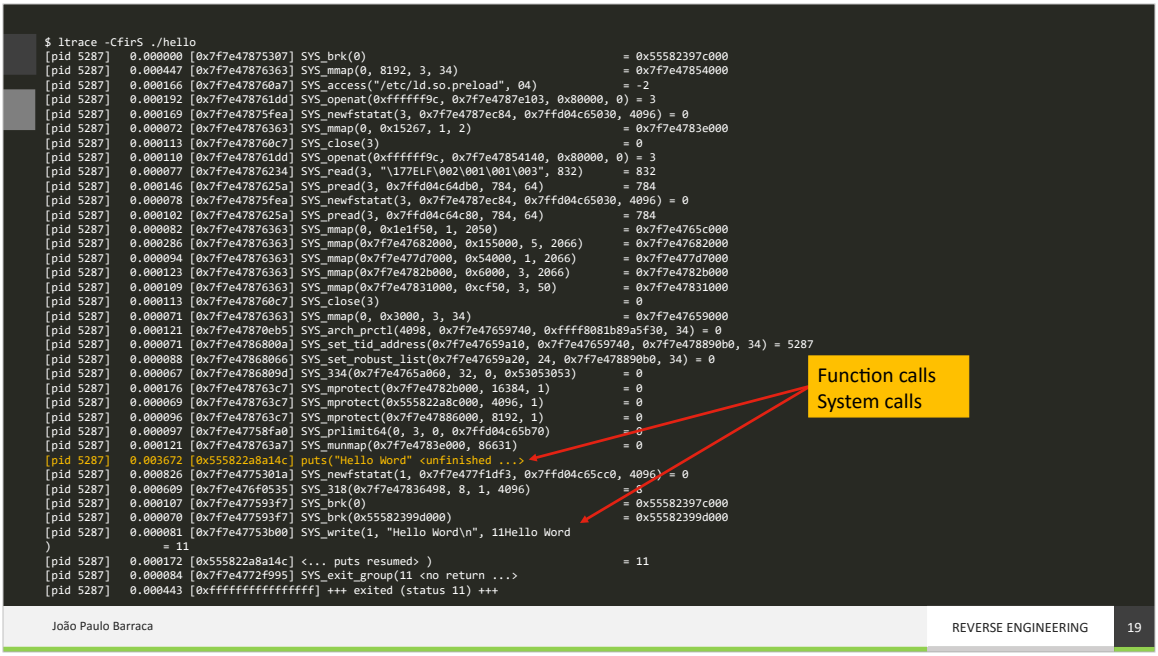
Analysis Depth: Tracers can only inspect interactions between the application and the external environment (system calls, library calls), blinding the analyst to internal algorithmic computations.

Architecture Constraints: The host environment must be strictly compatible with the target binary. You cannot trace Windows binaries natively on Linux, or embedded ARM systems on x86 Windows.

Common Tooling:

Linux: Includes `ltrace` (library traces), `strace`/`ptrace` (system calls), `bpftrace`, `wireshark`, and the Valgrind suite (`cachegrind`, `callgrind`, `helgrind`).

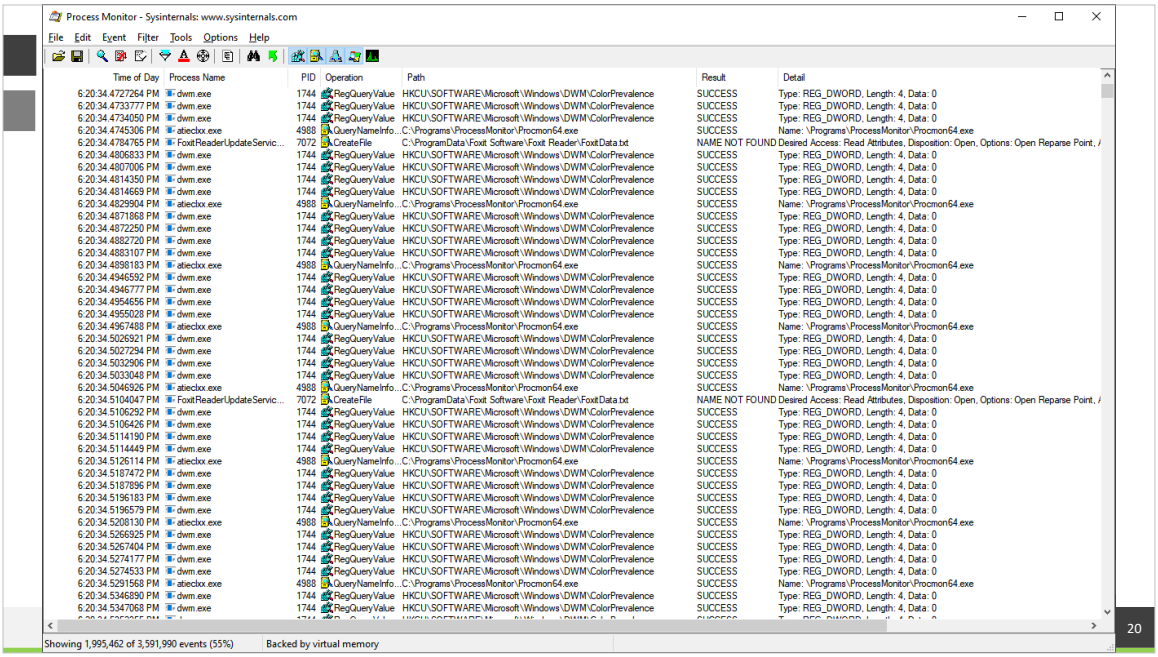
Windows: Process Monitor (ProcMon) and Wireshark.

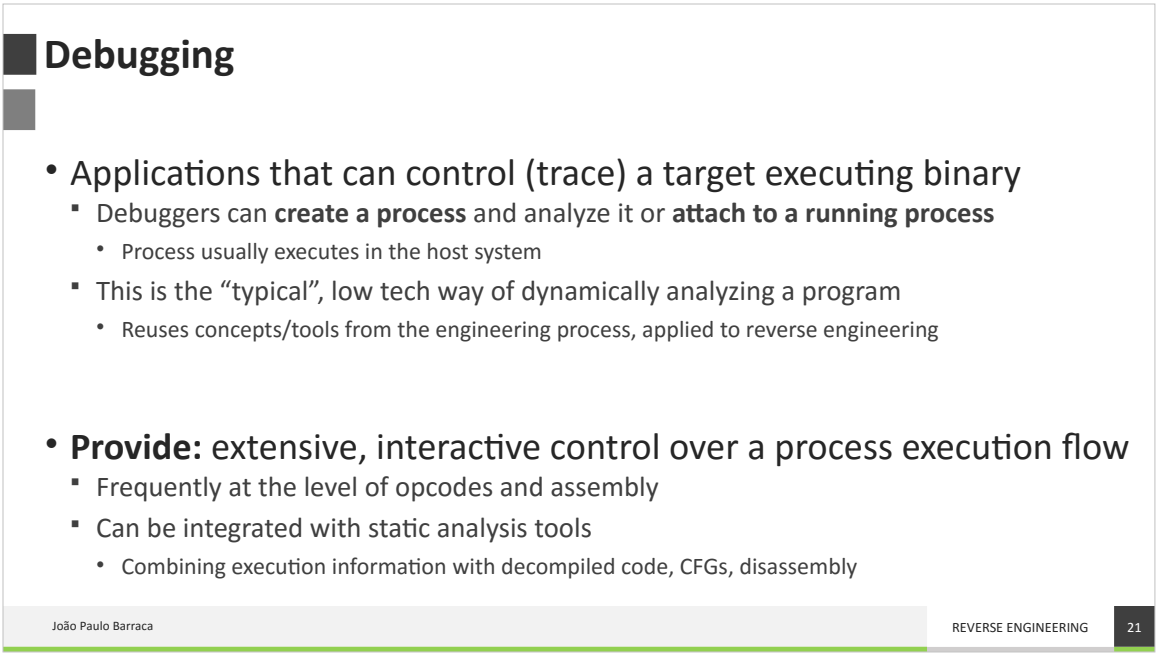


Interpreting the Output: This slide provides a raw terminal output of running `$ ltrace -CfirS ./hello`.

System and Function Calls: The output demonstrates how `ltrace` captures the exact sequence of events during program startup. Students can observe the program mapping memory (`SYS_mmap`), checking for preloaded libraries (`SYS_access("/etc/ld.so.preload")`), opening files (`SYS_openat`), and eventually executing the standard C library function `puts("Hello Word")` before making the `SYS_write` syscall to print the output to standard out.

Practical Use: This granular log is invaluable during initial triage to see exactly what libraries a binary is dynamically loading and what files it requires to function, without needing to open a disassembler.


Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help
Time of Day | Process Name | PID | Operation | Path | Result | Detail
6:20:34.4727264 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4733777 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4734050 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4745306 PM | atieclxx.exe | 4988 | QueryNameInfo... | C:\Programs\ProcessMonitor\Procmon64.exe | SUCCESS | Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.4784765 PM | FoxitReaderUpdateServic... | 7072 | CreateFile | C:\ProgramData\Foxit Software\Foxit Reader\FoxitData.txt | NAME NOT FOUND | Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, A
6:20:34.4806833 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4807006 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4814350 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4814669 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4829904 PM | atieclxx.exe | 4988 | QueryNameInfo... | C:\Programs\ProcessMonitor\Procmon64.exe | SUCCESS | Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.4871868 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4872250 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4882720 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4883107 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4898183 PM | atieclxx.exe | 4988 | QueryNameInfo... | C:\Programs\ProcessMonitor\Procmon64.exe | SUCCESS | Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.4946592 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4946777 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4954656 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4955028 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4967488 PM | atieclxx.exe | 4988 | QueryNameInfo... | C:\Programs\ProcessMonitor\Procmon64.exe | SUCCESS | Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5026921 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5027294 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5032906 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5033048 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5046926 PM | atieclxx.exe | 4988 | QueryNameInfo... | C:\Programs\ProcessMonitor\Procmon64.exe | SUCCESS | Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5104047 PM | FoxitReaderUpdateServic... | 7072 | CreateFile | C:\ProgramData\Foxit Software\Foxit Reader\FoxitData.txt | NAME NOT FOUND | Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, A
6:20:34.5106292 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5106426 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5114190 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5114449 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5126114 PM | atieclxx.exe | 4988 | QueryNameInfo... | C:\Programs\ProcessMonitor\Procmon64.exe | SUCCESS | Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5187472 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5187896 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5196183 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5196579 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5208130 PM | atieclxx.exe | 4988 | QueryNameInfo... | C:\Programs\ProcessMonitor\Procmon64.exe | SUCCESS | Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5266925 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5267404 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5274177 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5274533 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5291568 PM | atieclxx.exe | 4988 | QueryNameInfo... | C:\Programs\ProcessMonitor\Procmon64.exe | SUCCESS | Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5346890 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5347068 PM | dwm.exe | 1744 | RegQueryValue | HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence | SUCCESS | Type: REG_DWORD, Length: 4, Data: 0
Showing 1,995,462 of 3,591,990 events (55%)
Backed by virtual memory



Interactive Control: Debuggers are applications designed specifically to control and trace the execution of a target binary. Unlike tracers that simply log events, debuggers allow an analyst to completely manipulate the process flow.

The "Low Tech" Approach: While we will discuss advanced techniques like hypervisor-level instrumentation later, standard debugging is the foundational, "low tech" method of dynamic analysis. It directly reuses concepts and tools originally built for software engineering and applies them to reverse engineering.

Granular Visibility: A debugger provides extensive control over execution flow at the lowest levels—typically at the opcode and assembly level. You can see the exact contents of the CPU registers and memory space before and after every single instruction executes.

Integration with Static Analysis: Modern debuggers do not operate in a vacuum. They are increasingly integrated with static analysis tools (like decompilers and disassemblers). This allows the analyst to map dynamic execution states back to the statically recovered Control Flow Graphs (CFGs) and pseudo-C code, massively accelerating the understanding of complex algorithms.

Debugging

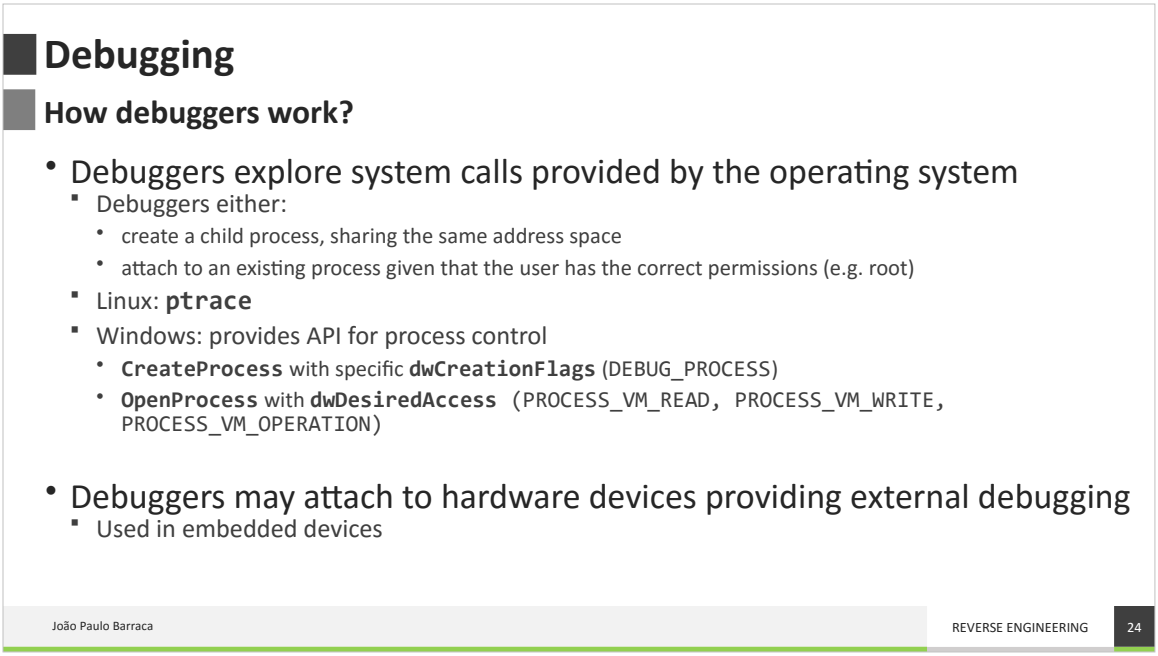
Limitations

- Debugging **can be detected** and subverted by the target application
  - Especially popular in malware and DRM systems
- Target application must be executed in a full hosted environment
  - Without isolation measures, this provides a serious security risk
  - Remote debugging may be used to circumvent this limitation
- Host system architecture must match the target binary architecture
  - Binary is loaded to the host system as a standard process
  - No debugging of windows in Linux, ARM or MIPS in x86
  - No direct way of debugging shellcode or a binary blob (e.g firmware).

The Evasion Problem: The most significant limitation of standard debugging is that it is highly detectable. Because the debugger controls the target via standard OS APIs, the target application can easily query the OS to check if it is being debugged, or look for artifacts left by the debugger (like software breakpoints). This is incredibly common in malware and Digital Rights Management (DRM) software.

Security Risks: Standard debugging typically requires executing the binary as a process directly on the host operating system. Without strict isolation measures (like running the debugger inside a heavily sandboxed Virtual Machine), executing malware is extremely dangerous.

The Architecture Gap: A standard debugger expects to run on the same architecture as the target binary. You cannot natively debug an ARM binary on an x86 host machine, nor can you easily debug raw shellcode or a raw firmware blob (which lacks an executable file format like ELF or PE) without specialized setups or emulators.



OS-Level Mediation: Debuggers do not magically control the CPU; they rely entirely on specialized system calls provided by the host operating system to manage process execution.

Process Attachment: A debugger can either create the target as a new child process (sharing the same address space context) or attach to an already running process (provided the analyst has sufficient privileges, like `root` or `SeDebugPrivilege` on Windows).

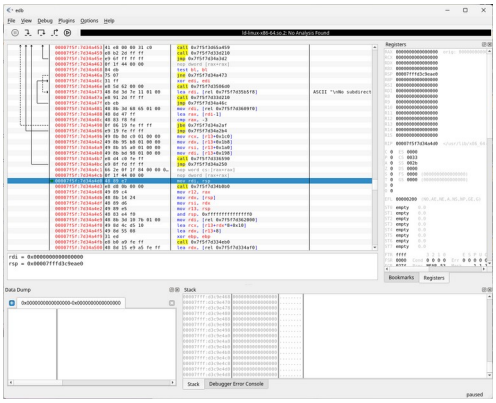
The Underlying APIs: * **Linux:** The magic happens via the `ptrace` system call. It allows one process to examine and change the core image and registers of another process.

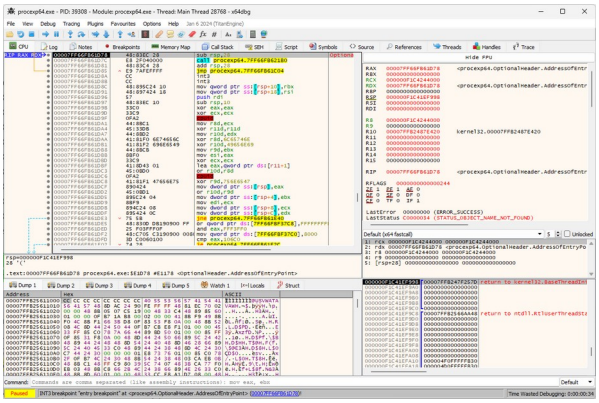
- **Windows:** Uses specific API flags. `CreateProcess` is called with the `DEBUG_PROCESS` flag. To attach to a running process, `OpenProcess` is used with access rights like `PROCESS_VM_READ` and `PROCESS_VM_WRITE`.

Hardware Debugging: For embedded devices, software debuggers often attach to physical hardware interfaces (like JTAG or SWD) via a debugging probe, allowing control over the physical microcontroller itself.

Debugging

edb and x86dbg



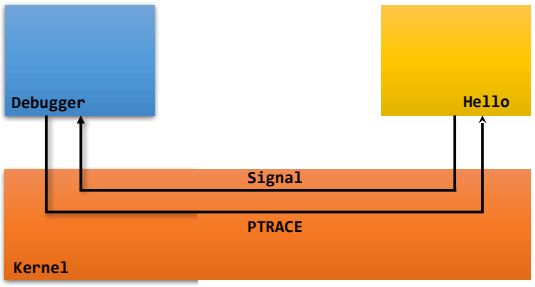


(This slide visually demonstrates modern debugger interfaces.)

Notice the standard layout required for effective dynamic analysis:

1. **Disassembly View:** Showing the assembly instructions where the instruction pointer (`EIP`/`RIP`) currently is.
2. **Registers View:** Showing the live, real-time values of all CPU registers, frequently highlighting which ones changed during the last instruction.
3. **Hex Dump View:** Allowing the analyst to inspect raw memory addresses.
4. **Stack View:** Crucial for understanding function parameters, local variables, and the call chain.

Debugging



Debugger set breakpoints which Trigger SIGTRAP, returning control to the debugger.

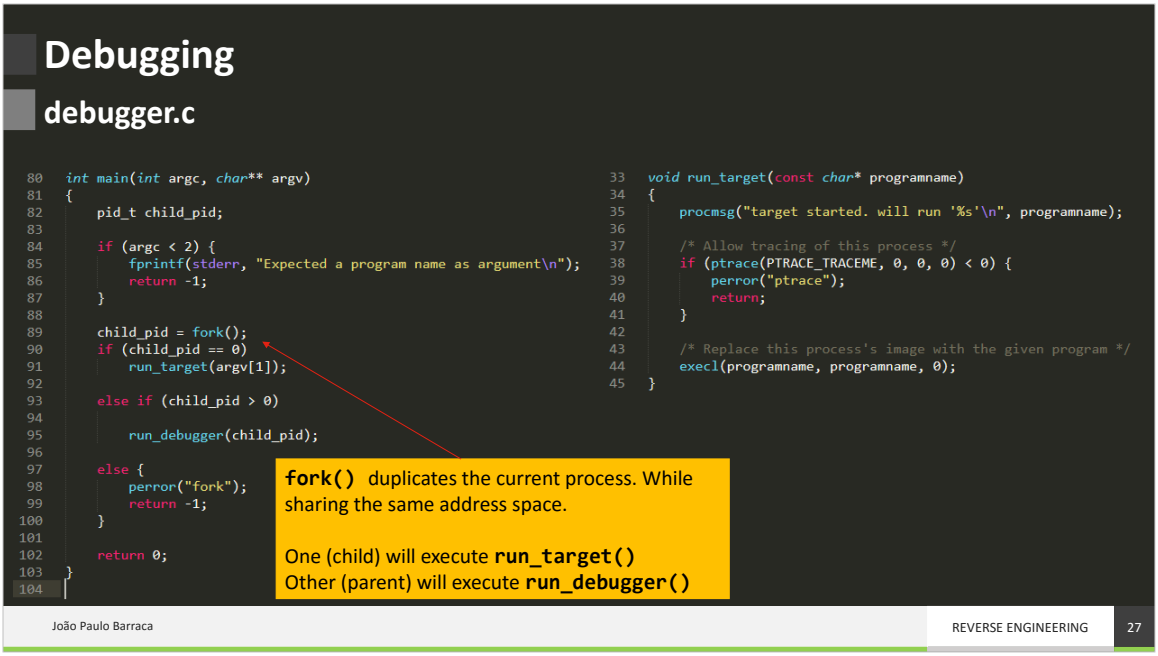
Patching the code with **0xCC** or using Hardware breakpoints (through **PTRACE**)

The Execution Loop: How does a debugger actually stop a program? On Linux, it relies on the `ptrace` interface and system signals.

Software Breakpoints (`int3`): When you set a breakpoint at a memory address, the debugger saves the original opcode at that address and replaces it with a special opcode: `0xCC` (the `int3` instruction on x86).

The Trap: When the CPU executes `0xCC`, it generates an interrupt (a trap). The Kernel catches this trap, suspends the target process, and sends a `SIGTRAP` signal to the debugger process. The debugger then restores the original opcode, allowing the analyst to inspect the state.

Hardware Breakpoints: Alternatively, debuggers can use special Debug Registers (DR0-DR7 on x86) provided by the CPU itself. These can break on execution, but more importantly, they can break on memory *reads* or *writes* without modifying the code itself, making them harder for malware to detect.

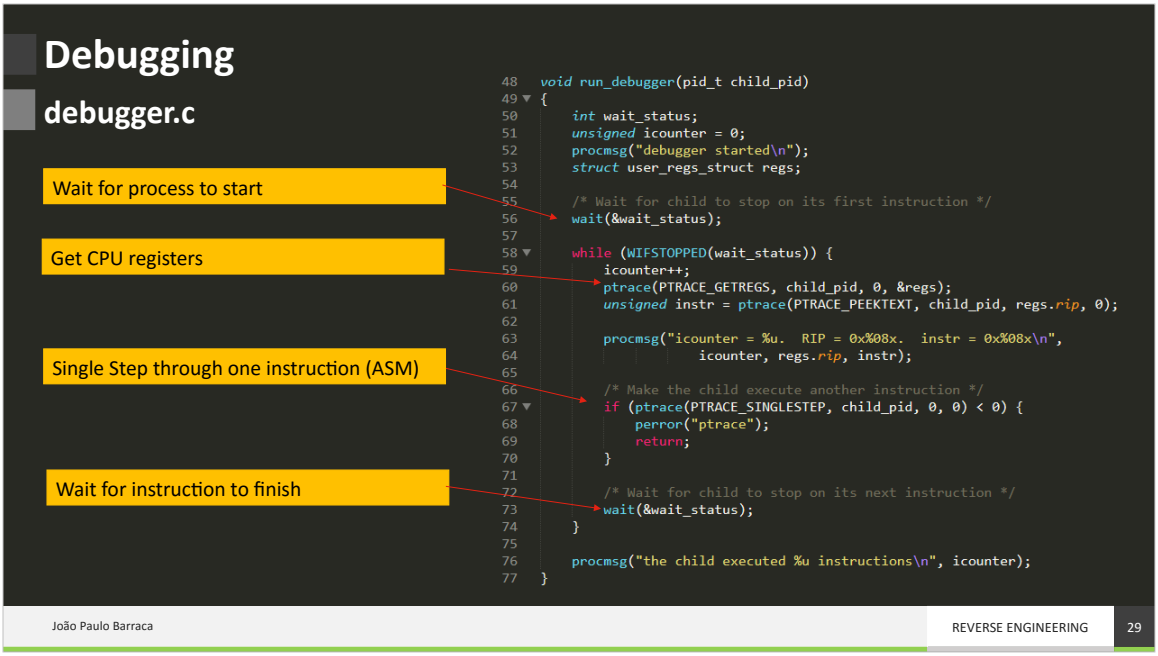


The `fork()` call: To trace a program from its very first instruction, the debugger must spawn it. We start by calling `fork()`. This creates a child process that is an exact duplicate of the parent debugger process, sharing the same address space.

Branching Logic: We use the return value of `fork()` to branch the code. The child process will prepare to execute the target binary (`run_target()`), while the parent process will act as the controller (`run_debugger()`).

The Child's Role: Inside the child process, the first thing it must do is call `ptrace(PTRACE_TRACEME, 0, 0, 0)`. This critical call tells the Linux kernel, "I am allowing my parent process to trace and control me."

Replacing the Image: Once tracing is enabled, the child calls `execl()`. This system call overwrites the current process's memory space with the actual binary we want to analyze (the target from disk). Because `PTRACE_TRACEME` was set, the kernel will immediately halt the child process right before the very first instruction of the new binary executes, and it will notify the parent.



The Parent's Role: Meanwhile, the parent process enters a `wait()` loop, waiting for the child to change state (e.g., waiting for the kernel to signal that the child hit a breakpoint or finished loading).

State Inspection: Once the child is halted, the parent can use `ptrace(PTRACE_GETREGS, ...)` to extract the exact state of the child's CPU registers.

Execution Control: The parent can then command the child to execute just a single assembly instruction using `ptrace(PTRACE_SINGLESTEP, ...)` and wait again. This forms the fundamental stepping loop of every low-level debugger.

Sandboxing

- Sandboxing improves the control that debuggers provide
  - Creation of a distinct execution environment
    - Different libraries? Restricted view of the filesystem (minimal access to files)
  - Isolate some actions, providing some safety to analyze malicious applications
- Implementation: lightweight virtual machines or namespaces/containers
  - Supported my mechanisms of the Operating System or additional tools
  - Tools: `sandboxie, fame, pyReBox, cuckoo, joe, any.run`
- An agent monitors interactions of the application inside the environment and may allow instrumentation
  - File access, network communication
  - Remote debugging

Moving Beyond Standard Debuggers: As we established, standard debuggers are highly detectable and run natively on the host, which is a major security risk when dealing with malware. Sandboxing solves this by improving the control and safety debuggers provide.

Confinement and Safety: A sandbox creates a distinct, isolated execution environment. It provides a restricted view of the filesystem (often a dummy OS), isolates potentially harmful actions, and allows analysts to observe malicious applications without infecting their host machine.

Implementation Techniques: Sandboxes are typically implemented using lightweight virtual machines, namespaces, or containers. These are supported by OS-level mechanisms or third-party tools.

The Tooling Landscape: Popular sandboxing tools include Sandboxie, FAME, pyREBox, Cuckoo, Joe Sandbox, and ANY.RUN.

The Monitoring Agent: Inside the sandbox, a specialized agent continuously monitors the application's interactions with the environment. It logs file access, registry modifications, and network communications, while frequently allowing for remote debugging and active instrumentation.

Emulators

- Emulators are common backends for secur sandboxes
  - May provide much better isolation as the guest and host environments are distinct
    - Kernel is not shared, hardware is emulated
  - Tools: **QEMU**, Virtualbox, Vmware
- Emulation types
  - Full system emulation
  - User mode emulation

The Backbone of Sandboxes: While some sandboxes rely on OS-level virtualization, true emulators are common backends for the most secure sandboxes.

Hardware-Level Simulation: Emulators provide a much higher degree of isolation because the guest and host environments are completely distinct. The kernel is not shared between them; instead, the entire underlying hardware is simulated in software.

Popular Emulators: Standard tools in this space include QEMU, VirtualBox, and VMware.

Granularity of Emulation: For binary analysis, we generally categorize emulation into two distinct types: Full System Emulation and User Mode Emulation.

Emulators

User Mode Emulation

- Launches **a processes** directly, but on a restricted environment
  - Process may be compiled for one CPU and executed on another CPU
  - Address space is restricted, such as filesystem and libraries available
  - Interaction with Host OS is mediated by the emulator
- Emulator process native CPU instructions (emulation/translation) and:
  - Provide means **to translate syscalls** from guest to host OS
  - Understand intrinsic characteristics such as clone
    - Clone is used to spawn new processes and will require the creation of a new emulation environment
  - Handle signals between analyzed binary and the host system
- May provide integration with debugging tools

Targeted Execution: User Mode emulation launches a specific process directly, but confines it within a restricted emulated environment.

Bridging the Architecture Gap: Its most powerful feature is allowing a process compiled for one CPU architecture (e.g., ARM) to be executed on a completely different host CPU (e.g., x86).

How it Operates: The address space is restricted, and the available filesystem and linked libraries are usually provided via a dedicated "sysroot" folder. All interaction with the Host OS is mediated.

The Emulator's Duties: To achieve this, the emulator must:

- Process and translate native CPU instructions.
- Translate system calls from the guest architecture to the host OS.
- Understand intrinsic OS characteristics, such as the `clone` syscall, which spawns new processes and requires the emulator to dynamically create new emulation environments.
- Handle system signals between the analyzed binary and the host.
- Provide integration interfaces with debugging tools.

Emulators

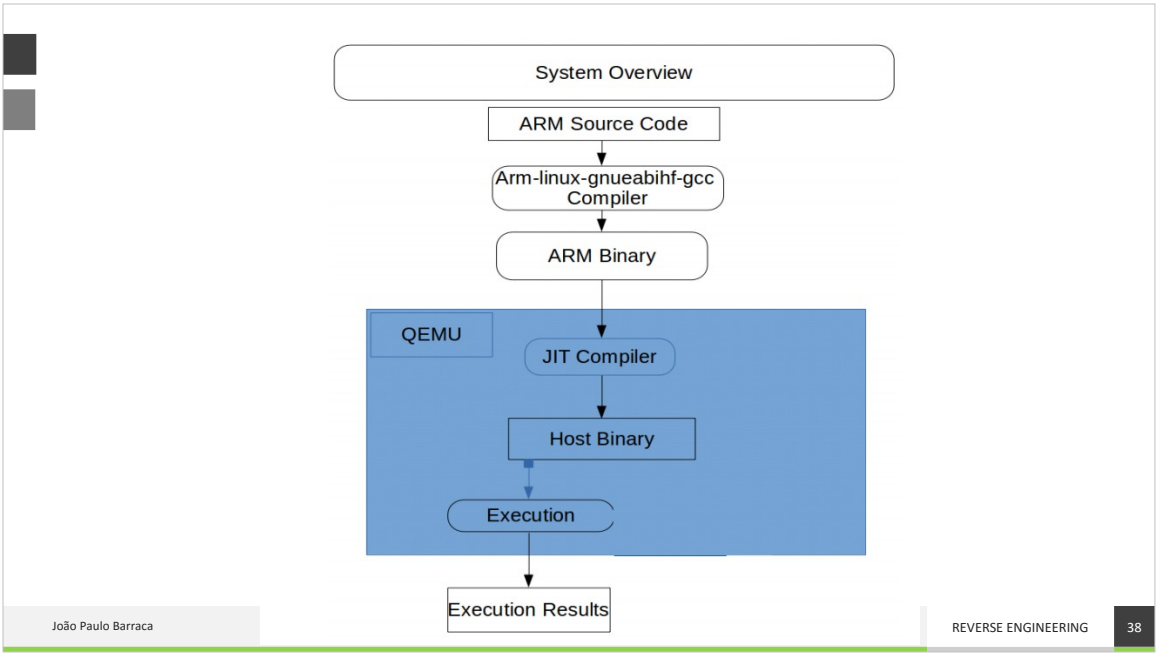
User Mode Emulation with QEMU

- QEMU allows user mode emulation **as long as the OS is kept the same**
- What it does:
  - Machine code translation from any CPU to any CPU
  - Syscall mapping
  - Data structure conversion (Bit-order and Bit-width conversions)
  - Extensive tracing capability to the level of Micro Ops
- Provides a **`gdbserver`** interface for interaction with GDB
- **Usefulness**: reverse engineering applications compiled to other architectures

The QEMU Engine: QEMU is the industry standard for this task. It allows user mode emulation provided the host and guest Operating Systems are the same (e.g., running an ARM Linux binary on an x86 Linux host).
Translation Mechanics: QEMU achieves this via several complex mechanisms:

- **Machine Code Translation:** It decodes instructions from any supported CPU and translates them to the host CPU via its Tiny Code Generator (TCG).
- **Syscall Mapping:** It intercepts the guest's syscalls and maps them to equivalent host syscalls.
- **Data Structure Conversion:** Crucially, it handles bit-order (endianness) and bit-width conversions (e.g., 32-bit to 64-bit pointer translations) dynamically.

Analysis Features: For reverse engineers, QEMU provides extensive tracing capabilities down to the level of Micro Ops (the intermediate representation QEMU uses before compiling to native host code). Furthermore, it provides a `gdbserver` interface, allowing seamless interaction with GDB.



In a practical scenario, we would visualize how a binary compiled for a foreign architecture interacts with QEMU's TCG. The original instructions are lifted into an Intermediate Representation (IR), optimized, and then JIT-compiled into the host's native assembly. This intermediate step is where analysts can inject instrumentation hooks.

Emulators

Full System Emulation

- Basically: a full-blown virtual machine
  - Emulates a highly configurable set of hardware, including embedded devices
  - Maps interactions to Host resources (screen, disk, network)
  - RE aware software tools expose debugging interfaces (usually to **gdb**)
- Provides the best level of isolation
  - All accesses are mediated by the emulator, reducing the attack surface to emulator components
  - Allows analyzing other binaries besides standard executable files
    - Firmware, MBR, UEFI
- Malware frequently try to detect Virtual Machines, emulators and debuggers…
  - With variable sophistication

The Ultimate Sandbox: Unlike user-mode, full system emulation is basically a full-blown virtual machine.

Hardware Simulation: It emulates a highly configurable set of hardware, including specific CPU variants, memory controllers, and embedded device peripherals. It maps all these virtual interactions to physical host resources (screen, disk, network).

The Ultimate Isolation: It provides the absolute best level of isolation. Because every single access is mediated by the emulator, the attack surface is drastically reduced to just the emulator's components.

Beyond Standard Binaries: This approach is mandatory when analyzing code that runs outside of a standard OS, such as raw firmware blobs, Master Boot Records (MBR), or UEFI components.

The Cat-and-Mouse Game: Students should note that sophisticated malware is aware of this. Malware frequently employs evasion techniques to detect if it is running inside a VM or emulator (e.g., checking CPUID leaves, specific driver MAC addresses, or timing discrepancies).

Remote debugging with emulators

gdb and gdbserver

- **gdb** can debug remote applications
  - It can even debug remote kernels and firmware
  - Why? Consider embedded devices, software inside an emulator
- **gdbserver** is launched on the target system, with the arguments:
  - Either a device name (to use a serial line) or a TCP hostname and portnumber, and the path and filename of the executable to be debugged
  - It then waits passively for the host **gdb** to communicate with it.
- **gdb** is run on the host, with the arguments:
  - The path and filename of the executable (and any sources) on the host, and
  - A device name (for a serial line) or the IP address and port number needed for connection to the target system.
- **Alternative**: the remote application is compiled with a stub that provides a **gdbserver** interface when the application is launched

Bridging the Gap: We previously mentioned QEMU's `gdbserver`. How does this work under the hood? GDB is designed to debug remote applications, even remote kernels and firmware.

The GDB Remote Serial Protocol (RSP): This is heavily used in embedded device analysis. `gdbserver` is launched on the target system (or within the emulator). It takes arguments for either a serial device name or a TCP hostname/port, along with the path to the executable.

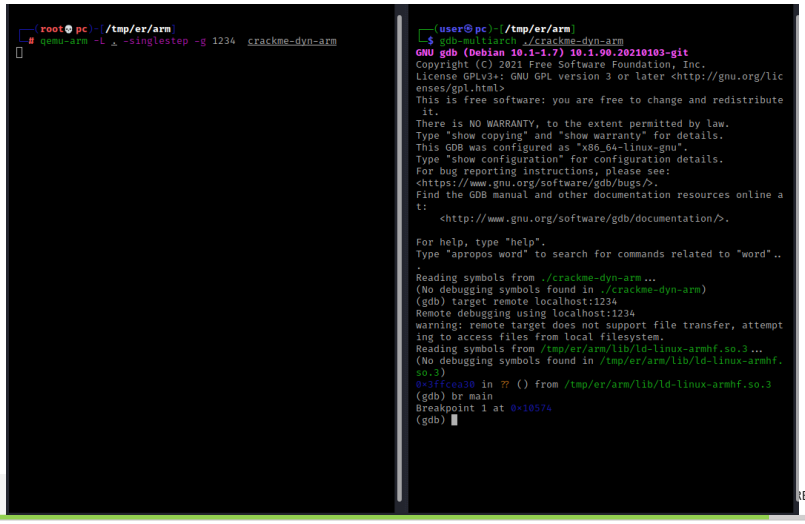
Passive Waiting: The `gdbserver` then waits passively for the host GDB to initiate communication.

Connecting from the Host: The analyst runs GDB on their host machine, loading the local unstripped binary (to get symbols) and issuing a `target remote <IP:PORT>` command to connect to the target.

Stubs: Alternatively, if `gdbserver` cannot run on the target OS, the remote application itself can be compiled with a small "stub" that provides this interface natively.

Example

Reversing an ARM binary



In a live lab environment, students would now see the GDB interface successfully connecting to the QEMU instance. They would observe the ARM registers (R0-R15, CPSR) populated with data, despite running the terminal on an x86 machine. This highlights the seamless integration modern dynamic analysis frameworks provide.

Example

unknown.bin

- Remember the **`unknown.bin`** file?
  - Well… looks like a PDF (is a PDF)
  - but **`$ file unknown.bin`** returns "**`unknown.bin: DOS/MBR boot sector`**"

- What we may extrapolate from that:
  - Seems to be a DOS/Master Boot Record (Master boot record – Wikipedia)
  - DOS was only released for i386 (16bits and 32bits)
  - `qemu-system-i386` may boot it if used as a hard disk or floppy disk

A Real-World Mystery: The presentation shifts to analyzing a file named `unknown.bin`.

Polyglot and Disguised Files: Initially, the file might look like a PDF document (perhaps matching a PDF signature or extension). However, running the Linux `$ file unknown.bin` command reveals its true nature: it is identified as a "DOS/MBR boot sector".

Extrapolating the Context: What does this mean for our analysis?

- We are likely dealing with a Master Boot Record (MBR).
- DOS and legacy MBRs were designed exclusively for Intel i386 (16-bit and 32-bit real mode) architecture.

The Emulation Strategy: Because this isn't a standard executable (like an ELF or PE), user-mode emulation won't work. However, we can use full system emulation. Specifically, `qemu-system-i386` can load and boot this raw binary file if we pass it to the emulator as a virtual hard disk or floppy disk.

Example

unknown.bin

- How to address such files?
  - Binary files other than ELFs (or PE or other similar) obey to a fixed set of rules
  - It is required to check the datasheets and gather information required to load the file.
  - Important:
    - CPU used, CPU mode, relevant or required peripherals: to know how to decode the binary instructions
    - Program Entry Point: to know where the program starts, and where disassembly should start
- From a Master Boot Record we may know:
  - MBR is loaded to address 0x7C00
  - MBR code runs in Intel x86 Real Mode (16bits)
  - There are quite a few limitations and assumptions: IBM DOS 2.00 Master Boot Record (pcministry.com)
  - There is no OS running. Input/Output must use BIOS Interrupts

Beyond Standard Executables: How do we approach a file like an MBR that lacks an executable header (like ELF or PE)? Standard tools don't know where the code starts, what architecture it uses, or how it maps into memory.

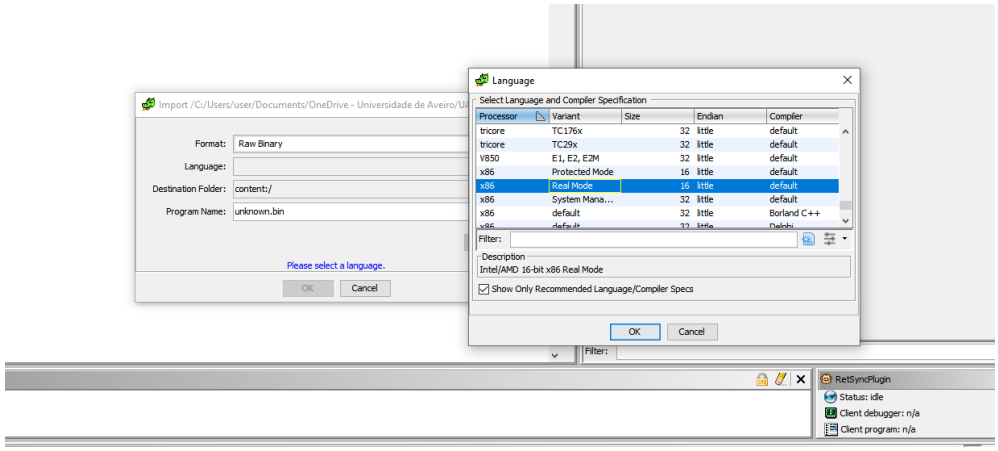
The Rule of Context: For raw binaries, the reverse engineer must provide the context. This involves consulting hardware datasheets and understanding the boot sequence.

MBR Specifics: * We know from IBM PC standards that the BIOS always loads the MBR into the physical memory address `0x7C00`.

- It executes in **Intel x86 16-bit Real Mode** (not the 32-bit or 64-bit protected modes modern OSes use).
- There is no underlying operating system. If the code wants to print to the screen or read from the disk, it cannot make a Linux or Windows syscall; it must trigger **BIOS Interrupts** directly.

Example

Loading the unknown.bin in ghidra

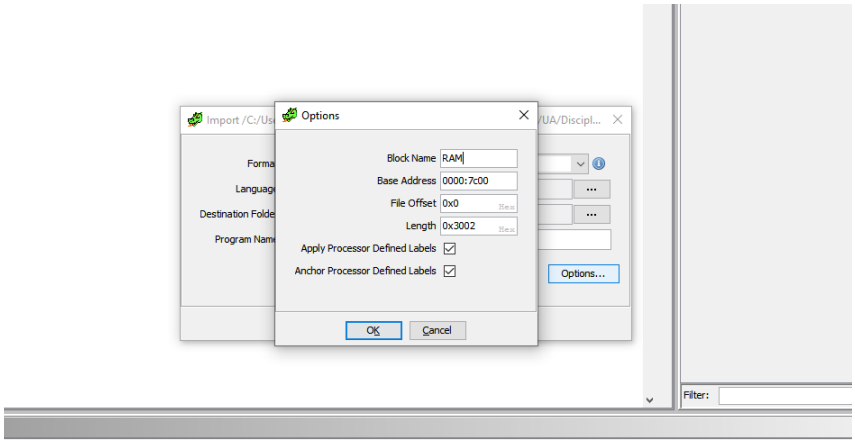


The Static Analysis Attempt: *(These slides visually demonstrate importing the raw binary into Ghidra.)*

Import Configuration: Because it's a raw binary, Ghidra will not automatically detect the architecture. The analyst must manually specify the language as `x86`, `16-bit`, `Real Mode`.

Example

Loading the unknown.bin in ghidra

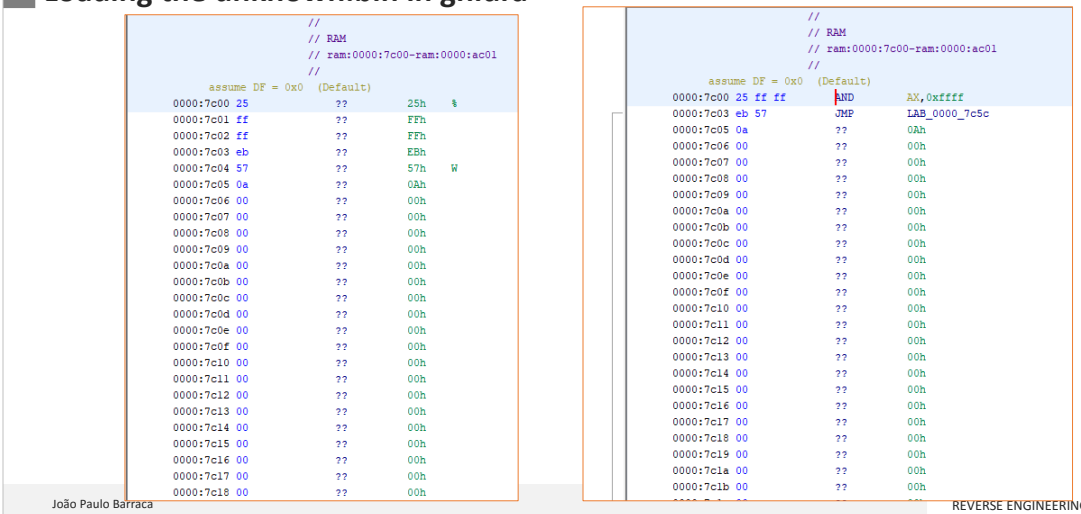


The Initial Confusion: If you just load it at address `0x0000`, the disassembly will look like garbage because all absolute memory references (jumps, data reads) will point to the wrong locations.

Example

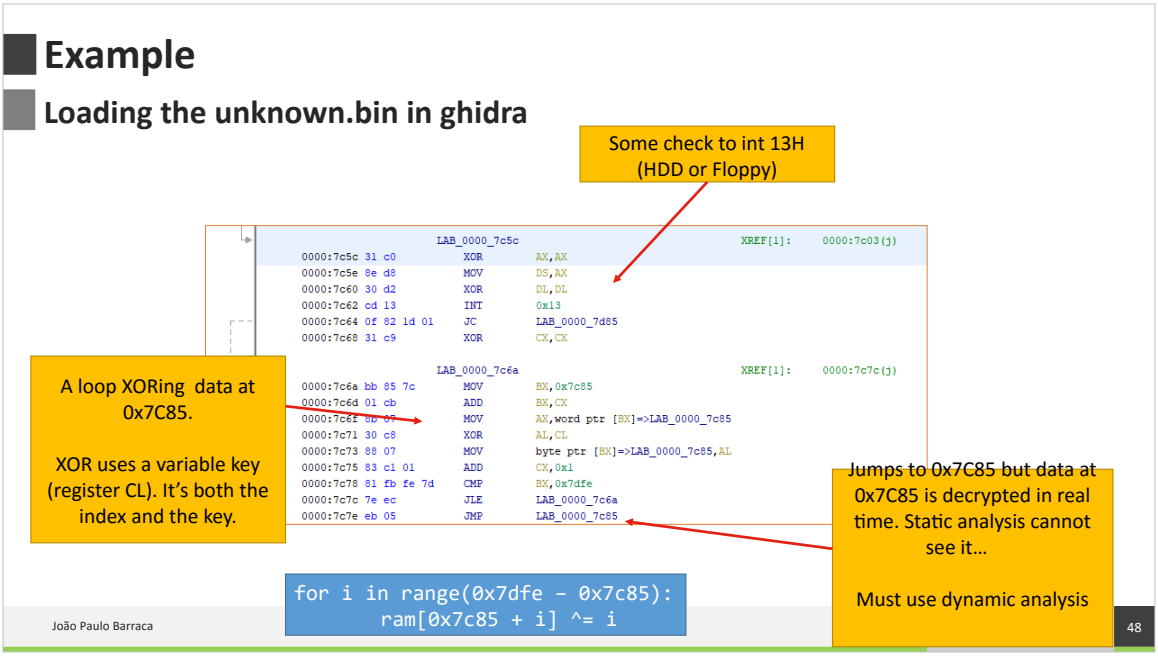
Loading the unknown.bin in ghidra

If we state that 0x7C00 has code, looks like we have something



Setting the Base: To get a coherent disassembly, we must tell Ghidra to map the block of data to the base address `0x7C00`.

Structuring the Code: Once mapped correctly, we can tell Ghidra to start disassembling at `0x7C00`. Suddenly, recognizable 16-bit assembly instructions appear, confirming our hypothesis about the file's nature.

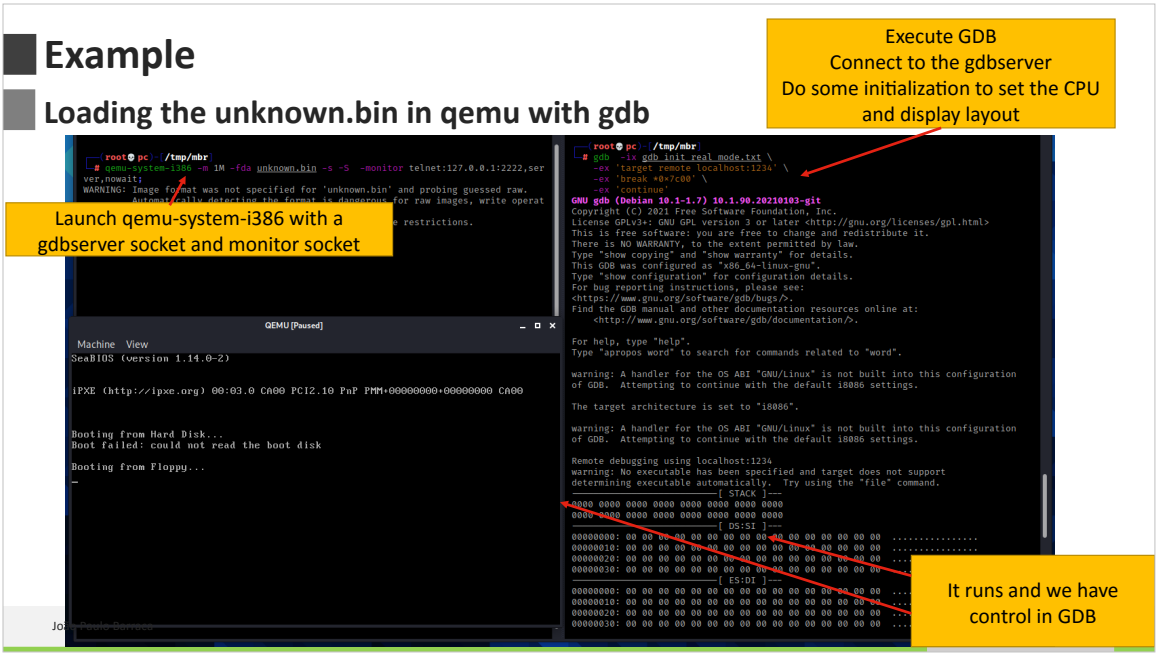


Analyzing the Behavior: Reviewing the recovered assembly, we notice calls to `int 13H`. In BIOS interrupts, `13H` is the low-level disk service routine, perfectly normal for a boot sector.

The Roadblock: However, we immediately spot a suspicious loop. The code is reading data starting at address `0x7C85` and XORing it using a variable key derived from the `CL` register.

Self-Modifying Code: The code then jumps *into* the data it just modified. This is a classic packer/obfuscator routine. The actual payload at `0x7C85` is encrypted. Because it's decrypted in real-time during execution, our static analyzer (Ghidra) can only see the encrypted bytes, not the underlying logic.

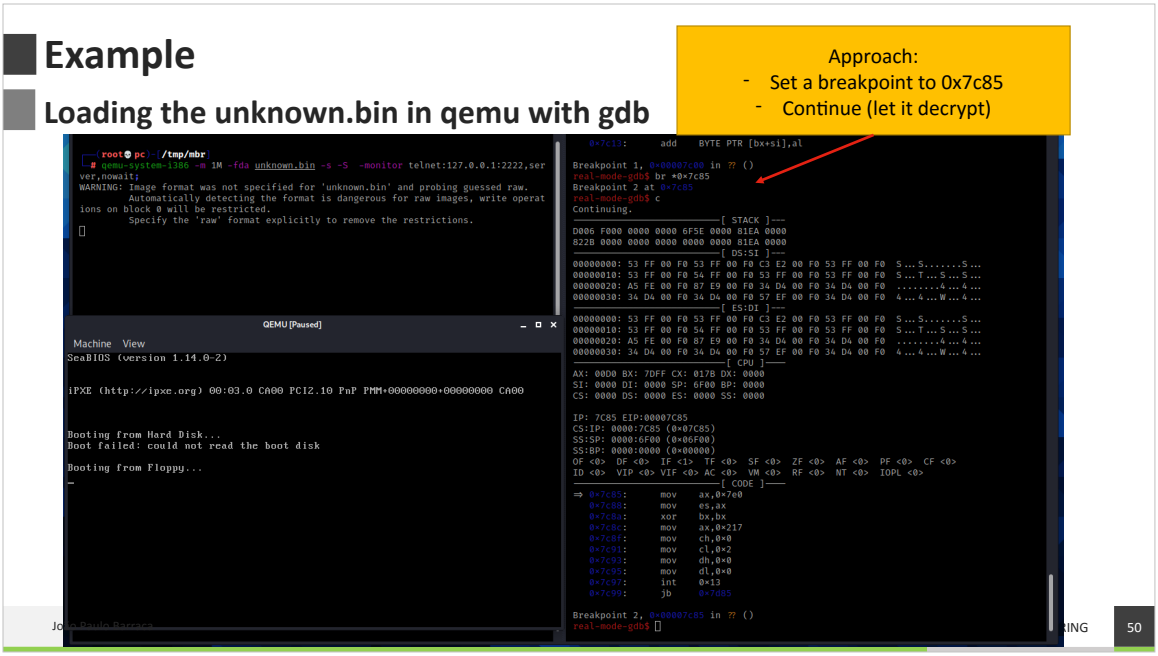
The Need for Dynamics: This perfectly illustrates why static analysis alone is insufficient. We must transition to dynamic analysis to let the code decrypt itself.



The Dynamic Setup: We launch `qemu-system-i386`, passing `unknown.bin` as a floppy disk image. We also append the `-s -S` flags (which spawn the `gdbserver` and freeze the CPU before the first instruction).

Connecting the Debugger: We execute GDB locally. Because we are debugging 16-bit real-mode code, we must do some specific GDB initialization (setting the architecture to `i8086`) so GDB interprets the registers and memory correctly.

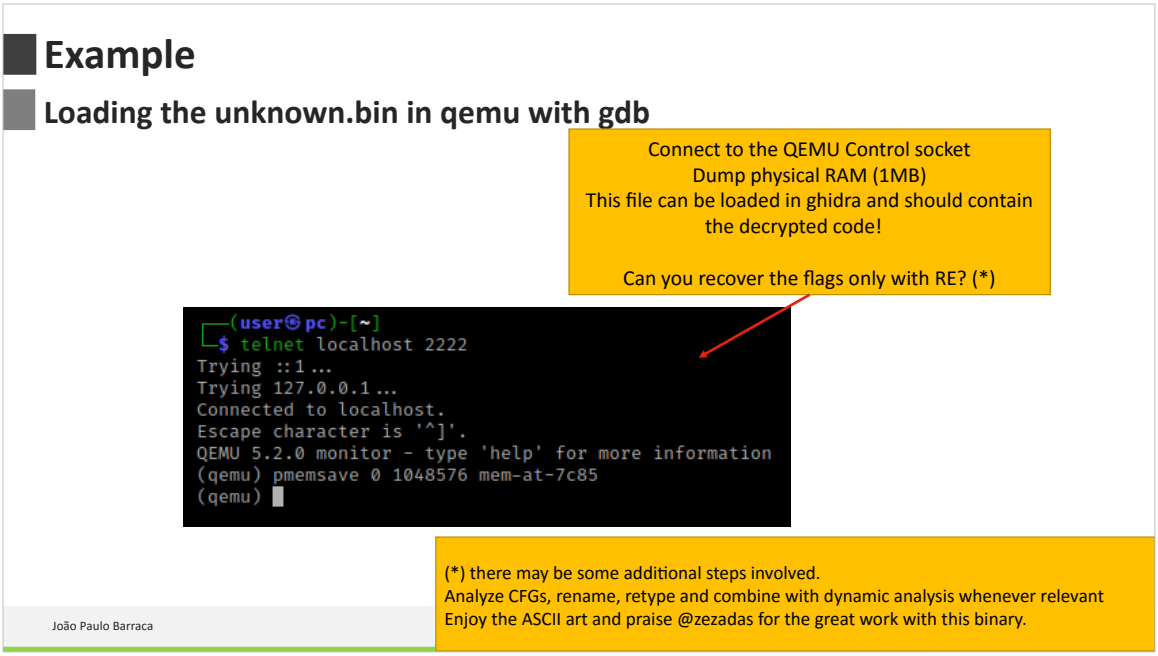
Taking Control: We then connect to the QEMU `gdbserver`. We now have interactive, instruction-level control over a simulated bare-metal PC.



The Strategy: We know from our static analysis that the decryption routine finishes and jumps to `0x7C85` to execute the payload.

Setting the Trap: Therefore, we use GDB to set a hardware or software breakpoint exactly at `0x7C85` (`break *0x7C85`).

Execution: We issue the `continue` command. The emulator runs the MBR natively, the decryption loop executes thousands of times, and the instant it tries to execute the first decrypted instruction at `0x7C85`, the emulator halts and returns control to our debugger.



Extracting the Payload: Now that the code is halted *after* decryption, the payload sits plainly in the emulator's virtual RAM.

The Dump: We can use QEMU's monitor console or GDB's memory dumping commands to dump the physical RAM (e.g., the first 1MB) to a new file on our host machine.

Back to Static: We can take this new memory dump file, load it *back* into Ghidra, and statically analyze the now-decrypted, fully visible malicious payload. This loop—Static -> Dynamic -> Static—is the core workflow of advanced reverse engineering.

Dynamic Binary Instrumentation (DBI)

What are they

- DBI system as an application virtual machine that interprets the ISA of a specific platform
  - usually (but not always) coinciding with the one where the system runs
  - offering instrumentation capabilities to support monitoring and altering instructions and data from an analysis tool component
  - Up to the level of a single instruction
- DBI systems expand standard Dynamic Binary Analysis tasks by
  - Fine grained monitoring capabilities
  - Full control over data and instructions, potentially increasing Reverse Engineering Scope
- Uses
  - Measure performance, Detect vulnerabilities, Force code execution, Fuzz binary programs at the scale of a group of instructions

Defining DBI: We now move to the most advanced topic: Dynamic Binary Instrumentation. A DBI system acts as an application virtual machine that interprets the Instruction Set Architecture (ISA) of a platform.

The Difference from Debugging: While a debugger uses OS interrupts to stop execution, a DBI framework recompiles and injects custom analysis code *directly into the execution stream* of the target program at runtime.

Granularity: This allows for incredibly fine-grained monitoring, down to the level of a single instruction. You can write scripts that intercept every memory read, every register change, or every function call without permanently modifying the binary on disk.

Use Cases: DBI is used for performance profiling, detecting vulnerabilities (like memory leaks via Dynamic Taint Analysis), forcing code execution paths, and performing highly targeted fuzzing.

Dynamic Binary Instrumentation (DBI)

caveats

- DBI is vulnerable to specific attacks targeting the emulator
  - Purpose: avoid the use of emulators or induce incorrect results
  - Exploit the fact that DBI tools are slow
  - Exploit the fact that the system is emulated and differs from a real system

- Some approaches
  - Extensive loops Timing measurements
  - Timing measurements
  - Testing for system specific behavior

```
128  for ( n = 0; n < 2000000; ++n )
129  {
130    EnterCriticalSection(&CriticalSection);
131    mw_junk_0();
132    v6[1] = (int)v6;
133    v6[0] = 707220816;
134    *(_DWORD *)sz = dword_41A4F4;
135    CharUpperW(sz);
136    for ( ii = 0; ii < 5; ++ii )
137    {
138      v6[2] = -199066008;
139      v8 = 0;
140    }
141    LeaveCriticalSection(&CriticalSection);
142  }
143  DeleteCriticalSection(&CriticalSection);
```

João Paulo Barraca

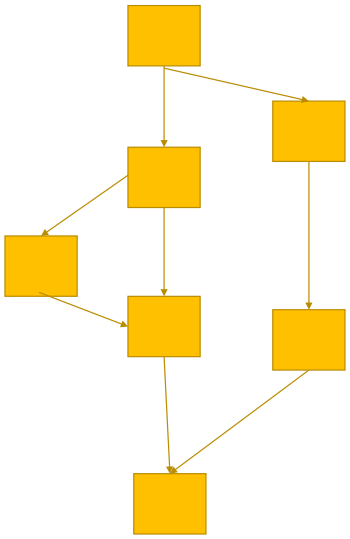
The Evasion Arms Race: Because DBI frameworks effectively act as emulators/JIT compilers, they carry the same fundamental weakness: they introduce massive artifacts into the execution environment.
The Transparency Problem: Advanced malware is designed to detect these artifacts to avoid analysis.
Detection Techniques:

- **Timing Measurements:** Emulating and instrumenting code is fundamentally slower than native execution. Malware uses instructions like `RDTSC` (Read Time-Stamp Counter) to detect if a block of code took "too long" to run.
- **Extensive Loops:** Malware might run a massive, pointless loop millions of times. Natively, it takes a fraction of a second. Under DBI, it might take hours, effectively "stalling" the analysis.
- **System Specifics:** Because DBI mimics the OS/Hardware, subtle differences in how edge-case instructions or interrupts are handled can be tested by the malware to detect the instrumentation matrix.

Dynamic Binary Instrumentation (DBI)

What are they

- Instrumentation
  - Insert Code
- Dynamic Binary Instrumentation
  - “Running” Code



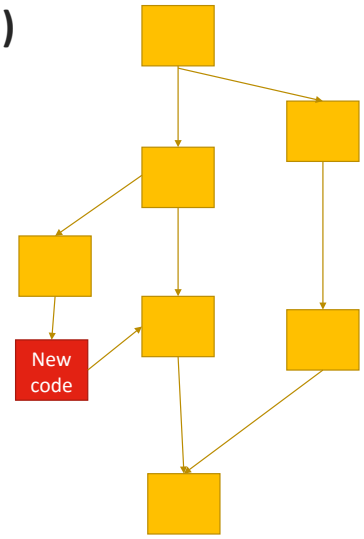
Under the Hood: How do DBI frameworks actually insert our analysis code into a running binary without breaking it? At a master's level, you must understand the two primary architectural approaches: Probe-based and Just-In-Time (JIT) based instrumentation.

The Choice: The framework you choose dictates the performance overhead and the level of granularity you can achieve. Each approach handles the "transparency problem" and execution control very differently.

Dynamic Binary Instrumentation (DBI)

What are they

- Instrumentation
  - Insert Code
- Dynamic Binary Instrumentation
  - “Running” Code



The Trampoline Technique: Probe-based systems (like Microsoft Detours or early versions of Frida) work by modifying the target binary's memory directly in place.

How it Works:

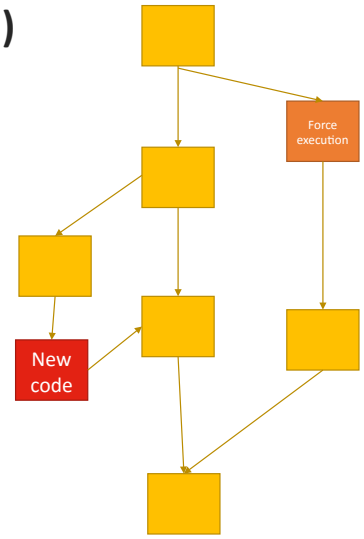
1. The framework identifies the target instruction or function entry point.
2. It overwrites the first few bytes with a highly specific unconditional jump instruction (a "probe" or "trampoline") pointing to the analyst's custom code payload.
3. Because overwriting bytes destroys the original instructions, the framework carefully relocates those overwritten bytes to a safe area.
4. After the custom payload executes, it runs the relocated original instructions and jumps back to the original execution stream.

Pros and Cons: This is incredibly fast because the un-instrumented code runs natively. However, it is coarse-grained (usually limited to function-level hooking) and highly susceptible to detection by malware that performs integrity checks on its own code.

Dynamic Binary Instrumentation (DBI)

What are they

- Instrumentation
  - Insert Code
- Dynamic Binary Instrumentation
  - "Running" Code



Total Control: Frameworks like Intel Pin, DynamoRIO, and Valgrind use a JIT-compiler approach. They never allow the original code to execute natively.

The Fetch-Execute Cycle: 1. The DBI engine fetches a Basic Block of code from the target. 2. It lifts this code into an intermediate representation. 3. It interleaves the analyst's instrumentation instructions directly into this representation. 4. It recompiles the combined code on the fly.

Granularity: Because the engine rebuilds the code instruction-by-instruction, you can achieve infinite granularity. You can instrument every single memory read or track the exact state of the `EFLAGS` register after every arithmetic operation.

Dynamic Binary Instrumentation (DBI)

How they work?

- Rebuild a program binary code using some JIT technique
  - Insert trace points and hooks for inspection
  - Divert execution to **additional user specified functions**
  - Monitor access to memory regions
    - Potentially triggering callbacks on access
  - May reimplement access to IOs or even `syscalls` and interrupts
  - May create a fully Emulated Execution Environment
    - Can be combined with an Emulation platform such as QEMU or Unicorn (a fork from QEMU)
- Popular tools: valgrind, DynamoRIO, Intel PIN, DynInst, Qiling, Frida

Mitigating Overhead: JIT compilation is incredibly slow. If a DBI engine recompiled a loop of 10 instructions every time it ran a million times, the program would stall entirely.

The Solution: The Code Cache. Once a basic block is instrumented and compiled, it is stored in a dedicated region of memory. The next time the execution flow hits that block, the DBI engine simply executes the cached version.

Context Switching: A major challenge in JIT-DBI is the context switch between the target application's state and the DBI engine's state. Advanced frameworks optimize this by keeping the execution inside the Code Cache as long as possible, only dropping back into the DBI engine to compile newly discovered code paths.

Dynamic Binary Instrumentation (DBI)

APPLICATION DOMAIN	DBI PRIMITIVES								
	INSTRUCTIONS				SYSTEM CALLS	LIBRARY CALLS	THREADS & PROCESSES	CODE LOADING	EXCEPTIONS & SIGNALS
	MEMORY R/W	CALLS/RETS	BRANCHES	OTHER					
CRYPTOANALYSIS	✓	✓	✓	✓					
MALICIOUS SOFTWARE ANALYSIS	✓	✓	✓	✓	✓	✓	✓	✓	✓
VULNERABILITY DETECTION	✓	✓	✓	✓	✓	✓			
SOFTWARE PLAGIARISM	✓				✓				
REVERSE ENGINEERING	✓	✓	✓	✓	✓	✓			
INFORMATION FLOW TRACKING	✓	✓	✓	✓		✓			✓
SOFTWARE PROTECTION	✓	✓	✓	✓	✓	✓	✓	✓	✓

Daniele D'Elia et al, SoK: Using Dynamic Binary Instrumentation for Security, AsiaCCS, 2019

The Core Use Case: Why do we need the instruction-level granularity of JIT-DBI? The most prominent academic and industrial application is Dynamic Taint Analysis (DTA).

The Concept: DTA tracks how specific pieces of data (usually user input) flow through a program's memory and registers during execution. It allows us to mathematically prove if attacker-controlled data can influence critical execution states.

Shadow Memory: To accomplish this, DTA frameworks maintain a separate memory map (Shadow Memory) that mirrors the target's RAM. Every byte in the target memory has a corresponding "taint bit" in the shadow memory indicating if it is clean or tainted.

DBI with Qiling

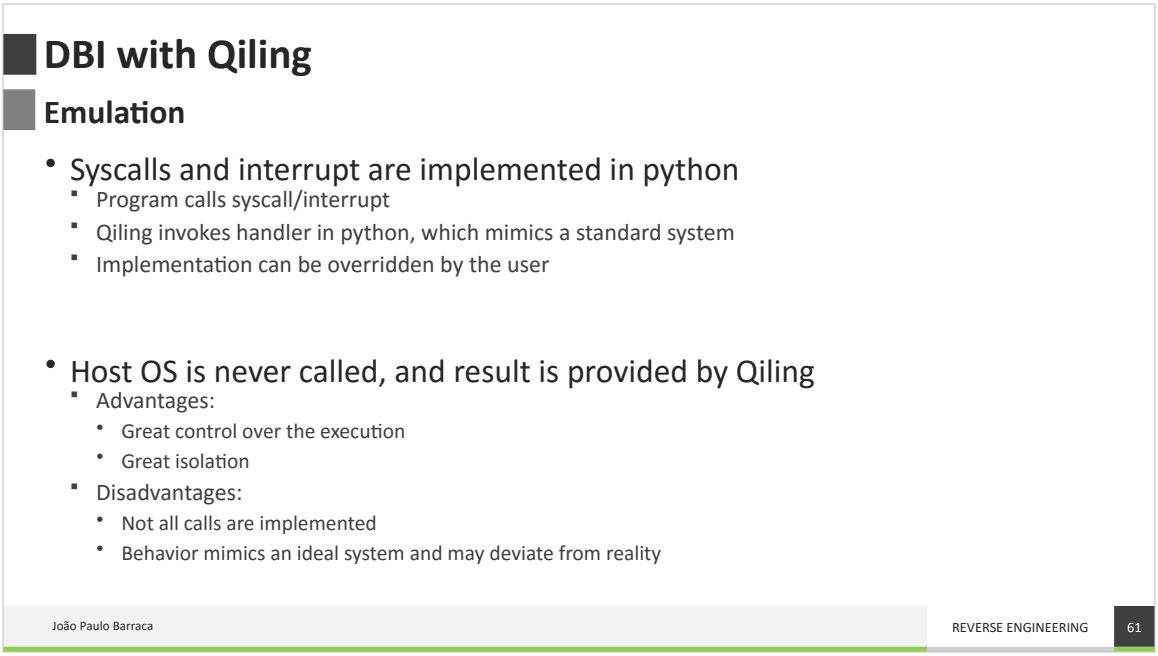
DBI tool that can perform:

- **Emulation**: Executes binary code step by step, replacing instructions
- **Binary instrumentation**: allows injection of user specified code
- **Cross-platform and cross-architectural analysis**: analyze one architecture or OS on another
- **Sandboxing**: I/O is redirected to fake devices (files, sockets)
- **On raw binaries**: used to analyze blobs from binary devices or shellcode

Introduction to Qiling: Moving beyond generic frameworks, we introduce Qiling, an incredibly powerful, advanced Dynamic Binary Instrumentation (DBI) framework built on top of the Unicorn engine.

Core Capabilities: Qiling is a true multi-tool for reverse engineers. It can perform:

- **Emulation:** It executes binary code step-by-step, replacing standard execution with controlled software emulation.
- **Binary Instrumentation:** It allows the seamless injection of user-specified Python code directly into the execution flow.
- **Cross-Platform/Cross-Architecture Analysis:** It empowers analysts to run and analyze a binary built for one architecture or OS directly on another (e.g., analyzing a Linux ARM binary on a Windows x86 machine).
- **Sandboxing:** It heavily isolates the execution by redirecting all input/output (I/O) requests to fake devices, mock files, and mock sockets.
- **Raw Binaries Execution:** It isn't limited to standard executables; it can be used to analyze arbitrary binary blobs extracted from IoT devices or raw shellcode.



The OS Illusion: Unlike user-mode emulators that map system calls to the host operating system, Qiling takes a more radical approach. In Qiling, all system calls and hardware interrupts are implemented entirely in Python.

The Execution Loop: When the target program attempts to call a syscall or trigger an interrupt, Qiling intercepts it and invokes a dedicated Python handler that mimics the behavior of a standard system. The host OS is never actually called, and the result is artificially provided back to the binary by Qiling.

Customization: Because the OS is mocked in Python, the user can easily override any standard implementation to suit their analysis needs.

Trade-offs: * *Advantages:* This yields unparalleled control over the execution flow and absolute isolation (true sandboxing).

- *Disadvantages:* Because it mimics an "ideal" system, its behavior may sometimes deviate from a real-world system. Furthermore, not all obscure or undocumented system calls are fully implemented in the framework.

DBI with Qiling

Instrumentation

- User can define hooks to triggering callbacks on an event
  - Because an emulator is translating code in real time, instruction level hooks are possible

- Example
  - Code execution reaches a specific address
  - An address is written or read
  - A function is called, or is leaving
  - An instruction is executed

Event-Driven Analysis: Qiling allows the analyst to define custom Python hooks that trigger callbacks whenever specific execution events occur.

Infinite Granularity: Because the underlying emulator is translating code in real-time, Qiling can support true instruction-level hooks.

Common Hook Types: We can instrument the binary to pause execution or run our Python scripts when:

- Code execution reaches a specific memory address.
- A specific memory address is written to or read from.
- A specific function is called or is returning.
- Every single instruction is executed.

Application: This is how you implement customized Dynamic Taint Analysis or automated unpacking scripts. You hook memory writes to trace where data goes, or hook execution to dump memory right after a decryption loop finishes.

DBI with Qiling

Cross Platform and Cross Architecture

- Binary code is emulated, allowing cross architecture execution
  - Target architecture instructions are compiled to native instructions
- Because all syscalls and interrupts are emulated, host platform can differ from target platform
  - As Qiling is based on Unicorn (Qemu), a wide range of possibilities is available

The Architectural Abstraction: By combining the Unicorn CPU emulator with Qiling's OS-level emulation, the binary code is emulated at an architectural level. The target architecture instructions are compiled to the host's native instructions on the fly.

True OS Independence: Because all syscalls and interrupts are completely emulated by Qiling in Python, the host platform can completely differ from the target platform. You do not need a Linux kernel to run Linux binaries, nor do you need a Windows kernel to run PE files.

The Unicorn Advantage: Qiling's foundation on the Unicorn engine (which itself is a lightweight fork of QEMU's CPU emulator) provides an incredibly wide range of possibilities and supported architectures out of the box

DBI with Qiling

Loading an Elf

- Qiling has several loaders
  - MBR
  - PE, ELF, MachO
  - Unstructured binary (shellcode)

```python
from qiling import *

def sandbox(path, rootfs):
    ql = Qiling(path, rootfs)
    ql.run()

if __name__ == '__main__':
    sandbox(['./hello'], '.')
```

- Loader will make code available to be **emulated** on a secure **rootfs**
  - Calls to interrupts and syscalls are implemented in python

The Loader Ecosystem: Before code can execute, it must be mapped into memory correctly. Qiling ships with a rich set of proprietary loaders designed for several formats.

Supported Formats: It can natively load standard executables like PE (Windows), ELF (Linux), and MachO (macOS), as well as Master Boot Records (MBR) and unstructured binary data like raw shellcode.

The `rootfs` Concept: The loader makes the code available to be emulated within a secure, mock root filesystem (`rootfs`). This ensures that when the binary requests `/etc/passwd` or `C:\Windows\System32`, Qiling serves the file from the safe `rootfs` directory rather than your actual host disk.

Execution Handoff: Once mapped, execution begins, and any subsequent calls to interrupts and syscalls are handled by the Python implementation.

```
[=]     brk(input = 0x0)
[=]     uname(address = 0x80000000d960)
[=]     access(path = 0x7ffff7dfa9b0, mode = 0x4)
[=]     openat(fd = 0xffffff9c, path = 0x7ffff7df7b67, flags = 0x80000, mode = 0x0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     read(fd = 0x3, buf = 0x80000000d0f8, len = 0x340)
[=]     fstat(fd = 0x3, buf_ptr = 0x80000000cfa0)
[=]     mmap(addr = 0x0, length = 0x1c4508, prot = 0x1, flags = 0x802, fd = 0x3, pgoffset = 0x0)
[=]     mprotect(start = 0x7fffb7dfb000, mlen = 0x196000, prot = 0x0)
[=]     mmap(addr = 0x7fffb7dfb000, length = 0x14b000, prot = 0x5, flags = 0x812, fd = 0x3, pgoffset = 0x25000)
[=]     mmap(addr = 0x7fffb7f46000, length = 0x4a000, prot = 0x1, flags = 0x812, fd = 0x3, pgoffset = 0x170000)
[=]     mmap(addr = 0x7fffb7f91000, length = 0x6000, prot = 0x3, flags = 0x812, fd = 0x3, pgoffset = 0x1ba000)
[=]     mmap(addr = 0x7fffb7f97000, length = 0x3508, prot = 0x3, flags = 0x32, fd = 0xffffffff, pgoffset = 0x0)
[=]     close(fd = 0x3)
[=]     mmap(addr = 0x0, length = 0x2000, prot = 0x3, flags = 0x22, fd = 0xffffffff, pgoffset = 0x0)
[=]     arch_prctl(ARCHX = 0x1002, ARCH_SET_FS = 0x7fffb7f9bf40)
[=]     mprotect(start = 0x7fffb7f91000, mlen = 0x3000, prot = 0x1)
[=]     mprotect(start = 0x555555557000, mlen = 0x1000, prot = 0x1)
[=]     mprotect(start = 0x7ffff7dff000, mlen = 0x1000, prot = 0x1)
[=]     fstat(fd = 0x1, buf_ptr = 0x80000000d630)
[=]     ioctl(fd = 0x1, cmd = 0x5401, arg = 0x80000000d590)
[=]     brk(input = 0x0)
[=]     brk(input = 0x55555557c000)
[=]     write(fd = 0x1, buf = 0x55555555b2a0, count = 0x6)
Hello [!]       0x7fffb7e9bc08: syscall ql_syscall_clock_nanosleep number = 0xe6(230) not implemented
[=]     write(fd = 0x1, buf = 0x55555555b2a0, count = 0x6)
World
[=]     exit_group(exit_code = 0x0)
```

DBI with Qiling

Overriding a library function

- Functions can be overridden with custom implementations
  - Code can access arguments of basic types (Strings, Ints, Floats)
  - Inside function, other external functions can be called
  - Entire set of registries and memory can be manipulated
  - Return is provided to the calling function to be **emulated** on a secure **rootfs**
  - Calls to interrupts and syscalls are implemented in python

```python
from qiling import *
from qiling.os.const import UINT
import time

def my_sleep(ql):
    args = ql.os.resolve_fcall_params({'seconds': UINT})
    seconds = args['seconds']
    print(f"Sleep: {seconds}")
    if seconds > 10:
        print("QL: Limiting sleep to 10s")
        time.sleep(10)
    else:
        time.sleep(seconds)

def sandbox(path, rootfs):
    ql = Qiling(path, rootfs, log_file="hello-2.log", verbose=0)
    ql.set_api('sleep', my_sleep)
    ql.run()

if __name__ == '__main__':
    sandbox(['./hello'], '.')
```


High-Level Hooking: Beyond simple instruction tracing, Qiling allows you to completely hijack and override standard library functions with your own custom implementations.

Context Access: When your custom Python hook intercepts a function call, your code can easily access and parse the arguments passed to it, translating them into basic Python types (Strings, Integers, Floats).

State Manipulation: Inside your custom function hook, you have omnipotent control. You can call other external functions, and manipulate the entire set of CPU registers and memory.

Faking the Return: Once your custom logic is complete, you can provide a spoofed return value back to the calling function, which continues its emulation seamlessly within the secure `rootfs`. This is exceptionally useful for bypassing cryptographic checks or forcing a malware sample to believe it successfully connected to a command server.