

Reverse Engineering Software: Selection, Requirements, and Workflow



Choose the right RE approach for each target



Establish legal and ethical requirements before analysis



Run a repeatable, documented analysis workflow



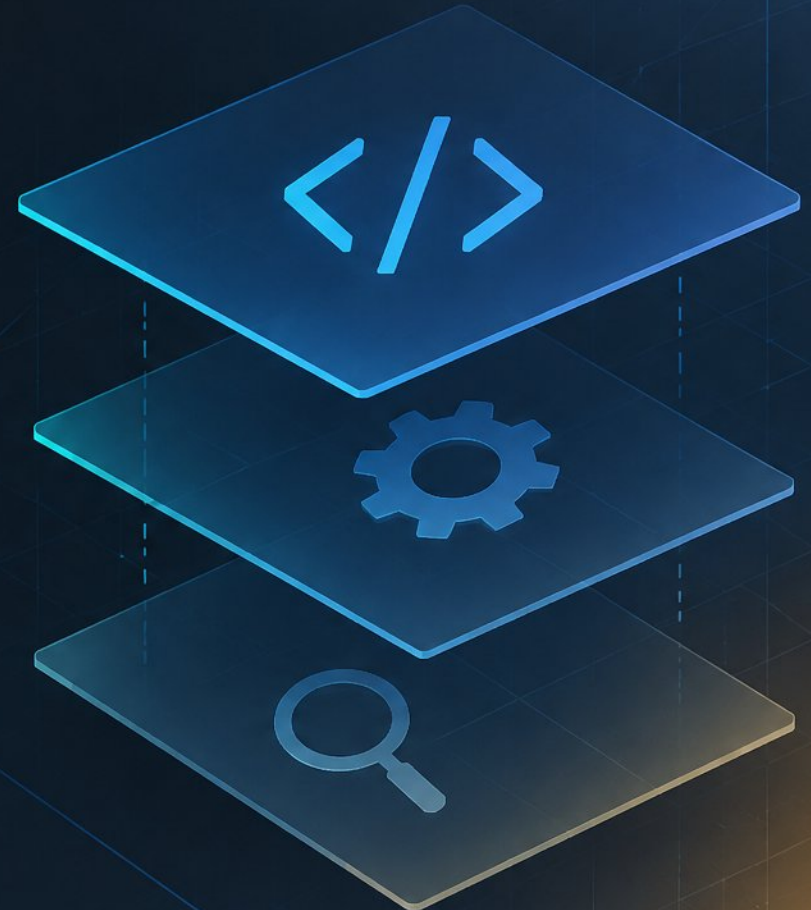
Analyze only software you own or are authorized to inspect



Supporting interoperability, security review, and legacy recovery — not circumvention



Outcomes: tooling rationale, analysis plan, reproducible evidence, clear reporting



What Reverse Engineering Is, and When It Is Legitimate

ANALYZE
UNDERSTAND
BUILD COMPATIBILITY
DO THINGS RIGHT



Definition:

studying an artifact's construction, behavior, and exposed interfaces without original source



Core subfields:

static analysis, dynamic analysis, binary analysis, protocol/format analysis



Legitimate uses:

interoperability, vulnerability review, malware understanding, legacy recovery, compatibility testing



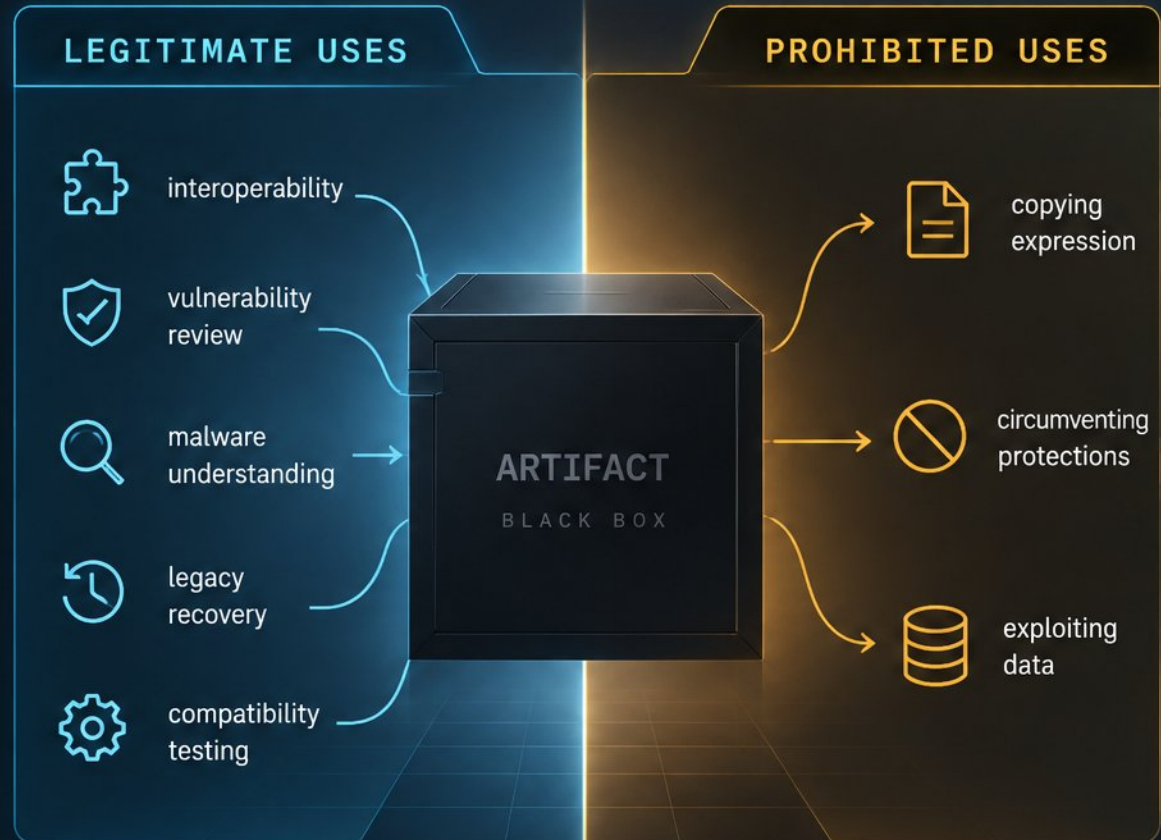
Boundary:

understanding behavior and interfaces differs from copying expression or circumventing protections

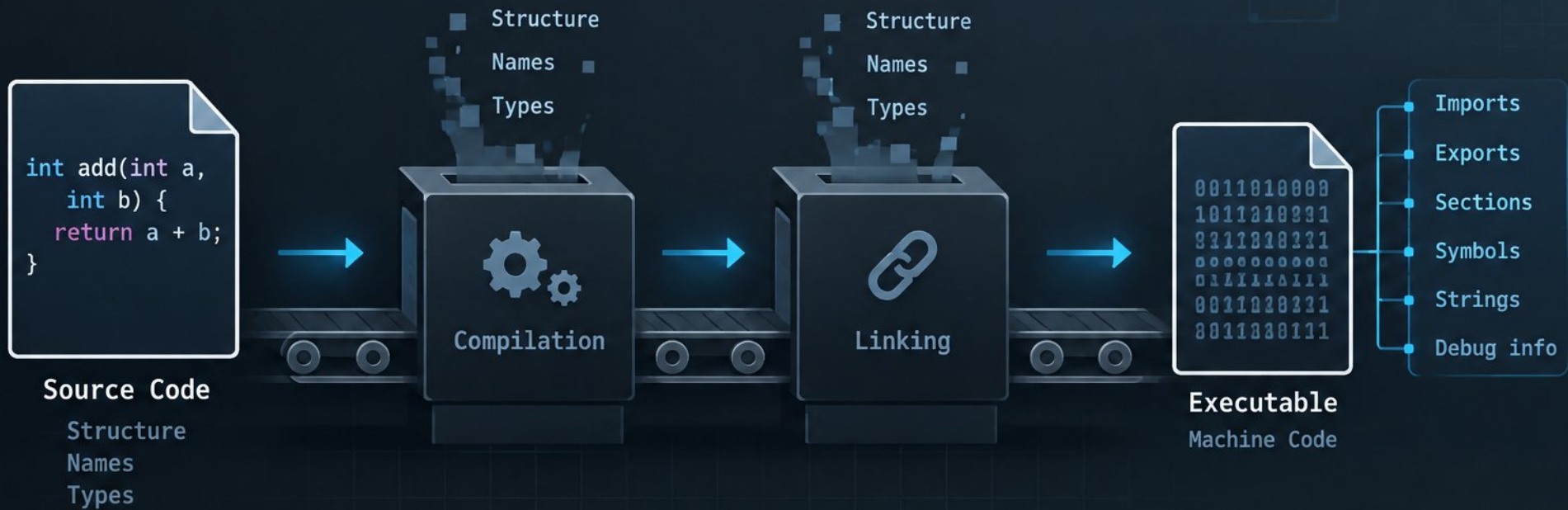


Method over talent:

undocumented RE work is not reproducible, reviewable, or defensible

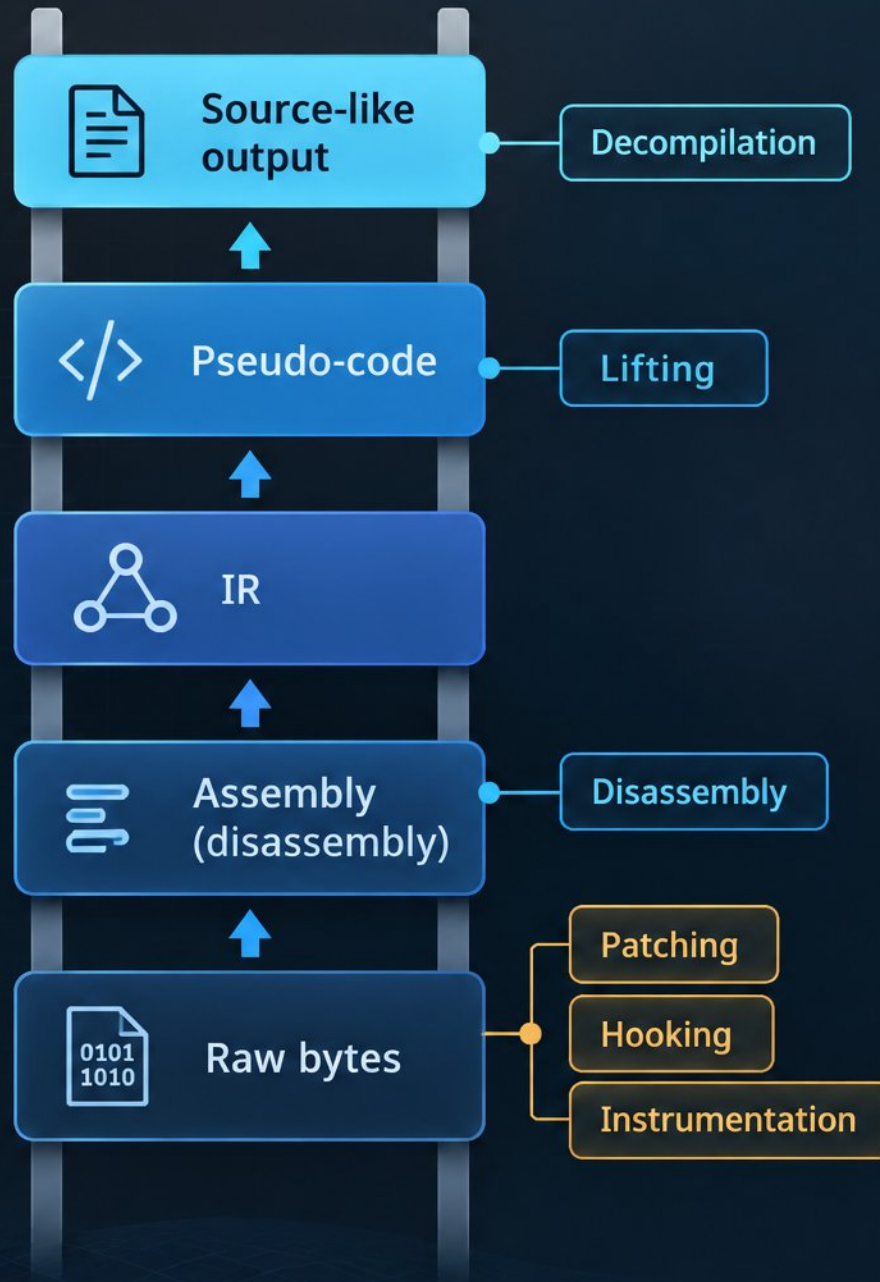


Background: From Source Code to Machine Code



- Compilation and linking discard structure, names, and types progressively
- What survives guides analysis: **imports**, **exports**, **sections**, **symbols**, **strings**, **debug info**
- ISAs in practice: **x86-64**, **ARM64**, plus bytecode targets like **JVM** and **.NET**
- Formats vary: **PE**, **ELF**, **Mach-O**, **APK/AAB**, **JAR/Assembly**, and firmware images
- **Static** and **dynamic** analysis are complementary; mature workflows alternate between them

Levels of Abstraction and Core Vocabulary



- Abstraction ladder: raw bytes → assembly → IR → pseudo-code → source-like output
- Core terms: disassembly, decompilation, lifting, patching, hooking, instrumentation
- Binary Ninja exposes LLIL, MLIL, and HLIL layers for scripted analysis
- ABI and calling conventions: arguments, return values, stack frame layout
- Decompiler pseudo-code is a reconstruction — validate against assembly
- Obfuscation, packing, anti-analysis raise cost, they don't block analysis

Define the Analysis Goal Before Choosing Tools



- Start from the question: interoperability, vulnerability review, malware triage, or legacy recovery
- Goal drives depth: interface-level review needs far less than full algorithm reconstruction
- Goal drives target class: native binary, bytecode, mobile package, firmware, or protocol traffic
- Write a falsifiable objective with an explicit success condition and stopping point
- Record the goal in writing before analysis begins—it anchors scope, tooling, and reporting

Tool Selection: Matching Platforms to the Target



Disassemblers and decompilers

Ghidra, IDA, Binary Ninja, radare2/Rizin with Cutter



Debuggers and tracers

GDB, LLDB, WinDbg, x64dbg, plus syscall and API tracing



Dynamic instrumentation with Frida

inject JavaScript into live processes across platforms



Specialized toolchains

JADX and apktool (Android), ILSpy/dnSpyEx (.NET), binwalk/Unblob (firmware)



Key Considerations

Decide by target format, depth needed, scripting, headless support, skills, cost, reproducibility

Building a Selection Matrix and a Student-Friendly Stack

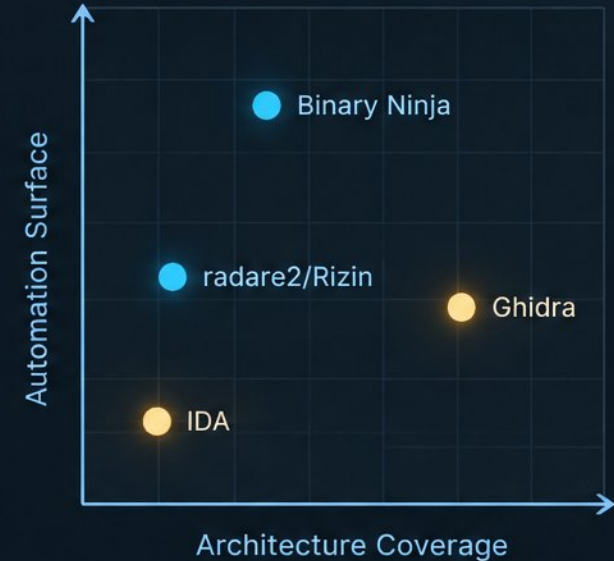
ANALYZE
COMPARE
CHOOSE
LEARN

- Score each tool on architecture coverage, decompiler quality, debugging, and automation
- Licensing: Ghidra and radare2/Rizin are free; IDA and Binary Ninja are commercial
- Trade-offs: Ghidra broad but heavy; Binary Ninja fast with strong API; radare2 CLI-first
- Minimum stack: Ghidra + native debugger + Frida + triage utilities
- Write one paragraph justifying your choices; include it in the final report

Cost vs. Decompiler Quality



Architecture Coverage vs. Automation Surface



Minimum Practical Student Stack

Ghidra

+

native
debugger

+

Frida

+

triage
utilities

Requirements: Authorization, Law, and Ethics



Establish authorization first: ownership, written permission, or scope document



EU interoperability exceptions differ from US anti-circumvention rules; EULAs can constrain independently



Key risks: anti-circumvention claims, ToS breach, unauthorized access, tool distribution



Minimize data exposure, avoid production impact, stay within declared scope



binwalk and Unblob invoke third-party extractors on untrusted input: isolate extraction



Required Documentation Before Analysis Begins



Authorization and scope:

systems, copies, accounts, permitted methods, explicit exclusions



Analysis plan:

objectives, hypotheses, tool versions, and a defined stopping condition



Chain of custody:

acquisition method, hashes, provenance, storage location



Reproducibility record:

exact commands, tool versions, environment, configuration changes



Data handling rules:

retention and redaction for binaries, memory dumps, captured traffic

Preparation and Reconnaissance: The Isolated Lab



Lab design: isolated VMs or containers, snapshots, restricted network, no production credentials



Artifact inventory: hashes, file type, architecture, size, build metadata, provenance



Static triage: strings, symbols, imports, sections, entropy, packer detection



Vendor security check and sandbox awareness; assume VM escape risk with untrusted samples

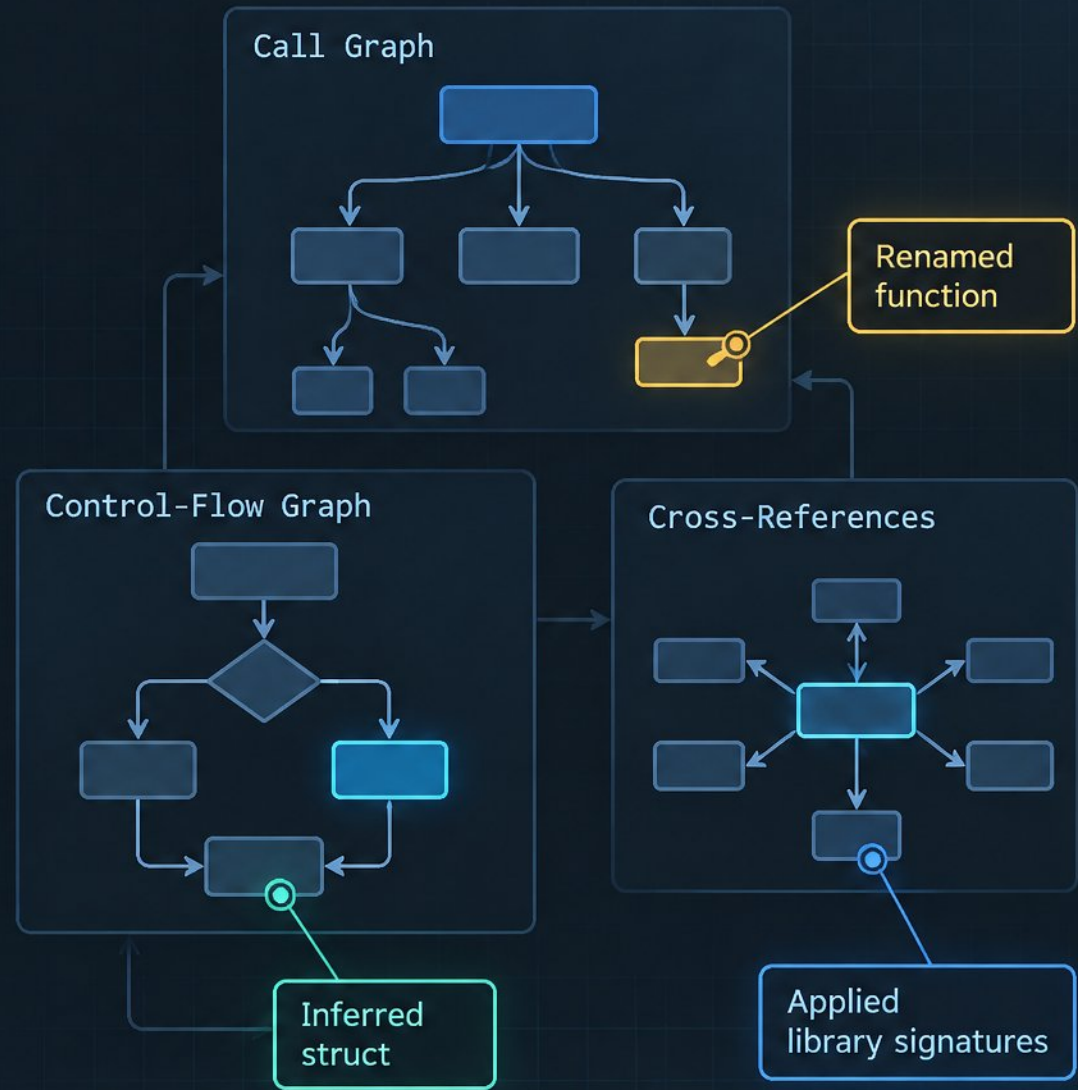


Form hypothesis: likely behavior, interesting logic locations, definition of a successful result



Static Analysis Workflow

- **Navigation basics:** functions, cross-references, control-flow graphs, call graphs
- **Recognize compiler patterns:** loops, switch tables, inlined functions, struct access
- **Recover data and types** from usage, not declarations
- **Apply library signatures** (FLIRT), type libraries, and community function naming databases
- **Annotate and automate** via IDA Python, Ghidra scripts, or Binary Ninja API



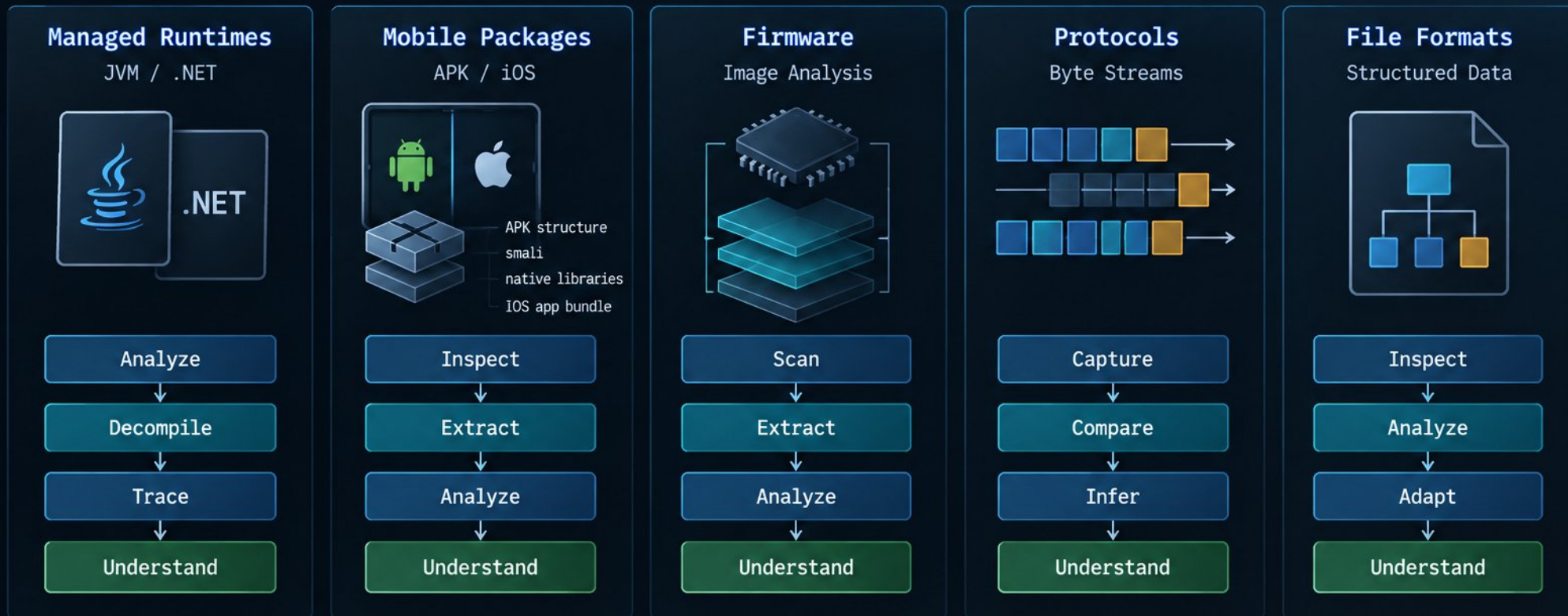
Dynamic Analysis and Debugging Workflow

- **Run-to-observe discipline:**
clean snapshots, controlled inputs, logged system activity
- **Debugger workflow:**
breakpoints, watchpoints, single-stepping, stack and register inspection, experimental patching
- **Tracing and instrumentation:**
syscall/API tracing, Frida hooks, and coverage collection
- **Frida in practice:**
resolve modules with current APIs, hook by offset in stripped PIE binaries, patch return values
- **Iterate between static hypotheses and runtime behavior;**
recognize anti-debugging and environment checks within scope



Specialized Targets: Bytecode, Mobile, Firmware, and Protocols

- **Managed runtimes:** decompile JVM and .NET assemblies, trace with runtime tooling
- **Mobile packages:** APK structure, smali, native libraries, iOS app bundle basics
- **Firmware:** signature scanning, recursive extraction, entropy graphs, filesystem carving
- **Tooling reality:** binwalk v2 vs v3 flags differ; external extractors like sasquatch required
- **Protocols and file formats:** structured comparison, format inference, adapted workflow per target class



Documenting Findings, Communicating Results, and Course Wrap-Up



- Report structure: objective, scope, method, evidence, findings, limitations, recommendations
- Evidence quality: commands, hashes, annotated listings, reproducibility notes for re-runs
- Separate observed facts from inference; state confidence levels explicitly
- Audience-aware communication: engineer, manager, legal, and security variants
- Responsible disclosure: private report first, CVE assignment where appropriate
- Safe practice platforms: crackmes.one, picoCTF, ReRange, pwn.college

PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/ec2c618f/public