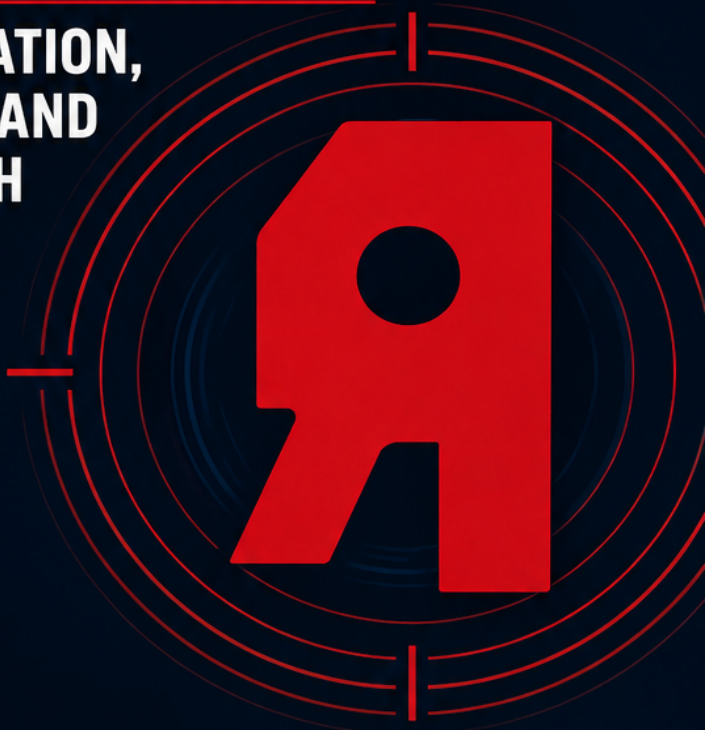
# MASTERING FRIDA

```
Interceptor.attach(ptr, {
  onEnter(args) {
    console.log(
    "onEnter", this.context
    );
  },
  onLeave(retval) {
    console.log(
    "onLeave", retval
    );
  }
});

Memory.scanSync(address, size
  onMatch(address, size) {
   console.log(
    "match at", address,
    size
  ));
});
```

## DYNAMIC INSTRUMENTATION, REVERSE ENGINEERING AND SECURITY TESTING WITH FRIDA.RE



**HOOK ANYTHING. SEE EVERYTHING.**
Instrument functions, trace execution, and inspect behavior in real time.

**UNDERSTAND WHAT RUNNING CODE IS REALLY DOING.**
Go beyond the disassembly. Observe, modify and control live processes.

**BUILD POWERFUL TOOLS FOR REVERSE ENGINEERING AND SECURITY TESTING.**
From quick scripts to production-quality instrumentation.

**CROSS-PLATFORM. ONE WORKFLOW.**
Windows, Linux, macOS, Android, and iOS.

```
var api = new NativeFunction(
  ptr, "int", ["int", "pointer"]));
var result = api(42, ptr("hello"));
console.log("result:", result);
```

| WINDOWS | LINUX | macOS | ANDROID | iOS |
|---|---|---|---|---|
| 10 / 11 | x64 / ARM | INTEL / APPLE SILICON | ARM / ARM64 | JAILBROKEN / NON-JAILBROKEN |

GEORGE CRAWFORD

# Mastering Frida

## Dynamic Instrumentation, Reverse Engineering and Security Testing with frida.re

Steve Publications

This book is available at https://leanpub.com/masteringfrida

This version was published on 2026-08-15

# Contents

# Dynamic Instrumentation, Reverse Engineering and Security Testing with frida.re

**About this book.** This book takes you from your first `frida-ps` command to building production-quality instrumentation tools on top of Frida, the open source dynamic instrumentation toolkit at the center of modern reverse engineering and security testing workflows. Every concept is grounded in how things actually work: what happens inside a process when a hook is installed, why the host/agent split exists, how the native API surface maps to operating system primitives and where each platform (Windows, Linux, macOS, Android, iOS) diverges. Examples run against software you own or are explicitly authorized to test, and every chapter explains not just what an API does but when to reach for it, what it costs and what breaks when you use it wrong.

# Introduction: The Art of Asking a Running Program Questions

Imagine an application sitting on your screen about to transmit data over the network. You know, from reading its source or from a static analysis pass, that somewhere between the moment the data is assembled in memory and the moment it leaves as ciphertext, there is a function call where the plaintext still exists in clear form. Static tools will never show you that moment. The bytes are not on disk, they are not in a binary section, they exist only for a few microseconds while a particular instruction sequence is executing on a particular thread. If you want to see them, you have no choice but to ask the running program itself.

That is exactly what Frida does. It lets you inject code into a process that is already alive and make it report what it is doing, what it is calling and what it is passing around. You can read its memory, scan it for byte patterns, step through its control flow block by block, replace the behavior of any exported function with your own JavaScript, or call its internal functions as if they were ordinary API methods. The toolkit describes itself as "Greasemonkey for native apps" [1], and that framing is more accurate than it sounds: just as a userscript rewrites what a web page does without touching the browser's source, an injected Frida script rewrites what a native program does without recompiling it.

A concrete first taste. Install the command line tools with `pip install frida-tools`, then run [2]:

```
$ frida-trace -i "recv*" -i "read*" twitter
recv: Auto-generated handler: .../recv.js
recvfrom: Auto-generated handler: .../recvfrom.js
...
Started tracing 21 functions. Press Ctrl+C to stop.
```

Frida injected itself into the running Twitter process, enumerated its shared libraries, found every function whose name starts with `recv` or `read`, and

generated an editable JavaScript handler for each one. When the application next calls `recvfrom`, your handler fires and can print something like [3]:

```
recvfrom(socket=70, buffer=0x32cc018, length=65536, flags=0x0,
↪  address=0xb0420bd8, address_len=16)
```

No source code. No recompilation. No debugger breakpoints slowing the program down. Just a live process that has been taught to narrate its own behavior. This book is about making that moment routine, and then going far beyond it.

## What this book promises

By the end of this book you will be able to:

- Set up complete instrumentation laboratories on Windows, Linux, macOS, Android, and (where feasible) iOS, and verify each layer works before depending on it.
- Use the Frida command line tools fluently: listing processes, spawning versus attaching, interactive exploration in the REPL, and function tracing with `frida-trace`.
- Write injected agents in JavaScript and TypeScript that observe, modify, and drive native code, using the full GumJS API surface: process and module introspection, memory reading, scanning, and writing, function interception, native calls, callbacks, and control flow tracing.
- Instrument managed runtimes: Java and Kotlin on Android through the Java bridge, Objective-C (and Swift interop) on iOS through the Objective-C runtime.
- Build host-side tools in Python that manage devices, sessions, scripts, and RPC calls, with proper error handling and structured output.
- Engineer production-quality Frida tooling: reusable agent architectures, performance optimization, reliability under failure, version maintenance, and an honest understanding of how instrumentation is detected and what the authorization boundaries are.

The through-line of the book is a single mental model with three layers. The first layer is the host/agent split: your Python or Node.js program on

the outside talks to an injected agent inside the target, and everything about Frida's ergonomics follows from that boundary. The second layer is the native API surface (GumJS), the JavaScript-visible view of a C instrumentation library called Gum, which gives you pointers, modules, threads, memory, interceptors, and tracers. The third layer is the platform runtimes: the ABI and module system of the operating system, plus managed environments like Android's ART and Apple's Objective-C runtime, each with their own rules about when and how you may observe them. Most Frida confusion comes from mixing these layers up. A hook that works on Linux but not on macOS is usually a platform-runtime problem wearing a native-API costume. Every chapter in this book names which layer it is operating on.

## How the book is organized

The material is arranged in five parts that build on each other. Part I establishes foundations: what dynamic instrumentation actually is at the operating system level, how Frida's components fit together, and how to stand up a laboratory you can trust. Part II develops core skills: the command line tools, the agent runtime and host/agent messaging, and the navigation APIs for processes, modules, threads, and memory. Part III covers the instrumentation core: function interception with `Interceptor`, calling native code in both directions with `NativeFunction` and `NativeCallback`, and control flow tracing with `Stalker`. Part IV is platform-specific: Android and Java, iOS and Objective-C, and desktop plus server-side and remote scenarios. Part V is engineering: the Python bindings as a tool-building foundation, and the practices that separate a throwaway script from a maintainable instrument, including performance, reliability, detection awareness, authorization, and long-term maintenance.

The book ends with a conclusion that ties the three layers together, followed by reference material: a command quick reference, an API quick reference organized in context, a systematic troubleshooting reference, a glossary, and a curated list of authoritative sources. There are no end-of-chapter exercises; instead, each chapter teaches through complete, annotated walkthroughs you can reproduce against targets you own or are authorized to test.

## Authorization, ethics, and the boundaries of this book

Before any technique, one rule that governs everything else: you may instrument software you own, software you have written, deliberately vulnerable laboratory targets, or systems where you hold explicit written authorization to perform dynamic analysis. Frida is a general-purpose tool with no built-in sense of who owns the process it touches. The legal and moral boundary is entirely on you, and it varies by jurisdiction, by employment contract, and by the scope document you signed. This book frames every dual-use technique around controlled labs: your own applications, open-source intentionally vulnerable targets, or explicitly authorized assessments. Where a technique is commonly used offensively (for example, defeating integrity checks), the book explains the mechanism so you understand the threat model and can test for it in authorized engagements, but it does not provide evasion recipes. Chapter 15 treats detection and authorization in depth, including what defenders should look for and why "stealth" is an engagement-scoping question rather than a technical footnote.

A practical note on versions. This book targets Frida 17.x, the current release line as of mid-2026 (the latest stable at writing time is 17.17.0) [4]. Frida 17.0.0, released in May 2025, was a breaking release: the Java, Objective-C, and Swift runtime bridges were unbundled from the core runtime, legacy enumeration and memory-read API styles were removed, and the Module API moved to an instance-based pattern [4]. Much of the public tutorial material online predates that change. Wherever this book's examples diverge from what you find in older articles, this book follows the current official documentation, and it calls out the difference explicitly so you can read both.

## What you need to get started

For most of the book, a single computer with Python 3.x is enough: `pip install frida-tools` gives you the command line tools, and `pip install frida` gives you the Python bindings [2]. You will also want one small target program that you own. The official documentation walks through compiling a tiny C program (`hello.c`) whose function address you can print and then hook from Python [5]; that pattern is the backbone of the early chapters, and it works identically on Windows, Linux, and macOS.

The platform chapters need more. The Android chapter wants either a rooted device or an emulator where you can place a `frida-server` binary, plus at least one intentionally vulnerable open-source app to test against [6]. The iOS chapter works best with a jailbroken device for full coverage; without a jailbreak, the realistic options are instrumenting your own debuggable apps via the Frida Gadget or using a simulator where available [7]. The desktop chapters run entirely on whatever machine you are reading this book on. A recommended directory layout for keeping all of this organized is given in Chapter 3, and it stays useful through the rest of the book.

## Prerequisites and how to read

You should be comfortable in a terminal and fluent in at least one programming language; Python and JavaScript appear throughout, and code examples assume you know what a function, an object, and a callback are. You do not need prior reverse engineering experience. Where C concepts matter (pointers, calling conventions, struct layout), they are introduced in context with enough depth to be useful, but the book is not a C textbook.

Read the parts in order if you are new to Frida; each chapter states its prerequisites at the top and only relies on concepts from earlier chapters. If you already know the basics and want a specific capability, Part II and Part III are self-contained enough to jump into once you have a working laboratory from Part I. The appendices are designed for lookup: Appendix A for commands, Appendix B for API signatures in context, Appendix C for troubleshooting by symptom, Appendix D for terminology.

One last expectation to set. Dynamic instrumentation is an interactive discipline. Reading about `Interceptor.attach` teaches you the shape of the tool; watching a hook fire against your own process and seeing the argument values scroll past is what makes it stick. Every walkthrough in this book is written so that you can run it, break it deliberately, and see why it broke. That loop, form a hypothesis, write the smallest hook that tests it, read the output, refine, is the actual skill this book is teaching.

# Part I: Foundations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 1: The Process You Can't Stop: Dynamic Instrumentation Fundamentals

**Prerequisites:** none. **Feeds into:** Chapters 2 through 15 (the injection model described here is the foundation of everything that follows).

## Static vs Dynamic Analysis

Every analysis technique you have ever used falls on one side of a line: did it observe the program while it was executing, or not? Static analysis reads the program as data. Disassemblers like Ghidra, IDA Pro, or radare2 turn machine code back into readable instructions. Decompilers approximate source. String extractors find constants. This approach is powerful and safe: the binary never runs, so it cannot phone home, corrupt your disk, or notice you.

But static analysis has a hard wall. It sees what the program can do in all possible worlds, not what this particular run actually did. Consider four concrete limits. First, runtime state is invisible: the value of a counter, the contents of a decrypted buffer, the configuration loaded from a server none of these exist anywhere in the file. Second, addresses are lies at rest: with address space layout randomization (ASLR), the same binary loads at different base addresses on every run, so any static offset you compute must be re-based dynamically anyway. Third, code that is assembled or fetched at runtime (just-in-time compiled code, data-driven dispatch tables, downloaded payloads) simply is not in the file you are reading. Fourth, control flow under obfuscation (opaque predicates, indirect call chains, virtualized dispatch) can be nearly unreadable statically while being perfectly ordinary to a tracer that just watches which instructions actually execute.

Dynamic analysis crosses the line: it observes the program while it runs. You get real values, real addresses, real control flow, in the order they actually happened. The price is that you are now coupled to a live process. It can crash,

and with it your instrumentation dies. Its behavior may depend on inputs you cannot fully reproduce. And, critically for security work, the program may notice you. Debuggers have been detected since the 1980s; modern applications check for traces of instrumentation as a matter of course.

The two approaches are complementary rather than competing. In practice, dynamic instrumentation is almost always guided by static analysis: you read the binary to find candidate functions (an AES key schedule, a network send wrapper, a license check), then you hook those candidates dynamically to see what actually flows through them. Frida sits squarely on the dynamic side, and it is worth understanding why that side requires the machinery described next.

## What Happens Inside a Running Process

To instrument a process you need a working picture of what a process is in the eyes of the operating system. A process is, at its core, an address space plus the execution context (threads) that operates on it. The address space is a contiguous virtual range, typically 48 bits wide on x86-64 and ARM64 user space, divided into regions with different permissions:

```
Low addresses
+-----------------------------------------+
| stack (grows down)                      |  RW-
| ...                                     |
| heap: malloc'd chunks                   |  RW-
| ...                                     |
| shared libraries (.so / .dylib / .dll)  |  R-X, then RW- for data
| ...                                     |
| executable image (text, rodata, data)   |  R-X / R-- / RW-
+-----------------------------------------+
```

A few properties of this layout matter for everything that follows. Code and data are separated by page permissions: instruction pages are read-execute, data pages are read-write, and the operating system will fault a process that tries to write through an instruction page or execute a data page. Frida's hooking machinery has to respect this rule, which is one reason its internals are more elaborate than "just patch a few bytes." Shared libraries are mapped into every process that loads them, each at its own ASLR-chosen base address, so the same symbol (`open`, for example) lives at a different address in every

process and on every boot. The kernel maintains, per process, tables of these mappings (visible on Linux as `/proc/<pid>/maps`), the loaded threads, and the file descriptors.

Symbols are the bridge between names and addresses. A compiled function has a name in its symbol table (`send`, `CCCryptorCreate`, `Java_com_example_App_login`), and the dynamic linker resolves those names to concrete addresses at load time. When symbols are stripped (common in release builds), the names vanish from the binary but the functions remain; you then find them by other means, which is exactly what Frida's module and symbol APIs help you do in Chapters 6 and 8.

A second concept: a process is multithreaded by default on every modern platform. Each thread has its own stack, registers, and program counter, but all threads share the same address space. This matters for instrumentation because hooks are usually installed globally (every call to `open` from any thread fires your hook), while the state you observe (registers, backtraces, the current thread ID) is per-thread. Keeping those two scopes straight prevents an entire class of confusing bugs.

## Operating-System Injection Primitives

Dynamic instrumentation requires getting code into another process's address space and making that process execute it. No operating system offers a polite "run this function in process X" call, so every injection framework is built from the same three low-level moves: gain control of the target, write your payload into its memory, and redirect execution to your payload's entry point. The specific primitives differ per platform, and knowing them explains both what Frida can do and where it fails.

**Linux and Android: ptrace and process_vm_writev.** On Linux, the `ptrace` system call lets one process observe and control another thread: it can attach to a running thread (which stops it), read and write its memory and registers, single-step it, and restart it [32]. The manual page makes two details explicit that you will feel later in this book. Tracing is per-thread, not per-process, so a multithreaded target means coordinating many tracees. And attachment is gated by permission checks: user ID matching, the `CAP_SYS_PTRACE` capability, the target's dumpable status, and security modules such as Yama (which implements `/proc/sys/kernel/yama/ptrace_scope`, where

the default value 1 restricts tracing to parent processes) and SELinux (the reason Android setups need a rooted shell) [32]. For bulk memory transfer, Linux provides `process_vm_readv` and `process_vm_writev`, which copy data directly between two address spaces with no kernel-space bounce buffer [33]. A typical Frida-style injection on Linux therefore looks like: `PTRACE_ATTACH` (or seize) the target's threads, stop them, use memory writes to place the agent shared library and a small bootstrap stub in the target's heap, rewrite one thread's program counter and return address to run the stub, which calls the platform loader (`dlopen`) on the injected library, then restore everything and detach. The target process now contains a full instrumentation runtime that never appeared in its original image.

**macOS and iOS: Mach task ports.** Apple platforms use the Mach kernel layer, where processes are "tasks" and inter-process control flows through ports: secure unidirectional channels whose access is governed by port rights that act as capabilities [35]. The key primitive is `task_for_pid`, which asks the kernel for a send right to another process's task port. With that right you can stop tasks, kill them, and manipulate their address space. On macOS this right is mediated by the `taskgated` daemon, which checks entitlements and user privileges (the `system.privilege.taskport` authorization); the official troubleshooting documentation for Frida points at exactly this gate when spawning fails with "task_for_pid returned (os/kern) failure" [8]. On iOS the same mechanism has been enforced strictly for a long time, which is why jailbreaks historically included a "tfp0" patch to lift it, and why non-jailbroken iOS instrumentation goes through different doors entirely (the Gadget, covered in Chapters 2 and 12). Once Frida holds the task port on Apple platforms, injection proceeds by mapping the agent into the target's address space via Mach VM operations and scheduling a thread to run its initializer.

**Windows: CreateRemoteThread.** Windows offers no ptrace at all. The classic injection path uses `CreateRemoteThread`, which creates a new thread whose start function must already exist inside the target process [34]. Since you cannot point it at your own code, the standard recipe is: open the target with sufficient access rights (`PROCESS_CREATE_THREAD`, `PROCESS_VM_WRITE`, and friends), write the path of a DLL into the target's memory, create a remote thread whose start address is the export `LoadLibraryA` inside the already-loaded kernel32.dll, and let that thread load your DLL, whose entry point runs your bootstrap [34]. Microsoft's own documentation includes a warning worth quoting in full: injecting remote threads "can make single-threaded apps multithreaded, change timing/memory layout, trigger DLL entry points,

and cause deadlocks" [34]. That sentence is the honest summary of a core tradeoff in all dynamic instrumentation: you are changing the program you are studying, and the change has side effects.

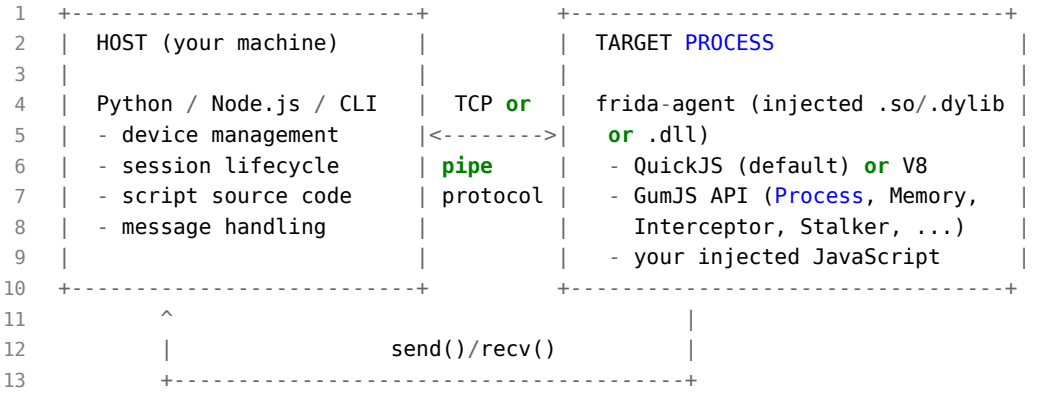
Notice what all three platforms have in common. The injected payload is a shared library (`.so`, `.dylib`, or `.dll`) whose constructor runs inside the target. That library is Frida's **agent**: it sets up the JavaScript runtime, opens a communication channel back to your host program, and starts executing your scripts. Everything else in this book is built on that one fact.

# Chapter 2: Frida's Architecture and Ecosystem

**Prerequisites:** Chapter 1 (process model and injection primitives). **Feeds into:** every subsequent chapter; the terminology defined here is used without re-explanation.

## The Host-Agent Model

Strip Frida down to its essence and two programs remain, separated by a process boundary. On one side is the **host**: your Python script, your Node.js tool, or the `frida` command line itself. It lives in its own address space, runs with your privileges on your machine (or connects over the network), and owns the user experience: listing devices, attaching to processes, loading scripts, printing results. On the other side is the **agent**: a shared library injected into the target process that hosts a JavaScript engine and exposes an API for poking at the target's internals [10].




The channel between them is a bidirectional message stream. The agent sends you `send()` payloads and uncaught exceptions; you post messages back with the host-side equivalent of `post()`, and call into the agent through RPC

exports (Chapter 5). This split is not an accident; it is the single most important design decision in Frida, and it buys you three things. First, crash isolation: a bug in your analysis code (an infinite loop in Python, a bad regex) does not corrupt the target process, and a crash in the target does not take down your host tooling. Second, language freedom: anything that can speak the protocol becomes a frontend, which is why official bindings exist for Python, Node.js, Go, Swift, .NET, and QML [1]. Third, deployment flexibility: the same agent works whether it was injected by `frida-server` on an Android phone, embedded in your own app as a Gadget, or preloaded via `LD_PRELOAD` on a server [10].

Worth understanding at this level of abstraction: the agent's JavaScript runs inside the target's address space. When your script reads a word from target memory with `ptr.readU32()`, there is no network round trip for the read itself; the GumJS layer (the C code behind the JavaScript API) performs the read directly in-process [9]. Only host-bound messages cross the channel. This distinction, in-process work versus cross-channel messaging, reappears constantly when we discuss performance in Chapter 15.

The agent runtime is a full JavaScript engine embedded in C. Since Frida 14, the default engine is QuickJS, a small embeddable engine; V8 is available as an alternative and can be selected per script or per tool invocation (`--runtime qjs` or `v8`) [4, 15]. The runtime executes on a single dedicated thread inside the target (you will see it named `gum-js-loop` in thread listings), driven by a GLib main loop. Your script's timers (`setTimeout`, `setImmediate`) and message handling all run on that one thread, which has real consequences for how you write agents: long-running synchronous work in your script blocks the agent's own event processing. Chapter 5 covers this constraint in depth.

## Component Tour

Frida is a family of components built around one core library. Here is the map, with each component's job and when you will meet it in this book.

**frida-core (Gum, GumJS, frida-agent).** The heart of everything. `frida-gum` is the cross-platform C instrumentation and introspection library: module enumeration, memory access, function interception, code tracing [1]. Layered on top is GumJS, the JavaScript bindings that turn that C API into the `Process`, `Memory`, `Interceptor`, and friends you will use throughout this book [9]. frida-core packages these into the injectable agent shared library. You rarely interact

with frida-core directly except when building from source or embedding it in your own tools (the C API is documented separately for that audience).

**frida-server.** A standalone daemon that runs on a device and exposes frida-core over TCP, listening on `localhost:27042` by default [10]. This is the component you install on Android devices, jailbroken iOS devices, and remote Linux servers. It performs the actual injection work (the ptrace or Mach operations from Chapter 1) locally, which is why it must run with sufficient privileges on the target system. The release artifacts are named `frida-server-{platform}-{arch}` and distributed as compressed binaries on GitHub Releases [31, 46].

**frida-tools.** The Python package that provides the command line suite: `pip install frida-tools` [2]. As of the current release line (frida-tools 14.x against Frida 17.x), it installs seventeen console scripts: `frida` (the interactive CLI/REPL), `frida-ps`, `frida-ls`, `frida-ls-devices`, `frida-kill`, `frida-discover`, `frida-trace`, `frida-strace`, `frida-itrace`, `frida-create`, `frida-compile`, `frida-pm`, `frida-apk`, `frida-push`, `frida-pull`, `frida-rm`, and `frida-join` [25]. Older releases included tools like `frida-bgs` and `frida-pk` that are no longer present; if a tutorial references them, you are reading pre-17 material. Chapters 4 and 13 cover the tools in depth.

**Language bindings.** `frida-python` (the `frida` PyPI package) gives you the device/session/script/RPC model programmatically [28]. `frida-node`, `frida-go`, `frida-swift`, `frida-clr`, and `frida-qml` provide the same model in other ecosystems [1]. Chapter 14 is built around the Python bindings; the others follow the same shape.

**frida-gadget.** A shared library you embed into a program you control, for the cases where injection mode is not available or not appropriate: non-rooted Android apps, App Store iOS apps you are developing, or any target where you can modify the build [10]. The Gadget starts from the dynamic linker's constructor, reads an adjacent configuration file, and then behaves according to its configured interaction mode (listen for a client, run scripts autonomously, or join a portal cluster) [12]. Chapters 3, 11, and 12 use it for lab setup.

**frida-portal.** A connection relay and cluster manager that lets Gadget instances and servers join a shared instrumentation network; the Gadget's `Connect` mode points at a portal (default `127.0.0.1:27052`) [12, 31]. Portals are relevant when you are orchestrating many devices or building distributed

tooling; this book mentions them where they matter but does not require them for any workflow.

**frida-inject.** A small command line utility that injects a Gadget or agent into an already-running process on the local machine [31]. Useful for experiments and for understanding what the server does internally, one step at a time.

Two more artifacts round out the ecosystem. **Bridges** are the runtime interop layers for managed environments: `frida-java-bridge` (Android/-Java), `frida-objc-bridge` (Objective-C), and `frida-swift-bridge` (Swift). Since Frida 17 they are no longer baked into the core agent; a compiled agent must explicitly import the bridge it needs (`import Java from "frida-java-bridge"`), while the REPL and `frida-trace` still bundle all three for compatibility [11]. **Build options** let you trim or extend components: the footprint documentation lists build flags such as `FRIDA_JAVA_BRIDGE`, `FRIDA_OBJC_-BRIDGE`, and `FRIDA_DATABASE` (for the in-agent SQLite API), which matters when building custom servers with smaller attack surfaces or extra capabilities [24].

## Supported Platforms and Build Targets

Frida injects into native applications on Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX [1]. In practice, the platforms you will work with most are the five in your laboratory: a desktop OS for development, Android for mobile instrumentation, and (where available) an Apple mobile platform. The support matrix has some version-dependent edges worth knowing up front:

| **Platform** | **Typical setup** | **Notes (version-sensitive)** |
|---|---|---|
| Windows (x86, x86-64, ARM64) | `frida-server.exe` or local attach | Windows on ARM support landed in 16.5 [4]; some protected processes remain out of reach even as administrator [41] |

| Platform | Typical setup | Notes (version-sensitive) |
|---|---|---|
| macOS (Intel, Apple Silicon) | local attach; `frida-server` for remote | System binaries may be protected by SIP and code signing; the troubleshooting docs describe the authorization workarounds and their cost [8] |
| GNU/Linux (x86, x86-64, ARM, AArch64) | local or `frida-server` | Yama `ptrace_scope` restricts tracing non-child processes by default [32]; eBPF-based spawn gating added in 17.16 [4] |
| Android (ARM, ARM64; x86/x86-64 less tested) | rooted device or emulator with `frida-server`; Gadget without root | Official docs recommend recent AOSP on Pixel/Nexus or arm/arm64 emulators [6]; 17.6 replaced intrusive Zygote instrumentation with a lightweight external mechanism for stability [4] |

| Platform | Typical setup | Notes (version-sensitive) |
|---|---|---|
| iOS (ARM64, ARM64E) | jailbroken device with `frida-server`; Gadget for your own apps | Rootless jailbreak support since 16.1.5; arm64e (pointer authentication) is supported with ongoing fixes through 17.16 [4]; non-jailbroken use is limited to debuggable apps via auto-injected Gadget [7] |

Three cross-cutting version facts matter for everyone reading this book. First, Frida ships a continuous stream of minor releases; the news page on frida.re is the authoritative changelog, and major releases (14.0, 15.0, 16.0, 17.0) have each changed behavior in ways that invalidate older tutorials [4]. Second, platform support moves: capabilities like hardware breakpoints and watchpoints (16.5), Apple simulator access via the CoreDevice protocol (16.3), and the `frida-strace` system call tracer (17.8) did not exist a year earlier [4]. Third, when in doubt about whether a feature exists in your version, check `Frida.version` from inside an agent or the release notes; Chapter 15 turns this into a maintenance habit.

## Terminology and Versioning

A compact glossary of the terms that recur in every Frida conversation, defined precisely:

- **Device**: an abstraction over "somewhere I can list processes and attach to them." Devices come in types: `local` (your own machine), `usb` (an Android or iOS device over USB), `remote` (a host running frida-server, addressed as `host:port`), plus special cases like emulators discovered through platform tooling [28, 30]. The Python API's `enumerate_devices()` and the CLI's `frida-ls-devices` both list what is currently reachable.

- **Session**: an active attachment to one process on one device. A session owns scripts; detaching a session removes your instrumentation from the target. Sessions can die for several distinct reasons (the process exited, you detached, the connection dropped), and mature tools handle each case deliberately (Chapter 14).
- **Script**: a unit of injected JavaScript with a lifecycle: created from source (or a compiled bundle), loaded into the agent, optionally unloaded or eternalized [29]. Scripts carry messages both directions and expose RPC exports.
- **Export**: a function you declare in `rpc.exports` inside an agent so the host can call it by name (Chapter 5).
- **Gadget**: the embeddable shared-library variant of the agent, configured by an adjacent file (Chapter 3, Chapter 12) [12].
- **Bridge**: a runtime interop layer for Java, Objective-C, or Swift; explicit imports in compiled agents since Frida 17 [11].
- **Realm**: on Android, processes can contain multiple "realms" of code: the native realm and emulated realms created by the NativeBridge (for running e.g. x86 apps on ARM hardware). Frida can target a specific realm when attaching or enumerating modules [4].
- **Spawn gating**: the mechanism by which Frida holds a newly spawned process suspended at its entry point until your scripts are loaded, guaranteeing your hooks exist before any application code runs (Chapter 6) [30].

And then the rule that causes more confusion than any technical detail: **client and server versions must match exactly.** The `frida` Python package, `frida-tools`, and the `frida-server` binary on each target device all carry a version number, and the protocol between them is not tolerant of mismatch. If your host runs 17.17.0 and the Android device runs 16.5.3, you will get connection failures or bizarre behavior, and no amount of debugging your script will fix it. Check versions on both sides before troubleshooting anything else: `frida --version` on the host, and `frida-server --version` (or the version stamped into the release artifact name) on the device [27, 46].

The same strictness applies to the bridge packages in compiled agents: a `frida-java-bridge` built for Frida 17 will not load under a Frida 16 agent. This is deliberate; the protocol and the C API move together, and silent compatibility would hide real breakage.

Finally, a note on the breaking release that defines the current era. Frida 17.0.0 (May 2025) made three changes that you will encounter throughout this book: bridges unbundled from GumJS [11], legacy enumeration and memory-read API styles removed in favor of modern iteration and pointer-chained access, and static `Module.*` convenience methods replaced by an instance-based pattern [4, 42]. Concretely, code written for Frida 16 that calls `Process.enumerateModulesSync()` or `Memory.readU32(ptr)` no longer works; the 17.x equivalents are `Process.enumerateModules()` (which returns the collection directly) and `ptr.readU32()` [42]. All examples in this book use the current style, and Chapter 15 shows you how to keep your own tooling honest as versions move.

With the component map in hand, the next chapter builds the laboratory: installing everything on your machines, standing up an Android target, deciding what is realistic for iOS, and verifying each layer with concrete checks before any instrumentation begins.

# Chapter 3: Building Your Instrumentation Lab

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Host Installation on Desktop Platforms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Android Lab: Emulator, Root, and Targets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## iOS Lab Options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Verifying the Stack and Laying Out Your Workspace

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Part II: Core Skills

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 4: The CLI and the REPL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Reading Processes: frida-ps and Friends

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Attaching and Spawning from the Command Line

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## The Interactive REPL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Script Injection with -e and -l

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 5: Your First Agent: JavaScript on the Other Side

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## The Agent Runtime and Its Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Messaging: send, recv, and Post

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## RPC Exports in Both Directions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Structuring Reusable Agents (JavaScript and TypeScript)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 6: Processes, Modules, and Threads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## The Process Object and System Topology

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Modules: Where Code Lives at Runtime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Threads and Backtraces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Spawn Versus Attach: Timing the Instrumentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 7: Memory: Reading, Scanning and Writing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Pointers and the Memory Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Reading and Writing Primitives and Strings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Scanning and Enumeration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Protection, Allocation, and Lifetimes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Part III: Instrumentation Core

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 8: Interceptor: Hooking Native Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## attach: Observing Calls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## replace and implement: Rewriting Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Hooking C++ Members and Vtables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Mistakes, Deadlocks, and Safer Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 9: Calling Native Code — NativeFunction and NativeCallback

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## NativeFunction: The Signature System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Calling Conventions and Structs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## NativeCallback: Giving Native Code a Callback

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Pitfalls: Reentrancy, Lifetimes, and Errors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 10: Stalker — Control-Flow Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Following Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Events: Instructions, Blocks, Slices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Transforms: Instrumenting the Trace Itself

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Costs, Limits, and Practical Use

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Part IV: Platform Instrumentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 11: Android and Java

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Running Frida on Android

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## The Java Bridge and the VM Lock

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Hooking Java Methods

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Finding Classes, Instances, and Native Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 12: iOS and Objective-C

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## The Objective-C Runtime and Frida

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Running Frida on iOS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Hooking and Swizzling Objective-C Methods

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Swift Interop and Limitations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 13: Desktop, Server-Side, and Remote Instrumentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Windows Instrumentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Linux and macOS Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Remote Devices and Networked frida-server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Server-Side Use Cases: Daemons, CI, and Automation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Part V: Engineering with Frida

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Chapter 14: The Python Bindings — Host-Side Engineering



## Devices and Sessions in Python



## Scripts and Messages from the Host



## RPC from Python



## Building a Small Instrumentation Tool

# Chapter 15: Production-Quality Frida Tooling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Agent Architecture That Survives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Performance Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Reliability and Failure Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Detection, Safety, and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Keeping Your Tooling Current

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Conclusion: From Hook to Habit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Appendix A: Command Quick Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Tool overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## `frida` (REPL/runner) key flags [13, 49]

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## `frida-ps` flags [14]

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## `frida-trace` key flags [15]

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Appendix B: API Quick Reference in Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Process / Thread / Module

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Process.id`, `Process.arch`, `Process.platform`, `Process.pointerSize`, `Process.pageSize`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Process.mainModule`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Process.enumerateModules()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Process.getModuleByName(name)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Process.enumerateThreads()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Process.enumerateRanges(prot?)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Process.attachModuleObserver(...)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Process.setExceptionHandler(fn)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Module.base`, `Module.size`, `Module.path`, `Module.name`, `Module.version`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `module.getExportByName(name)` / `module.findExportByName(name)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Module.getGlobalExportByName(name)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `module.enumerateExports()` / `module.enumerateImports()` / `module.enumerateSymbols()` / `module.enumerateRanges()` / `module.enumerateSections()` / `module.enumerateDependencies()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `ModuleMap()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Module.load(path)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Thread.backtrace(...)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Thread.sleep(ms)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Memory / pointers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `ptr(...)`, `NULL`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `p.readU8/U16/U32/U64/S*/Float/Double()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `p.readCString/Utf8String(len)/Utf16String/AnsiString/ByteArra`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `p.writeU*/writeCString/writeUtf8String/writeByteArray`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `p.readPointer()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `p.add(n)/sub(n)`, `p.toMatchPattern()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Memory.alloc(size)` / `allocAligned` / `allocUtf8String/Utf16String/AnsiString`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Memory.scan(base,size,pattern,{onMatch,onError,onComplete})` / `scanSync`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Memory.protect(addr,size,prot)` / `queryProtection(addr)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Memory.patchCode(addr,size,apply)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `MemoryAccessMonitor.enable(range,{onAccess})` / `disable()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Int64` / `UInt64`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Interceptor / NativeFunction / NativeCallback / Stalker

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Interceptor.attach(target,{onEnter,onLeave})`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Interceptor.replace(target, impl)` / `replaceFast`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Interceptor.revert()` / `flush()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `retval.replace(value)` (in `onLeave`)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `new NativeFunction(addr, retType, argTypes, cconv?)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `new NativeCallback(fn, retType, argTypes)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `SystemFunction`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Stalker.follow(threadId,{events, transformSync, onReceiveOutput})`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Stalker.unfollow(threadId)` / `flush()` / `invalidate()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### Events: `calls/returns/blocks/instructions/slices`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Java bridge (Android; import `frida-java-bridge` since 17) [9, 11]

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Java.available`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Java.perform(fn)` / `performNow(fn)` / `scheduleOnMainThread(fn)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Java.use("com.x.Y")`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `.method.overload(sig...)` / `$init` / `$new`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `.implementation = fn` / `.unhook()`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### Field access: `obj.field.value`, `this._field.value` on clash

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Java.enumerateLoadedClasses({onMatch,onComplete})`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Java.choose(Class,{onMatch,onComplete})` / `chooseSync`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Java.openClassFile(path)` / `registerClass` / `cast` / `array`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Objective-C bridge (iOS/macOS; import `frida-objc-bridge` since 17) [9, 11]

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `ObjC.availableClasses` / `ObjC.classes.Name`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `Interceptor.attach(cls["- sel:"].implementation, ...)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `method.implementation = ObjC.implement(method, fn(self, sel, ...))`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `new ObjC.Object(ptr)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `ObjC.chooseSync(Class)` / `choose`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `ObjC.registerClass` / `parseHeader`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `ObjC.schedule(ObjC.mainQueue, fn)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Messaging / RPC (agent side) [19]

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `send(payload)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### Uncaught exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `recv(type, cb)`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### `rpc.exports = {...}`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

### Timers: `setTimeout/setInterval/setImmediate` + clears

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Appendix C: Systematic Troubleshooting Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Stage 1: Installation and versions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Stage 2: Connecting and attaching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Stage 3: Hooks not firing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Stage 4: Crashes and corruption

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Stage 5: Messaging and performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Stage 6: Platform-specific

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# Appendix D: Glossary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

# References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.

## Where Frida Fits in the Landscape

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringfrida.