



GHOST
PROTOCOL

RE

THE FIELD LIBRARY

DEEP TRACK

FROM THE COMPILED FORM

Reverse Engineering

What reverse engineering is, the tools,
static analysis, dynamic analysis, and
practical targets.



Ghost Protocol (Pvt) Ltd · Colombo · ghosts.lk

Security internship training · The Field Library · Edition 1 · 2026

THE GHOST PROTOCOL FIELD LIBRARY · DEEP TRACK

Reverse Engineering

What reverse engineering is, the tools, static analysis, dynamic analysis, and practical targets.

Ghost Protocol (Pvt) Ltd · Colombo

The Field Library, Edition 1 · 2026

Copyright 2026 Ghost Protocol (Private) Limited. All rights reserved.

Sold as a PDF for the buyer's own reading. Please do not share or resell the file.

The library

The Field Library is a whole shelf, not one book. The Core Path is the spine everyone reads. The rest is depth and breadth, taken on as the work calls for it. The shelf is complete now, and you are reading the one marked here.

A · THE CORE PATH

O The Oath BUILT

I Groundwork BUILT

II The Web BUILT

III The Wider Field BUILT

IV The Work BUILT

V The Craft BUILT

B · DEEP TRACKS

AI and Agents BUILT

Reading Code BUILT

Internal and Active Directory BUILT

The Web, Deeper BUILT

Cloud and Containers BUILT

Mobile and APIs BUILT

Reverse Engineering YOU ARE HERE

Hardware and the Edges BUILT

C · THE OPERATIONS

Red and Purple Team BUILT

OSINT and Recon BUILT

Report Writing Mastery BUILT

The Consulting Craft BUILT

Bug Bounty BUILT

D · THE OTHER SIDE

Detection and Response BUILT

Threat Hunting and Forensics BUILT

Hardening and Architecture BUILT

E · THE GROUND RULES

The Law, Deeper BUILT

Standards and Compliance BUILT

Ethics and Disclosure BUILT

F · COMPANIONS

The Reference Pack BUILT

The Lab Workbook BUILT

Case Files BUILT

The Bestiary BUILT

The Glossary and the Ledger BUILT

Contents

- Opening 5
- 1** What Reverse Engineering Is 6
- 2** The Tools 10
- 3** Static Analysis 14
- 4** Dynamic Analysis 18
- 5** Practical Targets 22
- Worked answers 27

Read the metal



The target came with no source code, just a compiled binary that did something the client could not fully explain. Blink poked at it, baffled.

BLINK "There is no code to read. It is just a file of noise."

SOL "The code is still in there. The compiler turned it into the machine's language, not into noise, and the machine's language can be read back. Reverse engineering is learning to read it: what the program does, where it decides, what it hides. Slower than reading source, and sometimes the only way in."

Worm's eye was on the one thing a binary cannot hide: what it actually does when it runs.

WORM "read the metal."

So the book is that reading: what reverse engineering is, the tools that open a binary, reading it still and reading it running, and the first targets to cut your teeth on, every one of them yours to take apart.

CHAPTER ONE

What Reverse Engineering Is

A compiled program is the exact truth of what the software does, with everything the authors meant stripped away. Reverse engineering is reading that truth. This chapter is what RE is, static against dynamic, the mindset it takes, and the moment a tester reaches for it.

In Volume II you read the web by watching a request go out and a response come back, because the logic lived on a server you could only poke from outside. Reverse engineering is for the other case: the logic is in your hands, as a file, with no source and no documentation. You open the compiled program itself and work backwards to enough understanding to answer one question. Every technique in this track is a way of recovering meaning from a thing that was built to run, not to be read.

The source is gone; the behaviour is not

When a developer compiles a program, the compiler throws most of the human meaning away. Variable names, comments, the shape of the author's types, the file the function came from: most of that does not survive into the binary, and what does survive is there by accident or for the linker, not for you. What is left is machine code, the exact instructions the processor will run, plus some data and whatever symbols were not stripped out. That is a loss, and it is also the point. The binary cannot lie about what it does, because it is the thing that does it. The marketing can be wrong and the documentation can be out of date, but the `.text` section runs exactly as written.

So reverse engineering is not decompiling a whole program back into tidy source. It is recovering the specific understanding a question needs. Where is the licence key checked. What does this function send on the network, and to where. Why does the app behave differently when a file is present. You almost never read the whole binary, and a tester who tries to is wasting the week. You find the part that answers the question and you read that, closely.

Static and dynamic, the two eyes

Static analysis is reading the binary without running it: disassembly, decompilation, the strings it carries, the places one piece of code refers to another. You see every path the program could take, including the branch that did not fire today and the error handler nobody reached. **Dynamic analysis** is running the program under control and watching: a debugger that stops it where you ask, the real value in a register, the system call it actually made. You see the one path this input took, with the real data on it, including a string the program only built at run time.

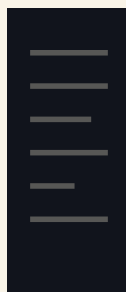
Each eye sees what the other cannot. Static shows you the whole map but only the shapes; dynamic shows you one real journey in full colour. The habit this track teaches, the same habit Volume II calls verify-it, is to use both and let one confirm the other. A branch you read in the decompiler is a hypothesis. The compared values you watch go past in the debugger are evidence. A claim that rests on only one of them is not yet a finding.

two eyes, one binary

Static

(read it still)

- every path at once
- paths that never ran
- shapes, not values



the binary

Dynamic

(watch it run)

- one path, this run
- the live values
- only what ran

Two eyes on one binary: static reads it still and sees every path, dynamic watches it run and sees the real values.

What the compiler leaves you

A compiled program arrives in a container format: **ELF** on Linux, **PE** on Windows, Mach-O on macOS. Inside are sections with jobs. In an ELF file `.text` holds the code, `.rodata` holds read-only data such as string constants, `.data` and `.bss` hold variables (PE names its read-only data `.rdata`). There may be a symbol table naming functions, or the binary may be **stripped**, with those names removed so that everything is an address until you give it a name. Underneath, the processor runs instructions over a set of registers and a stack. The tools in Chapter Two exist to translate that machine-level detail back up into something you can reason about, and Chapters Three and Four are how you read each eye.

a binary's layout



- format and map
- the code that runs
- constants, strings
- initialised variables
- zeroed, no file space

What a compiled ELF binary holds: the header and the `.text`, `.rodata`, `.data` and `.bss` sections.

The mindset

The work rewards curiosity kept on a leash. You form a small hypothesis, you test it against the binary or the running process, and you throw it out the moment the bytes disagree. You name things as you learn them, because a function called `FUN_00401a30` means nothing and a function you have renamed `check_licence` carries your understanding forward. You read toward the question and resist the pull to understand everything, because most of a program is plumbing that has nothing to do with what you were asked. And you stay honest about the difference between what you have shown and what you suspect, which is the whole of the house standard and the reason this track reads the way it does.



whose binary?

Before the tools come out, one question decides whether any of this is allowed: is this a binary you own, or one you are authorised in writing to analyse. Wyrms asks it first every time, because the most impressive teardown in the world is worthless, and worse than worthless, if you had no right to open the file.

When a tester reaches for it

Most web testing needs no reverse engineering at all, because the logic sits on a server and you study it by behaviour, which is Volume II's whole method. You reach for RE when the logic is in a binary you hold. A desktop or thick-client application that does its checks locally. A mobile app, where the client is a package you can pull apart (the Mobile and APIs track goes deep on that, with its own tooling). A piece of firmware pulled off a device, which Chapter Five opens. A proprietary protocol or file format with no spec. A sample you suspect is malicious and have to triage. A crackme you set yourself to build the eye. In each the shape is the same: no source, a question, and a file that holds the answer if you can read it.



SOL SAYS

The beginner's instinct is to load the binary and start reading from the top, as if a program were a novel. It is not, and that instinct will cost you days. The senior move is to decide the one question first, then go looking for the single function that answers it, usually by anchoring on a string or an imported call and following it inward. The skill in this track is not reading assembly fast. It is choosing what not to read.

■ EXERCISE

Pick a binary that is unambiguously yours: a small C program you compile yourself, or a system tool like `/bin/true`. Run `file` on it and read what it tells you: the format, the architecture, whether it is stripped, whether it is dynamically linked. Then run `strings` on it and skim the output for anything human: messages, paths, format strings. You are not attacking anything and you are not yet reading code. You are learning to get a first read on a file in thirty seconds, which is the move you will make on every target before you decide what to do next.

■ VERIFY IT

A string you spotted with `strings` is a lead, not a fact about behaviour. The binary may carry that string and never use it, or use it on a path your input never takes. Before you build anything on "the program contains the text X", confirm the program actually reaches and uses it: find where the code refers to it (Chapter Three), or watch it appear when you run the program (Chapter Four). Static presence is not proof of use, and in this track proof of use is the bar.

■ THE LINE

The law in this track is not re-taught here. It is Volume 0, in full, and it governs every page that follows. The working rule is short: you reverse only a binary you own, or one you are licensed or authorised in writing to analyse, and many software licences themselves forbid reverse engineering, which is a term you check before you start. Anything you do not fully trust runs only inside an isolated, disposable virtual machine with no network, and only when you are authorised to handle that sample. A malicious binary never runs on your workstation or on anything connected to it. Chapters Four and Five carry the detail. The permission comes first, here as everywhere in the Library.

■ ON YOUR FEET

A client hands you the installer for their desktop licensing application and asks, "could someone bypass our licence check?" You have the binary and a verbal yes. Before you open a disassembler, what do you establish, and what is the first question you ask the client?



WHAT WYRM KEEPS

Reverse engineering recovers understanding from a compiled binary when there is no source. Static reads it cold; dynamic watches it run. You answer one question, not the whole program, and only on a file you are allowed to open.

- Compilation drops names and comments but not behaviour: the machine code is exactly what runs, so the binary is the ground truth when the documentation is not.
- Static analysis shows every possible path but only its shape; dynamic analysis shows one real path with real values. Use both, and let one confirm the other before it is a finding.
- A binary is a container (ELF, PE, Mach-O) of sections (in ELF, `.text` code, `.rodata` constants, `.data` / `.bss` variables), with symbols either present or stripped.
- Reach for RE when the logic lives in a binary you hold: a thick client, a mobile app, firmware, an unknown protocol, a sample to triage, a crackme. Ownership or written authorisation comes before the first tool.

CHAPTER TWO

The Tools

The kit, what each piece is actually for, and how to pick. None of them is magic. They translate machine code and organise it so a human can reason; the understanding stays yours.

The tools fall into the two camps from Chapter One. Static tools read the binary without running it. Dynamic tools watch it run. A small set covers almost everything a tester meets, and the senior habit is to know two or three of them well rather than to install twenty. This chapter names the ones worth learning and says plainly what each does, so that later chapters can use them without stopping to explain.

The triage trio: file, strings, objdump

Three tools ship with almost any Linux box and answer the first thirty seconds of every job. `file` identifies what you are holding by its magic bytes, using the `libmagic` database: the format, the architecture, whether it is stripped, whether it is dynamically or statically linked. `strings` prints the runs of printable characters in the file, down to a minimum length (four by default, raise it with `-n 8`); a surprising amount of a program's intent leaks through its messages, paths, URLs and format strings. `objdump`, from GNU binutils, is a straight disassembler: `objdump -d` dumps the code, `-M intel` switches it to Intel syntax, and it can dump a named section too. Its sibling `nm` lists the symbol table when one survives. These are not beginner tools you grow out of. They are the first read, every time.

```
# first read on a file you own
file ./target
strings -n 8 ./target | less
objdump -d -M intel ./target | less
```

The disassemblers and decompilers: Ghidra, IDA, radare2 and rizin

When the triage trio has pointed you somewhere, you move to a tool that understands the whole binary as a program: functions, cross-references, a graph of the control flow, and often a decompiler that reconstructs a C-like view.

Ghidra is the software reverse engineering suite the United States National Security Agency released as open source in 2019. It is written mostly in Java, it is free, it reads the usual formats across many processor architectures, and it includes a decompiler that turns machine code into readable pseudo-C alongside the disassembly. For a tester with no budget it is the obvious first serious tool, and its decompiler is good enough that much of the industry now reaches for it.

IDA, the Interactive Disassembler from Hex-Rays, is the long-standing commercial option and the house tool in many shops. Its decompiler, the Hex-Rays Decompiler, comes inside the paid IDA plans rather than as a separate purchase, and IDA Free, licensed for non-commercial use only, includes a