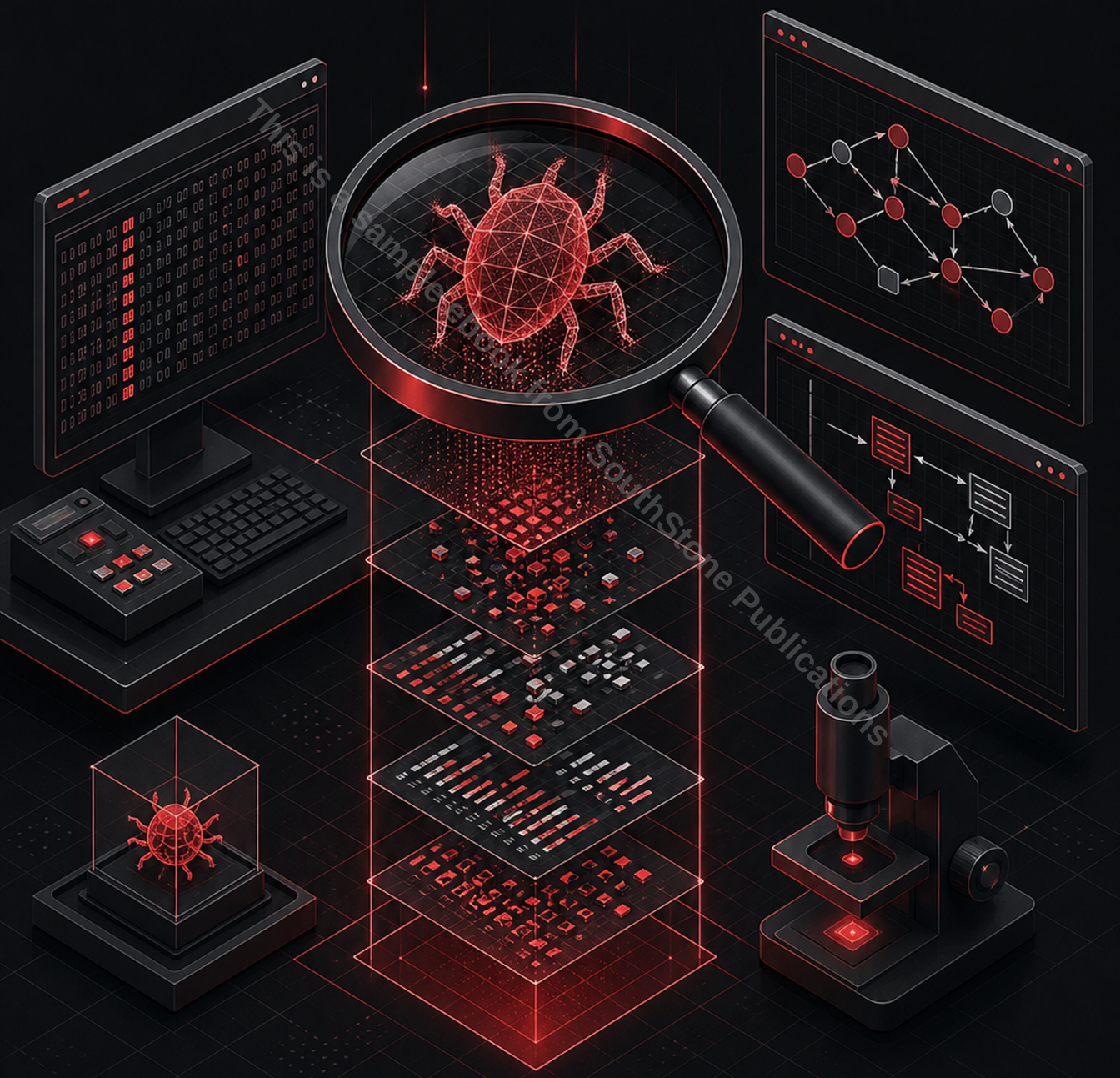
Malware Analysis

FROM SAMPLE TO SIGNATURE



SOUTHSTONE PUBLICATIONS

Malware Analysis

From Sample to Signature Across Static, Dynamic, and Code Analysis

by SouthStone Publications

Published 2026 · © 2026 SouthStone Publications

Table of Contents

About this book: This book teaches SOC analysts, incident responders, threat intelligence analysts, and aspiring reverse engineers how to take a piece of malware from a suspicious file to a defensible detection. The premise is practical: every sample is a small story written in code, and your job is to read it carefully enough to write the next chapter — a YARA rule, a Sigma detection, or a behavior-based hunt. Part I covers the foundations: malware categories, analysis tiers, lab setup, and the rules of safe handling. Part II walks the analyst from the outside in, starting with static triage, moving to detonation and behavior, and finishing with code and memory analysis. Part III turns the artifacts into operational value through classification, YARA, attribution, and reporting. By the end you will have built a real lab, detonated and unpacked a sample end to end, written production-quality YARA, and produced an analyst-grade report.

Introduction: Why Malware Analysis Still Matters

Every detection engineer, incident responder, and threat intelligence analyst eventually has to answer the same question: what does this sample *do*? Vendor write-ups answer that question for the malware of last week. They do not help you when the alert queue contains a file with a brand-new hash, an attachment from an unknown sender, or a suspicious binary recovered from a host you do not own. At that point you become the analyst. The choices you make in the next hour — what to run, what to look for, what to write down — determine whether the next incident is detected in minutes or in months.

The good news is that the fundamental craft of malware analysis has never been more accessible. Free distributions like FLARE VM and REMnux bundle the disassemblers, debuggers, and deobfuscators that used to cost tens of thousands of dollars. Open-source frameworks such as Volatility 3, oletools, capa, and FLOSS take care of the most tedious parts of reverse engineering. Cloud sandboxes can give you a first cut of behavior in minutes. And the body of community knowledge — reference YARA rules, capa capabilities, Malpedia entries, MITRE ATT&CK mappings — keeps growing.

The bad news is that malware authors have noticed. Modern samples check for virtualization, debuggers, sleep duration, and a thousand other host attributes before detonating. They run entirely in memory, hide behind multi-stage loaders, and split their behavior across signed and unsigned components. They pull configurations from cloud object storage, sign their payloads with leaked certificates, and impersonate entire supply chains. A first scan will not always tell you what the binary does. Sometimes the second, third, or fourth scan will not either.

This book is designed for the gap between "I have a suspicious file" and "I have a defensible detection." It does not assume you have ever loaded a binary in a debugger, read x86 assembly, or unpacked a packer. It does assume you are comfortable on a Linux or Windows command line, you have written a few scripts, and you can read code without panicking. If you have those foundations, the rest is technique, vocabulary, and practice.

We start by mapping the territory: what malware is, how it is categorized, and which questions you should be trying to answer when you analyze it. We then build a lab that is safe enough for real samples, then move from the outside of a binary in

(static analysis, dynamic analysis, code analysis, memory analysis), and finally turn the artifacts into operational detections and reports. Each chapter ends with a hands-on exercise so the techniques stick.

A note on ethics. The methods in this book are dual-use: anyone who can analyze malware can also write it more effectively. We discuss techniques, including anti-analysis tricks, so that defenders can recognize and defeat them. The companion rules are the same as every reverse engineering community has held for decades: do not target systems you do not own, do not redistribute live samples outside of controlled channels, and when in doubt, do the analysis in a sandbox. The art is open; the targets are not.

By the end of this book, you should be able to walk up to an unknown binary, decide within minutes which tier of analysis it needs, run a safe and defensible detonation, extract a usable configuration, identify the family or the developer toolkit behind it, and ship a YARA rule and a behavioral detection your whole team can use. That is what malware analysis is for. Let us begin.

Chapter 1: Malware Analysis Fundamentals

Before you load a single tool, you need a mental model of the problem. What kinds of code are we dealing with, what questions are we trying to answer, and what does the rest of the team's workflow actually need from us? This chapter sets the vocabulary for the rest of the book.

The Malware Zoo: Categories by Behavior

Malware is not one thing. It is a sprawling family of related behaviors, and the categories matter because they predict what the sample is going to try to do — which in turn determines what you should be looking for.

Trojans and loaders. A *trojan* is a program that pretends to be something legitimate but carries a hidden payload. The classic example is a PDF viewer that quietly drops a second executable in the background. A *loader* is a specialization: a small piece of code whose entire job is to download or unpack the next stage and hand off execution. Modern loaders such as IcedID, Bumblebee, and Smoke Loader are responsible for most initial-access intrusions, because once they have run, the attacker can decide which more expensive tool to deploy next.

Droppers and downloaders. *Droppers* unpack their embedded payload directly out of themselves. *Downloaders* fetch the payload over the network. The distinction matters because droppers often have everything they need inside one file (good for static analysis), while downloaders often look innocuous on disk and only become interesting when detonated.

RATs and stealers. A *RAT* (remote access trojan) gives the attacker an interactive shell on the host: a desktop, a file browser, a process manager, sometimes a microphone and a webcam. AsyncRAT, Quasar, and njRAT are typical examples. A *stealer* is narrower: it grabs credentials, cookies, crypto wallets, and system information, exfiltrates them, and exits. LummaC2, RedLine, and Raccoon are the stealer families you will see most often in 2025 and 2026.

Ransomware. When the goal is encryption of the host's data for extortion, you are looking at ransomware. Modern families are usually operated by an affiliate model (the developers rent out the encryptor to other criminals) and use techniques from both RATs and stealers — double-extortion (encrypt files *and* leak

them) is standard. LockBit, BlackCat, and Akira are names you should be familiar with even at the triage tier.

Wipers. Wipers look superficially like ransomware — they often overwrite files — but their goal is destruction, not extortion. There is no decryption tool because the attacker has no financial motive; the goal is to make systems unusable. WhisperGate, HermeticWiper, and AcidPour are the families that have dominated the news in recent years.

Rootkits. A *rootkit* is malware that hides itself and other components from the operating system. User-mode rootkits hook APIs. Kernel-mode rootkits load a driver and operate below the kernel. Bootkits infect the boot process and persist across reinstalls. Analyzing rootkits well requires kernel debugging or memory forensics — both of which we will touch later in the book.

Fileless and "living-off-the-land." Strictly speaking, these are not categories so much as techniques. Many recent intrusions never drop a traditional executable to disk. They run PowerShell, signed Microsoft binaries, or JS/VBA directly in memory. The "category" matters for analysis: when the sample is fileless you may have nothing to statically analyze and must rely on memory and logs.

When in doubt, treat the first observable stage — the attachment, the script, the dropped binary — as a *loader*, and ask what it loads. This mindset is more useful than trying to slot families into a single label.

The Four Tiers of Analysis

Malware analysis is not one skill. It is four, applied in increasing order of depth and cost.

Tier 1: Automated sandboxing. A sandbox runs the sample in an instrumented environment and reports on its behavior. Cuckoo, CAPE, and cloud sandboxes such as Hybrid Analysis, Joe Sandbox, and Any.Run are typical examples. Tier 1 is fast — usually minutes per sample — and a good sandbox will give you a process tree, network traffic, dropped files, registry modifications, and a rough MITRE ATT&CK mapping. Tier 1 has two main weaknesses: anti-analysis checks (the sample just exits) and shared infrastructure (some families fingerprint public sandboxes and behave differently inside them).

Tier 2: Static analysis. Static analysis means inspecting the file without running it. Hashing, magic-byte identification, embedded strings, header parsing, and import-table analysis are all static. Tier 2 is reproducible — running the same tool twice gives you the same answer — and it is the first thing every report should

include. Tools: `file`, `exiftool`, PE-bear, CFF Explorer, `pescanner`, FLOSS, capa, and the relevant section-viewer for whatever container the file is in.

Tier 3: Dynamic analysis. Detonate the sample under controlled conditions and watch what it does. Process Monitor, Process Hacker, Wireshark, Regshot, and specialized tools like ProcDOT or fakenet-NG give you the picture. Dynamic analysis is the bridge between static and code analysis: it tells you what the sample is *for* so you know what to look for when you open it in a disassembler.

Tier 4: Code analysis. Open the sample in a disassembler (Ghidra or IDA) or run it in a debugger (x64dbg, WinDbg). Read the assembly. Understand the algorithm. This is the most expensive tier — it can take hours per function — but it is the only way to recover an encrypted configuration, identify a custom C2 protocol, or attribute a sample to a specific developer. We will spend a full chapter on it, but the rest of the book is built so you reach for code analysis only when the cheaper tiers have run out of answers.

In practice, you rarely treat the tiers as a strict ladder. A good analyst mixes tiers: triage with the sandbox, confirm with static analysis, detonate under controlled conditions to fill in any gaps, and then open the disassembler only for the parts the other tiers cannot explain.

Analysis Goals: What the Rest of the Team Needs

It is easy to lose hours chasing an interesting implementation detail. The job of a malware analyst is to produce specific, actionable artifacts, and each goal drives a different section of your eventual report.

Triage. Is this sample dangerous? Is it part of an active campaign? Triage is mostly a Tier 1 question and it is what most automated alerts eventually need.

IOC extraction. Indicators of Compromise (hashes, domains, IPs, file paths, registry keys, mutexes, user-agent strings) feed blocklists and detection rules. Every modern report has an IOC table.

Detection content. YARA, Sigma, KQL, Snort, and Suricata rules give the rest of the SOC something to alert on. Goal: a behavior that you can detect, not just a hash that will expire tomorrow.

Attribution and family classification. "This is LummaC2 variant X" or "this looks like work from group Y" carries more strategic weight than "this is a stealer." Goal: the name of a family or a toolset, with confidence.

Behavioral understanding. Sometimes the report is not about a sample at all but about a TTPs document — what does the actor do, what does their toolkit look like, how do they pivot. Goal: a paragraph or two of operational behavior.

When you start a new sample, ask which of these it serves. It will save you from analysis that is technically interesting but operationally useless.

Safe Handling Rules

Malware is, by definition, hostile. Treat every sample as a weapon until proven otherwise.

Isolate. Analysis happens in a VM with no shared folders, no host clipboard, no ability to talk to corporate networks. Network traffic should be tightly limited; ideally, host-only networking with a fake internet appliance such as INetSim or FakeNet-NG catches outbound calls.

Defang. Never email raw samples, never paste live URLs into chats. Hashes can travel freely. Domains get a `.defanged` suffix or brackets: `example[.]com`. URLs use `<...>` or `hxxp:`. This culture exists precisely because a single pasted live URL can detonate a phish.

Hash first. Compute the MD5, SHA-1, and SHA-256 of every sample before you do anything else. A hash lookup against VirusTotal, MalwareBazaar, Malpedia, or your in-house threat intel platform will often give you a confident family label in seconds.

Snapshot before detonation. Take a clean VM snapshot before every run. Reverting to that snapshot should take seconds, not minutes. If your analysis environment is harder to reset than rebuilding it, you will avoid re-runs and your conclusions will be biased.

Containment vs analysis. A common question is whether to detonate a suspicious file you just received. The answer is governed by policy, not by analysis. If your team has a sandbox and a sample, detonate. If the sample is on a real host, contain first and analyze after. "Detonate the host" is rarely the right answer.

Operational hygiene. Sample repositories should be encrypted at rest, access-controlled, and version-controlled. Every analyst who touches a sample leaves a record. Treat sample storage as you would treat any other production secrets store.

We will put all of these rules into practice in the next chapter, when we build the lab you will be using for the rest of the book.

SOUTHSTONE PUBLICATIONS

Thank You for Sampling

This book is available in https://southstonepub.com