



GHOST
PROTOCOL

RC

THE FIELD LIBRARY

DEEP TRACK

THE WHITE-BOX EYE

Reading Code

Secure code review: the white-box eye,
taint, the classes in source, reviewing a diff.



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

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

THE GHOST PROTOCOL FIELD LIBRARY · DEEP TRACK

Reading Code

Secure code review: the white-box eye, taint, the classes in source, reviewing a diff.

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 YOU ARE HERE

Internal and Active Directory BUILT

The Web, Deeper BUILT

Cloud and Containers BUILT

Mobile and APIs BUILT

Reverse Engineering BUILT

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** The White-Box Eye 6
- 2** Following Untrusted Input 10
- 3** The Classes, in Source 14
- 4** Per-Language Gotchas 19
- 5** Reviewing a Diff 23
- Worked answers 29

The line that made it



You had found the hole from the outside, the way the earlier books taught: a request, a response, a finding you proved by hand. Sol pulled up something you had not seen on any engagement before. The source. The actual lines the developer wrote.

SOL "You found it black-box, by knocking. Now read the one line that let it in. When a client hands you the source, you stop guessing where the bug might be and start seeing exactly where it is, and every other place the same mistake was made."

BLINK "Is it not cheating, seeing the answers?"

SOL "It is the same eye, turned inward. Outside, you ask what the server does. Inside, you read why. The best testers can do both, and switch between them without thinking."

Wyrn, who has read more code than anyone in the house, said nothing, which is Wyrn for yes. This book is reading code the way an attacker does: following the untrusted thing to the dangerous place.

CHAPTER ONE

The White-Box Eye

Reading the source is the shortest path to the bug, because the flaw you had to guess at from outside is sitting there in plain text. This chapter is how to read it the way the attacker will.

Every volume so far has you outside the application, sending requests and reading what comes back. This one puts you inside, with the source open in front of you. The questions do not change. You are still asking where untrusted input goes and what dangerous thing it reaches. What changes is that you no longer have to infer the answer from behaviour. You can read it.

Black box, white box, the same question

Black-box testing, which is most of Volumes I to III, works from the outside. You probe the running application, watch the responses, and reason backward to code you cannot see. It is honest about what an attacker with no inside knowledge can reach, and it misses whatever the surface never exposed. White-box review hands you the source instead. The vulnerability that took twenty probes to confirm from outside is often one search and three lines of reading from inside.

The mental model does not change, and you already have it from Volume II: a vulnerability is untrusted data reaching a dangerous operation, across a trust boundary, without being made safe first. In the running application you watched that happen and inferred the code. In the source you read the code and predict the behaviour. Same shape, read instead of guessed. Everything in this volume is that one habit practised with the files open: find the source, find the sink, and look hard at the boundary the author drew between them.

What reading the source adds

Black-box tells you what an attacker can do today. White-box tells you why, and tells you about the paths no attacker has found yet. Four things come for free once the code is in front of you:

- You see the sink directly. You do not have to wonder whether the search box builds a SQL string. You open the function, and the string is either glued together or bound to a placeholder.
- You see where the author decided to trust. The most useful line in many reviews is a comment like `// already checked on the client` sitting above an unescaped use. That is the author telling you, in writing, exactly which boundary they assumed was already behind them.
- You see paths the interface never offers: a route no link points at that still answers, a debug parameter, an older version of an endpoint still wired up, a second handler that does the same work without the guard the first one has.
- You find it before it ships. Black-box finds the bug in production, which is the most expensive place it will ever live. White-box finds it in the pull request, which is the cheapest.

Where the bugs actually hide

Not in the clever code. In the code that looked correct. The line that ships broken is the one that passed review because it read fine on a tired Thursday, and the reviewer who goes hunting for something exotic walks straight past it. A short list of the usual hiding places will catch most of what you will ever find:

- The boundary the author assumed was already crossed. "The frontend validates it" (the frontend is the attacker's tool, Volume I). "That value is internal" (it arrived on a webhook). "It is already escaped" (it was escaped for HTML and is now going into a query).
- A check that was removed or moved. A change that deletes an ownership clause from a query, or relocates a validation into client-side JavaScript where the attacker edits it freely.
- The swallowed error: a `try/catch` that catches an authorisation failure and carries on as if nothing happened.
- The place the types stopped helping. A fresh `any`, an `@ts-ignore`, a `# type: ignore`, sitting exactly where untrusted data crosses into trusted code. The annotation did not close the hole. It hid it.



SOL SAYS

A new reviewer wants to find something impressive, so they read looking for cleverness and feel let down by ordinary code. Turn that around. Assume the author was competent and tired, because they usually were, and competent tired people miss the plain thing, not the subtle one. The reviewer who treats the most boring-looking line as the likeliest bug finds more than the one chasing a trophy. The mistake you are looking for is almost always a small, reasonable-looking decision made at the wrong boundary.

The diff-First mindset

You cannot re-read a large codebase on every change, and you should not try. The new risk arrived on the lines that changed, so most of your reading life is the diff: a pull request, a commit, a patch. You read the change, read enough around it to know what it touches, and ask the source-to-sink question of the new lines. Chapter Five is that whole pass, in order, with the triage that turns it into a review someone will act on. But a diff only means something against a codebase you understand at least a little, which is the other half of the work and the harder half the first time.

Reading a codebase you have never seen

Do not read it front to back. You will drown in files that do not matter and lose the one that does. Read it along the path the data takes, and let four questions pull the map out of it:

1. Where does input enter? Find the routes, the controllers, the request handlers. That list is your sources.
2. Where is identity decided? Find the one place a request is tied to a user: the session middleware, the auth filter, the token check. Then, for every handler, ask whether it actually passes through that place or quietly skips it.

3. Where does data get stored or executed? Find the database layer, the shell calls, the template renderer, the file writes. That list is your sinks.
4. Where do secrets and configuration live? The environment file, the config module, the deploy settings.

With those four lists and a search for the dangerous functions each language favours (Chapter Four names them), you can map every source and every sink in an afternoon without reading most of the files. Let the framework tell you its own conventions, because they rarely move: Express routes are wherever `app.get` and `router.post` are called, a Laravel app keeps them under `routes/`, a Django project in `urls.py`. Learn the five files that matter in a stack and you can orient in any project built on it.



not yours.

Reading source is access. A repository you were not authorised to read is a system you were not authorised to touch, whether it reached you as a clone or a zip. Before the first file, the permission exists, the same rule as the first packet in Volume 0.

■ THE LINE

Source code is a client's property, and often the most sensitive thing they own. You read it under the same written authorisation that covers the rest of the engagement, and what you read stays inside the engagement exactly like anything else you see (Volume 0). A repository handed to you "just to glance at", with nothing signed, is the Chapter One problem from Volume 0 wearing different clothes. Three yeses, or you do not clone it.

■ EXERCISE

Take a repository you wrote, or an intentionally vulnerable application whose source ships with it (DVWA for PHP, OWASP Juice Shop for JavaScript). Do not read a single file end to end. Instead, answer the four orientation questions: list the entry points, find the one place identity is decided, list the sinks (queries, shell calls, template renders, file writes), and find where secrets live. Write the four lists out. You now have the map most reviewers never draw, and the rest of this volume is what you do with it.

■ ON YOUR FEET

A friend sends you a zip of their startup's backend, privately, and asks you to "have a look and tell me if it's hackable, just between us". You have the skill now, and an evening would be enough to read it. There is nothing in writing, and your friend does not own the company. What do you do, and what do you offer them instead?



WHAT WYRM KEEPS

White-box is the same source-to-sink question as black-box, read instead of guessed. The bug hides in the code that looked correct, at the boundary the author assumed was already behind them.

- Reading the source adds the sink in plain view, the author's trust assumptions (often written in a comment), the paths no link reaches, and the catch before it ships.
- Read a new codebase along the data's path, not top to bottom: entry points, where identity is decided, the sinks, where secrets live.
- Most reviewing is diff-first: the new risk is on the changed lines (Chapter Five).
- Source is access. You read only what you are authorised to read, and the silence rule from Volume 0 covers everything in it.

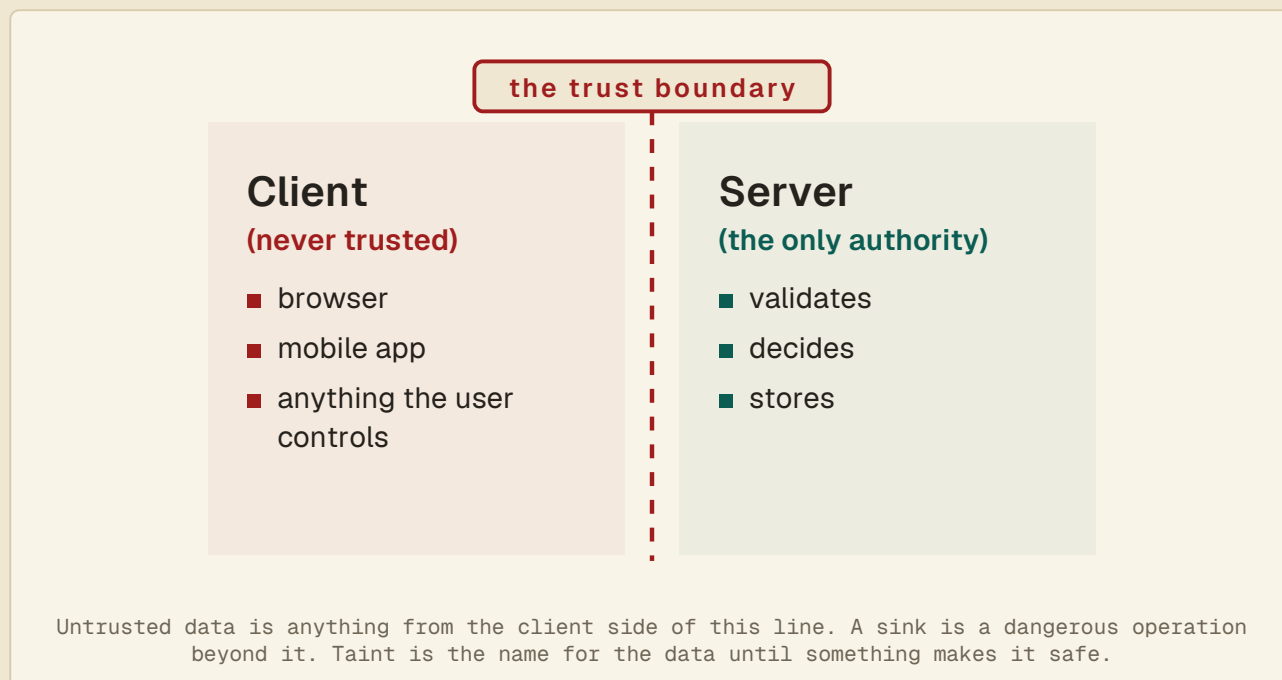
CHAPTER TWO

Following Untrusted Input

A vulnerability is a value that travels from somewhere an attacker controls to somewhere dangerous without being made safe on the way. Learn to follow that value through source and you can find any class, including ones you have never seen named.

The last chapter gave you the question. This one is the method, and it is almost mechanical once you trust it. Pick a value an attacker controls, and follow it. At every step ask what happens to it, and keep going until it reaches something dangerous or is genuinely made safe. That walk has a name, taint tracking, and it is the single most useful thing you can do with source in front of you.

Sources, sinks, and the line between



A **source** is attacker-controlled input. The obvious ones are the request body, the query string, the headers, the cookies, and the URL path parameters. The ones people forget are the uploaded file's contents and its name, the webhook payload, the response from a third-party API your server called, and the one that catches almost everyone: a row read back from your own database that was itself once user input. If a value can be influenced by someone on the other side of the wire, it is a source, no matter how many hops it took to arrive.

A **sink** is an operation where data can do harm: a SQL or shell command, an `eval` or a template render, a `fetch` of a URL, a filesystem path, HTML written to a page, a deserialiser, an HTTP redirect, or a merge of attacker-controlled keys into an object. The **trust boundary** is the line where your code starts treating a value as safe. Draw those three on any handler and the bug, if there is one, is almost always a boundary the author thought was already crossed.