



GHOST
PROTOCOL

GE

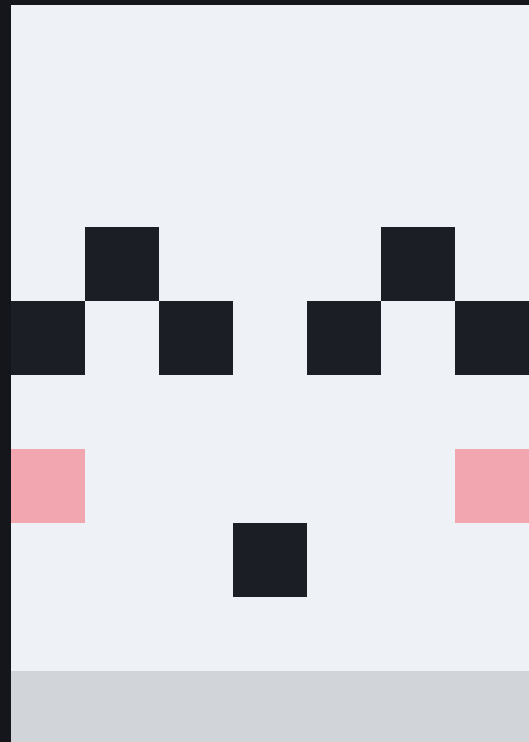
THE FIELD LIBRARY

COMPANION

KNOW THE CREATURE TO CATCH IT

The Gestiary

A field guide to the vulnerability classes as creatures: their tells, their habits, and how to cage them.



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

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

THE GHOST PROTOCOL FIELD LIBRARY · COMPANION

The Bestiary

A field guide to the vulnerability classes as creatures: their tells, their habits, and how to cage them.

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 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

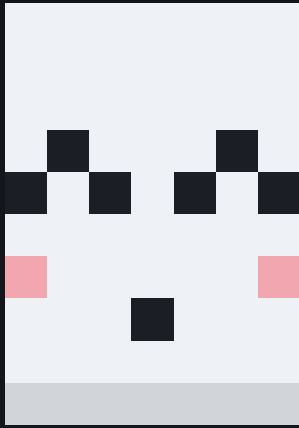
The Bestiary YOU ARE HERE

The Glossary and the Ledger BUILT

Contents

- Opening 5
- 1** The Web Beasts 6
- 2** The Auth Beasts 11
- 3** The Infra Beasts 15
- 4** The AI Beasts 19
- 5** The Human Beasts 23
- Worked answers 28

Give the bug a Face



Blink had taken one look at the chapter titles, all beasts and creatures, and was already delighted past the point of usefulness.

BLINK "Monsters! We get to fight monsters! Which one is the biggest?"

SOL "Settle. We give each bug a face because a face is easy to remember, and a creature you can picture is a creature you will spot in the wild. But the fun is a handle on a real and dangerous thing. The beast is cute on the page. The bug it stands for drains an account or leaks a database. Learn the creature, and never mistake the cartoon for a reason to relax."

Wurm's eye was level, unamused and entirely serious.

WURM "a face. real teeth."

So the book is the field guide: the web beasts, the auth beasts, the infra beasts, the AI beasts, and the human beasts, each with its tell, its habit, and exactly how you catch and cage it.

CHAPTER ONE

The Web Beasts

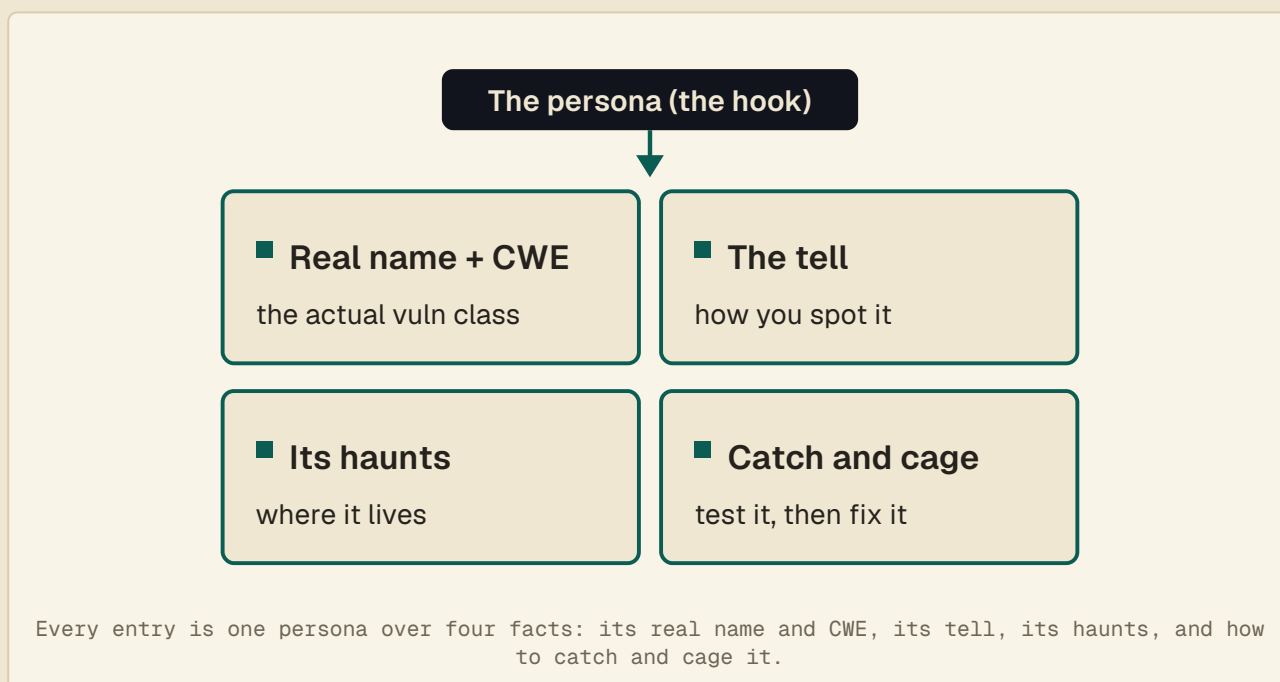
The four oldest beasts of the web, by persona and by proper name. Learn each one's tell and you recognise it in a codebase you have never read, then prove it with the smallest safe demonstration and know the fix that cages it for good.

A vulnerability class is a habit, not a trick. Learn the habit and you know the beast on sight, in a language you barely read and an application you have never opened. This chapter is a field guide to the web's four oldest beasts: what each one really is, the tell that gives it away, where it lives, and how you catch it and cage it without harming anyone. The names are ours, a mnemonic and no more. The mechanisms underneath are the ones the Web volume teaches in full, so when an entry here runs short for the work in front of you, that is the shelf to reach for.

the five families

- **Web beasts** the request and the page
- **Auth beasts** sessions, tokens, logins
- **Infra beasts** config, services, patches
- **AI beasts** the model's context window
- **Human beasts** the person at the keyboard

The five families in this volume, and the part of a system where each one lives.



The Mimic

Feed the Mimic a sentence and it cannot tell your words from its own. You hand it data, a name to look up, and somewhere in the sentence you slip an instruction. The Mimic reads the whole line in one breath and acts on all of it, because to the interpreter underneath there was only ever one line.

Real name. Injection. The worked example is SQL injection, **CWE-89** (improper neutralisation of special elements in an SQL command), and the family runs on to OS command injection (**CWE-78**) and LDAP, XPath and XML injection. In the OWASP Top Ten it is **A05:2025**. The single fault under all of them is input that becomes part of a command instead of staying data.

The tell. The application builds a command by gluing your input into a string. Send one character that matters in that syntax, a single quote into a SQL field, and watch the response change: an error, a different row count, a `500`. That change is the parser telling you your input reached the code and not just the data. A quote echoed back as inert text on the page is telling you nothing reached the query.

Its haunts. Anywhere input meets an interpreter: a login form, a search box, a filter, an id in the query string, an export that shells out, an XML upload. The numeric id glued into a lookup is the classic den.

Catch and cage. Catch it with a boolean pair, not a stack trace: one injected condition that is always true, one always false, everything else identical, and a response that differs only because the logic flipped. That proves the mechanism and touches no data you were not already shown. Cage it by never building the command from strings: parameterised queries (prepared statements) send the query text and the data to the database apart, so the data can never become syntax. Where a value cannot be bound, a table name for instance, allowlist it. Filtering bad characters is a cage with a gap in it.

The Puppeteer

The Puppeteer never touches the server. It reaches past it, to the next person who loads the page, and moves their browser as though the strings were its own. Your script runs in their session, on the application's origin, doing what they could do and nothing marks it as yours.

Real name. Cross-site scripting, **CWE-79** (improper neutralisation of input during web page generation). OWASP folds it under injection, **A05:2025**. It takes three shapes by where the input travels: reflected (echoed straight back in the response to the request that carried it), stored (saved and served later to other viewers), and DOM-based (client-side JavaScript reads an attacker-controlled source such as the URL fragment and writes it into the page).

The tell. Your input comes back inside the page as markup rather than as text. The giveaway is not that your input appears, it is that the browser parses it: angle brackets that survive, an attribute you can close, a script context you can reach. For the DOM shape the reading is of the page's own JavaScript, tracing a source like `location.hash` to a sink that writes HTML.

Its haunts. Search results that echo the query, comment fields and profile names shown to others (stored), administrative views that render user content, and any client-side code that writes a URL parameter or fragment into the page.

Catch and cage. Catch it with a benign marker that could only run if script executed, a call that pops the page's own origin into a dialog or writes one fixed, unmistakable string into the page. Execution is the finding, not reflection, and you never wire up a payload that steals a real user's cookie to make the point. Cage it by treating output as text and letting the framework encode for its context, because HTML body, attribute, JavaScript and URL are four different encodings; where HTML from users genuinely must render, sanitise it against a strict allowlist with a vetted library; set a Content-Security-Policy as the second line of defence, not the first.

The Skeleton Key

The Skeleton Key fits every lock because none of them was ever locked against it. You are let in at the door, you are logged in, and after that the building assumes anyone inside belongs in every room. Change the number stamped on the key and the next door opens.

Real name. Insecure direct object reference, the object-level face of broken access control. **CWE-639** (authorisation bypass through a user-controlled key), a child of **CWE-863** (incorrect authorisation) in the **CWE-284** (improper access control) pillar; OWASP **A01:2025**. In APIs the same fault is named broken object level authorisation, BOLA (OWASP API1:2023), and the Wider Field volume treats it there.

The tell. A request carries an identifier of something owned, an invoice number, a user id, a document reference, and the server returns or changes that object on the strength of your being logged in, without checking it is yours. Sequential identifiers are the open invitation: change `5001` to `5002` and see whose invoice you are handed.

Its haunts. Any endpoint that takes an id for an owned or scoped object. Predictable identifiers are the den; a long random token is a wall.

Catch and cage. Catch it cross-account, never self-account: hold two accounts you own, note an object's id as account A, then request that id from account B's session. B receiving A's data is the finding, and no third party's data is touched. Fetching your own object proves nothing, of course you can read your own. Cage it by scoping every query to the authenticated principal in the data access itself (fetch only where the record's owner is the current session user), checking at both the object level and the function level, server-side on every request, and answering `404` rather than confirming an id exists.

The Errand-Runner

The Errand-Runner is trusted inside the walls, so it goes where you cannot. You write an address on a slip, a URL for it to fetch, and it walks the request into rooms that open only for insiders: an internal service, a cloud metadata endpoint, something behind the firewall that trusts its own kind. The service on the other end answers because the knock came from a machine it trusts.

Real name. Server-side request forgery, **CWE-918**. In the 2025 OWASP Top Ten it is folded into **A01:2025** Broken Access Control, no longer a category of its own as it was in 2021 (A10). The server makes a request to a destination you chose, and that is the whole of the mechanism.

The tell. A feature where the server fetches a URL you influence: an avatar-by-URL, a webhook, a link preview, an import-from-URL, a PDF or screenshot renderer. If you set the address, the server is a candidate proxy into its own network.

Its haunts. Those URL-fetching features, and in cloud estates the link-local metadata address `169.254.169.254`, which can hand out credentials. That reach into credentials is why SSRF rises to critical in the cloud.

Catch and cage. Catch it by pointing the fetch at a listener you control and watching the server-originated request arrive in your own log: proof of the mechanism without reaching anything sensitive. Prove the capability and stop, because the metadata endpoint and the internal range are past your authorisation. Cage it with an allowlist of permitted hosts and schemes; resolve the hostname and block private, loopback and link-local ranges after resolution; and pin the connection to that one resolved address, so a name cannot pass the check and then re-resolve inward (the DNS-rebinding trick an allowlist alone does not stop). Refuse non-HTTP schemes such as `file:`, and re-validate redirects so a permitted host cannot bounce the request inward.



prove it. take nothing.

Every beast in this chapter is proven with the smallest safe demonstration: a true/false pair, a benign marker that runs, one object a second account of yours reaches, a request arriving at your own listener. The moment you reach for the whole table, the real cookie, the metadata credentials, you have stopped proving and started taking.

■ VERIFY IT

A beast is confirmed by what it does, not by what it accepts. Input echoed in the raw HTML is not the Puppeteer until a marker actually runs in the rendered page. A URL the form swallowed is not the Errand-Runner until a request reaches your listener. An id that returns your own record is not the Skeleton Key until a second account returns the first one's data. Capture the pair for each, the request and the response, both halves. A single screenshot of normal behaviour is a lead, never a finding.

■ ON YOUR FEET

A teammate shows you a search box that reflects your query on the results page. They paste a classic script tag, see it sitting in the page source, and write it up as cross-site scripting. You load the same page and the browser shows the tag as plain text on screen: no dialog, nothing runs. Based only on this chapter, is this a finding yet, what single thing would settle it, and what do you tell your teammate before the report goes out?



WHAT WYRM KEEPS

Four web beasts, one habit each. Injection is data that became code. Cross-site scripting is your code in another browser. The skeleton key is the door nobody scoped to its owner. The errand-runner is the server fetching where you point. Read the habit and you need no catalogue of payloads.

- The Mimic (injection, CWE-89 and family, A05:2025): input that becomes a command. Prove it with a boolean true/false pair, not a stack trace; cage it with parameterised queries, never blacklists.
- The Puppeteer (XSS, CWE-79, A05:2025): your script in a victim's browser, in three shapes (reflected, stored, DOM). Prove it with a benign marker that runs; cage it by encoding per context and sanitising by allowlist, CSP second.
- The Skeleton Key (IDOR, CWE-639 in the CWE-284 access-control pillar, A01:2025; BOLA = API1:2023 in APIs): an owned object fetched with no ownership check. Prove it cross-account; cage it by scoping every query to the caller at object and function level, 404 not 403.
- The Errand-Runner (SSRF, CWE-918, folded into A01:2025): the server fetches an attacker-chosen URL. Prove it with a request to your own listener, then stop; cage it with an allowlist and connection-pinning against DNS rebinding.