



GHOST
PROTOCOL

WD

THE FIELD LIBRARY

DEEP TRACK

PAST THE BASICS

The Web, Deeper

Deserialization, template injection, request smuggling, OAuth and token attacks, and the modern client side.



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

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

THE GHOST PROTOCOL FIELD LIBRARY · DEEP TRACK

The Web, Deeper

Deserialization, template injection, request smuggling, OAuth and token attacks, and the modern client side.

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 YOU ARE HERE

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** Deserialization and Object Injection 6
- 2** Template Injection 12
- 3** Request Smuggling and Cache Poisoning 17
- 4** OAuth, OIDC, SAML and JWT 22
- 5** Prototype Pollution, GraphQL and WebSockets 27
- Worked answers 33

The basement



The scope was a web application, and the first pass had come back tidy: a few soft headers, a reflected parameter, nothing that would frighten anyone. Blink was pleased with it.

BLINK "Clean. We found the usual and wrote it up."

SOL "You found the first floor. Everything in Volume II lives there, and it is real work. But the serious money, and the serious damage, is in the basement: a cookie the server trusts enough to rebuild into an object, a template that runs what a name field feeds it, two servers that disagree about where one request ends. This book is the stairs down."

Worm opened the one green eye at the word trusts, because trust is the hinge every bug in this volume swings on.

WYRM "deeper."

That is the whole of it. The web you learned is the surface. Under it is the same web, carrying objects, templates, tokens and requests that each assume someone upstream already checked. This track is where you stop assuming.

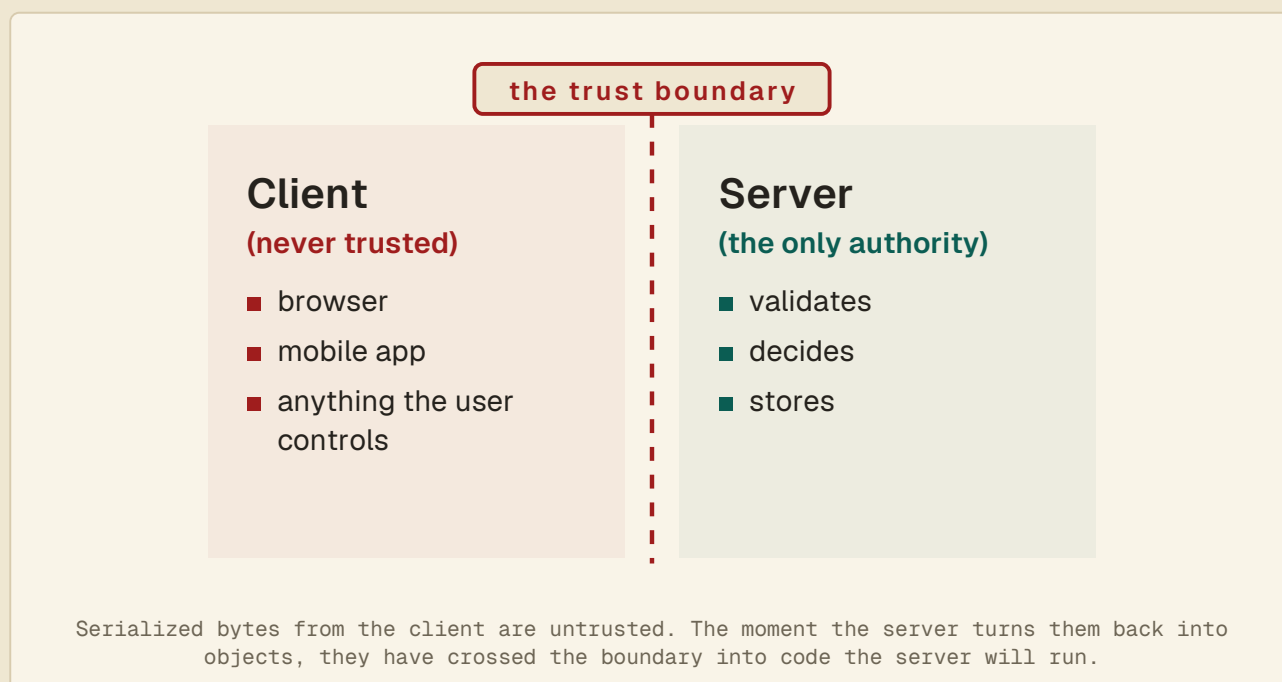
CHAPTER ONE

Deserialization and Object Injection

When an application rebuilds objects from bytes it was handed, the bytes decide what gets built. Learn to spot a serialized blob, prove it is being deserialized without running a command on someone else's server, and read why the fix is to not do it at all.

Volume II taught injection as input that becomes code. This is the same confusion one layer deeper: input that becomes an object. An application takes a stream of bytes from a cookie, a cache, or a message queue and reconstructs live program objects from it. If those bytes are yours, you are no longer supplying data to the program. You are telling it what to build and, on some runtimes, what to run while it builds. The whole class lives in that handover, and the safe way to prove it is quieter than it looks.

How it works: the bytes choose the type



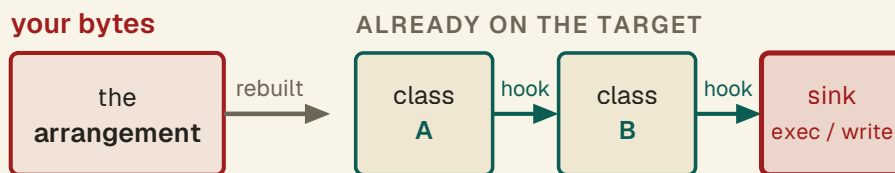
Serialization is how a program flattens an object (its class, its fields, their values) into bytes it can store or send. Deserialization is the reverse: bytes back into a live object. Frameworks lean on it constantly, for session state in a cookie, for a cached value, for a job on a queue, for an API body. The trouble starts because deserialization is not a passive copy. To rebuild an object the runtime calls code: constructors, and the language's lifecycle hooks (PHP's `__wakeup` and `__destruct`, Java's `readObject`, Python's `__reduce__`, .NET's deserialization callbacks). Those run on the attacker's structure before the application has looked at a single value.

That is the first thing to hold on to: an ordinary injection changes a value inside a query the developer wrote. A deserialization attack changes the type and shape of the objects the program instantiates,

and the program then runs its own normal code on an arrangement of objects its author never intended. You are not smuggling a value past a parser. You are handing the runtime a blueprint.

Gadget chains: you bring the arrangement, not the code

You rarely get to serialize a class that says "run this command". The classes that do something dangerous are already present, in the application and its libraries. A **gadget chain** is a sequence of those existing classes, wired together by their own lifecycle methods, so that rebuilding your object sets off a cascade that ends at a dangerous sink: a process execution, a file write, an outbound request. In Java this is what tools like `ysoserial` assemble from libraries such as Commons Collections. In PHP the same idea is a POP chain (property-oriented programming), generated by `phpggc`. You supply the arrangement; the target supplies the code. Which is why two servers running the same application can be exploitable or not depending only on which libraries sit on the classpath or the autoload path.



You supply the **arrangement**.

The target supplies the **code**.

Same app, different libraries on the path: exploitable or not.

A gadget chain wires classes already present on the target into a sequence that ends at a dangerous sink. You supply the arrangement; the code is the target's, which is why an identical app is exploitable on one host and not another.

How to Find it

Find the places where the application accepts bytes that it will later rebuild. The surface is wider than a form field: session cookies, hidden fields, view state, cache entries, API bodies, message queues, anything marked as a serialized object by its content type. You are looking for two things together: a serialized blob under your control, and a runtime that will deserialize it natively.

The fastest tell is the shape of the blob. Learn these on sight:

```
# Java, raw, hex 0xAC 0xED 0x00 0x05; base64 begins r00AB; gzip'd begins H4sIA
# content type is often application/x-java-serialized-object
r00ABXNyABFqYXZlLnV0aWwuSGFzaE1hcAUH2sHDFmDRAwACRgAK...

# PHP serialize(): O: is an object, a: an array, s: a string
O:4:"User":2:{s:4:"name";s:5:"blink";s:7:"isAdmin";b:0;}

# .NET ViewState in __VIEWSTATE, base64 often starting /wE
# Python pickle and Ruby Marshal (\x04\x08) are binary; decode and read the opcodes
```

When you can read the source (the white-box eye from the Reading Code track), the search is even more direct: find every call to the native deserializer (`unserialize`, `ObjectInputStream.readObject`, `pickle.loads`, `yaml.load`, `Marshal.load`, `BinaryFormatter.Deserialize`) and ask whether anything attacker-controlled can reach it. That single grep finds the class faster than any payload.

How to confirm it safely

This is where interns get into trouble, because the loud proof and the safe proof point in opposite directions. The loud proof is a full command-execution gadget: satisfying, and on a live multi-user system it is also you running arbitrary code on a box that is not yours, potentially destabilising it, for no evidence you could not get another way. You do not need it to prove the flaw.

On Java, the clean proof is the **URLDNS gadget**. It is built into `ysoserial`, needs no third-party library on the target, and does exactly one thing when deserialized: it performs a DNS lookup of a hostname you choose. Point it at a subdomain of an out-of-band listener you control (Burp Collaborator, interact-sh, or the house OAST collector), send the blob, and watch for the DNS hit. A lookup arriving at your listener is undeniable proof that your bytes were deserialized, and nothing ran, nothing broke, no command touched the host. That is the whole finding: the bytes were rebuilt into an object, and the object did something only your input could have caused.

The same principle carries to the other runtimes. On PHP, if you have source, demonstrate **object injection** by crafting an object whose property the application trusts (flip an `isAdmin` field, point a path somewhere benign) and showing the changed behaviour, no shell involved. Black-box, reach for a detection gadget that makes a harmless outbound callback to your listener rather than one that executes a command. On Python and Ruby, where `pickle.loads` and `Marshal.load` on untrusted bytes are game over by design, prove it with a benign out-of-band callback, never a destructive command. The rule is constant: prove the deserialization happens, not how much damage you can do once it does.

■ VERIFY IT

A recognisable blob is a lead, not a finding. A base64 string that decodes to a serialized object tells you the application serializes; it does not prove the application deserializes your copy without checking it. The confirmation is the callback: your unique hostname resolving at your listener, or your crafted property visibly changing the application's behaviour, reproduced by hand with the request that carried the blob. No callback, no confirmed deserialization vulnerability. A tampered blob that the server rejects with an integrity error is the good case, and worth recording as the control working.

Impact

Where a command-execution chain exists, insecure deserialization is **CRITICAL**: remote code execution on the server, which is as bad as it gets. But the class does not always reach a shell, and you rate the instance, not the headline. With the gadgets actually present, the realistic outcome might be object injection that bypasses an authentication check, an arbitrary file read or write, server-side request forgery, or a denial of service from a hostile object graph. Each of those is a real finding at its own severity. Argue from the gadget you can actually build on this target, the way Volume II taught you to rate the specific injection rather than the worst the class can theoretically do.

How to Fix it

The honest fix is the same shape as the injection fix: stop mixing the two things. Do not deserialize data you do not trust with a native deserializer at all. Move state to a data-only format (JSON, with a schema you validate against) so the bytes can only ever become data, never an object graph of the attacker's choosing. When a value must survive a round trip through the client, sign it with a keyed MAC and verify the signature before you deserialize, so a tampered blob is rejected before it is rebuilt.

Where a native deserializer genuinely cannot be removed, constrain what it is allowed to build. Java supports look-ahead deserialization and an `ObjectInputFilter` allowlist of permitted classes. PHP's `unserialize` takes an `allowed_classes => false` option that refuses to rebuild objects at all. Python's answer is blunt and correct: never `pickle` untrusted input, use `json`; and use `yaml.safe_load`, never `yaml.load`. On .NET, `BinaryFormatter` is unsafe for exactly this reason: since .NET 9 the runtime's own implementation always throws, but older .NET Framework applications still use it, so the fix there is a serializer that binds to known types. The defender's tell is on the changed line: a native deserializer pointed at request-derived bytes is the bug, an allowlisted or data-only path is the fix.

■ EXERCISE

On the PortSwigger Web Security Academy, work the insecure deserialization labs in order. Start with the PHP lab where you modify a serialized object in a cookie to change a trusted property, and write one sentence naming the field the application trusted and why serializing it client-side was the mistake. Then do a lab that uses a pre-built gadget chain (`phpggcg` for PHP, or the Java labs with `ysoserial`), and before you fire a command-execution gadget, generate the Java URLDNS gadget against a Collaborator or interactsh host and confirm the DNS hit first. Capture the blob you sent and the callback you received. You are proving the mechanism, in the order a professional proves it.

■ THE LINE

A DNS callback or a flipped property on your own lab is harmless. A command-execution gadget is a working exploit, and running one on a client's server is running arbitrary code on a system you were trusted to test, not own. On an engagement you confirm with the quietest proof that settles it, and you escalate to command execution only where the Rules of Engagement explicitly authorise it, with a benign marker, and never on shared infrastructure you could knock over. The habit on the lab, where no one is harmed, is the habit that keeps you clean on a client, where someone would be. The law is in Volume O.

**dns first.**

The shell is louder, not stronger. A lookup that only your hostname could have caused proves the bytes were deserialized as completely as a command does, and it leaves the box standing. If you are reaching for the command before the callback, you have skipped the proof and gone straight to the damage.

**SOL SAYS**

Notice that the dangerous code was never yours. It was already in the application, in a library nobody thought about, waiting to be wired up in an order its authors never pictured. That is the real lesson of this class and it outlives any one gadget: the risk is not a payload you carry in, it is the arrangement of trusted parts you can summon from outside. Learn to see the blob and the deserializer, and you will find this on targets whose exact gadget you have never seen before.

■ ON YOUR FEET

On a client's live, shared production server you find a Java serialized object in a session cookie, and `ysoserial` has a Commons Collections chain that would almost certainly land remote code execution. The Rules of Engagement authorise testing the application and say nothing either way about running commands on the host. What do you send to confirm the flaw, what do you deliberately not send, and how do you word the impact of a critical you proved by a DNS lookup rather than a shell?