The Oracle Problem — How HTTP Compliance Leaks Information

March 24, 2026

Your API returns `403 Forbidden` when a user requests a document they don't own. The response is semantically correct per RFC 9110 (https://www.rfc-editor.org/rfc/rfc9110). It's also a vulnerability.

By distinguishing "you can't access this" from "this doesn't exist," you've confirmed the resource's existence to an attacker. That confirmation—a single bit of leaked state—is an oracle.

Not every oracle is a vulnerability — a public-facing catalog leaking product existence is a non-issue, while a patient records API doing the same is a compliance disaster. Like most vulnerabilities — and really, most of life — context matters.

This series examines six classes of HTTP oracles — patterns where standards-compliant server behavior creates deterministic information channels that attackers exploit to infer protected state. More importantly, it examines the deliberate anti-patterns that close those channels: intentional RFC violations, the kind of things your professor docked points for back in the day.

Part 1 establishes the framework and tackles the most intuitive oracle class: the **Existence Oracle**.

What Is an HTTP Oracle?

An oracle, in the security sense, is any system that answers questions it shouldn't. This system doesn't hand over your secrets voluntarily, but instead by letting you, the attacker, infer them from its behavior.

Think of it like a locked filing cabinet. You don't have the key, so you can't read the documents inside. But if the cabinet rattles when you pull on a drawer that has files in it and stays silent when you pull on an empty one, you've just learned which drawers contain documents without ever opening them. The cabinet answered a question you never should have been able to ask.

That's an oracle. Not a break-in. Not a bypass. Just observation of a system doing exactly what it was designed to do…and also leaking information as a side effect.

Now consider what an oracle is *not*: it's not an exploit, and it's not unauthorized access. The system isn't malfunctioning. It's behaving correctly, by specification. The problem is that correct behavior, viewed from the outside, reveals something about internal state that was supposed to stay internal.

> *Definition: An HTTP oracle exists whenever a server produces a deterministic, observable difference in behavior based on state the client is not authorized to know about.*

What makes HTTP oracles so cool is that they don't require any sophistication, just observation. The server is already giving you the answers. You just have to know which questions to ask.

That "difference" can manifest across multiple dimensions:

- **Response status codes:** `403` versus `404`
- **Response body content:** "Invalid username" versus "Invalid password"
- **Response timing:** 50ms versus 200ms
- **Response headers:** presence or absence of rate-limit or CSRF headers
- **Error detail granularity:** "expired" versus "invalid signature"

The attacker doesn't need to break encryption or bypass authentication. They just need to ask the right question and observe the answer.

A Taxonomy of HTTP Oracles

This series covers six distinct oracle classes, grouped by their primary leakage channel:

The model is simple: *any deterministic difference in externally observable server behavior that lets an unauthenticated or under-privileged client infer protected state is an oracle*. The defense, in every case, involves making the server's external behavior identical regardless of internal state—even when that means violating the semantic contracts defined by HTTP RFCs.

The Core Tension: Compliance vs. Security

RFC 9110 — *HTTP Semantics* [Fielding et al., 2022] (https://www.rfc-editor.org/rfc/rfc9110) — defines status codes as carrying specific, distinct meanings. The specification is built on an assumption that accuracy is helpful:

- **`401 Unauthorized`:** "The request has not been applied because it lacks valid authentication credentials for the target resource" (§15.5.2)
- **`403 Forbidden`:** "The server understood the request but refuses to fulfill it" (§15.5.4)
- **`404 Not Found`:** "The origin server did not find a current representation for the target resource" (§15.5.5)[1]

Each code communicates something different. That's the intent. The problem is that **information flows in both directions**. When your server accurately reports *why* a request failed, it's answering questions the client may have had no right to ask in the first place.

This tension, between RFC compliance and information security, is the central theme of this series. And as you'll learn, even the RFC authors knew it.

The Existence Oracle: When 403 Meets 404

The existence oracle is the simplest and most common HTTP oracle. The classic version exploits the semantic gap between two status codes, although many more exist[2].

- `403 Forbidden`: *This resource exists, but you can't have it*
- `404 Not Found`: *This resource does not exist…or the server doesn't want to give it to you*

See the issue here?

An attacker who receives a `403` when requesting `/api/v1/documents/foobar` now knows three things:

1. The document **exists**
2. The document ID format is **valid**
3. Their current credentials are **insufficient**; but someone's aren't

That's three inferences from a single response code. Against a UUID-based resource scheme, this transforms a brute-force search from 2^122 attempts (guessing a valid UUID in the full space)[3] to a targeted privilege escalation: the attacker now has a confirmed-valid document ID and just needs to find or forge credentials to gain access.

Compounding with IDOR

This is why existence oracles compound with Insecure Direct Object Reference (IDOR) vulnerabilities. IDOR alone means the server doesn't enforce authorization properly. An existence oracle means the server actively tells you *which objects to target*. Combined, they're devastating: the attacker enumerates valid resource IDs via the oracle, then exploits the IDOR to access them.

How This Plays Out

Consider a document-sharing API with user-scoped access control:

```
# Vulnerable: existence oracle
@app.route("/api/v1/documents/<doc_id>")
def get_document(doc_id):
    document = db.find_document(doc_id)
    if document is None:
        return jsonify({"error": "Not found"}), 404  # doesn't exist
    if not current_user.can_access(document):
        return jsonify({"error": "Forbidden"}), 403  # exists, no access
    return jsonify(document.to_dict()), 200
```

An attacker scripts requests across a range of document IDs. Every `403` is a confirmed hit—a resource that exists and belongs to someone. The `404`'s are noise. Within minutes, they've enumerated the entire document space.

For time-ordered UUID variants such as v6 and v7, an attacker who knows an object's approximate creation time can restrict probing to the subset of UUID prefixes consistent with that time window, turning the oracle from a confirmation channel to a search-space reduction tool.

Github documents the same anti-enum pattern in its REST API. If you request a private resource without proper authentication, Github returns a `404` instead of a `403` specifically to avoid confirming the existence of private repositories. This is a clean production example of intentional semantic flattening.

RFC 9110's Quiet Admission

Here's the thing: RFC 9110 (https://www.rfc-editor.org/rfc/rfc9110) knows about this.

§15.5.4—*403 Forbidden*—includes a note that is easy to miss but profound in its implications:

> *"An origin server that wishes to 'hide' the current existence of a forbidden target resource* ***MAY***

This single sentence, one glossed over by the 1% of folks who actually have read the RFC specs, is remarkable. The specification explicitly acknowledges the need to hide forbidden resource existence.

But note the language: **MAY.** For readers who don't live in RFCs, **MAY** denotes permission, not prescription. The spec allows oracle prevention; it does not require it. And that restraint is understandable, because not every endpoint needs masking. Even the word 'hide' appears in scare quotes—a small signal that this is treated as an exception, not the rule.

This creates a dangerous default: developers who follow the spec's primary guidance or who are just getting their footing in web development, can introduce information leaks on security-sensitive endpoints. Only developers who read the fine print, or who already think in terms of information

leakage, reach the safer behavior.

For API designers, the safer choice is framed as an exception; the ordinary, semantics-first implementation remains the default.

The Anti-Pattern: Uniform Denial

The defense is straightforward in principle: **return `404` for all unauth'd resource access**, regardless of whether the resource exists. From the client's perspective, the server's response must be identical whether:

- The resource doesn't exist
- The resource exists but the client lacks access
- The resource exists, the client has an account, but lacks the specific permission

This is the anti-pattern: an intentional, permanent deviation from the semantic model. You're lying to the client about *why* the request failed. That's the point.

```
@app.route("/api/v1/documents/<doc_id>")
def get_document(doc_id):
    document = db.find_document(doc_id)

    # Timing-safe: evaluate both conditions to prevent timing side-channel
attacks
    exists = document is not None
    allowed = current_user.can_access(document) if exists else False

    if not (exists and allowed):
        # Identical response regardless of reason
        return jsonify({"error": "Not found"}), 404

    return jsonify(document.to_dict()), 200
```

The key detail: **the response body must also be identical**. Returning a `404` with different JSON structures or error messages for the two cases just moves the oracle from the status code to the body.

Caching Implications

RFC 9111 permits caching of `404` responses by default (they're heuristically cacheable), while `403` responses are not cacheable unless explicitly marked as such. When you collapse `403` into `404`, you inherit `404`'s behavior by default. This can be desirable: it reinforces the deception at the infrastructure layer.

However, if your 404 responses include user-specific content — e.g. different error messages for authenticated vs. unauthenticated users — you'll need `Cache-Control: private` or `no-store` to prevent cross-user cache pollution. There is no *one-size-fits-all* solution; the right configuration requires case-by-case evaluation.

What You Lose

This anti-pattern has real costs:

1. **Debugging complexity**: When a legitimate user reports "I can't find my document, please help me!" your support team can't tell from the client's perspective whether the document simply doesn't exist or the user lacks access. Server-side logs must capture the real reason; the client only ever sees the sanitized version.
2. **User experience**: Legitimate users who lack access see "not found" instead of "access denied", breaking the traditional UX experience. The clean way to handle this without leaking state it to decouple internal domain errors from external API responses. Using a `Result` pattern (like `neverthrow` in TypeScript or Rust's native `Result`), your core logic returns an explicit `Err()` pattern. The API boundary layer then safely squashes both into an identical `404`, while preserving the more granular context for internal logs.
3. **API documentation drift**: Your API's documented error semantics now diverge from its actual behavior. This confuses API consumers and creates a maintenance burden.

Bridging the Gap: The Anti-Oracle Pipeline

You can mitigate these costs by enforcing a strict, code-genned boundary between internal domain errors and external API responses. The pattern will vary based on ecosystem and specific use-case, so the following is based on what I followed within one of my projects:

1. **Coarse-grained external contracts**: Use a standard like RFC 7808 (Problem Details) but intentionally restrict the error codes to a shared enum (e.g. `DOC_NOT_FOUND`, `FORBIDDEN`) defined in your spec. If five different domain failures all map to a `DOC_NOT_FOUND`, the oracle is closed.
2. **Generated client stubs**: Generate client validation schemas directly from your spec. If the backend sends an error code not in the allowed enum, the client simply rejects it. This catches doc drift at build time.
3. **Internal error hierarchies**: Keep rich, typed error discriminants (e.g. `kind: 'doc', step: 'read'`) inside the application boundary. The backend logs these detailed errors before returning the coarse HTTP response.
4. **The one-way valve**: On the client, wrap the coarse API error in a domain-specific result type. When it's time to render, pass that typed error through a one-way mapping function that returns *only* hardcoded, static strings to the UI. No dynamic interpolation, no raw error messages.

This tradeoff deliberately degrades the self-service debugging experience for third-party API consumers, but for security-critical endpoints or applications, that friction is the exact mechanism that prevents state leakage.

Key Takeaways

1. **An HTTP oracle is any deterministic difference in server behavior that leaks protected state.**
2. **RFC semantics for status codes are, by design, an information channel**; accuracy and security are in tension.
3. **RFC 9110 explicitly permits the anti-pattern**—returning `404` instead of `403` is spec-sanctioned divergence, not rogue engineering.
4. **Uniform denial requires normalizing the entire response**—status code, body, headers, and all other variables must be identical across denial paths.

What's Next

Part 2 "Silent Equals Secure (/series/http-oracles/structure)" tackles the **Authentication Oracle**: how login, registration, and recovery endpoints leak valid usernames through error message differences, and why RFC semantics themselves are an enumeration surface. We'll take a look at [MITRE, 2024] (https://cwe.mitre.org/data/definitions/204.html) (CWE-204) and [OWASP, 2025] (https://owasp.org/Top10/2025/A07_2025-Authentication_Failures/) (A07:2025), alongside the real UX cost of generic error messages.

References

Bleichenbacher, D. (1998). Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. *Advances in Cryptology — CRYPTO '98*. Springer. https://link.springer.com/chapter/10.1007/BFb0055716 (https://link.springer.com/chapter/10.1007/BFb0055716)

Fielding, R., Nottingham, M., & Reschke, J. (2022). HTTP Semantics. RFC 9110, IETF. https://www.rfc-editor.org/rfc/rfc9110 (https://www.rfc-editor.org/rfc/rfc9110)

Fielding, R., Nottingham, M., & Reschke, J. (2022). HTTP Caching. RFC 9111, IETF. https://www.rfc-editor.org/rfc/rfc9111 (https://www.rfc-editor.org/rfc/rfc9111)

GitHub Docs. (2026). Troubleshooting the REST API. *GitHub Docs*. https://docs.github.com/en/rest/using-the-rest-api/troubleshooting-the-rest-api (https://docs.github.com/en/rest/using-the-rest-api/troubleshooting-the-rest-api)

OWASP. (2025). A07:2025 — Authentication Failures. *OWASP Top 10:2025*. https://owasp.org/Top10/2025/A07_2025-Authentication_Failures/ (https://owasp.org/Top10/2025/A07_2025-Authentication_Failures/)

OWASP. (2025). Testing for Account Enumeration and Guessable User Account. *OWASP Web Security Testing Guide*. https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/03-Identity_Management_Testing/04-Testing_for_Account_Enumeration_and_Guessable_User_Account (https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/03-Identity_Management_Testing/04-Testing_for_Account_Enumeration_and_Guessable_User_Account)

MITRE. (2024). CWE-204: Observable Response Discrepancy. *Common Weakness Enumeration*. https://cwe.mitre.org/data/definitions/204.html (https://cwe.mitre.org/data/definitions/204.html)

Footnotes

1. The full RFC 9110 §15.5.5 definition continues: "…or is not willing to disclose that one exists." This means the spec's own definition of 404 already anticipates its use as a masking response, even before the explicit **MAY** permission in §15.5.4. ↩
2. The 403/404 mapping here is an analytic shorthand, not a literal claim about every denial path. Under RFC 9110, 404 can also mean the server is unwilling to disclose that a resource exists, and equivalent existence leaks can arise from other differentials such as 401 vs 404, redirect behavior, response bodies, or timing. ↩
3. The attack is even more efficient with sequential or predictable IDs. UUIDs raise the bar significantly, but the existence oracle still leaks confirmation when an attacker obtains candidate IDs through other channels — leaked URLs, email links, browser history, or log files. The oracle doesn't help *find* IDs in a vast space; it helps *confirm* IDs obtained elsewhere. ↩
