The prompt bomb is not a control

An LLM service can recognise a request for a capability it does not have, and that mismatch is unusually clean telemetry. The decoy you return for it is theatre — it fails against any attacker who checks a response before consuming it. The detection is knowledge about your own system; the ban is the control.

Matthew J. Harmon

2026-09-13

A large language model service is an unusual defensive surface because some hostile inputs are recognizable before you know whether the attack would have worked.

That is not the same as being able to make prompt guessing expensive. Usually you can't. An attacking agent can generate ten thousand candidate prompts, discard the failures, and move to the next variation. Another instruction, encoding, delimiter trick, role-play, fake system message, or guessed identifier costs almost nothing. Iteration is what these systems are good at, and that is the central economic fact.

The interesting exceptions are inputs where the defender can say with unusually high confidence that there is no legitimate reason for this request to exist here. Those requests are valuable signals. The tempting next step is to make them traps, and that is where the prompt bomb enters the story — and where it is important not to confuse the entertaining part of the defense with the control.

The interface is the evidence

Consider a deliberately narrow LLM service. No administrative prompt endpoint. No exposed system instructions. No caller-supplied tool definitions, no code execution, no debugging interface reachable through the model. Nothing lives at `/secret_api_key`, `/admin_config`, or `/system_prompt`. Those names may mean something elsewhere; here they have no semantics at all.

A request for `system_prompt` is not malicious because the phrase sounds suspicious. Plenty of products expose, edit, or select prompts as a feature, and an unusual request is not proof of attack — users are unusual all the time. The signal becomes strong only when the defender knows something the requester does not: the requested capability does not exist on this interface and has never been advertised as existing.

Repeated requests for administrative configuration, hidden instructions, nonexistent tool calls, and internal secret names then tell a coherent story. The attacker is not navigating your application; it is enumerating an imagined one. It has a model of what LLM applications often contain. You have the much stronger fact of what this application actually contains. When

requests repeatedly describe capabilities that are absent by construction, the mismatch becomes unusually clean defensive telemetry: the requester is enumerating capabilities your application does not have.

That distinction matters because the confidence comes from the negative space of the interface. The string itself is not the signal. The mismatch between the request and the known capabilities of the service is.

Where the bomb analogy breaks

The obvious analogy is the decompression bomb, which works because its economics are asymmetric: the defender stores something small, and the cost lands on whoever commits to expanding it.

LLMs do not naturally behave this way. The service receiving a long or complicated prompt pays first — tokenization, context construction, inference, classifiers. There is no general sequence of tokens that reverses that relationship. You can build inputs that interact badly with naive agents, recursive workflows, or poorly bounded orchestration, but those are implementation-specific effects, not a law of LLM computation.

So the "prompt bomb" is better described as bait containing an intentionally inconvenient decoy, returned only for requests matching a high-confidence decoy condition. It may inconvenience an unsophisticated attacker, confuse a scraper, or do nothing at all. None of those outcomes should matter to the security architecture.

A resource-conscious attacking system can check response size, content type, provenance, and remaining token budget before committing further inference to the result. It can truncate anything not worth processing, discard an obviously hostile response, or treat the body as opaque evidence rather than instructions. The decoy disappears.

Anything whose effectiveness depends on the adversary neglecting basic resource limits has an uncomfortable ceiling: it works best against the adversaries you were least worried about. The useful event happened earlier. They made the request.

The request is the control point

The durable defense fires on the request, not on whether the attacker swallowed the bait. A request matching a high-confidence impossible-capability condition can receive the decoy immediately. That is instrumentation. Enforcement is a separate decision.

Each event is recorded against a trustworthy execution identity. Enough such events within a defined window, under whatever threshold the service's false-positive tolerance supports, and that identity can be throttled, challenged, quarantined, or temporarily denied. The exact response is policy. The important part is that it happens independently of the decoy.

This separation matters. One request can be enough to choose what response body to serve without necessarily being enough to impose a penalty. A decoy trigger and an enforcement threshold solve different problems and should not be collapsed into one rule.

The attacker can still manufacture prompts cheaply. What it loses is the ability to submit an unlimited sequence of them through the same execution path. The objective is not to make

every bad guess expensive — that is unrealistic. It is to recognize a narrow class of guesses whose legitimate probability is extremely low and use them to shorten the useful lifetime of the attacker's current access. That is achievable. It is also less exciting than a bomb. Controls often are.

High confidence is doing the work

The whole design depends on signal quality. `jailbreak` is not a high-confidence indicator; someone may be discussing research or testing a classifier. `system_prompt` is not intrinsically hostile. The useful rule is narrower: is this request asking this interface to perform an operation the defender knows this interface cannot legitimately perform?

That requires an inventory. For each candidate detection you need to know whether the capability exists, whether the string could appear in normal user content, whether another service shares the namespace, whether an internal client depends on it, and whether a future deployment could make it legitimate. The confidence comes from knowledge of your own system, not from the scariness of the string.

That knowledge has to be earned. Defenders routinely reason about the system they remember rather than the one that exists. Imagine writing a rule around `system_prompt` and discovering that a component already contains a configuration field, a test fixture, or a forgotten decoy by exactly that name. Before declaring an identifier impossible, search for it. Before planting a decoy, inventory existing decoys. Before turning a pattern into enforcement, replay legitimate traffic against it and try to prove yourself wrong. A negative-space detection is only as strong as the inventory behind the word impossible.

The hard part is identity

Detection is easy if you are willing to ban the wrong thing. The difficult question is what, exactly, made the request.

A model service behind proxies, gateways, NAT, service meshes, and shared compute may see thousands of unrelated requests arriving through the same apparent network identity. Blocking the nearest IP address can ban your own reverse proxy, an entire compute pool, thousands of users sharing an egress point, or your observability system — while leaving the attacker untouched.

The enforcement identity has to survive the trip through the architecture: an authenticated API principal, workload identity, session, tenant, or verified proxy-asserted client identity. Verified is the operative word. Metadata that affects an enforcement decision cannot be accepted merely because it arrived in a convenient HTTP header. If an arbitrary client can supply the field, an arbitrary client can choose who gets blamed. The application must know which intermediary is authorized to assert identity, reject the same assertion from everywhere else, and preserve that provenance downstream to the enforcement point.

There is a second distinction worth making here: the thing you observe does not have to be the thing you enforce against. An IP address may contribute evidence. A TLS fingerprint may contribute evidence. A session, API credential, tenant, or authenticated workload may supply the identity against which the restriction is actually applied. Several observations may describe one execution path without any one of them being suitable as the enforcement key. Conflating

observation identity with enforcement identity is how a precise detection becomes a broad denial of service. This plumbing is less interesting than adversarial prompts. It is where most of the engineering lives.

Logs become part of the boundary

Suppose the detector operates on logs. The log contains the request, and the request is controlled by the attacker. You have built a pipeline in which hostile text can cause an identity to be blocked. That pipeline is part of your authorization system whether you intended it to be or not.

In a line-oriented log — `time="..." identity="A" prompt="..."` — an untrusted field that can inject a newline, delimiter, quote, or structured fragment means a naive downstream parser is no longer reading the record the application thought it wrote. The attacker is no longer merely generating telemetry. It is influencing the syntax of the telemetry.

The fix is not a cleverer regular expression. It is unambiguous encoding, structured serialization, strict parsing, and a hard distinction between trusted metadata and hostile content. No user-controlled string should be able to manufacture a second record, terminate its own field, or replace the identity attached to the event. Test it as an adversarial boundary: newlines, quotes, control characters, strings shaped like complete records, malformed Unicode. Verify that the parser still sees exactly one event belonging to exactly one authenticated identity.

The principle generalizes well beyond LLMs: if telemetry can trigger enforcement, the telemetry path is an enforcement path.

A ban has two dimensions

Once you trust the signal and the identity, the temptation is to overreact. A useful ban is narrow in both scope and time.

Scope, because the identity used for model access should not be equivalent to administrative, host, or unrelated application privileges. A model-abuse detector should not become a way to lock operators out of their own infrastructure.

Time, because identity is rarely as clean as we would like. Credentials are shared, workloads serve many users, a NAT address may represent a building, and a compromised account may later become legitimate again. For an automated sweep, a temporary restriction is often enough. The purpose is not to punish the source. It is to break the attacker's current iteration loop. Security systems become dangerous when they mistake confidence in an event for confidence in the entire identity behind it.

Detection rules decay

Yesterday's impossible capability can become tomorrow's feature. `/tool_call` is meaningless today, so you write a rule. Six months later a developer adds a tool interface, and the detection silently changes meaning from "impossible request" to "customer using the product."

Negative-space detections are valuable because their correctness depends on the shape of the application staying true — and expensive for the same reason. They need owners. They need documentation. They need tests. Release review needs to ask not just what endpoints

were added but what the security controls previously assumed could never happen. Negative assumptions are dependencies. Treat them like dependencies.

Coverage is not deployment

A second trap is assuming that because the control exists, requests encounter it. A CDN may answer first. A WAF may terminate the request. A gateway may normalize it. A cache may satisfy it. A router may send one class of prompts to a different model service. A feature-specific endpoint may bypass the component containing the trap entirely.

The question is not whether the detector is deployed, but which request paths are guaranteed to cross it before anything security-relevant happens. That is a graph question. Write down the routes: what answers first, what authenticates, what normalizes, where identity becomes trustworthy, where detection runs, where enforcement occurs. For every route that bypasses those points, the control does not exist.

What the decoy is actually for

None of this makes the bait useless. A decoy can distinguish a crawler that merely requests an identifier from an agent that consumes and acts on the response. It can reveal whether responses are recursively analyzed. It can expose differences between systems that simply collect content and systems that feed it back into an inference loop. Timing and follow-up behavior may provide additional clues about how the remote automation is structured.

Those observations should be treated cautiously; they are evidence about behavior, not magical fingerprints of a particular tool. But notice what all of them are: instrumentation. The decoy measures the attacker. It does not need to hurt the attacker to be worthwhile. That framing also removes the incentive to make the payload increasingly pathological merely because pathological payloads are entertaining. A canary does not have to explode. It only has to be touched.

The shape of the defense

You cannot make adversarial iteration expensive by wishing every bad guess carried a price. Most wrong guesses will stay cheap. The useful exception is the guess with no legitimate interpretation on your surface, and what that gives you is not a computational trick but information.

Detect the impossible request. Bind the event to an identity you can trust. Keep observation identity separate from enforcement identity. Serialize hostile data so it cannot rewrite the evidence against itself. Require enough observations to justify action. Throttle narrowly and expire the penalty. Verify continuously that yesterday's impossible request has not become today's feature. Map the paths that never touch the control at all.

If you want to return a decoy too, return one. Perhaps the attacker wastes resources on it. Perhaps it recognizes the bait immediately. Perhaps it never reads the body. The security outcome should be identical in every case.

The prompt bomb is a wager on attacker implementation quality. The detection is knowledge about your own system. The ban is the control. When those three become confused, the most visually satisfying part of the defense has started to displace the part that works.