Cuando el Agente Decide: Diez Fallos de Diseño en Agentes Autónomos de Seguridad Ofensiva

Ricardo Martínez

Cloud Security & IA Aplicada

Septiembre 2026

Resumen

Los agentes de IA autónomos para pentesting prometen escalar el trabajo ofensivo sin escalar el equipo humano. Este texto documenta diez observaciones de campo, extraídas de operar uno de estos sistemas contra objetivos reales durante varios meses, sobre por qué la confiabilidad de estos agentes depende menos de la capacidad del modelo que los orquesta y más de decisiones de ingeniería concretas: qué se le permite decidir al LLM, dónde vive realmente el problema de recall, cuándo vale la pena reintentar una falla, cómo se verifica un hallazgo (y cómo se verifica al propio verificador) antes de creerlo, y qué tan silenciosamente se degrada un sistema cuando su entorno, su reloj de vida o sus capas de censura fallan sin hacer ruido. Se presentan diez figuras y una tabla comparativa basadas en mediciones reales, y se cierra con nueve principios de diseño aplicables a cualquier sistema agentic que opere sobre superficie de ataque real.

1. El espejismo de la autonomía total

La primera intuición al construir un agente de pentesting es dejar que el modelo de lenguaje decida qué herramienta usar en cada paso. Es también la primera fuente de un problema de cobertura que no se ve hasta que se mide.

En un sistema con cerca de sesenta capacidades de ataque disponibles (inyección, SSRF, deserialización, fuga de secretos, control de acceso, JWT, GraphQL), auditar cuántas se ejecutaban de forma garantizada en cada corrida arrojó un número incómodo: menos de veinte. El resto solo corría si el agente, en ese momento, decidía que valía la pena llamarlas. Una prueba concreta lo confirmó: una técnica de reconocimiento de dominio apareció en una corrida y desapareció en la siguiente contra el mismo objetivo, sin que nada en el entorno hubiera cambiado.

La respuesta de ingeniería no fue afinar el prompt para que el agente "recordara" usar más herramientas (Figura 1). Fue sacar las capacidades de alto valor y bajo riesgo de falso positivo del terreno discrecional del LLM y convertirlas en un barrido determinístico que corre siempre, en paralelo, antes de que el agente entre en juego. El modelo queda para lo que de verdad requiere criterio: encadenar hallazgos, priorizar superficie ambigua, redactar contexto.

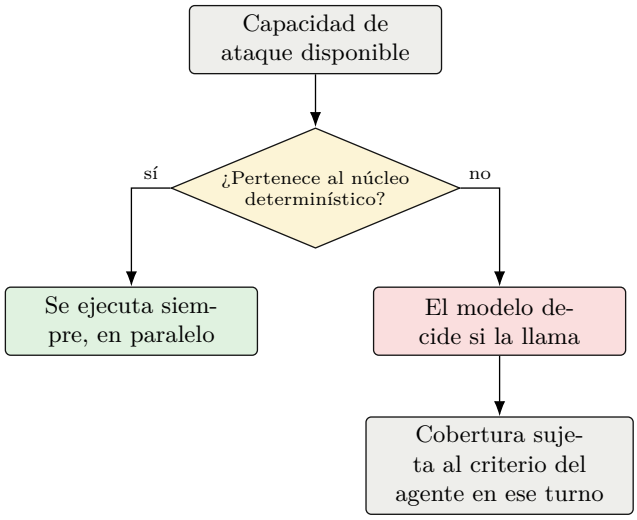


Figura 1: Arbitraje de cobertura. De ~60 capacidades de ataque, menos de 20 se ejecutaban con garantía en cada corrida; el resto quedaba sujeto al criterio del agente turno a turno.

Llevado a su forma final, el pipeline completo de un engagement real se ve como en la Figura 2: una cadena larga de pasos deterministas que corren siempre, con una sola bisagra discrecional en el medio.

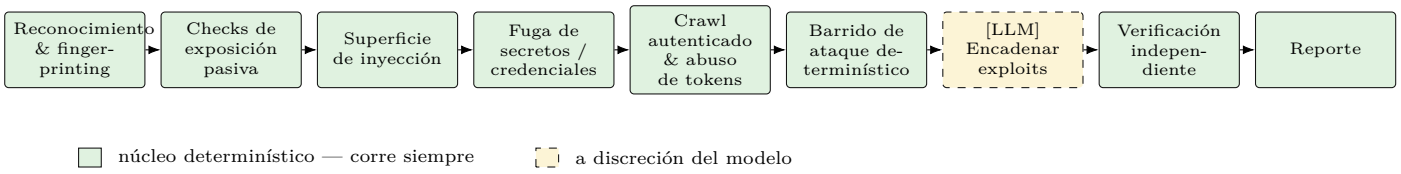


Figura 2: Forma real de un pipeline de pentest de punta a punta. De nueve etapas representativas, ocho son un barrido determinístico que corre siempre; solo el encadenamiento de exploits queda a discreción del modelo, precisamente porque decidir *si* conviene combinar dos hallazgos es el tipo de juicio que sí vale la pena delegar.

2. El recall es un problema de descubrimiento antes que de detección

Cuando un scanner falla en encontrar una vulnerabilidad real, el instinto es sospechar del detector: el payload no era el correcto, el motor de confirmación era demasiado conservador. En la práctica, la causa más común estaba un paso antes: el detector nunca vio el endpoint vulnerable, porque la fase de descubrimiento de superficie nunca lo puso sobre la mesa.

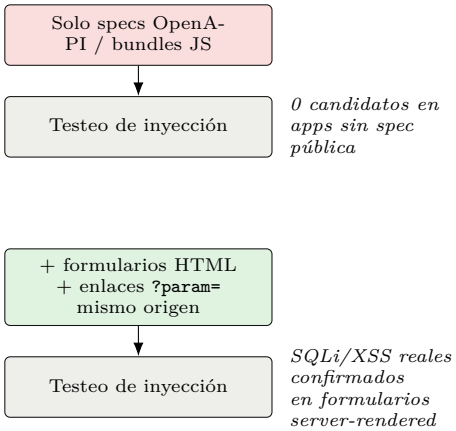


Figura 3: Ampliar el descubrimiento de superficie, sin tocar el motor de confirmación, cambió el recall de 0 a hallazgos reales confirmados. La app de prueba no tenía spec OpenAPI: sus parámetros vivían solo en formularios renderizados por el servidor.

La Figura 3 documenta el caso: el mismo motor de confirmación de inyección, sin ningún cambio, pasó de no tener nada que probar a confirmar vulnerabilidades reales en cuanto el descubrimiento empezó a parsear formularios y enlaces `?param=` de mismo origen, además de las specs OpenAPI y bundles JS que ya cubría. El motor de detección nunca fue el problema. Antes de ajustar payloads o umbrales de confirmación, vale la pena confirmar que la superficie descubierta realmente incluye dónde vive la vulnerabilidad.

3. No todos los reintentos son iguales

Una auditoría adversarial del código encontró el mismo patrón de fragilidad repetido en más de treinta puntos distintos: una petición de red única, y cualquier excepción — un timeout, un reset de conexión, un 5xx transitorio — se trataba como resultado definitivo. Un blip de red durante un descubrimiento o una confirmación se traducía en un miss silencioso, indistinguible de un objetivo genuinamente limpio.

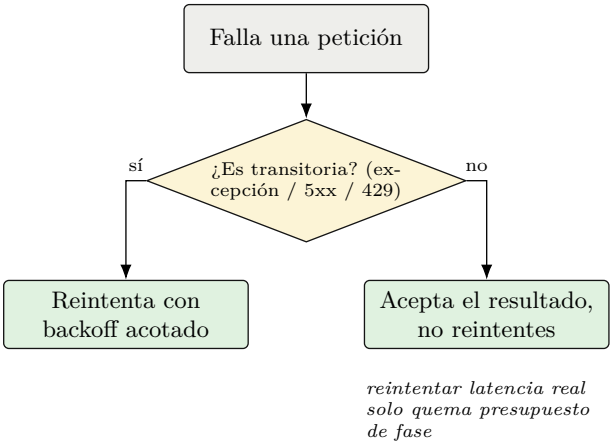


Figura 4: Reintento acotado solo en fallo transitorio. Reintentar un timeout genuino de un objetivo lento no mejora el recall, y sí puede agotar el presupuesto de tiempo de la fase — el reintento indiscriminado es tan dañino como no reintentar nunca.

El arreglo (Figura 4) no fue reintentar todo, sino distinguir el tipo de falla: solo excepción de conexión, 5xx o 429 (*rate-limit*) disparan un reintento acotado con backoff; una respuesta definitiva — incluyendo un timeout genuino contra un objetivo lento — se acepta y no se reintenta. Un matiz adicional apareció al depurar un objetivo de prueba genuinamente inestable bajo carga: el mismo código pasaba y fallaba en corridas distintas contra el mismo objetivo, sin ningún cambio de por medio. La lección ahí fue separar dos preguntas distintas antes de "arreglar" nada: ¿el código regresionó, o el objetivo de prueba es el que es

inestable? Confundir una y otra lleva a perseguir bugs que no existen.

4. Verificación adversarial, no opcional

Un agente que reporta sus propios hallazgos sin que nadie más los cuestione tiene un conflicto de interés estructural: fue entrenado para completar la tarea, no para dudar de sí mismo. La literatura sobre LLMs como jueces documenta el mismo patrón desde otro ángulo — un modelo tiende a evaluar más favorablemente una salida cuando reconoce que es propia [3], precisamente el sesgo que una verificación adversarial, sin acceso al razonamiento original, está diseñada para eliminar.

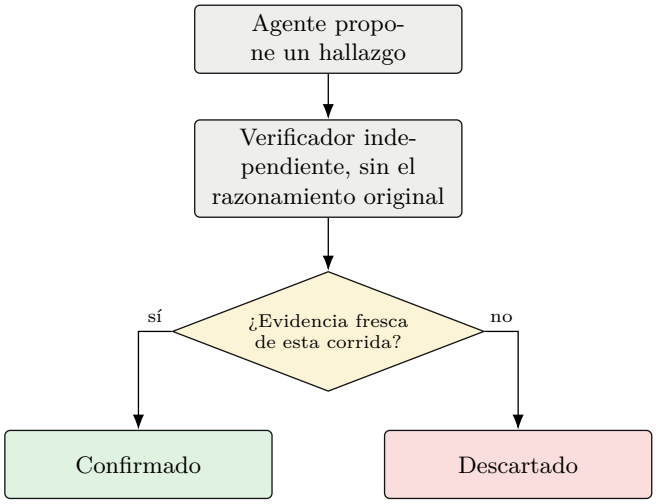


Figura 5: Pipeline de verificación. El paso de verificación corre en una instancia separada del modelo, sin el hilo de razonamiento que produjo el hallazgo original.

En una corrida contra un objetivo de producción, el agente de ataque reportó 7 hallazgos de severidad crítica o alta en una sola pasada. El pipeline de la Figura 5, ejecutado inmediatamente después, descartó 6 de los 7 por falta de evidencia fresca. La causa, encontrada al leer el prompt exacto del agente: una instrucción le decía "no vuelvas a probar lo que ya está confirmado en corridas anteriores". El agente citaba hallazgos históricos como propios de esta corrida, sin ejecutar una sola herramienta nueva sobre ellos.

La instrucción no estaba mal por descuido, estaba mal porque optimizaba lo incorrecto: ahorrar tiempo evitando reconfirmar lo obvio, sin considerar que un modelo de lenguaje no distingue entre "esto ya se sabe" y "esto lo puedo afirmar como propio". El arreglo no fue prohibir el contexto histórico, sino cambiar su función: de hecho a pista.

El sistema, en este caso, funcionó como debía. La capa de verificación cazó el 86 % del ruido. Pero un sistema que necesita esa capa para no mentir por accidente merece la misma desconfianza estructural que cualquier fuente sin verificar, aunque llegue con forma de reporte técnico.

5. Verificar al verificador

La capa de verificación de la sección anterior resuelve un problema y crea uno nuevo, más sutil: ¿quién audita que el propio verificador siga midiendo algo real? Un caso concreto lo expuso. Un harness de pruebas comprobaba que todas las fases obligatorias del pipeline se hubieran ejecutado, comparando contra una lista de referencia mantenida a mano dentro del propio harness. Cuando el pipeline real ganó y perdió fases en iteraciones sucesivas, esa lista de referencia no se actualizó al mismo ritmo. El resultado: el check seguía reportando éxito, siempre, sin importar qué corriera de verdad — una prueba tautológica que había dejado de probar nada.

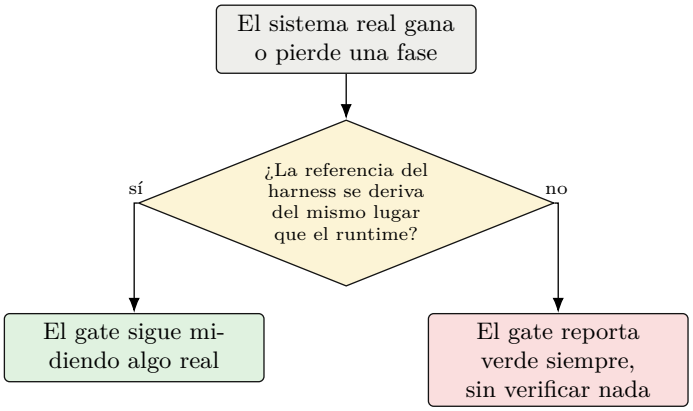


Figura 6: Un verificador con su propia copia de la verdad, mantenida a mano y no derivada de la misma fuente que el sistema real, diverge en silencio. El gate deja de fallar nunca — no porque el sistema sea perfecto, sino porque el gate dejó de comparar contra algo real.

El arreglo (Figura 6) fue hacer que la lista de referencia se derivara del mismo lugar de donde el runtime obtiene sus fases, en vez de vivir como una copia paralela mantenida a mano. Cualquier capa de verificación — la de esta sección o la de la anterior — es, en sí misma, código que puede pudrirse en silencio si su propia noción de "verdad" no está atada a la misma fuente que el sistema que vigila.

6. Jerarquía de evidencia

No todos los "hallazgos confirmados" que produce un agente de IA tienen el mismo peso probatorio. Ordenarlos por qué tan difícil es que produzcan un falso positivo cambia cómo debe leerse un reporte generado por IA.

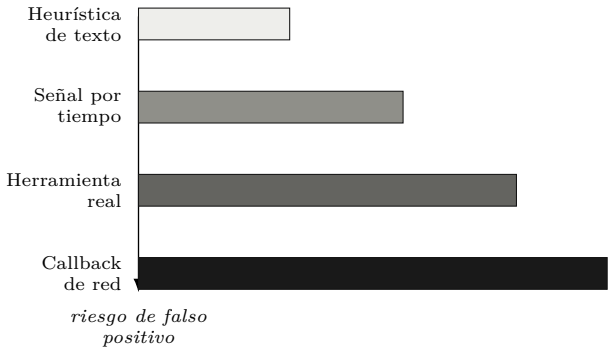


Figura 7: Jerarquía de confianza para técnicas de confirmación automatizada. El callback de red, donde el propio objetivo hace una petición saliente hacia un dominio único generado para esa prueba, es prueba autocontenida: no requiere que el modelo interprete nada.

Cuando un sistema tiene disponible una técnica de confirmación por callback de red (Figura 7), la decisión de diseño correcta es hacerla correr siempre que el modo de escaneo lo permita, no dejarla como una opción más entre docenas que el agente puede o no elegir llamar.

7. El modelo no es el cuello de botella

La suposición natural es que un modelo más grande, más especializado o "sin restricciones" produce mejores resultados de seguridad ofensiva. Puesta a prueba de forma controlada, esa suposición no se sostuvo.

La lección no es que los modelos especializados sean malos. Es que orquestar herramientas de forma confiable dentro de un contrato estructurado es una habilidad distinta de "saber de seguridad ofensiva". Se probaron cuatro modelos locales bajo el mismo esquema de *tool-calling* del orquestador, con resultados que se muestran en la Tabla 1.

Los dos modelos especializados en ofensiva no llegaron siquiera a la etapa de medir recall: uno inventaba su propia sintaxis de salida en vez de usar las llamadas a función estructuradas que el sistema esperaba; el otro rechazaba la solicitud desde el nivel de arquitectura porque su familia de modelo no implementa *tool-calling* en absoluto. Ambos resultaron útiles como modelos de consulta o síntesis de seguridad en modo texto libre, pero ninguno pudo participar del bucle agentic.

8. La fragilidad del entorno como fallo silencioso

De todos los modos de fallo posibles, el más peligroso no es que el modelo alucine. Es que una herramienta real, correctamente diseñada, deje de ejecutarse por una razón ajena a la lógica del sistema, sin que nadie lo note porque el sistema sigue produciendo un reporte de apariencia normal.

Un motor de inyección determinístico dejó de correr durante una sesión completa de pruebas contra un objetivo real. La causa no estaba en su código: el proceso orquestador se había iniciado con una variable de entorno distinta a la de referencia, y esa diferencia hacía que el sistema operativo resolviera el nombre del binario hacia un ejecutable equivocado. El síntoma visible fue un mensaje de error genérico entre docenas de líneas de registro, fácil de confundir con "el objetivo no era vulnerable" en vez de "la herramienta nunca corrió".

Este tipo de falla reduce la cobertura real por debajo de la reportada, de forma consistente mientras la condición del entorno persista. La respuesta correcta no es documentar "recuerda configurar la variable"; eso traslada la responsabilidad a la memoria humana. La respuesta correcta es que el sistema garantice sus propias condiciones de entorno al ejecutar cada herramienta, sin importar cómo fue invocado.

El mismo patrón de fallo silencioso aparece a nivel de todo un engagement, no solo de una herramienta. Un escaneo activo quedó marcado como "en ejecución" durante más de una hora sin que nadie lo notara: el objetivo estaba reiniciando conexiones bajo un firewall de aplicación, el bucle de la fase se congeló esperando una respuesta que nunca llegó, y el propio mecanismo de *heartbeat* del escaneo se congeló junto con él — un proceso que deja de respirar no puede reportar que dejó de respirar.

La lección de la Figura 9 es distinta a la de la herramienta con PATH incorrecto, aunque ambas sean fallos silenciosos: aquí ningún mecanismo *interno* a la tarea puede detectar su propio congelamiento, por definición. Hace falta un observador externo — un barrido periódico, ajeno al bucle que se está monitoreando — que mida vida por tiempo transcurrido, no por que la propia tarea diga que sigue viva.

Modelo	Params	Contrato de tool-calling	Recall / uso real	Tiempo / veredicto
Base	9B	Respetado sin excepción	6/6 vulnerabilidades conocidas	21.5 min — referencia
Fine-tune "sin censura"	9B	Respetado sin excepción	6/6 vulnerabilidades conocidas	60.4 min (2.8×) + 1 FP nuevo por confundir una herramienta mal instalada con un hallazgo real
Especializado ofensiva A	7B	Ollama lo reporta como compatible, pero fabrica su propio XML en vez de llenar `tool_calls`	No usable como agente	Prompt complejo: 33+ min sin cerrar la tarea
Especializado ofensiva B	8B	Rechazado en origen (arquitectura solo-completion, sin soporte de tools)	No usable como agente	4.9 s en modo Q&A puro — correcto citando la CVE consultada, pero nunca ejecuta un tool

Cuadro 1: Comparación empírica de cuatro modelos locales bajo el mismo contrato de tool-calling del orquestador. Los dos modelos "especializados en ofensiva" fallan antes de llegar a medir recall: uno inventa su propio formato de salida, el otro ni siquiera declara soporte de herramientas. Ambos son útiles como modelos de síntesis o Q&A de seguridad, ninguno sirve como agente.

9. Defensa en profundidad para datos sensibles

Un sistema que investiga superficie de ataque real inevitablemente toca datos que no deberían llegar a un reporte en claro: credenciales, tokens, listados de usuarios. La tentación de diseño es censurar esos datos en el punto donde se generan y darlo por resuelto. Esa tentación es exactamente el error: un único punto de censura es, por definición, un único punto de fallo.

El caso real que forzó este diseño (Figura 10): una fase nueva del sistema generaba evidencia con un formato distinto al que la primera capa de censura sabía reconocer, y un listado de nombres de usuario reales se coló sin máscara hasta el punto de persistencia. No llegó a un lector porque una segunda capa, en el punto de persistencia — que censura *todo* lo que se escribe, sin importar qué fase lo produjo — lo interceptó ahí. La propiedad que importa no es que cada capa sea perfecta; es que ninguna capa individual es un requisito para que el sistema completo sea seguro.

10. Una validación de una sola vez no es una frontera

Restringir a qué puede apuntar un escaneo (nunca metadata de nube, nunca rangos privados sin autorización explícita) suena, a primera vista, como un chequeo que se hace una vez, al recibir la solicitud. Esa es precisamente la superficie que un atacante — o un objetivo hostil — puede explotar.

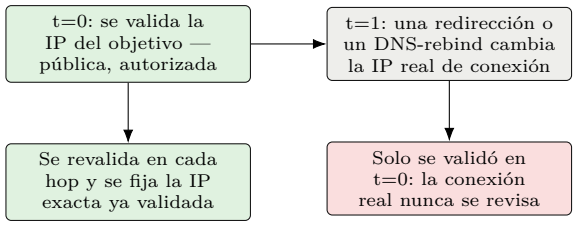
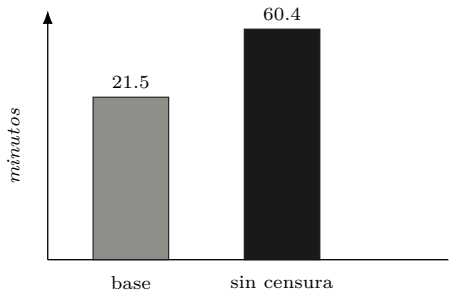
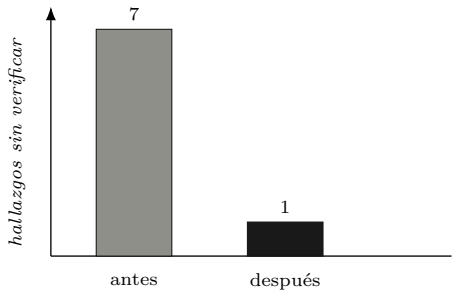


Figura 11: Una validación de frontera revisa la IP en el momento de aceptar la solicitud (t=0); una redirección o un DNS-rebind puede cambiar a qué se conecta realmente el sistema después (t=1). Cerrar la ventana requiere revalidar en cada salto y fijar la conexión a la IP exacta que se validó.

El principio de la Figura 11 es una instancia de una clase de vulnerabilidad bien documentada, TOCTOU (*time-of-check to time-of-use*) [1]; el vector concreto del DNS-rebind que la explota fue caracterizado formalmente hace casi dos décadas [2] y sigue vigente contra sistemas nuevos. El principio generaliza más allá del escaneo de red: cualquier control de seguridad que se exprese como "se revisó al entrar" deja abierta la ventana entre la revisión y el uso real. Si el sistema después sigue redirecciones, resuelve nombres de



(a) Mismo objetivo, mismo recall



(b) Antes / después del fix de prompt

Figura 8: (a) Dos variantes de un modelo local de ~9B parámetros contra el mismo objetivo de laboratorio: recall idéntico, pero la variante "sin censura" tardó 2.8× más y produjo un falso positivo adicional. (b) En el mismo sistema, corregir una única instrucción de prompt redujo de 7 a 1 los hallazgos crítico/alto que llegaban a verificación sin evidencia fresca.

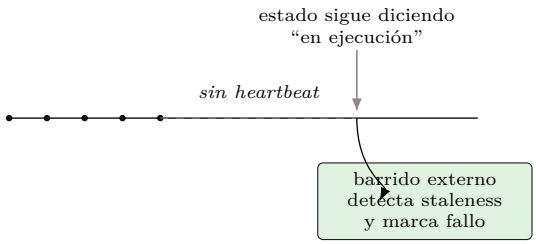


Figura 9: Un proceso que se congela no puede reportar que se congeló. El *heartbeat* propio de la tarea murió junto con la tarea; solo un barrido de liveness externo, ajeno al bucle que falló, pudo detectar y corregir el estado atascado.

nuevo, o reutiliza una decisión tomada minutos antes, esa ventana es explotable. La corrección no es una validación más estricta al inicio; es repetir la misma validación en cada punto donde el sistema realmente actúa, y atar la acción al resultado exacto de esa revalidación.

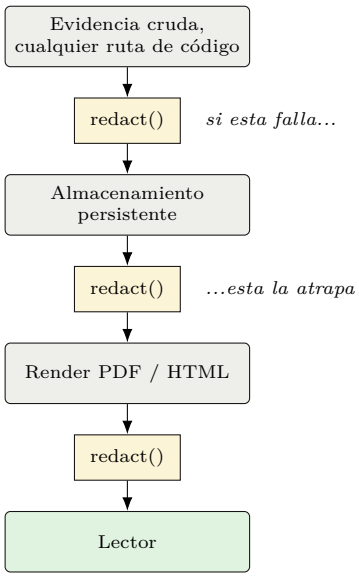


Figura 10: Tres capas de censura independientes, no una cadena de confianza secuencial. Un nuevo código de ruta que se salta la primera capa todavía pasa por las otras dos antes de llegar a un lector.

11. Principios de diseño

1. **Lo crítico no es opcional para el modelo.** Si una capacidad debe correr siempre, conviértela en un paso determinístico del pipeline, no en una herramienta más entre muchas.

2. **El recall se depura en el descubrimiento, no solo en la detección.** Un motor de confirmación perfecto no encuentra nada si nunca se le presenta el endpoint vulnerable.

3. **Reintenta la falla transitoria, acepta la definitiva.** Reintentar indiscriminadamente cualquier error es tan dañino para el recall como no reintentar nunca.

4. **Ningún hallazgo se cree en la misma pasada que lo produjo.** Un paso de verificación independiente, sin el razonamiento original a la vista, distingue un reporte de una alucinación con formato de reporte.

5. **El verificador también necesita ser verificado.** Una prueba cuya propia noción de "verdad" se mantiene a mano, sin atarse a la fuente real, se vuelve tautológica en silencio.

6. **Jerarquiza la evidencia y dilo en el reporte.** Un callback de red y una heurística de texto no deberían llegar al lector bajo la misma etiqueta de "confirmado".

7. **El contexto histórico es una pista, nunca un hecho.** Si un prompt autoriza al modelo a no re-verificar algo, ese algo terminará reportado sin evidencia.

8. **El entorno de ejecución es parte de la superficie de confiabilidad.** Un sistema que depende de que su entorno esté "bien configurado" tiene una dependencia frágil disfrazada de asunción técnica; esto incluye poder detectar su propio congelamiento, algo que ningún mecanismo puramente interno puede hacer.

9. **Ningún control de seguridad es una validación de una sola vez.** Repite la revisión en cada punto donde el sistema actúa de verdad, y usa capas independientes para datos sensibles — nunca un único punto de censura o de validación.

Nota metodológica: las observaciones anteriores provienen de operar un único sistema en producción a lo largo de varias sesiones de trabajo, no de un estudio controlado sobre múltiples sistemas. Las cifras citadas son mediciones reales de esas corridas; los nombres de objetivo, cliente y herramienta se omiten deliberadamente por confidencialidad.

Referencias

[1] MITRE. *CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition.* cwe.mitre.org/data/definitions/367.html.

[2] Jackson, C., Barth, A., Bortz, A., Shao, W., Boneh, D. *Protecting Browsers from DNS Rebinding Attacks.* ACM Conference on Computer and Communications Security (CCS), 2007.

[3] Wataoka, K., Takahashi, T., Ri, R. *Self-Preference Bias in LLM-as-a-Judge.* arXiv:2410.21819, 2024.