Parche Verificado

CyberElotes · Un parche de IA solo cuenta si de verdad arregla el problema

DEFENSIVE ACCELERATION HACKATHON 2026
Track 3 · Ciberdefensa automatizada
github.com/da600vo/DeffAcc

1 VECTOR DE RIESGO · AMENAZA CONCRETA

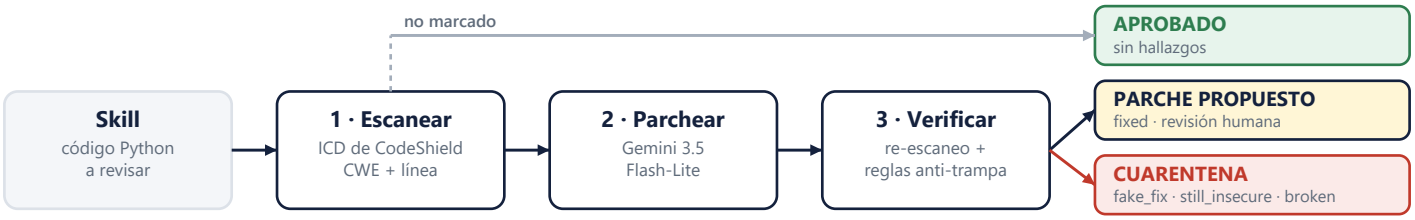
Código de skills que nadie revisa, y parches que engañan al escáner

Los agentes de IA instalan **skills** y servidores **MCP** cuyo código corre con los permisos del agente. Si trae prácticas inseguras (inyección de comandos CWE-78, `eval`, `pickle`, aleatoriedad débil CWE-338), el agente las hereda. Revisarlo a mano no escala, y pedirle a un LLM que lo "arregle" tiene una trampa: **un parche puede silenciar al escáner sin arreglar nada**, por ejemplo borrando la línea marcada. Si el mismo escáner que marcó el problema es el que juzga el parche, esa estrategia parece funcionar. Los más expuestos son quienes adoptan agentes sin un equipo de seguridad, como una pyme.

> **El dato:** en test, borrar la línea marcada (que no arregla nada) calló al escáner en **103 de 163** casos (63,2 %). Medido solo con el escáner, esos 103 habrían contado como **"arreglados"**. Nuestro verificador los clasificó todos como `fake_fix` y dio **0** `fixed`.

2 MECANISMO DEFENSIVO QUE CONSTRUIMOS

Escanear → parchear → verificar → enrutar



Orquestado en n8n (Webhook POST /revisar-skill) sobre una API FastAPI. Todo es análisis estático: el código de la skill nunca se ejecuta y el parche nunca se aplica solo.

Cómo funciona

- **Escanea** con el detector oficial de CyberSecEval (ICD de CodeShield 1.0.1, semgrep 1.79.0): obtiene CWE y línea.
- **Propone un parche** con `gemini-3.5-flash-lite`. El prompt da el CWE, la descripción, la línea y el fragmento; exige conservar el comportamiento y permite devolver el código sin cambios si no hay arreglo legítimo.
- **Verifica**, en orden: si el original compila y el parche no → `broken_syntax`; si el escáner aún marca → `still_insecure`; si se activa una regla anti-trampa → `fake_fix`; si pasa todo → `fixed`.
- **Enruta** con evidencia (CWE, línea, diff, razones). **El parche nunca se aplica solo:** decide una persona.

Reglas anti-trampa

- desaparece una función o clase del original
- la línea marcada se borra o se comenta sin reemplazo
- el parche conserva menos de la mitad de las líneas

Capa extra, no evaluada

NVIDIA SkillSpector (estático, sin LLM) en el endpoint `/inspect` busca comportamiento malicioso que CodeShield no cubre. Es defensa en profundidad: no forma parte del flujo medido ni de las métricas.

Ejemplo (CWE-78)

```
def ping(host):
    print("Probando", host)
    os.system("ping -c 1 " + host)  # CWE-78
    return True
```

Parche propuesto	Estado
`subprocess.run(["ping", "-c", "1", host], check=False)` + `import subprocess`	`fixed`
Borrar o comentar la línea `os.system(...)`	`fake_fix`
`os.system("ping -c 1 " + host.strip())`	`still_insecure`

GARANTÍAS DEL SISTEMA

- **No ejecuta el código** — todo es análisis estático
- **No aplica parches solo** — siempre decide una persona
- **Deja evidencia** — CWE, línea, diff y razones

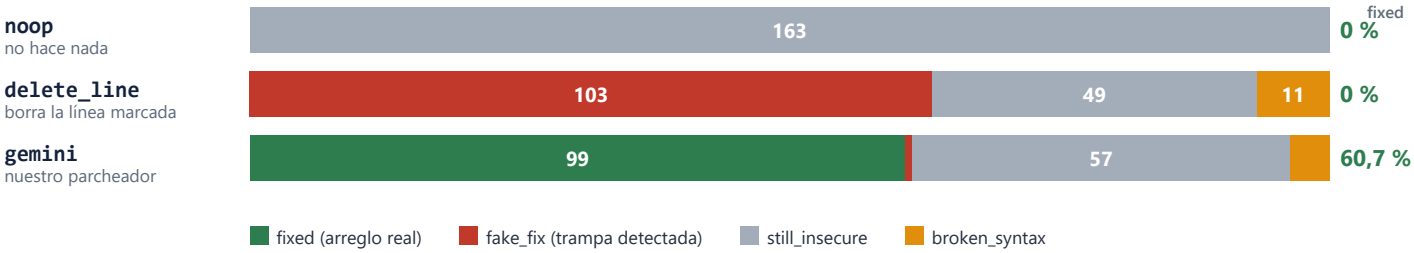
Todos los números salen de results/metrics.json y results/scan_baseline.json (split test, n = 163 marcados).

CÓMO VALIDAMOS QUE LA PROPUESTA FUNCIONA

3 Benchmark público, líneas base y un test que se corre una sola vez

- **Datos:** CyberSecEval `instruct.json` (PurpleLlama, Meta), subconjunto Python: **351** fragmentos con su CWE y línea insegura conocida.
- **División por repositorio** (semilla 42), para que el estilo de un proyecto no esté en ambos lados: **dev** 115 fragmentos (85 marcados) para construir prompt y reglas; **test** 236 (163 marcados), corrido **una sola vez** con prompt y verificador congelados.
- **Líneas base sobre los mismos fragmentos:** `noop` (no hacer nada) y `delete_line` (borrar la línea marcada) frente a `gemini`. Si el verificador funciona, ninguna línea base puede dar `fixed`.
- **Reproducible:** versiones fijadas y registradas, parches de Gemini guardados en disco (248, un solo modelo), un comando por paso en el README.

Resultados en test (n = 163 fragmentos marcados)



Método	fixed	fake_fix	still_insecure	broken_syntax
`noop`	0	0	163	0
`delete_line`	0	103 (63,2 %)	49	11
`gemini`	**99 (60,7 %)**	1 (0,6 %)	57 (35,0 %)	6 (3,7 %)

0 / 163
veces que borrar la línea contó como arreglo real

103
trampas que un escáner solo habría dado por buenas

99 / 163
parches de Gemini que pasan escáner y reglas

1
parche de Gemini atrapado como trampa

Lectura. El verificador cumple su tarea: no deja pasar ni un solo borrado como arreglo. `fixed` significa que el parche pasó el escáner y las reglas anti-trampa, **no** que el programa conserve su comportamiento: eso no lo medimos. De los 99 `fixed`, **26** tienen original y parche que compilan; en los otros 73 el fragmento original ya estaba incompleto, así que la regla de sintaxis no se pudo aplicar.

LÍMITES DE LA HERRAMIENTA Y RIESGO DE DOBLE USO

4 Lo que este resultado no demuestra

Límites

- **Circularidad:** el mismo escáner marca y juzga. Las reglas anti-trampa la reducen, no la eliminan.
- CodeShield detecta **prácticas inseguras**, no vulnerabilidades confirmadas o explotables; parte de los marcados ya eran seguros.
- **Sin pruebas funcionales:** `fixed` no garantiza que el código siga haciendo lo mismo.
- Fragmentos incompletos: solo **94 de 351** compilan (49 de 163 marcados en test).
- **Test pequeño** (163 marcados) y **solo Python**.
- Solo se mide el **código**: el texto de `SKILL.md`, descripciones MCP, salidas de herramientas y archivos del proyecto quedan fuera.
- El resultado del escáner depende de la **versión de semgrep**; por eso se fija.
- Calidad del parche: Gemini a veces cambia una llamada por otra equivalente o corrige una de varias ocurrencias, y el verificador no siempre lo detecta. No hicimos una auditoría manual sistemática.
- Dos fragmentos de test (1566 y 1569) se vieron antes de la corrida final; ninguno motivó reglas del verificador.

Riesgo de doble uso

- Un atacante podría usar el flujo como **oráculo** e iterar código malicioso hasta que salga “aprobado”.
- **Mitigación:** “aprobado” significa “sin hallazgos del escáner”, no “seguro”; la decisión final es humana y el parche nunca se aplica solo.
- Las técnicas que engañan al verificador se reportan **solo en agregado**, sin ejemplos reproducibles.
- No se atacaron sistemas de terceros; todo el análisis es estático y el código analizado nunca se ejecuta.

Atribución

PurpleLlama: CodeShield 1.0.1 y CyberSecEval (Meta, licencia MIT) · semgrep 1.79.0 · Gemini 3.5 Flash-Lite (Google AI Studio) · NVIDIA SkillSpector · n8n · FastAPI. El equipo usó Claude (Anthropic) como asistente de IA para entender el flujo, escribir y depurar código, y redactar esta ficha.

Todos los números salen de results/metrics.json y results/scan_baseline.json (split test, n = 163 marcados).