// RECOVERED RESEARCH LOG · NOVEMBER 2026

The Ghost in the Patch

Hybrid Vuln Research: known CVEs, and bugs they leave behind

Sergei Zaiats

CHAPTER 01

The Mission

> October, as usual

Every year since 2014, **Hacktoberfest** meant one thing:
send pull requests to open source.

In 2026 the rules changed.

> “Pull requests and merge requests will no longer count toward Hacktoberfest rewards. It's easier than ever to submit low-effort spam PRs…”
>
> — Hacktoberfest 2026 FAQ

I decided to take part the old way anyway.

But for a security person, the most useful contribution is not a typo fix.

It's a vulnerability, found and reported responsibly.

> Flashback: 2022

I gave a talk about fuzzing.
Link to that talk

The main conclusion:

> the fuzzer is not the bottleneck.
> **Writing harnesses is.**

A harness is slow, careful work:

- understand the API
- decide what an input means
- decide what counts as “broken”
- repeat for every entry point

HYPOTHESIS

Harnesses are small, repetitive pieces of code with clear goals.

What if an AI agent writes them?

CHAPTER 02

Fuzzing 101

> What is a fuzzer?

A program that runs your code **millions of times** with generated inputs, looking for the one input that breaks it.

- a unit test checks the inputs you thought of
- a fuzzer finds the ones you did not

```
corpus = seeds
while True:
    data = mutate(pick(corpus))
    result = run(data)
    if result.crashed:
        save(data, "crashes/")
    elif result.new_coverage:
        corpus.add(data)
```

> Seeds, corpus, oracle

seeds

A few valid example inputs to start from.

Here: the project's own test files.

corpus

Starts as the seeds, then keeps every input the fuzzer finds interesting, usually because it reached new code.

The fuzzer's memory: a working set to mutate, not a map of paths.

oracle

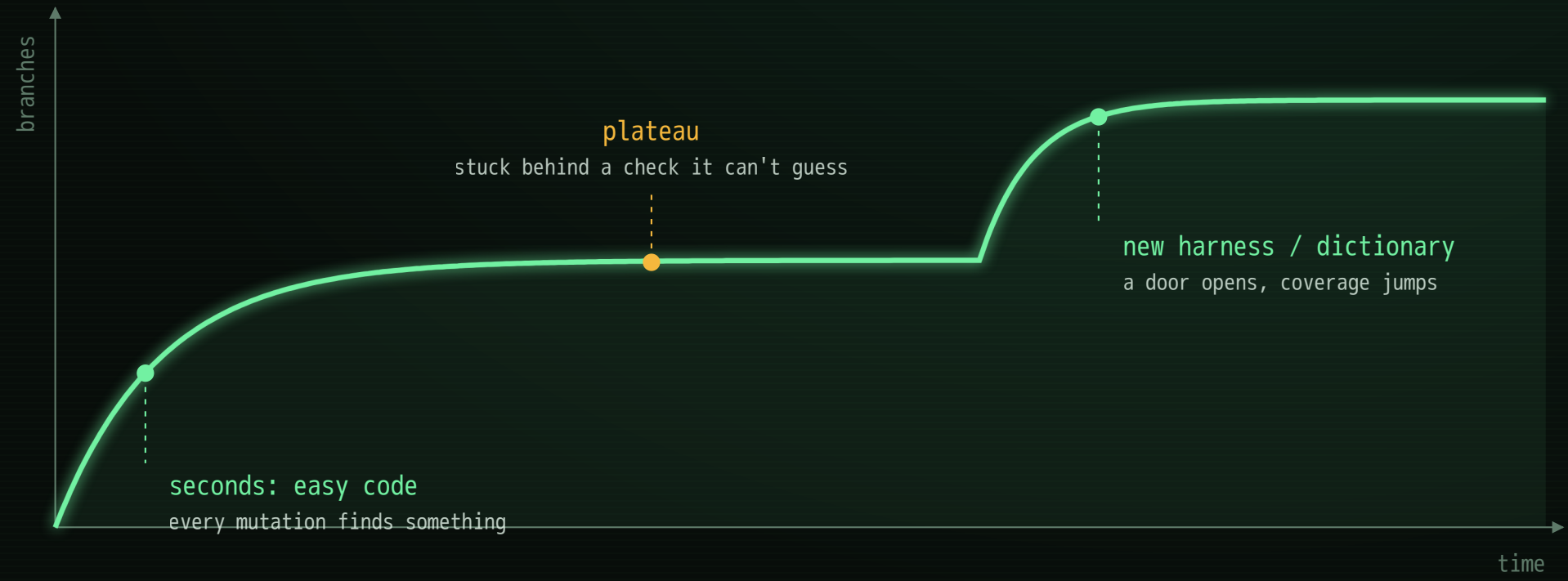
A check that signals when a run may have exposed a bug.

A crash, timeout, or broken rule can be a signal.

> How coverage grows

the compiler puts a counter on every branch; a new branch hit = keep the input

illustrative curve



> A fuzzer needs a harness

the fuzzer brings bytes; the harness turns them into a call and judges the result



coverage feedback: “this input reached new code, keep it in the corpus”

> What a harness looks like

```
func FuzzScriptBreakout(f *testing.F) {
    m := newMinifier()
    f.Fuzz(func(t *testing.T, data []byte) {
        // 1. shape: hide the fuzz bytes inside a JS string
        src := `<script>var s="` + hexEscape(data) + `";</script>`

        // 2. call the library
        out, _ := m.String("text/html", src)

        // 3. check: the output must not close the <script> early
        if breaksOutOfScript(out) {
            t.Fatalf("minifier introduced a breakout: %q", out)
        }
    })
}
```

Real harness from this research, slightly simplified. We will meet it again.

> The oracle: what counts as a bug?

Oracle	Detects	Example
crash	panic, memory corruption (caught by a sanitizer)	nil dereference, out of bounds
budget	too slow, too much memory	quadratic parsing, deep recursion
property	output breaks a rule	“safe input stays safe”
race detector	concurrent memory access	shared state between requests

Without a property oracle, a fuzzer never sees XSS or a logic bug: the program does not crash.

> Words we will use

Term	Meaning
crash / hang	an input that broke the oracle / never finished
triage	group crashes by root cause, rank by severity
minimization	shrink the input to the smallest one that still fails
PoC	minimal input + steps to reproduce, sent with the report
patch bypass	a new input that still triggers a bug the patch was meant to fix

CHAPTER 03

Experiment 0

> Can the agent find known bugs?

A Markdown library, pinned to an old commit with **published** vulnerabilities.

- the agent writes a harness from scratch
- no advisories, no hints
- seeds: only the project's own test files
- the score: how many known bugs come back

t = 0 · the library as a grid of code blocks

0 execs

code coverage 0.0%

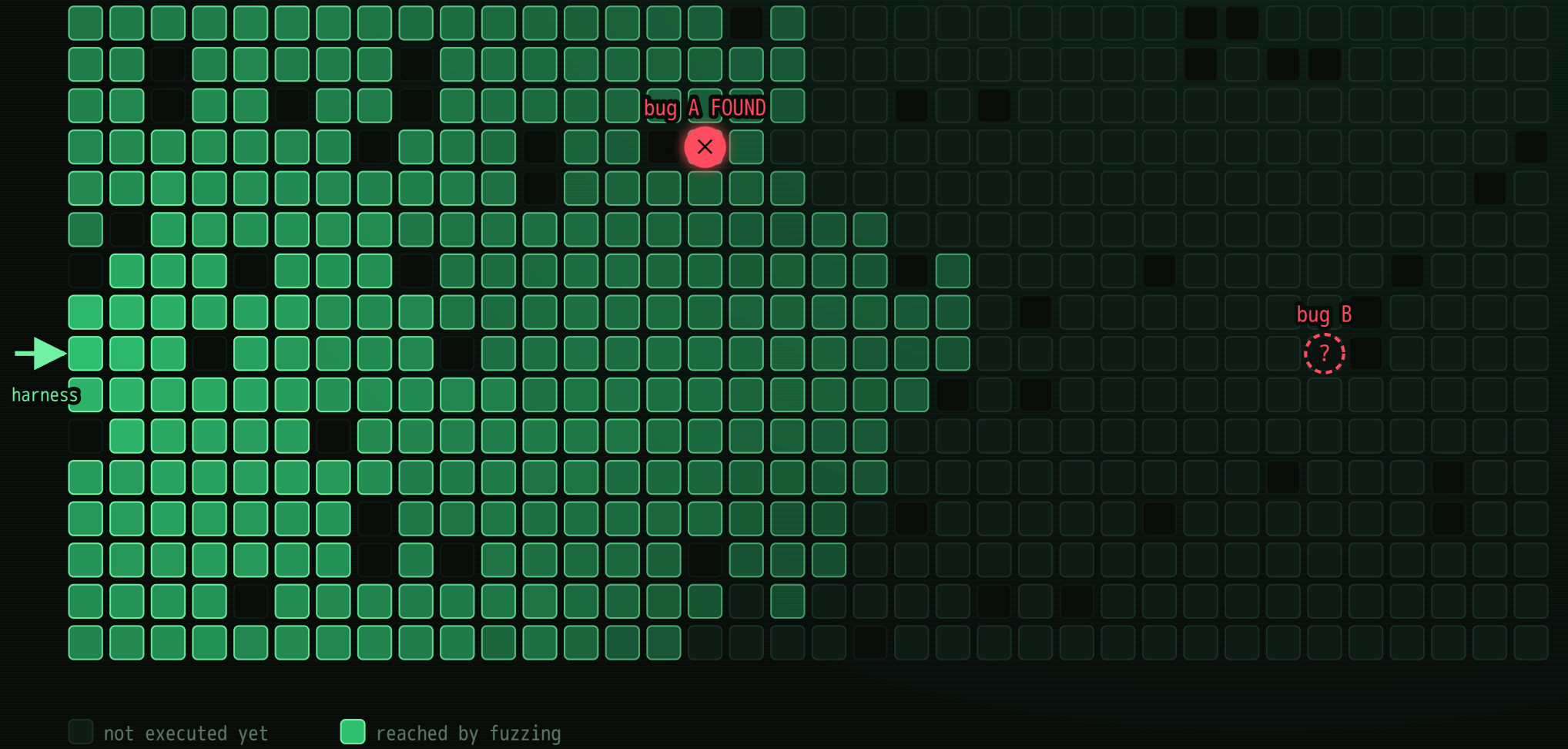


not executed yet reached by fuzzing

t = 2 s · coverage starts to spread

2 seconds

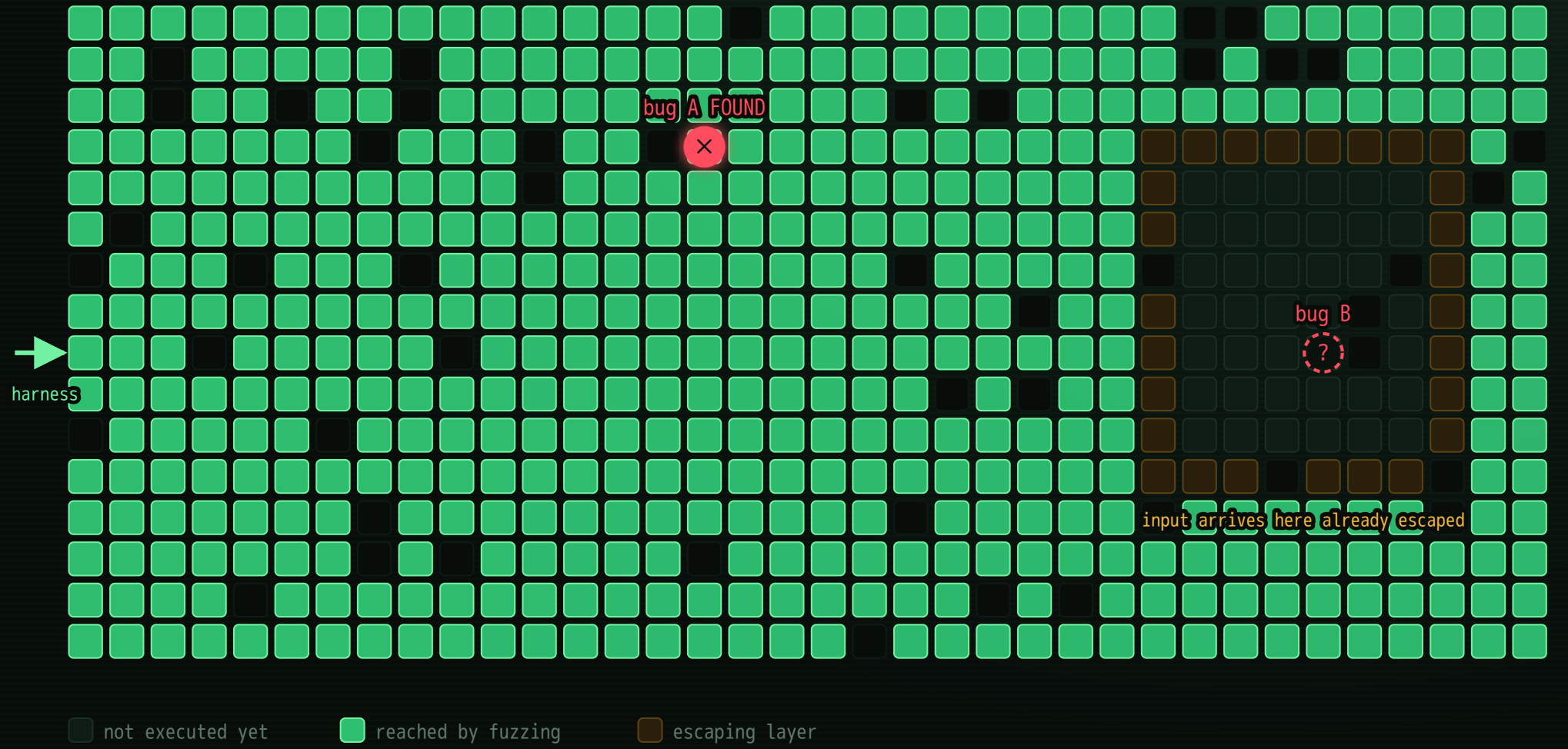
code coverage 53.4%



t = 10 min · coverage plateau

40,000,000 execs

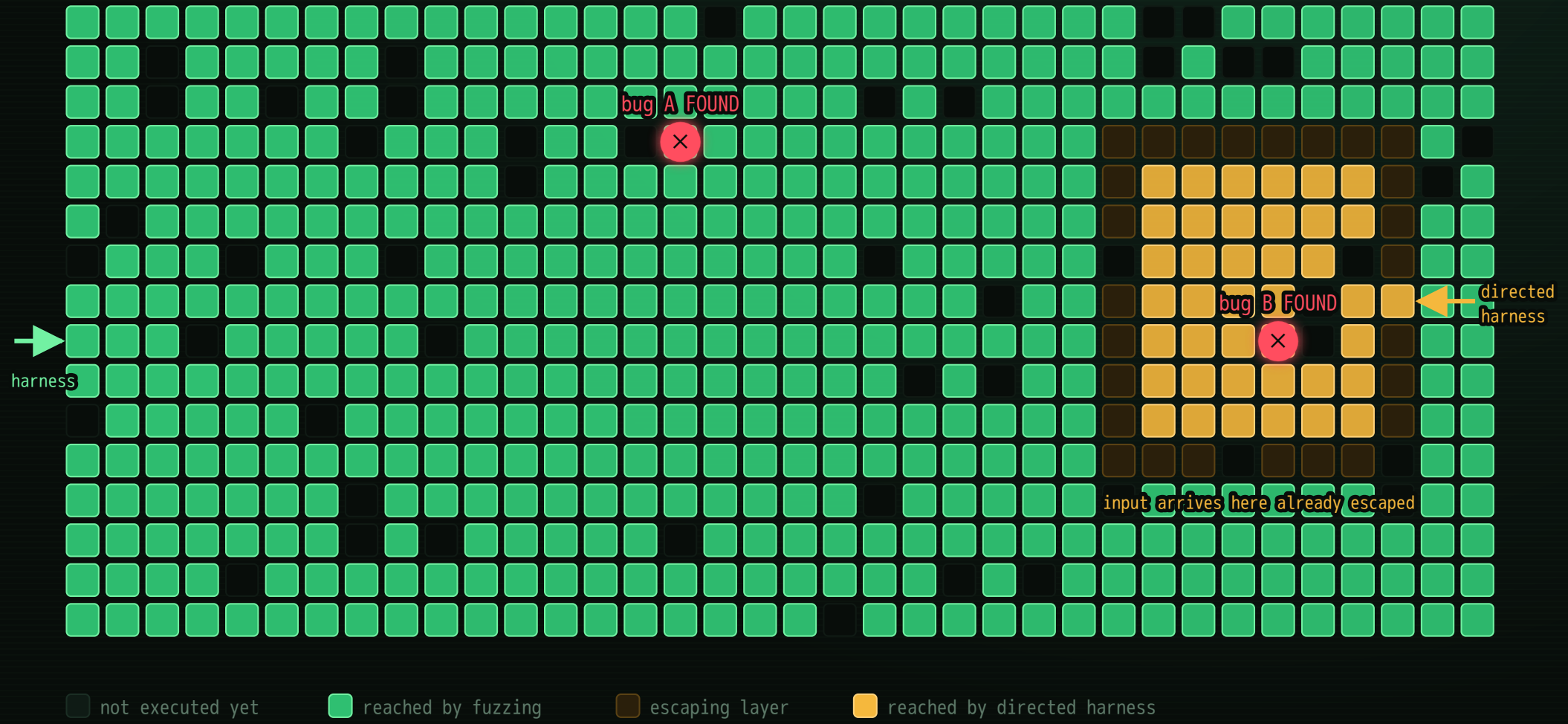
code coverage 87.5%



+ 1 s · a second harness calls the component directly

< 1 s

code coverage 95.1%



A fuzzer only finds what the harness can **reach**…

…and only what the oracle can **see**.

So the real question is: where do we aim?

CHAPTER 04

CVE North Stars

> Use known bugs as a compass

CVE North Stars (cve-north-stars.github.io)

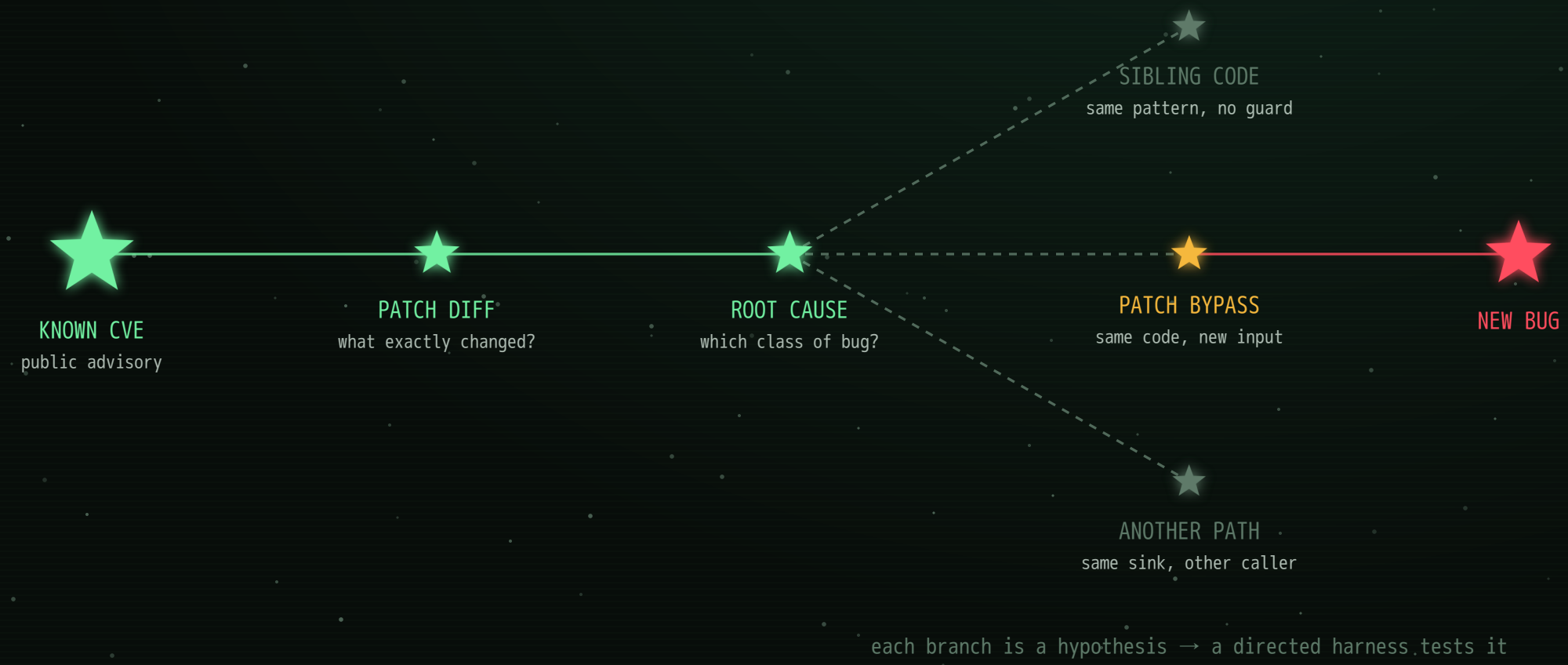
A research method: start from a public CVE, not from a blank page.

1. pick a known vulnerability
2. **diff the patch**: what exactly changed?
3. **root cause**: was it fixed, or just one symptom?
4. look for the same bug **elsewhere**

Why it works:

- the patch points to the exact function
- the advisory names the bug class
- developers fix the reported case, rarely the pattern

> From one star to a constellation



each branch is a hypothesis → a directed harness tests it

> The patch is a map

The guard that a fix adds marks the **boundary of the vulnerability**.

```
// after the fix: re-escape dangerous sequences in decoded JS strings
if b[i] == '<' {
    if bytes.Equal(b[i+1:i+4], []byte("!--")) {
        // escape <!--
    } else if parse.EqualFold(b[i+1:i+9], []byte("/script>")) {
        // escape </script>
    }
}
```

Questions for every diff: Which inputs does this check cover? Which does it miss? Where else does this pattern appear?

> Three shapes of an incomplete fix

ONE PATH FIXED

The same unsafe pattern exists in sibling functions. Only the reported one got a guard.

→ grep the pattern, not the function name

CHECK TOO NARROW

The guard matches the exact PoC, not the whole class of inputs.

→ fuzz around the boundary

LIMIT IN THE WRONG PLACE

The parser gets a limit, but the dangerous operation is still reachable without the parser.

→ call the operation directly

> The agent's workflow

Skill	What the agent does
`select-target`	pick an active, fuzzable project with a security history
`cve-seed`	diff known fixes, write down hypotheses
`write-harness`	one harness + oracle per hypothesis
`build-target` · `run-fuzzer`	instrumented build and fuzzing, in Docker
`triage-crash`	deduplicate, classify, “is this already known?”
`write-poc`	minimal PoC + report for the maintainer

Each skill is a short Markdown runbook. Every mistake the agent makes becomes a new rule in one of them.

CHAPTER 05

Case File: the Minifier

> The known bug

A popular Go minifier for HTML, CSS, JS. A fresh advisory:

> The JS minifier decodes escapes in strings. `"\x3c/script\x3e"` becomes a literal `</script>` and closes the script tag. **XSS.**

The fix: after decoding, re-escape `</script>` and `<!--`.

> The hypothesis

The fix looks for exactly 8 characters: `/script>`.

What does the **HTML specification** say ends a script element?

THE FIX ASSUMES

`</script>`

THE BROWSER ACCEPTS

`</script` followed by `>`, space, tab, newline, form feed, or `/`

Shape #2: the check is too narrow.

> Turning the hypothesis into a harness

The oracle is a **property**, not a crash:

> If the input cannot break out of the script tag, the minified output must not break out either.

- the fuzz bytes are always hex-escaped, so the source is **always safe**
- the oracle re-implements the HTML rule for ending a script
- any breakout is therefore **created by the minifier**

> Found

Safe input — no literal `</script` anywhere

```
<script>var s="\x3c/script\x20\x3e\x3cimg src=x onerror=alert(1)\x3e";</script>
```

Minified output

```
<script>var s="</script ><img src=x onerror=alert(1)>"</script>
```

The browser ends the script at `</script` , then the `<img>` runs: XSS.

Reported → fixed in **v2.24.24** → advisory published. Fix re-verified with the same harness.

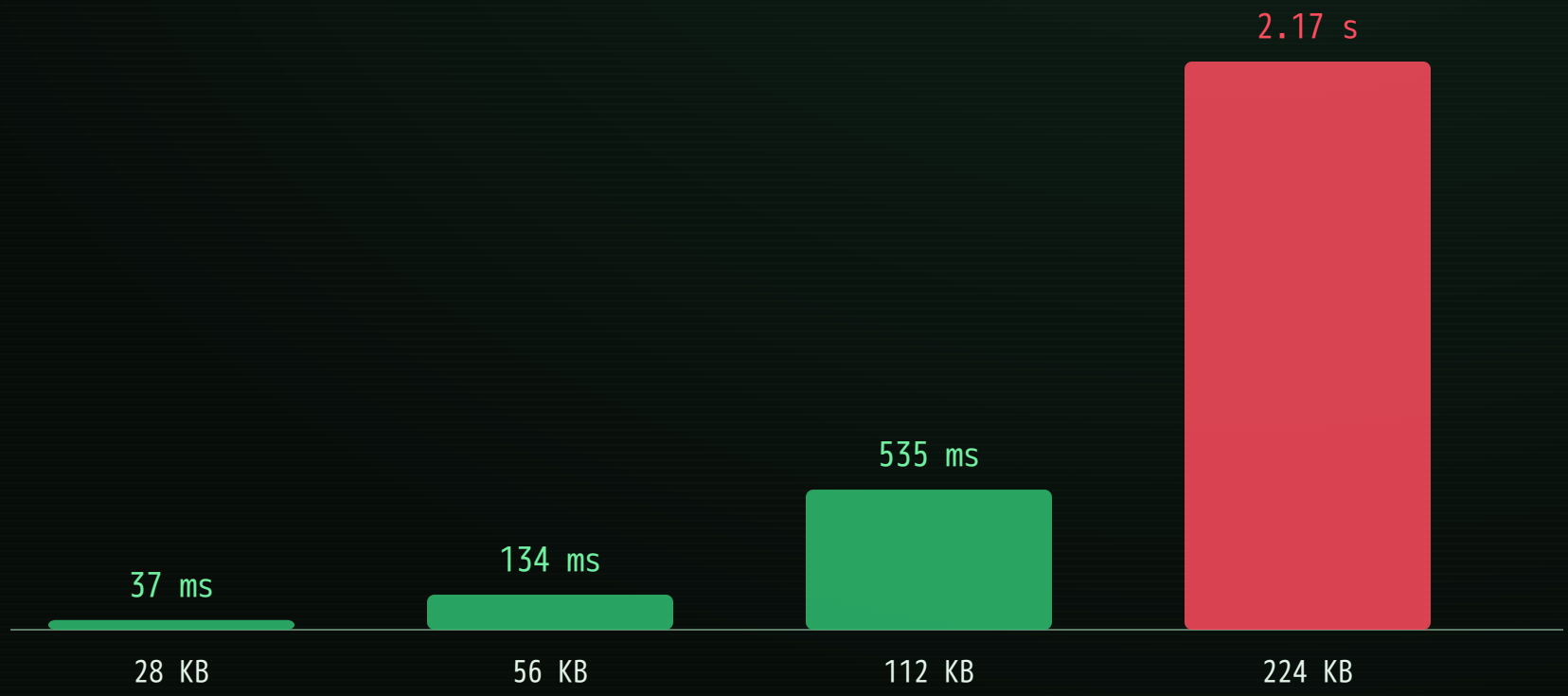
CHAPTER 06

More Ghosts

> When “slow” is a vulnerability

input size ×2 → parse time ×4

repeated unclosed tags, default parser, a single request



> Round three

An OpenAPI request validator. The advisory history already told a story:

Round	What happened
1	crash on a missing field, fixed in 3 code paths
2	“incomplete fix”: 3 more paths, same file
1′	infinite recursion on self-referencing schemas, fixed in 2 functions
3	**2 more functions recurse the same way, no guard. One HTTP request kills the process.**

An “incomplete fix” advisory is a strong hint there is another one.

OCTOBER, IN NUMBERS

10

vulnerabilities reported to maintainers, across **7 projects**

> The board

01 XML library
algorithmic DoS
blind fuzzing

02 XML library
recursion, patch gap
patch diff

03 SAML
signature bypass via dependency
patch diff

04 OpenAPI validator
stack overflow, 3rd round
patch diff

05 WAF
argument name collision
variant analysis

06 WAF
data race, log misattribution
impact escalation

07 WAF
data race, cross-request data
impact escalation

08 JOSE / JWK
nil dereference DoS
blind fuzzing

09 Markdown
quadratic DoS
variant analysis

10 Minifier
XSS, incomplete fix
patch diff

CHAPTER 07

What Actually Mattered

> Who did what

THE AGENT

- read advisories and diffs
- grep for sibling patterns
- write and rewrite harnesses
- build, run, minimize, re-run
- draft PoCs and reports

THE HUMAN

- choose targets worth the time
- question every “found it!”
- decide if the impact is real
- set a fair severity
- talk to maintainers

> Where the agent goes wrong

- **overclaims**: a panic behind a recovery handler is not a DoS
- **cheats by accident**: uses the known PoC as a seed, then “finds” it
- **trusts silence**: “0 crashes” after a harness that never reached the code
- **rediscovers**: a bug that is already reported, fixed, or in an open PR

Every one of these became a rule in a skill.

> Takeaways

HARNESS = REACH

A fuzzer only explores what the harness exposes. Aim it.

PATCHES = MAPS

A fix shows where the bug lived. Ask what it did not cover.

AI = SPEED

The agent removes the slow part. Judgment stays human.

// END OF LOG

Questions?

CVE North Stars · cve-north-stars.github.io

Go fuzzing · go.dev/doc/security/fuzz

AFL++ · aflplus.plus