THOMAS MORE

Lab 3.3

Security Scanning with Semgrep, Trivy, and SBOM

Toepasbare Cloud Security voor IT-Professionals

2025 - 2026

Contents

1. Security Scanning with Semgrep, Trivy, and SBOM

1.1. Context

In the previous lab, the pipeline checked whether the application was built and tested correctly. In this lab, you add security scanning to the same delivery flow.

You will work with three security checks:

- **Semgrep** scans the source code for insecure patterns.
- **Trivy** scans dependencies and the filesystem for known vulnerabilities and secrets.
- **CycloneDX SBOM** creates an inventory of the npm packages used by the backend.

The theory behind these tools is explained in **Before Lab 3.3 — Security Gates: Semgrep, Trivy, and SBOM**. This lab focuses on using the tools, reading the output, and adding the scan jobs to Azure DevOps.

> **Local versus pipeline environment**
>
> In class, local work is done on **Windows with PowerShell**. The Azure DevOps pipeline still runs on an **Ubuntu agent**, so the YAML snippets may use Bash-style commands. Use the PowerShell commands in this lab for local work, and keep the Linux commands inside the pipeline unless instructed otherwise.

1.2. Learning goals

By the end of this lab, you can:

- Run Semgrep and Trivy against the backend application.
- Generate and inspect a CycloneDX SBOM.
- Read security findings and classify them as fix, suppress, accept, or investigate.
- Add Semgrep, Trivy, and SBOM generation to the Azure DevOps Validate stage.
- Explain why the first version of the security gate runs in audit mode.

1.3. Estimated time

75 minutes

Time	Activity
10 minutes	Check local tooling and repository state
15 minutes	Run Semgrep and review one finding
15 minutes	Run Trivy and review one dependency finding
10 minutes	Generate and scan an SBOM
10 minutes	Classify findings
15 minutes	Add the security scan jobs to the Azure DevOps pipeline

1.4. Before you start

Make sure you have:

- [ ] Repository cloned locally.
- [ ] Backend dependencies installed with `npm install`.
- [ ] `.semgrep.yml` present in the repository.
- [ ] `trivy.yaml` present in the repository.
- [ ] Access to the Azure DevOps pipeline.

Recommended local tools:

- Python and pip, for installing Semgrep.
- Trivy for Windows.
- Node.js and npm.
- Optional: Docker Desktop, if you prefer container-based scans.

> **Note**
>
> If you cannot install Semgrep or Trivy locally, continue with the pipeline part of the lab and review the scan results in the Azure DevOps logs and artifacts.

1.5. Repository paths used in this lab

Use these placeholders consistently:

Placeholder	Meaning
`<REPOSITORY_ROOT>`	The root folder of your cloned repository
`<BACKEND_APP_FOLDER>`	The backend application folder, for example `course-cloud-security-app/apps/backend`

In PowerShell, move between folders like this:

```
cd <REPOSITORY_ROOT>
cd <BACKEND_APP_FOLDER>
```

> **No chmod needed locally**
>
> You do not need `chmod`, `ls -la`, or Unix file permission commands on Windows for this lab. Those commands are only relevant inside Linux/macOS environments or Ubuntu-based pipeline jobs.

2. Part 1 — Run Semgrep locally

2.1. Install or verify Semgrep

Check whether Semgrep is available:

```
semgrep --version
```

If the command is not found, install it with pipx.

You can obtain pipx from: https://pipx.pypa.io/stable/how-to/install-pipx/

```
pipx install semgrep
```

Then check the version again:

```
semgrep --version
```

2.2. Run the scan

From the repository root:

```
cd <REPOSITORY_ROOT>
semgrep --config .semgrep.yml
```

If you use Docker Desktop instead, run (not verified):

```
docker run --rm -v ${PWD}:/src semgrep/semgrep semgrep --config /src/.semgrep.yml /
src
```

2.3. Review the result

Answer the following questions.

How many findings does Semgrep report?

Your answer:

Which rule IDs are triggered?

Your answer:

Pick one finding and inspect it.

Question	Your answer
Rule ID	

File and line number	
Short description	
Why could this be risky?	

2.4. Inspect the Semgrep rules

Open `.semgrep.yml` in your editor.

Which rule checks for hardcoded secrets?

Your answer:

Which rule checks for SQL injection risk?

Your answer:

> **Reflection**
>
> Semgrep looks for patterns in source code. Can it prove that every possible SQL injection bug is absent, or can it only detect patterns that its rules describe?

Your answer:

3. Part 2 — Run Trivy locally

3.1. Install or verify Trivy

Check whether Trivy is available:

```
trivy --version
```

If it is not installed, install Trivy for Windows using the installation instructions at: https://trivy.dev/docs/latest/getting-started/installation/#windows-official make sure to add the Trivy executable to your system PATH.

3.2. Run a filesystem scan

From the backend folder:

```
cd <BACKEND_APP_FOLDER>
trivy fs --config ..\trivy.yaml .
```

If `trivy.yaml` is not one level above the backend folder, adjust the path to the actual location of the file.

If you use Docker Desktop from the repository root, run (not verified):

```
cd <REPOSITORY_ROOT>
docker run --rm -v ${PWD}:/src aquasec/trivy fs --config /src/trivy.yaml /src/course-cloud-security-app/apps/backend
```

3.3. Review the result

How many vulnerabilities does Trivy report?

Your answer:

Fill in the severity breakdown.

Severity	Count
CRITICAL	
HIGH	
MEDIUM	
LOW	

Pick one dependency finding.

Question	Your answer
CVE ID	
Affected package and version	
Severity	

Fixed version, if available	
Is this package used by the application?	
Initial decision: fix, suppress, accept, or investigate	

3.4. Check for secrets

Does Trivy report hardcoded secrets?

Your answer:

> **Reflection**
>
> A Key Vault URL is usually configuration, not a secret. A password, token, API key, or connection string with credentials is a secret. Does the Trivy output match that distinction?

Your answer:

4. Part 3 — Generate and scan an SBOM

An SBOM is a package inventory. The theory page explains why this matters; here you only generate and inspect one.

4.1. Generate the SBOM

From the backend folder:

```
cd <BACKEND_APP_FOLDER>
npx @cyclonedx/cyclonedx-npm --output-file sbom.json
```

Check that the file exists:

```
Get-ChildItem sbom.json
```

Open `sbom.json` and find one dependency.

Question	Your answer
Package name	
Version	

4.2. Scan the SBOM with Trivy

```
trivy sbom sbom.json
```

Did Trivy report the same dependency vulnerabilities as the filesystem scan, fewer, or more?

Your answer:

What is one reason to publish the SBOM as a pipeline artifact?

Your answer:

5. Part 4 — Classify findings

TODO: I WAS HERE

Use this small decision table while reviewing your scan results.

Decision	Use when...	Example action
Fix	The issue is real, relevant, and has a reasonable fix.	Update dependency, remove secret, refactor unsafe code.
Suppress	The tool is wrong or the finding does not apply.	Add a suppression with a short justification.
Accept	The risk is known and temporarily acceptable.	Document reason and review date.
Investigate	You cannot yet tell whether the finding applies.	Check code usage, release notes, or exploit conditions.

5.1. Classify one Semgrep finding

Question	Your answer
Rule ID and file	
Decision	
Justification	
Action taken or planned	

5.2. Classify one Trivy finding

Question	Your answer
CVE ID and package	
Decision	
Justification	
Action taken or planned	

6. Part 5 — experiment: introduce vulnerable code

This part helps you see that the scanners actually detect new problems.

> **Warning**
> Use a separate branch. Do not merge this code to `main`.

6.1. Create a branch

```
cd <REPOSITORY_ROOT>
git checkout -b security-scan-demo
```

6.2. Add an intentionally insecure route

Create this file:

`course-cloud-security-app/apps/backend/src/routes/insecure.ts`

Add the following code:

```
import { Router } from "express";
import cors from "cors";

export const insecureRouter = Router();

// INTENTIONAL VULNERABILITY: hardcoded API key
const THIRD_PARTY_API_KEY = "sk_live_1234567890abcdef";

// INTENTIONAL VULNERABILITY: unsafe CORS configuration
insecureRouter.use(cors({
  origin: "*",
  credentials: true,
}));

insecureRouter.get("/external-api", async (_request, response) => {
  const apiResponse = await fetch("https://api.example.com/data", {
    headers: {
      "Authorization": `Bearer ${THIRD_PARTY_API_KEY}`,
    },
  });

  const data = await apiResponse.json();
  response.json(data);
});

insecureRouter.get("/search", async (request, response) => {
  const { query } = request.query;

  // INTENTIONAL VULNERABILITY: user input is inserted directly into SQL
  const sql = `SELECT * FROM users WHERE name = '${query}'`;
```

```
  response.json({
    message: "Search executed",
    sql,
  });
});

insecureRouter.get("/calculate", (request, response) => {
  const { expression } = request.query;

  if (!expression || typeof expression !== "string") {
    response.status(400).json({ error: "Missing expression parameter" });
    return;
  }

  // INTENTIONAL VULNERABILITY: eval executes arbitrary JavaScript
  const result = eval(expression);

  response.json({ expression, result });
});
```

Register the route in `app.ts`:

```
import { insecureRouter } from "./routes/insecure.js";

// Add near the other routes:
app.use("/api/insecure", insecureRouter);
```

6.3. Add a vulnerable dependency

From the backend folder:

```
cd <BACKEND_APP_FOLDER>
npm install lodash@4.17.15
```

6.4. Re-run the scans

```
cd <REPOSITORY_ROOT>
semgrep --config .semgrep.yml

cd <BACKEND_APP_FOLDER>
trivy fs --config ..\..\..\trivy.yaml .
```

Which new Semgrep rules are triggered?

Your answer:

Does Trivy detect the vulnerable lodash version?

Your answer:

6.5. Clean up the experiment

Remove the insecure route file:

```
Remove-Item course-cloud-security-app/apps/backend/src/routes/insecure.ts
```

Remove these lines from `app.ts`:

```
import { insecureRouter } from "./routes/insecure.js";
app.use("/api/insecure", insecureRouter);
```

Upgrade lodash again:

```
cd <BACKEND_APP_FOLDER>
npm install lodash@latest
```

Verify the scans again:

```
cd <REPOSITORY_ROOT>
semgrep --config .semgrep.yml

cd <BACKEND_APP_FOLDER>
trivy fs --config ..\..\..\trivy.yaml .
```

Either commit the cleanup or delete the branch:

```
git checkout main
git branch -D security-scan-demo
git push origin --delete security-scan-demo
```

7. Part 6 — Add security scanning to Azure DevOps

The local commands used PowerShell because you are working on Windows. The Azure DevOps agent for this course uses Ubuntu, so the pipeline jobs may use Bash commands.

7.1. Review the pipeline snippet

Open:

`pipeline-snippets/lab-3-3-security-scanning.md`

This snippet should contain jobs for:

- Semgrep code scanning.
- CycloneDX SBOM generation.
- Trivy dependency or SBOM scanning.
- Publishing the security reports as pipeline artifacts.

7.2. Add the jobs to the Validate stage

Open:

`azure-pipelines.yml`

Find the `Validate` stage. Add the new jobs from the snippet at the same indentation level as the existing `QualityGates` job.

Expected result:

```
Validate
├─ QualityGates
├─ SecurityCodeAnalysis
└─ SecurityDependencyScanning
```

> **Important**
>
> Jobs inside one Azure DevOps stage run in parallel by default. Stages still run sequentially. In this lab, the security scan jobs should run in the Validate stage before the Build and Deploy stages. Keep in mind that this is limited by the amount of parallel jobs your Azure DevOps plan allows. If you have a free plan with only one parallel job, the security scan jobs will run sequentially, but they will still be part of the Validate stage and run before Build and Deploy.

7.3. Commit and push

```
cd <REPOSITORY_ROOT>
git add azure-pipelines.yml
git commit -m "feat: add security scanning jobs"
git push
```

7.4. Review the pipeline run

In Azure DevOps, open the new pipeline run.

Answer the following questions.

Question	Your answer
Do the Semgrep and Trivy jobs start in the Validate stage?	
Do they run next to the quality gate job?	
Does the pipeline continue even if findings are reported?	
Which security artifacts are published?	

7.5. Audit mode

The first version of these jobs normally runs in audit mode, for example with `continueOnError: "true"`.

Why is audit mode useful when introducing security scanning into an existing project?

Your answer:

When would you switch from audit mode to enforcement mode?

Your answer:

8. Reflection questions

8.1. Question 1 — Semgrep versus Trivy

Give one example of a problem Semgrep can detect better than Trivy.

Your answer:

Give one example of a problem Trivy can detect better than Semgrep.

Your answer:

8.2. Question 2 — False positive

Semgrep reports a hardcoded secret in a test file:

```
process.env.TEST_API_KEY = "test-key-12345";
```

Is this definitely a real vulnerability? What would you check before suppressing it?

Your answer:

8.3. Question 3 — CVE exploitability

Trivy reports a high severity CVE in a package. The package is installed, but you are not sure whether the vulnerable function is used.

What steps would you take before deciding to fix, suppress, accept, or investigate further?

Your answer:

8.4. Question 4 — Pipeline policy

Should every security finding immediately fail the pipeline? Why or why not?

Your answer:

9. Summary

In this lab, you:

- Ran Semgrep locally to inspect source-code security findings.
- Ran Trivy locally to inspect dependency and secret findings.
- Generated a CycloneDX SBOM and scanned it with Trivy.
- Classified findings using a simple decision table.
- Optionally introduced vulnerable code to test scanner behavior.
- Added security scanning jobs to the Ubuntu-based Azure DevOps pipeline.
- Reviewed security reports and artifacts in Azure DevOps.

> **Note**
>
> In the next lab, you will focus on infrastructure-as-code scanning with Checkov.