Trivy: A Comprehensive Learning Roadmap

Trivy is the "Swiss Army knife" of cloud-native security. Since you are a CSE student focusing on DevOps and K8s, mastering Trivy will be a massive asset for your projects like **K8s-Genesis** and your **GSoC** aspirations.

Here is a comprehensive roadmap from basic to advanced topics.

🟢 1. Core Fundamentals (The Basics)

- **Concepts of Vulnerability Scanning:** Understanding Common Vulnerabilities and Exposures (CVEs), CVSS scores, and the "Shift Left" security philosophy.
- **Installation & Environment Setup:** Installing the single binary on Linux/macOS/Windows and running Trivy via Docker.
- **Target Scanning Basics:**
  - **Container Images:** Scanning local and remote images (e.g., `trivy image nginx`).
  - **Filesystems:** Scanning a local directory for vulnerabilities in dependencies.
- **Understanding the Report:** Analyzing the output table: Vulnerability ID, Severity (Low to Critical), Installed Version, and Fixed Version.
- **Essential CLI Flags:**
  - `--severity`: Filtering results (e.g., `HIGH,CRITICAL`).
  - `--format`: Outputting to `json`, `table`, or `template`.
  - `--exit-code`: Using `1` to fail a process if vulnerabilities are found (critical for CI/CD).

🟡 2. Intermediate Scanning & Management

- **Infrastructure as Code (IaC) Scanning:**
  - Scanning **Dockerfiles** for best practices.
  - Scanning **Kubernetes Manifests**, **Terraform**, **CloudFormation**, and **Helm Charts**.
- **Secret Detection:** Scanning repositories for accidentally committed passwords, API keys, and SSH keys.
- **SBOM (Software Bill of Materials):**
  - Generating SBOMs in **CycloneDX** or **SPDX** formats.
  - Scanning an existing SBOM file for vulnerabilities.
- **License Scanning:** Identifying software licenses to ensure legal compliance (e.g., GPL vs. Apache).
- **Database Management:**
  - Understanding `trivy-db` and `trivy-java-db`.
  - Managing updates with `--skip-db-update` or `--download-db-only`.
- **Advanced Filtering:**
  - Using `.trivyignore` to suppress specific CVEs.

- Using `--ignore-unfixed` to focus only on actionable vulnerabilities.

🔴 **3. Advanced Infrastructure & Automation**

- **CI/CD Integration:**
  - **GitHub Actions:** Automating scans on every push or PR.
  - **GitLab CI / Jenkins / Azure DevOps:** Integrating Trivy into enterprise pipelines.
- **Kubernetes Advanced Scanning:**
  - **Cluster Scanning:** Using `trivy k8s cluster` to scan all resources in a live cluster.
  - **Namespace Filtering:** Targeting specific environments within K8s.
- **Trivy Operator:**
  - Installing and configuring the Operator for continuous security monitoring inside a K8s cluster.
  - Working with Custom Resource Definitions (CRDs) like `VulnerabilityReport`.
- **Client/Server Mode:** Setting up a centralized Trivy server to offload scanning from CI runners and centralize the vulnerability database.
- **Air-Gapped Environments:** Configuring Trivy for offline use by self-hosting the vulnerability databases.

🟣 **4. Specialized Features & Extensions**

- **Custom Policies (Rego):** Writing your own security rules using the **Rego** language (the same language used by Open Policy Agent/OPA).
- **VEX (Vulnerability Exploitability eXchange):** Using `.vex` files to document why a specific vulnerability is not exploitable in your specific context.
- **Advanced Output Templating:** Using Go-templates to create custom HTML or Markdown reports for stakeholders.
- **Trivy Plugins:** Installing and creating plugins to extend Trivy's CLI functionality.
- **Cloud Scanning:** Scanning AWS accounts (S3 buckets, EC2 configurations) directly via Trivy.

Would you like me to create a sample GitHub Action workflow that scans your Go port scanner code and fails the build if any Critical vulnerabilities are found?

Hands-on Introduction to Trivy This video provides a deep-dive walkthrough of how Trivy works, covering container scanning and integration strategies which are essential for your DevSecOps journey.

You're right—Trivy has evolved into a massive ecosystem that covers much more than just simple container scans. Since you are building **K8s-Genesis** and looking at **GSoC**, you need the "deep-dive" version that includes the specialized modules, internal mechanics, and extensibility.

Here is the exhaustive, categorized roadmap for Trivy.

🟦 **1. Architecture & Internal Mechanics**

To master Trivy, you must understand how it processes data under the hood.

- **Database Architecture:**
  - **trivy-db:** How vulnerability data is mirrored from GitHub.
  - **trivy-java-db:** Why Java needs a separate index for Maven/Gradle.
  - **Database Schema:** How Trivy maps packages to CVEs.
- **Scanning Engines:**
  - **Static Analysis:** How it parses lockfiles (`go.mod` , `package-lock.json`).
  - **Dynamic/Runtime:** How Trivy uses `fanotify` for runtime monitoring (advanced).
- **Cache Management:**
  - Understanding the `/root/.cache/trivy` directory.
  - Cleaning and managing the BoltDB cache.

🟩 **2. Expanded Scanning Targets (Beyond Images)**

Trivy can scan almost anything in a modern cloud-native stack.

- **Virtual Machine (VM) Images:** Scanning `.qcow2` , `.vmdk` , and EBS snapshots.
- **Remote Repositories:** Scanning directly via URL (`trivy repo https://github.com/user/repo`).
- **Specific Language Environments:**
  - **Go Specifics:** Scanning `go.mod` , `go.sum` , and handling "vendoring."
  - **Python/Node/Rust:** Differences in how Trivy handles different lockfiles.
- **SBOM Deep Dive:**
  - **Attestation:** Using `cosign` with Trivy to verify image signatures and SBOMs.
  - **Format Conversion:** Converting between CycloneDX and SPDX.

🟧 **3. Configuration & Operational Modes**

- **The Configuration File:** Using `trivy.yaml` instead of long CLI flags for reproducible scans.
- **Environment Variables:** Setting `TRIVY_AUTH_URL` , `TRIVY_USERNAME` , etc., for private registries.
- **Client/Server Architecture:**
  - Setting up a **Trivy Server** to act as a centralized DB provider.
  - Securing the connection with **TLS and Custom Tokens**.
- **Remote Private Registries:**
  - Configuring authentication for ECR, GCR, and DockerHub.
  - Using static credentials vs. IAM roles.

🟥 **4. Enterprise Cloud & K8s Security**

- **Cloud Provider Scanning:**
  - **AWS:** Scanning account-wide for misconfigured S3 buckets, IAM roles, and Lambda functions.
  - **Azure/GCP:** (Currently in preview/active development) Scanning cloud resources for drift.
- **Kubernetes Specifics:**
  - **trivy k8s cluster:** Detailed report of the entire cluster state.

- **Built-in Policies:** Understanding the automated checks for Pod Security Standards (Privileged, Root access).
- **The Operator:** * Resource overhead management.
    - Integrating Trivy reports with the K8s API via CRDs.

🟪 5. Compliance, Policy as Code & VEX

This is where professional security engineers spend most of their time.

- **Compliance Frameworks:**
    - Running scans specifically for **CIS Benchmarks**, **NSA**, or **PCI-DSS**.
    - Generating compliance-specific reports using `--report summary`.
- **Custom Policy (Rego):**
    - Writing custom `.rego` files to enforce organization-specific rules.
    - Using `trivy conf` to validate local config files against custom policies.
- **VEX (Vulnerability Exploitability eXchange):**
    - Understanding the **VEX Statement**.
    - How to "vouch" for a vulnerability (e.g., "Yes, the CVE is there, but the function is never called").

🟧 6. Extensibility & Development (The GSoC Path)

Since you know Go, this is your most important section.

- **Trivy Plugins:**
    - Installing community plugins (`trivy plugin install ...`).
    - **Building a plugin in Go:** Using the Trivy SDK to add custom CLI functionality.
- **Custom Templates:** Writing **Go Templates** to output security data into custom dashboards or Slack-compatible JSON.
- **Contributing to Trivy:**
    - Setting up the local dev environment (`make build`).
    - Understanding the **Misconfiguration (Defsec)** engine.
    - How to add a new "detector" for a new language or framework.

🟪 7. Integration Ecosystem

- **Harbor Registry:** Using Trivy as the default built-in scanner.
- **Prometheus:** Using the `trivy-exporter` to turn vulnerability counts into Grafana metrics.
- **IDE Integration:** Using the Trivy VS Code extension for "Shift-Left" security while coding.

🏁 Summary Table of Scan Commands

Command	Purpose
`trivy image`	Scan container images (OS + Language packages).
`trivy fs`	Scan local filesystem/directories.

Command	Purpose
`trivy config`	Scan IaC files (Terraform, Dockerfile, K8s).
`trivy rootfs`	Scan an unpacked root filesystem.
`trivy vm`	Scan virtual machine images.
`trivy aws`	Scan AWS account for misconfigurations.
`trivy k8s`	Scan live Kubernetes clusters.
`trivy sbom`	Generate or scan an SBOM file.

Would you like me to focus on the "Custom Policy (Rego)" section? Since you're interested in K8s security, learning to write custom rules is a huge level-up.

Trivy has evolved from a simple container scanner into a full-spectrum **Cloud-Native Security Platform**. To truly "cover everything," you need to move past simple image scanning and dive into supply chain security, policy-as-code, and the new 2025 AI integrations.

Here is the exhaustive, rebuilt roadmap from basic to "Architect" level.

🟢 1. The Essentials (Level: Beginner)

- **Scanner Targets:** Mastering the three core targets: `image`, `fs` (filesystem), and `repo`.
- **Severity Management:** Filtering by `--severity` (UNKNOWN, LOW, MEDIUM, HIGH, CRITICAL).
- **Actionable Results:** Using `--ignore-unfixed` (to find what you can actually patch) and `.trivyignore` (to silence false positives).
- **Output Formats:** Converting reports to `json`, `sarif` (for GitHub Security tab), and using `Go Templates`.
- **The DB Lifecycle:** Understanding `trivy-db` and `trivy-java-db`, and how to manage them in air-gapped (offline) environments.

🟡 2. Infrastructure & Configuration (Level: Intermediate)

- **IaC Misconfiguration Scanning:**
  - Scanning Dockerfiles, Terraform, CloudFormation, and Helm Charts.
  - **The Defsec Engine:** Understanding the library that powers Trivy's misconfiguration logic.
- **Secret Detection:** Using the built-in 50+ rules to find leaked API keys, AWS tokens, and SSH private keys.
- **License Compliance:** Scanning for high-risk licenses (GPL, AGPL) that could cause legal issues for your projects.
- **Client/Server Mode:** Deploying a **Trivy Server** to centralize vulnerability data and speed up CI/CD runners.
- **Advanced Container Tech:** Scanning unpacked root filesystems (`rootfs`) and Virtual Machine images (`.qcow2`, EBS snapshots).

🟠 3. Kubernetes & Cloud Security (Level: Advanced)

- **K8s Cluster Scanning:**
  - `trivy k8s cluster`: Generating a full security posture report of a live cluster.
  - **KBOM (Kubernetes Bill of Materials):** Generating a specialized inventory of your cluster's components.
- **The Trivy Operator:**
  - Continuous scanning of K8s resources.
  - **CRDs:** Working with `VulnerabilityReport` and `ConfigAuditReport`.
- **Cloud Infrastructure Scanning:**
  - **AWS Scanning:** Scanning S3, IAM, and EC2 for misconfigurations and drift.
  - **Azure/GCP/Alibaba:** Scanning for cloud-native security best practices.
- **Compliance Frameworks:** Running scans specifically for **CIS Benchmarks**, **NSA/CISA**, and **PCI-DSS**.

🔴 4. Supply Chain & Policy-as-Code (Level: Professional)

- **SBOM (Software Bill of Materials):**
  - Generating and scanning **CycloneDX** and **SPDX** files.
  - **SBOM Attestation:** Using `Cosign` to sign SBOMs and `Rekor` to store them.
- **VEX (Vulnerability Exploitability eXchange):**
  - Using VEX files to document why a vulnerability is *not* exploitable in your app.
  - **VEX Hub:** Integrating with community-driven exploitability data.
- **Custom Policy (Rego):**
  - Writing custom security rules using the **Rego** language.
  - Creating private policy bundles for your organization.
- **Attestation & Verification:** Verifying image signatures and provenance using `Trivy` + `Cosign`.

🟣 5. Extensibility & 2025 Features (Level: Expert)

- **WASM Modules:** Extending Trivy's scanning logic without changing the core Go binary.
- **Trivy Plugins:** Building your own CLI extensions (e.g., custom exporters or specialized detectors).
- **Trivy MCP Server (New):** Integrating Trivy with **AI Agents** (like Cursor or Claude) via the Model Context Protocol to allow AI to "see" and fix your security issues in real-time.
- **Runtime Security (Experimental):** Exploring how Trivy is beginning to integrate with eBPF-based tools for runtime monitoring.
- **SAST (Static Analysis):** Using the newer `--scanners sast` to find coding flaws like SQL injection or Buffer Overflows directly in your source code.

🛠️ Key Commands to Memorize

Category	Command
All-in-One Scan	`trivy fs --scanners vuln,secret,misconfig .`

Category	Command
K8s Summary	`trivy k8s cluster --report summary`
Generate SBOM	`trivy image --format cyclonedx --output sbom.json <image>`
Compliance	`trivy k8s cluster --compliance=nsa`
AWS Scan	`trivy aws --region us-east-1`
VEX Filter	`trivy image --vex repo <image>`

Would you like me to create a "Study Plan" for the next 2 weeks to help you master these topics for your K8s-Genesis project?

Welcome to the first module of your Trivy Masterclass! Since you're a CSE student building projects like **K8s-Genesis**, these notes are designed to take you from a total beginner to a "God Level" understanding of cloud-native security.

🛠️ Installation & Getting Started

Before we dive into the theory, let's get Trivy running on your machine. Trivy is a single binary, making it incredibly lightweight.

1. Installation Guide

Platform	Command
macOS (Homebrew)	`brew install aquasecurity/trivy/trivy`

Linux (Ubuntu/Debian)	`sudo apt-get install wget gnupg lsb-release`
	`wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | gpg --dearmor | sudo tee /usr/share/keyrings/trivy.gpg > /dev/null`
	`echo "deb [signed-by=/usr/share/keyrings/trivy.gpg] https://aquasecurity.github.io/trivy-repo/deb generic main" | sudo tee -a /etc/apt/sources.list.d/trivy.list`
	`sudo apt-get update && sudo apt-get install trivy`

Windows (Chocolatey)	`choco install trivy`
Docker (No Install)	`docker run aquasec/trivy image python:3.4-alpine`

> **Pro-Tip:** If you're on Windows, using the **Docker** method or **WSL2** is recommended for the best experience with Kubernetes tools.

2. Running Your First Scan

Once installed, verify it with `trivy --version`. To run your first scan on a common image:

```bash
trivy image nginx:latest
```

Trivy will download its vulnerability database (approx. 50MB) on the first run. Subsequent scans will be much faster due to local caching.

🟢 Topic 1: Core Fundamentals

In this section, we cover the "Why" and "How" of vulnerability scanning.

1. The "Shift Left" Philosophy

In the old days, security happened at the end of the development cycle. **Shift Left** means moving security to the beginning (the "left" side of the timeline).

- **Developer's machine:** Scan code/images before pushing.
- **CI/CD:** Scan during the build.
- **Registry:** Scan stored images.

2. Key Terms to Know

- **CVE (Common Vulnerabilities and Exposures):** A unique ID (e.g., `CVE-2024-1234`) assigned to a specific security flaw.
- **CVSS (Common Vulnerability Scoring System):** A numerical score (0.0 to 10.0) representing the severity.
- **False Positive:** When a tool reports a vulnerability that isn't actually exploitable in your specific context.

3. Basic Scan Targets

Trivy isn't just for images. It has three primary entry-level "targets":

A. Container Image Scanning

Scans the OS packages (like `apt`, `apk`) and language-specific dependencies (like `npm`, `pip`, `go.mod`).

```bash
# Basic image scan
trivy image alpine:3.15
```

B. Filesystem Scanning

Scans a local folder. This is perfect for checking your source code before you even build an image.

```bash
# Scan current directory
```

```
trivy fs .
```

C. Remote Repository Scanning

Scans a Git repo directly without needing to clone it first.

Bash

```
# Scan a public GitHub repo
trivy repo https://github.com/aquasecurity/trivy
```

📊 Understanding the Report

When you run a scan, Trivy generates a table. Here is how to read it:

- **Library:** The name of the vulnerable package (e.g., `openssl`).
- **Vulnerability ID:** The CVE number.
- **Severity:** CRITICAL, HIGH, MEDIUM, LOW, or UNKNOWN.
- **Installed Version:** What you currently have.
- **Fixed Version:** The version you need to upgrade to in order to fix the flaw.

Must-Know CLI Flags (The "Power User" Kit)

To make your life easier, use these flags:

Bash

```
# 1. Filter only High and Critical (ignore the noise)
trivy image --severity HIGH,CRITICAL nginx

# 2. Only show vulnerabilities that HAVE a fix available
trivy image --ignore-unfixed nginx

# 3. Output to JSON (for automation/scripts)
trivy image --format json --output result.json nginx

# 4. Fail the command (Exit Code 1) if a Critical is found
# (Crucial for CI/CD pipelines)
trivy image --exit-code 1 --severity CRITICAL nginx
```

📝 Module 1 Summary Checklist

1. [ ] Trivy is installed and `trivy --version` works.
2. [ ] You've scanned at least one image (`nginx`, `python`, etc.).
3. [ ] You understand that `trivy fs` scans local code and `trivy image` scans built artifacts.
4. [ ] You can filter results using `--severity`.

Would you like to move on to Topic 2: Intermediate Scanning (IaC, Secrets, and SBOMs) where we'll look at scanning your Terraform and K8s manifests?

How to Install Trivy on Windows, Linux, and macOS This video provides a clear, step-by-step visual guide on setting up Trivy across different platforms and performing your very first scan, which perfectly complements the installation notes provided above.

Install Trivy on Windows, Linux & macOS (2025) | Complete Setup + First Scan Guide

Shubham gour · 204 views

Since you've mastered the basic scans and installation, it's time to move into **Topic 2: Intermediate Scanning**. This is where Trivy starts acting like a true Security Engineer by looking inside your code, your secrets, and your infrastructure setup.

🟡 Topic 2: Infrastructure, Secrets, and SBOMs 🛡️

In this stage, we stop looking at just "packages" and start looking at **Configuration** and **Supply Chain**.

1. IaC (Infrastructure as Code) Misconfiguration Scanning

As a DevOps enthusiast, you'll be writing Dockerfiles, Kubernetes manifests, and Terraform. Even if your software is secure, a misconfigured "infrastructure" can leave the door wide open.

The Command: `trivy config [path]`

Example: Scanning a Dockerfile

Create a file named `Dockerfile` with this (purposely bad) content:

Dockerfile

```
FROM ubuntu:latest
USER root
RUN apt-get update && apt-get install -y telnet
```

Run the scan:

Bash

```
trivy config Dockerfile
```

What Trivy will find:

- **Root user:** Running as root is a security risk (Check ID: `DS002`).
- **Telnet:** Using unencrypted communication tools is a risk.

Example: Scanning Kubernetes Manifests

If you have a K8s YAML for your **K8s-Genesis** project:

Bash

```
trivy config k8s-deployment.yaml
```

Trivy will check against **CIS Benchmarks** and **Pod Security Standards**, flagging things like "Privileged containers" or "Missing resource limits."

---

2. Secret Detection (The "Leaked Password" Hunter) 🔑

One of the biggest causes of hacks is developers accidentally committing API keys or passwords to GitHub. Trivy has a built-in engine to catch these.

The Command: `trivy fs --scanners secret [path]`

- **How it works:** It uses Rego-based rules to look for patterns (Regex) like AWS Access Keys, GitHub Tokens, and Stripe Keys.
- **God-Level Tip:** By default, `trivy fs .` scans for vulnerabilities. If you only want to look for secrets to save time, use the `--scanners` flag.

Bash

```
# Scan only for secrets in the current directory
trivy fs --scanners secret .
```

---

3. SBOM (Software Bill of Materials) 📦

An SBOM is essentially a "Nutrition Label" for your software. It lists every single component, version, and license used in your application. In 2025, providing an SBOM is becoming mandatory for many government and enterprise contracts.

Generating an SBOM

Trivy supports the two industry standards: **CycloneDX** and **SPDX**.

Bash

```
# Generate a CycloneDX SBOM for an image
trivy image --format cyclonedx --output my-app-sbom.json nginx:latest
```

Scanning an existing SBOM

If someone gives you an SBOM file and asks, "Is this secure?", you don't need the actual image. You can scan the file:

Bash

```
trivy sbom my-app-sbom.json
```

---

4. License Scanning ⚖️

Some software licenses (like **GPL-3.0**) can legally force you to make your entire project's source code public if you use them. Trivy helps you find these "Legal Vulnerabilities."

Bash

```
# Scan for licenses in a project
trivy fs --scanners license .
```

⚙️ Advanced Configuration (The `trivy.yaml`)

Typing long commands every time is exhausting. Professionals use a config file. Create a file named `trivy.yaml` in your project root:

YAML

```
scan:
  scanners:
    - vuln
    - secret
    - misconfig
  exit-code: 1          # Fail the build if issues are found
  severity:
    - HIGH
    - CRITICAL
report:
  format: table         # Clean visual output
  output: security-report.txt
```

Now, instead of a long command, you just run:

Bash

```
trivy fs .
```

Trivy will automatically read your `trivy.yaml` and apply all those rules.

🧪 Practical Exercise for You

1. Go to your **Go Port Scanner** project folder.
2. Run `trivy fs --scanners vuln,secret,misconfig .`
3. Look for any "Hardcoded Secrets" (like a test API key you might have left).

4. Generate an SBOM for your project in `json` format.

📊 Summary of Topic 2 Tools

Feature	Target Command	Why use it?
Misconfigurations	`trivy config`	Check Docker/K8s/Terraform security.
Secrets	`trivy fs --scanners secret`	Find leaked API keys/passwords.
SBOM	`trivy image --format cyclonedx`	Create a list of all dependencies.
License	`trivy fs --scanners license`	Avoid legal issues with code usage.

Ready for Topic 3: Kubernetes and Cloud Security? We will dive into scanning live clusters and using the Trivy Operator!

Would you like me to show you how to write a custom `.trivyignore` file before we move on to K8s? This is vital for managing "False Positives."

Now we are entering the territory of a true **DevSecOps Architect**. Since you are building **K8s-Genesis** (a zero-downtime E-commerce platform), this module is the most critical for your project's survival.

🔴 Topic 3: Kubernetes & Cloud Security (The Cluster Level) ☸️

Up until now, we've scanned "static" things (files, images). Now, we will scan **living, running environments.**

1. Live Cluster Scanning

Instead of scanning a single YAML file, Trivy can talk to the Kubernetes API and scan everything currently running in your cluster.

The Command: `trivy k8s [target]`

A. The Full Cluster Report

This command scans every Pod, ConfigMap, and Service in your current context.

Bash

```
trivy k8s cluster --report summary
```

- **Why use `--report summary`?** Without it, Trivy will dump thousands of lines of text. The summary gives you a clean table of which namespaces are "clean" and which are "dirty."

B. Scanning a Specific Namespace

If you only want to scan your `genesis-prod` namespace:

Bash

```
trivy k8s -n genesis-prod all
```

C. Finding "Compliance" Issues

Trivy can check your cluster against official security frameworks like the **NSA (National Security Agency)** or **CIS Benchmarks**.

Bash

```
trivy k8s cluster --compliance=nsa
```

---

2. The Trivy Operator (Continuous Security)

Scanning manually is good, but in a real-world project like **K8s-Genesis**, things change every minute. You need an **Operator**—a pod that lives inside your cluster and scans things automatically.

Why use the Operator?

1. **Continuous Scanning:** It scans new images as soon as they are deployed.
2. **CRDs (Custom Resource Definitions):** It saves security reports as K8s objects. You can view them using `kubectl`.
3. **Prometheus Integration:** It exposes security metrics (e.g., "Number of Critical Vulns") so you can see them on a Grafana dashboard.

Quick Install (via Helm):

Bash

```
helm repo add aquasecurity https://aquasecurity.github.io/helm-charts/
helm repo update
helm install trivy-operator aquasecurity/trivy-operator \
  --namespace trivy-system \
  --create-namespace
```

How to view reports:

After installation, you don't use the `trivy` command anymore. You use `kubectl`:

Bash

```
# See all vulnerability reports in the cluster
kubectl get vulnerabilityreports -A

# See misconfiguration reports for your K8s manifests
kubectl get configauditreports -A
```

---

3. Cloud Infrastructure Scanning (AWS/Azure/GCP)

Trivy can now scan your cloud account setup. For example, it can find if you have an **S3 Bucket** that is accidentally public or an **IAM Role** with too many permissions.

The Command: `trivy aws [flags]`

- **Scan your entire AWS account:**

```bash
export AWS_REGION=us-east-1
trivy aws --region $AWS_REGION
```

- **Scan a specific service (e.g., S3):**

```bash
trivy aws --service s3
```

🛠️ God-Level Feature: The Client/Server Mode

If you have a large cluster or a fast CI/CD pipeline, downloading the 50MB vulnerability database every time is slow and hits GitHub's rate limits.

Solution: Run one Trivy instance as a **Server** and others as **Clients**.

Step 1: Start the Server

```bash
trivy server --listen 0.0.0.0:4954
```

Step 2: Run a Scan from the Client

```bash
trivy image --server http://<server-ip>:4954 nginx:latest
```

- **Benefit:** The client doesn't download the DB. It sends the "fingerprint" of the image to the server, the server checks its local DB, and sends the result back instantly.

📓 Module 3 Summary Checklist

1. [ ] You can run `trivy k8s cluster` to see the health of your local cluster (Minikube/Kind).
2. [ ] You understand that the **Operator** is for *continuous* monitoring, while the **CLI** is for *ad-hoc* checks.
3. [ ] You know how to check for **NSA/CISA compliance**.
4. [ ] You understand why **Client/Server mode** is used in professional DevOps environments.

🏏 Practical Exercise

1. Start a local Kubernetes cluster (Minikube or Kind).
2. Deploy a simple (and vulnerable) app like `ghost:0.11`.
3. Run `trivy k8s cluster --report summary`.

4. Note how many **Critical** vulnerabilities are running in your cluster right now.

Ready for Topic 4: The Advanced Supply Chain (SBOMs, VEX, and Rego Policies)? This is where we learn how to write our OWN security rules.

Would you like me to explain how to set up the **Trivy Exporter** for Grafana so you can visualize these vulnerabilities in a dashboard?

As a 2nd-year CSE student building **K8s-Genesis**, you are now entering the "Architect Phase." This topic is where you move from being a user to being a **Security Developer**. We are going to cover **Policy as Code (Rego)**, **VEX (Vulnerability Exploitability Exchange)**, and **Supply Chain Attestation**.

🟥 Topic 4: Advanced Supply Chain & Policy as Code 🛡️

In this stage, you don't just find vulnerabilities—you define what is "allowed" and "disallowed" in your specific infrastructure.

1. Custom Policy with Rego (Policy as Code)

Trivy uses **Rego** (the language for Open Policy Agent) to define security rules. If Trivy's 100+ built-in checks aren't enough, you can write your own.

Scenario: You want to ensure that *no* container in your K8s-Genesis project uses an image from an untrusted registry (only allow `ghcr.io` or `docker.io`).

Write the Policy (`my_policy.rego`)

Code snippet

```
package user.kubernetes.ID001  # Unique package name

import data.lib.result

# METADATA
# title: Trusted Registry Only
# description: All images must come from our approved internal registry.
# custom:
#   id: ID001
#   severity: CRITICAL
#   input:
#     selector:
#       - type: kubernetes

deny[res] {
    input.kind == "Pod"
    some container in input.spec.containers
    image := container.image
    not startswith(image, "ghcr.io/my-org/")

    msg := sprintf("Untrusted image registry: %v", [image])
    res := result.new(msg, container)
}
```

Run the Scan with your Policy:

Bash

```
# Point Trivy to your custom policy folder
trivy config --policy ./policies --namespaces user .
```

---

2. VEX (Vulnerability Exploitability eXchange) 📉

VEX is the "professional" way to handle **False Positives**. Imagine Trivy finds a "Critical" vulnerability in a library you use, but your app doesn't actually call the vulnerable function. Instead of just "ignoring" it (which is bad practice), you issue a **VEX statement**.

The VEX Hub: In 2024-2025, Trivy introduced **VEX Hub**, which automatically downloads "vouching" statements from software maintainers.

Using VEX to filter noise:

Bash

```
# Automatically filter out vulnerabilities that maintainers have marked as 'not_affected'
trivy image --vex repo nginx:latest
```

---

3. Supply Chain Attestation (Trivy + Cosign) 🔐

How do you prove that the image running in your **K8s-Genesis** production cluster is the *exact* one that passed the Trivy scan? You **Attest** it.

Step 1: Scan and generate a report

Bash

```
trivy image --format cosign-vuln --output scan.json my-app:latest
```

Step 2: Create the Attestation (using Cosign)

Bash

```
# This signs the scan results and attaches them to the image in your registry
cosign attest --predicate scan.json --type vuln my-app:latest
```

Step 3: Verify at Runtime

Now, your Kubernetes cluster can use a tool like **Kyverno** to say: *"If this image doesn't have a Trivy scan attestation signed by the DevOps key, don't let it run."*

---

4. Advanced MLOps Security (Trivy + AI) 🤖

Since you're interested in **MLOps**, you should know that Trivy can scan AI/ML artifacts.

- **Model Weights:** Scanning for "pickles" (malicious code hidden in model files).
- **AI Pipelines:** Scanning Kubeflow or Airflow manifests for misconfigurations.

Bash

```
# Scan a directory containing ML models and notebooks
trivy fs --scanners vuln,secret,misconfig ./ml-project
```

📈 Module 4 Summary Table

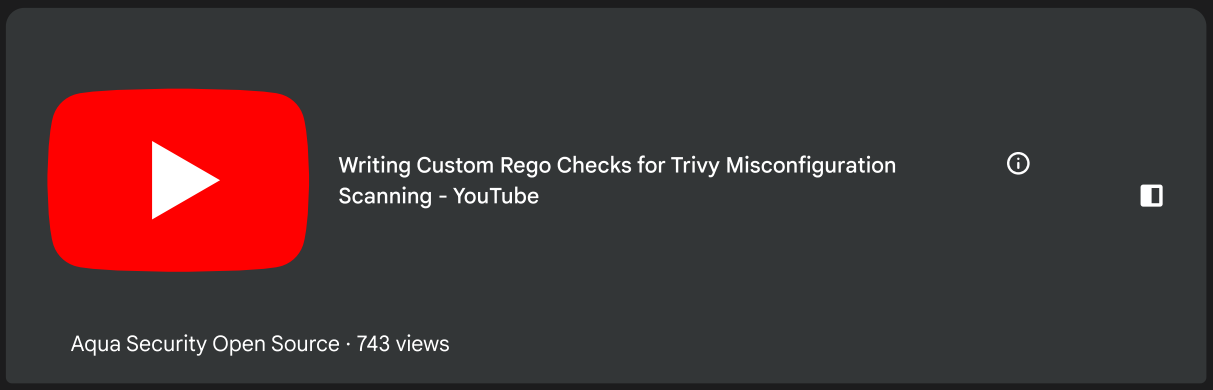
Tool/Feature	Command / Key File	Why you need it
Custom Rego	`--policy ./my-rules`	Enforce your own organization-specific rules.
VEX	`--vex repo`	Stop seeing vulnerabilities that aren't actually exploitable.
Cosign Attest	`cosign attest ...`	Create a cryptographic proof of your security scan.
SARIF Format	`--format sarif`	Export data to GitHub Security Dashboards.

🏁 Practical Exercise

1. Create a simple Rego policy that checks if your **Go Port Scanner** Dockerfile uses `latest` as a tag (which is bad practice).
2. Run `trivy config --policy . Dockerfile` and see it fail.
3. Try running `trivy image --vex repo python:3.9` to see how many vulnerabilities disappear when community VEX data is applied.

Would you like me to move on to Topic 5: CI/CD Automation and GitHub Actions? I can show you how to block a pull request if security checks fail.

Writing Custom Rego Checks for Trivy This video is essential because it walks you through the actual syntax and logic of writing Rego rules specifically for Trivy, which is the "God Level" skill for a DevSecOps engineer.



🟣 Topic 5: CI/CD Automation & AI-Driven Security 🤖

In professional environments, no human runs `trivy` manually. It happens automatically on every code change.

1. GitHub Actions: The Security Gate 🧱

GitHub Actions is the industry standard for CI/CD. We want to set up a "Gate" that blocks a Pull Request if a developer introduces a **Critical** vulnerability.

Example: `security-scan.yml`

Place this in `.github/workflows/` of your Go Port Scanner or K8s project.

YAML

```yaml
name: Security Scan
on:
  push:
    branches: [ main ]
  pull_request:

jobs:
  scan:
    name: Trivy Security Check
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Build local image (Optional)
        run: docker build -t my-app:${{ github.sha }} .

      - name: Run Trivy vulnerability scanner
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: 'my-app:${{ github.sha }}'
          format: 'sarif'
          output: 'trivy-results.sarif'
          severity: 'CRITICAL,HIGH'
          exit-code: '1' # This fails the build if vulnerabilities are found!

      - name: Upload results to GitHub Security Tab
        uses: github/codeql-action/upload-sarif@v3
        if: always() # Upload results even if the scan failed
        with:
          sarif_file: 'trivy-results.sarif'
```

Why this is "God Level":

- **SARIF Format:** This allows GitHub to show the security flaws directly in the "Security" tab of your repo, complete with line numbers and fix suggestions.
- **Exit Code 1:** This is the "Gatekeeper." It prevents insecure code from ever reaching your production cluster.

---

2. The 2025 Edge: Trivy MCP Server (AI Integration) 🧠

You are likely using AI (Gemini, Claude, or Cursor) to help you code. In 2025, Trivy introduced the **Model Context Protocol (MCP)** server. This allows your AI to "see" your security reports and fix them for you automatically.

How it works:

1. You run the **Trivy MCP Server** locally.
2. You connect it to your IDE (like Cursor or VS Code).
3. **The Result:** You can ask Gemini: *"Hey, look at my Trivy scan results and refactor my Go code to fix that SQL injection vulnerability."*

Quick Setup:

Bash

```
# Start the MCP server using the new 'uv' python manager
uvx trivy-mcp
```

Once configured in your IDE, the AI becomes a "Security Partner" that understands your specific scan data.

---

3. Dynamic vs. Static: The eBPF Frontier 🌊

Trivy is a **Static** scanner (it looks at files). But some "obfuscated" malware hides in a way that static scans can't see.

- **Trivy (Static):** Finds known CVEs in your `go.mod` or Docker layers.
- **Tracee/Trivy Runtime (Dynamic):** Uses **eBPF** (extended Berkeley Packet Filter) to watch what a container *actually does* in the kernel.

Example Scenario: A hacker hides a script in a `/tmp` folder inside your image. Trivy might miss it because it looks like a text file. But at runtime, if that script tries to connect to a crypto-mining pool, an eBPF tool (like Aqua's Tracee) will kill the pod instantly.

---

4. Integration Ecosystem (The Pro Tools)

To round out your knowledge, know these integrations:

- **Harbor Registry:** The most popular open-source registry. It has Trivy built-in so it scans images as soon as you `docker push`.
- **Prometheus/Grafana:** Use the `trivy-exporter` to create a dashboard. For your **K8s-Genesis** project, you could have a TV in your room showing a green/red "Security Health" status of your E-commerce platform.

🏛️ The Final Roadmap Summary

Level	Skill Mastered	Key Command
Beginner	Vulnerability Scanning	`trivy image <name>`
Intermediate	IaC & Secret Hunting	`trivy config .`
Advanced	Kubernetes Cluster Health	`trivy k8s cluster`
Architect	Policy-as-Code (Rego)	`trivy config --policy .`
God Level	AI-Driven Automation	`trivy-mcp` + GitHub Actions

🏁 Final Project for You

1. **Repo Setup:** Create a new GitHub repo for your **Go Port Scanner**.
2. **Workflow:** Add the `security-scan.yml` from above.
3. **Trigger:** Push a "Vulnerable" Dockerfile (e.g., `FROM ubuntu:14.04` and `USER root`).
4. **Observe:** Watch the GitHub Action fail and check the **Security -> Code Scanning** tab to see your results.

Congratulations! You have completed the full Trivy Masterclass. Would you like me to help you write that first custom Rego policy for your Go Port Scanner to ensure no one runs it as a root user?

You've covered the fundamentals, infrastructure, and policy-as-code. However, in a professional enterprise setting—or for a high-level **GSoC** project—there are "dark corners" of Trivy that separate the users from the experts.

Here is the **Final Missing Module** to complete your God-Level notes.

❄️ Topic 6: Air-Gapped & Offline Scanning

In high-security environments (banking, government, or isolated K8s clusters), your nodes won't have internet access to download the vulnerability DB.

1. Manual DB Management

You must manually transport the DB into the environment.

1. **On an internet-connected machine:**

```bash
# Use 'oras' or Trivy itself to download the DB bundle
trivy image --download-db-only
tar -czvf trivy-db.tar.gz ~/.cache/trivy/db
```

2. **On the Air-Gapped machine:**

Bash

```
mkdir -p ~/.cache/trivy/db
tar -xzvf trivy-db.tar.gz -C ~/.cache/trivy/db
# Run scan with skip flags
trivy image --skip-db-update --offline-scan my-app:latest
```

2. Private DB Mirroring

For a team, you shouldn't manually copy files. You can host the Trivy DB in your **own private OCI registry** (like Harbor).

Bash

```
# Point Trivy to your internal mirror
trivy image --db-repository my-registry.com/trivy-db:2 my-app:latest
```

🕴️ Topic 7: The Plugin Architecture (Developer Path)

Since you know **Go**, you can extend the Trivy CLI. Plugins allow you to add custom commands like `trivy email-report` or `trivy slack-notify`.

Anatomy of a Plugin

A plugin is just a directory with a `plugin.yaml`.

YAML

```
name: "my-custom-plugin"
version: "0.1.0"
usage: "scans and does something cool"
platforms:
  - selector:
      os: linux
      arch: amd64
    uri: "./bin/my-plugin-binary" # Your Go binary
```

Installation: `trivy plugin install github.com/your-user/trivy-plugin-name`

🔍 Topic 8: Advanced Secret Scanning (Custom Rules)

Trivy's default 50+ secret rules are good, but your company might have a custom token format (e.g., `GENESIS-TOKEN-XXXX`).

Custom Secret Config (`secrets.yaml`)

You can tell Trivy exactly what a "secret" looks like using Regex.

YAML

```
rules:
  - id: genesis-api-key
    name: Genesis Platform API Key
    category: custom
```

```
    regex: 'genesis-[a-z0-9]{32}' # Custom regex for your app
    keywords:
      - "genesis"
```

Run it: `trivy fs --secret-config secrets.yaml .`

🌉 Topic 9: The "Admission Controller" Pattern

For your **K8s-Genesis** project, you don't just want to *know* about vulnerabilities—you want to **block** them at the cluster door.

1. **The Flow:** A dev runs `kubectl apply`.
2. **The Gatekeeper:** The K8s API sends the request to an **Admission Controller** (like Kyverno or OPA).
3. **The Scan:** The controller asks the **Trivy Operator** for the security report.
4. **The Verdict:** If the report shows a CRITICAL vuln, the pod is rejected with an error message sent back to the developer's terminal.

☕ Topic 10: Java & "Fat Jar" Deep-Dive

Java security is notoriously hard because dependencies are often nested inside other dependencies (Transitive Dependencies).

- **Trivy-Java-DB:** Trivy maintains a specialized index for Java because Maven/Gradle metadata isn't always in the JAR.
- **Scan Flag:** Use `--scanners vuln` specifically if you find Trivy is skipping JARs in a complex filesystem.

📋 The Complete "God Level" Command Checklist

Feature	The Ultimate Command
Full Security Suite	`trivy fs --scanners vuln,secret,misconfig,license --severity HIGH,CRITICAL .`
Offline Scan	`trivy image --skip-db-update --offline-scan <image>`
K8s NSA Audit	`trivy k8s cluster --compliance=nsa --report summary`
Custom Registry DB	`trivy image --db-repository <your-registry>/trivy-db <image>`
Go/WASM Extension	`trivy module install github.com/org/trivy-wasm-module`

🚀 Final Next Step

You now have the full theory. The only thing left is to **implement it. Would you like me to guide you through the exact steps to integrate Trivy into your "Heimdall Protocol" project (the self-healing infrastructure using Terraform) so it automatically fixes misconfigurations?**

Since you're aiming for a "God Level" understanding to support your **K8s-Genesis** and **Heimdall Protocol** projects, we need to treat the `trivy.yaml` not just as a settings file, but as your **Security-as-Code engine**.

By default, Trivy looks for a `trivy.yaml` in your current directory. Using this file is the difference between a "script kiddie" running one-off commands and a "DevSecOps Architect" building reproducible security pipelines.

🟢 1. The Basic Level: Standard Clean Config

At the start, you use the YAML to replace long CLI flags. This is perfect for local development on your **Go Port Scanner**.

YAML

```yaml
# trivy.yaml - Basic Starter
quiet: false           # Set to true to hide progress bars in CI
debug: false           # Set to true only for troubleshooting
cache:
  dir: "/tmp/trivy-cache" # Custom cache directory

scan:
  scanners:            # What are we looking for?
    - vuln             # Vulnerabilities in packages
    - secret           # Hardcoded API keys
    - misconfig        # Dockerfile/YAML errors

vulnerability:
  severity:
    - HIGH
    - CRITICAL
  ignore-unfixed: true # Focus ONLY on what we can actually patch
```

🟡 2. The Intermediate Level: CI/CD & Reporting

When you move to **GitHub Actions**, your YAML needs to handle automated failures and output formats that other tools (like SonarQube or GitHub Security) can read.

YAML

```yaml
# trivy.yaml - Intermediate (CI-Optimized)
format: sarif          # Standard format for GitHub Code Scanning
output: "results.sarif" # Save the report to a file
exit-code: 1           # CRITICAL: Fail the CI pipeline if issues are found

# Registry auth (use Env Vars for these in practice)
registry:
  username: ["${DOCKER_USER}"]
  password: ["${DOCKER_PASS}"]

db:
  no-update: false     # Ensure we always have the latest CVE data
  auto-update: true
  repository: "ghcr.io/aquasecurity/trivy-db" # Mirror source
```

🔴 3. The Advanced Level: Infrastructure & K8s

For **K8s-Genesis**, you'll be scanning live clusters and complex IaC. This level includes compliance checks and specialized K8s settings.

YAML

```yaml
# trivy.yaml - Advanced (Infrastructure Focus)
kubernetes:
  context: "my-k8s-context"
  namespace: "genesis-prod" # Scope scan to your project namespace
  report: "summary"         # 'all' for deep dive, 'summary' for overview
  exclude-kinds:            # Don't waste time scanning these
    - "ConfigMap"
    - "Service"
  tolerations:              # If scanning nodes with taints
    - "key1=value1:NoExecute"

compliance:
  spec: "nsa"               # Check against NSA/CISA Kubernetes hardening guide

rego:
  check: ["./custom-policies"] # Path to your OWN Rego security rules
  namespaces: ["user"]         # Only run policies in this namespace
```

🟣 4. The "Architect" Level: Full Feature Reference

This is the exhaustive list of every section you can configure in a `trivy.yaml` in 2025.

Section	Key Feature	Description
`cache`	`backend`	Choose `fs` (filesystem) or `redis` for high-speed shared caching.
`db`	`java-repository`	Specify where to pull the heavy Java/Maven vulnerability index.
`image`	`docker.host`	Connect to a remote Docker daemon (e.g., `unix:///var/run/docker.sock`).
`pkg`	`types`	Filter targets: `os` (Alpine/Ubuntu) or `library` (Go/NPM/Python).
`secret`	`config`	Point to a `trivy-secret.yaml` for custom regex token detection.
`vulnerability`	`vex`	Path to a VEX file to silence non-exploitable vulnerabilities.
`module`	`enable-modules`	List of WASM modules to extend Trivy's logic at runtime.
`clean`	`scan-cache`	Boolean to automatically wipe cache after every scan (saves disk).

🛠️ Pro-Tip: Precedence Rules

Remember the **Precedence Hierarchy** when troubleshooting why a setting isn't working:

1. **CLI Flags** (e.g., `--severity HIGH`) always win.
2. **Environment Variables** (e.g., `TRIVY_SEVERITY=HIGH`) are second.
3. **trivy.yaml** is the base level.

🪄 Practical Exercise: Your "God Mode" Config

Create this file in your **Heimdall Protocol** repo. It combines vulnerability scanning, secret hunting, and custom Terraform policies:

YAML

```yaml
# Final 'God Mode' trivy.yaml
quiet: true
exit-code: 1

scan:
  scanners:
    - vuln
    - secret
    - misconfig
  skip-dirs:
    - "tests"
    - "scripts"

vulnerability:
  severity: ["HIGH", "CRITICAL"]
  ignore-unfixed: true

secret:
  config: "custom-secrets.yaml" # For your Heimdall-specific tokens

rego:
  check: ["./policies/terraform"] # Custom Heimdall security rules
```

Would you like me to generate a `custom-secrets.yaml` for you? Since you're building a port scanner, you'll want to ensure no SSH keys or private certs are accidentally included in your Go binaries.

Running a `trivy.yaml` file is straightforward because Trivy is designed to be "config-aware." You don't "execute" the YAML file like a script; instead, you point the Trivy binary to it, and it adopts those settings for the scan.

There are three ways to run your YAML configuration:

1. The Automatic Way (Zero Config)

Trivy is "lazy" in a good way. If you have a file named `trivy.yaml` in your current working directory, Trivy will **automatically** read it. You don't need to specify any flags.

Bash

```bash
# Just run the scan. Trivy finds the trivy.yaml in this folder.
trivy image nginx:latest
```

---

2. The Explicit Way (`--config` flag)

If your config file has a different name (like `prod-security.yaml`) or is in a different directory, you use the `--config` (or `-c`) flag.

Bash

```
# Pointing to a specific config file
trivy --config ./configs/production.yaml image nginx:latest
```

---

3. The Environment Variable Way

In CI/CD pipelines (like the one you'll use for **K8s-Genesis**), it's often cleaner to set an environment variable. Trivy will look at `TRIVY_CONFIG` to find its settings.

Bash

```
export TRIVY_CONFIG="./trivy.yaml"
trivy fs .
```

⚠️ Precedence Rules: Who Wins?

It is vital to understand what happens if you have a setting in your YAML but type a different one in your terminal. Trivy follows a strict **"Closest to the command"** hierarchy:

1. **CLI Flags:** If you type `--severity CRITICAL` in the terminal, it ignores whatever is in your YAML.
2. **Environment Variables:** These override the YAML file but are overridden by CLI flags.
3. **YAML Config File:** This is your "Base Truth."
4. **Trivy Defaults:** If a setting isn't anywhere else, Trivy uses its built-in defaults.

💡 "God Level" Pro-Tip: The `--generate-default-config`

If you are ever unsure about the exact syntax or keys available in a `trivy.yaml`, you can ask Trivy to generate a massive "cheat sheet" for you:

Bash

```
# This creates a file with every possible setting and its default value
trivy --generate-default-config write trivy-master.yaml
```

You can then open `trivy-master.yaml`, uncomment the sections you need, and delete the rest. This ensures you never have a syntax error.

🪠 Example: Running a Complex Scan

If your `trivy.yaml` contains all your scanners and severity levels, your terminal command stays incredibly clean:

Your `trivy.yaml`:

YAML

```
scan:
  scanners: [vuln, secret]
```

```
vulnerability:
  severity: [HIGH, CRITICAL]
```

Your Command:

Bash

```
# Even though we just typed 'fs .', it will scan for secrets
# and only show High/Critical because of the YAML.
trivy fs .
```

Would you like me to show you how to set up a specific "ignore" file (`.trivyignore`) to work alongside your YAML? This is the best way to handle vulnerabilities you've already reviewed.

Defining Trivy Security Scans through YAML This video is highly relevant as it provides a visual demonstration of how to structure a Trivy configuration manifest and the practical difference between using a YAML file versus a `.trivyignore` file.

Defining Trivy Security Scans through YAML - YouTube

Aqua Security Open Source · 1.2k views