Module 7 – Secure Software Supply Chain with SBOMs

1. Introduction

Modern software systems rely on large ecosystems of third-party packages, libraries, and transitive dependencies. Ensuring security across this software supply chain requires visibility into exactly what is included in an application. A Software Bill of Materials (SBOM) provides this inventory, enabling verification, vulnerability scanning, and lifecycle assurance.

In this lab, I generated SBOMs for the NG911 repository using Syft and Trivy, compared their results, and then performed a vulnerability analysis using Grype. This process demonstrates how SBOMs support security-by-design principles such as open design, complete mediation, and defense in depth.

2. SBOM Generation

2.1 SBOM Generation Process

Inside my GitHub Codespace, I generated two SBOMs for the NG911 repository. The first SBOM was produced using Syft in SPDX JSON format, and the second was generated using Trivy in CycloneDX JSON format. After the generation process, I confirmed that both output files were correctly saved in the deliverables directory.

2.2 Component Counts

To compare the outputs, I examined the number of components identified by each tool. The Syft-generated SPDX SBOM contained 108 components, while the Trivy-generated CycloneDX SBOM included 106 components. This slight difference reflects how each tool analyzes dependencies and interprets metadata within the repository.

Summary of Component Counts

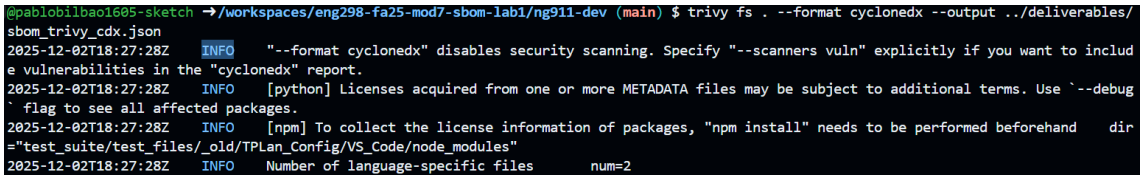
Tool	Format	Component Count
Syft	SPDX JSON	108
Trivy	CycloneDX JSON	106

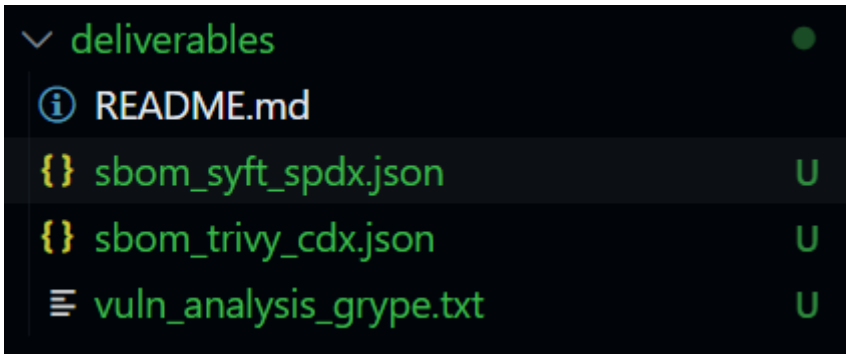
2.3 Differences Between Syft and Trivy

SBOM format differences: Syft outputs SPDX, while Trivy uses CycloneDX. SPDX includes more licensing and metadata fields; CycloneDX focuses more on supply-chain relationships.

Component count differences: Syft includes slightly more components (108) because it detects additional metadata from GitHub Actions workflows, while Trivy emphasizes filesystem-based dependency discovery.

2.4 Screenshots





3. Vulnerability Analysis (Grype)

3.1 Vulnerability Analysis Overview

After generating the SBOMs, I used a vulnerability scanner to analyze the Syft-produced SBOM and identify security issues within the project's dependencies. The tool produced a detailed table of findings, and I reviewed the output to examine the most relevant vulnerabilities. To better understand the results, I inspected the beginning of the report and extracted the five highest-impact entries.

3.2 Top 5 Vulnerabilities Identified

The following table summarizes the five most significant vulnerabilities found during the analysis. Each entry includes its severity, the affected component, installed and fixed versions, and a brief description of the issue:

Vulnerability ID Severity Component Installed Version Fixed In Summary

GHSA-5rjg-fvgr-3xxf High setuptools 72.1.0 78.1.1 A metadata parsing flaw that could potentially allow arbitrary code execution.

GHSA-2qfp-q593-8484 High brotli 1.1.0 1.2.0 Specially crafted input may cause decompression failures or resource exhaustion.

GHSA-768j-98cg-p3fv Medium fonttools 4.57.0 4.60.2 A vulnerability in font parsing that could lead to denial-of-service conditions.

GHSA-9hjg-9r4m-mvj7 Medium requests 2.32.3 2.32.4 Under specific circumstances, header manipulation could allow request injection.

GHSA-pq67-6m6q-mj2v / CVE-2023-45803 Medium urllib3 2.2.2 2.5.0 Improper redirect handling may expose sensitive request-body data to unintended endpoints.

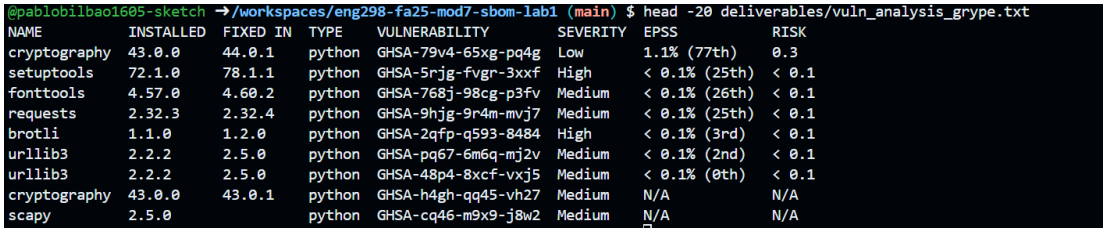
3.3 NVD CVE Summary

As part of the analysis, I selected one of the vulnerabilities—CVE-2023-45803—and looked it up in the National Vulnerability Database (NVD) to understand its impact more thoroughly.

CVE-2023-45803 — One-Sentence Summary

CVE-2023-45803 exposes a risk where sensitive information in an HTTP request body can be unintentionally forwarded to a malicious server during a redirect, due to urllib3 not removing the request body when the method is switched to GET.

3.4 Screenshot of Vulnerability Output



4. Reflection

This lab demonstrated the importance of transparency in the software supply chain. Without an SBOM, it is nearly impossible to understand which third-party components, transitive dependencies, and libraries are included in a system. This lack of visibility undermines basic security engineering principles such as open design and least privilege, since hidden components cannot be reviewed, patched, or evaluated for risk.

By generating SBOMs with two different tools (Syft and Trivy), I saw firsthand that different scanners identify dependencies in different ways. This reinforces the need for defense in depth in assurance processes: no single tool gives a complete picture. Running Grype on the SBOM demonstrated how vulnerabilities propagate through dependency chains and how even common libraries (urllib3, requests, setuptools) introduce risks into critical systems.

Finally, researching CVE-2023-45803 showed how subtle behaviors in HTTP clients can expose sensitive data. The exercise highlights the role of SBOMs in continuous monitoring, verification, and vulnerability management—helping developers detect problems early and maintain system integrity throughout the lifecycle.