

Báo cáo đề tài

DevSecOps: Tích hợp bảo mật vào CI/CD

Sinh viên thực hiện: Nguyễn Xuân Trường

Người hướng dẫn: Lê Trần Bá - Đơn vị: Viettel Software (VTIT)

Mục lục

1	Tổng quát	3
1.1	Vấn đề	3
1.2	Mục tiêu	3
2	Khái niệm sơ bộ	3
2.1	DevSecOps là gì	3
2.2	CI/CD là gì	4
2.3	Các mối đe dọa trong luồng CI/CD	5
3	Kiến trúc và công cụ DevSecOps được sử dụng	6
3.1	Kiến trúc tổng quát – Luồng CI/CD đề xuất	6
3.2	Gới thiệu các công cụ	7
3.2.1	SonarQube – Kiểm tra chất lượng mã nguồn (Static Application Security Testing - SAST) . . .	7
3.2.2	Snyk – Quét lỗ hổng thư viện phụ thuộc (Software Composition Analysis - SCA)	8
3.2.3	Trivy – Quét lỗ hổng container và cấu hình (Container Scanning + IaC Misconfiguration) . . .	8
3.2.4	DefectDojo – Tập trung hoá và quản lý lỗ hổng (Vulnerability Management Platform)	9
4	Mô phỏng và kết quả thực nghiệm	9
4.1	Môi trường thử nghiệm	9
4.2	Chi tiết lỗi phát hiện qua pipeline	10
4.2.1	Snyk – Quét thư viện phụ thuộc (SCA)	10
4.2.2	Trivy – Quét image Docker và cấu hình (Container & Config Scan)	11
4.2.3	SonarQube – Quét mã nguồn (SAST)	11
4.2.4	DefectDojo – Tập trung hoá kết quả và theo dõi	12
5	Tài liệu tham khảo	12

1 Tổng quát

1.1 Vấn đề

Trong bối cảnh phát triển phần mềm hiện đại, các hệ thống phần mềm ngày càng trở nên phức tạp, bao gồm nhiều thành phần như mã nguồn mở, thư viện bên thứ ba, container, hạ tầng triển khai động và quy trình CI/CD liên tục. Những yếu tố này làm tăng đáng kể bề mặt tấn công và nguy cơ rò rỉ, xâm nhập nếu không được kiểm soát tốt từ đầu.

Tuy nhiên, thực tế cho thấy nhiều tổ chức vẫn chưa thực sự tích hợp bảo mật vào xuyên suốt các giai đoạn của vòng đời phát triển phần mềm. Việc kiểm tra bảo mật thường chỉ được thực hiện ở giai đoạn cuối, hoặc sau khi sản phẩm đã được triển khai. Điều này dẫn đến:

- Lỗ hổng bị phát hiện muộn, làm tăng chi phí khắc phục và thời gian phản hồi.
- Dễ bị khai thác trong các tấn công chuỗi cung ứng phần mềm (software supply chain attack).
- Thiếu tính tự động và kiểm soát liên tục trong phát hiện rủi ro.

Câu hỏi cốt lõi đặt ra là: “Làm sao để phát hiện và xử lý sớm các rủi ro bảo mật?”

1.2 Mục tiêu

DevSecOps ra đời như một mô hình tích hợp bảo mật vào mọi giai đoạn của quy trình phát triển và triển khai phần mềm, nhằm trả lời trực tiếp cho câu hỏi trên. Thay vì coi bảo mật là bước cuối, DevSecOps thúc đẩy cách tiếp cận “bảo mật ngay từ đầu” (shift-left security), với các mục tiêu chính:

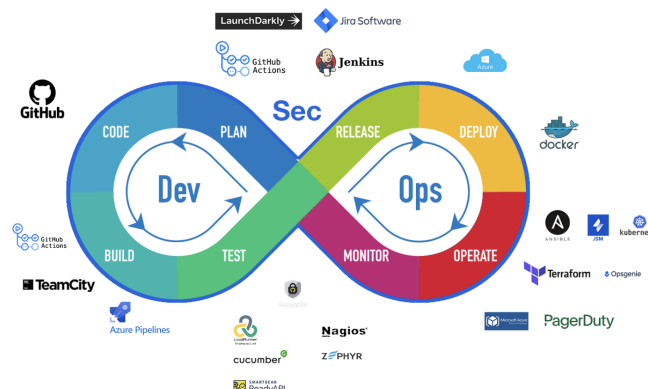
- Tích hợp các công cụ kiểm tra bảo mật tự động (static analysis, dependency scanning, container scanning, etc.) vào pipeline CI/CD.
- Phát hiện sớm các lỗ hổng trong mã nguồn, cấu hình hệ thống, và thành phần phụ thuộc.
- Cung cấp phản hồi nhanh chóng và có thể hành động được cho đội ngũ phát triển.
- Nâng cao nhận thức và trách nhiệm bảo mật trong toàn bộ chuỗi phát triển.
- Giảm thiểu chi phí và rủi ro liên quan đến các sự cố an ninh.

DevSecOps không chỉ là một tập hợp công cụ, mà còn là một văn hoá và cách tiếp cận giúp bảo mật trở thành một phần không thể tách rời của quy trình DevOps hiện đại.

2 Khái niệm sơ bộ

2.1 DevSecOps là gì

DevSecOps (viết tắt của Development – Security – Operations) là sự mở rộng tự nhiên của mô hình DevOps, với mục tiêu đưa bảo mật trở thành một thành phần xuyên suốt và tích hợp trong toàn bộ vòng đời phát triển phần mềm. Thay vì chỉ tập trung vào tốc độ phát triển và triển khai như DevOps truyền thống, DevSecOps thêm vào yếu tố bảo mật, giúp phát hiện và xử lý các rủi ro từ sớm.



So sánh DevOps và DevSecOps

Tiêu chí	DevOps	DevSecOps
Mục tiêu chính	Tăng tốc độ phát triển và triển khai	Tăng tốc kèm đảm bảo an toàn bảo mật
Bảo mật trong quy trình	Tách biệt, xử lý cuối chu kỳ	Tích hợp xuyên suốt pipeline
Vai trò bảo mật	Đội ngũ bảo mật chuyên trách xử lý riêng	Phân tán, chia sẻ trách nhiệm bảo mật giữa Dev, Sec và Ops
Tư duy	“Triển khai nhanh hơn”	“Triển khai nhanh nhưng an toàn hơn”

Table 1: So sánh giữa DevOps và DevSecOps

Khái niệm “Shift Left” trong DevSecOps

Một nguyên lý cốt lõi trong DevSecOps là “Shift Left” – tức là đưa các hoạt động kiểm tra bảo mật từ cuối chu kỳ (hậu kiểm) lên sớm ngay từ giai đoạn đầu (tiền kiểm). Điều này bao gồm:

- Phân tích mã nguồn (Static Analysis) trong giai đoạn viết mã.
- Quét lỗ hổng thư viện phụ thuộc trong giai đoạn tích hợp (dependency scanning).
- Kiểm tra cấu hình container và hạ tầng trong giai đoạn build.
- Áp dụng các chính sách bảo mật tự động trong quá trình CI/CD.

“Shift Left” giúp phát hiện lỗi sớm, giảm chi phí khắc phục và rút ngắn thời gian phản hồi. Đây là điểm khác biệt cốt lõi giúp DevSecOps nâng cao mức độ an toàn mà không làm chậm tốc độ phát triển phần mềm.

2.2 CI/CD là gì

CI/CD (Continuous Integration / Continuous Delivery hoặc Continuous Deployment) là một mô hình quy trình hiện đại giúp tự động hoá việc tích hợp, kiểm thử và triển khai phần mềm. Đây là một trong những trụ cột quan trọng trong các hệ thống DevOps và DevSecOps hiện đại, giúp đẩy nhanh tốc độ phát triển mà vẫn đảm bảo tính nhất quán và ổn định.

CI – Continuous Integration

Continuous Integration (CI) là quá trình trong đó các lập trình viên liên tục đưa các thay đổi mã nguồn của mình lên kho lưu trữ chung (repository), và mỗi thay đổi đều được hệ thống tự động:

- Kiểm thử (unit test, integration test)
- Phân tích chất lượng mã nguồn (code quality, static analysis)
- Kiểm tra định dạng, coding conventions

CI giúp phát hiện lỗi sớm, giảm thiểu xung đột khi tích hợp mã từ nhiều lập trình viên và tạo điều kiện cho việc phát triển theo nhóm quy mô lớn.

CD – Continuous Delivery / Deployment

Continuous Delivery đảm bảo rằng phần mềm luôn trong trạng thái sẵn sàng triển khai ở bất kỳ thời điểm nào. Khi kết hợp với các quy trình triển khai tự động (deployment pipeline), hệ thống có thể chuyển thành mô hình Continuous Deployment – triển khai ngay lập tức vào môi trường production sau khi vượt qua toàn bộ kiểm thử và kiểm tra bảo mật.

Tuy nhiên, trong thực tế, CD cũng có thể chỉ dừng lại ở bước tạo ra bản build (artifact) sẵn sàng triển khai, nhưng vẫn cần phê duyệt thủ công để đưa lên production.

Nguy cơ bảo mật trong CI/CD

CI/CD mang lại hiệu suất và tốc độ cao nhưng cũng tiềm ẩn nhiều rủi ro bảo mật nếu không được kiểm soát tốt:

- Lỗ hổng có thể được đưa vào từ mã nguồn, thư viện phụ thuộc hoặc cấu hình sai.
- Secrets (API key, mật khẩu, token) dễ bị rò rỉ trong quá trình build nếu không được kiểm tra và mã hoá đúng cách.
- Các hình ảnh container hoặc file triển khai có thể chứa lỗ hổng bảo mật chưa được kiểm soát.

Luồng CI được áp dụng trong dự án này

Trong phạm vi dự án hiện tại, hệ thống chỉ triển khai **luồng CI (Continuous Integration)** nhằm mục tiêu phát hiện và xử lý sớm các vấn đề bảo mật trước khi tiến hành triển khai thực tế

2.3 Các mối đe dọa trong luồng CI/CD

Mặc dù CI/CD giúp tăng tốc độ phát triển và triển khai phần mềm, nhưng nếu không được kiểm soát đúng cách, chính pipeline này lại trở thành điểm yếu trong toàn bộ hệ thống. Dưới đây là các mối đe dọa bảo mật phổ biến trong luồng CI/CD, kèm theo ví dụ minh họa:

1. Mã nguồn chứa lỗ hổng bảo mật

Các lỗi trong mã nguồn như *code injection*, sử dụng *API không an toàn*, hoặc thiếu xác thực đầu vào có thể bị khai thác để thực thi mã độc hoặc truy cập trái phép.

Ví dụ:

- Một đoạn mã Python sử dụng `eval(input())` để xử lý dữ liệu đầu vào, dẫn đến khả năng thực thi tùy ý mã độc.
- Giao diện API thiếu kiểm tra phân quyền, cho phép người dùng bình thường truy cập dữ liệu admin.

2. Thư viện phụ thuộc chứa lỗ hổng (CVE)

Việc sử dụng các thư viện bên thứ ba là rất phổ biến. Tuy nhiên, nếu không kiểm soát chặt chẽ, chúng có thể chứa các lỗ hổng bảo mật đã được công bố (CVE – Common Vulnerabilities and Exposures).

Ví dụ:

- Sử dụng `log4j` phiên bản cũ dễ bị tấn công thông qua lỗ hổng `Log4Shell` (CVE-2021-44228).
- Một ứng dụng Node.js phụ thuộc vào thư viện `lodash < 4.17.21` – có nhiều lỗ hổng XSS và prototype pollution.

3. Docker image chứa phần mềm lỗi thời hoặc độc hại

Hình ảnh Docker nếu được tạo từ base image lỗi thời (như Ubuntu 14.04, Alpine cũ, etc.) hoặc chứa phần mềm không được cập nhật, sẽ là mục tiêu dễ bị khai thác.

Ví dụ:

- Base image `python:3.6` đã hết vòng đời và chứa nhiều gói không còn được vá lỗi.
- Hình ảnh chứa gói `curl` phiên bản lỗi có thể bị tấn công qua HTTP response injection.

4. Cấu hình sai trong Dockerfile hoặc GitHub Actions

Cấu hình không đúng trong các tập tin hạ tầng như `Dockerfile`, `docker-compose.yml`, hoặc `.github/workflows/*.yml` có thể dẫn đến các rủi ro nghiêm trọng.

Ví dụ:

- Mở cổng container ra toàn bộ mạng bằng `EXPOSE 80` mà không hạn chế IP.
- Trong GitHub Actions: dùng `run: curl ${url}` mà không kiểm tra nguồn của biến `url`, dẫn đến lệnh shell bị can thiệp.
- Gán quyền `GITHUB_TOKEN` quá cao trong workflow, dẫn đến khả năng ghi vào repository từ job CI.

5. Lộ thông tin nhạy cảm (secrets/key)

Việc hardcode hoặc rò rỉ thông tin như API key, mật khẩu cơ sở dữ liệu, token truy cập... là mối đe dọa nghiêm trọng, đặc biệt nếu chúng bị lộ công khai trên GitHub.

Ví dụ:

- Đưa nhầm file `.env` chứa `DATABASE_PASSWORD`, `AWS_SECRET_ACCESS_KEY` vào commit và bị push lên GitHub.
- Dùng lệnh `echo $SECRET_KEY` trong GitHub Actions mà không che giấu output, khiến key xuất hiện trong log công khai.

Các lỗ hổng này có thể được khai thác không chỉ để tấn công hệ thống mà còn làm bàn đạp xâm nhập sâu hơn vào toàn bộ chuỗi cung ứng phần mềm. Do đó, tích hợp kiểm tra bảo mật tự động trong pipeline CI/CD là điều thiết yếu để phòng ngừa từ sớm.

3 Kiến trúc và công cụ DevSecOps được sử dụng

3.1 Kiến trúc tổng quát – Luồng CI/CD đề xuất

Kiến trúc DevSecOps trong dự án này được triển khai theo hướng tích hợp bảo mật trực tiếp vào luồng CI (Continuous Integration), tuân thủ nguyên tắc **Shift Left**. Điều này có nghĩa là các kiểm tra bảo mật được thực hiện sớm ngay trong giai đoạn phát triển, thay vì dời đến giai đoạn triển khai hoặc production.

Quy trình thực tế:

- Lập trình viên thực hiện chỉnh sửa mã nguồn trên một **feature branch**.
- Sau khi hoàn tất tính năng, developer **push code lên remote** và **tạo Pull Request (PR)** để merge vào nhánh dev.
- Hành động tạo PR này sẽ **kích hoạt pipeline CI** được định nghĩa sẵn trong GitHub Actions.

Luồng CI được thiết kế bao gồm các bước kiểm tra bảo mật sau:

1. Kiểm tra chất lượng mã nguồn với SonarQube (SAST):

- Phát hiện code smell, lỗi logic, và lỗ hổng bảo mật như SQL injection, sử dụng API không an toàn.

2. Quét thư viện phụ thuộc với Snyk (SCA):

- Phân tích các tệp như `requirements.txt`, `pom.xml`, `package.json` để phát hiện CVE từ dependency.

3. Build Docker Image: Tạo container từ source code đã kiểm thử.

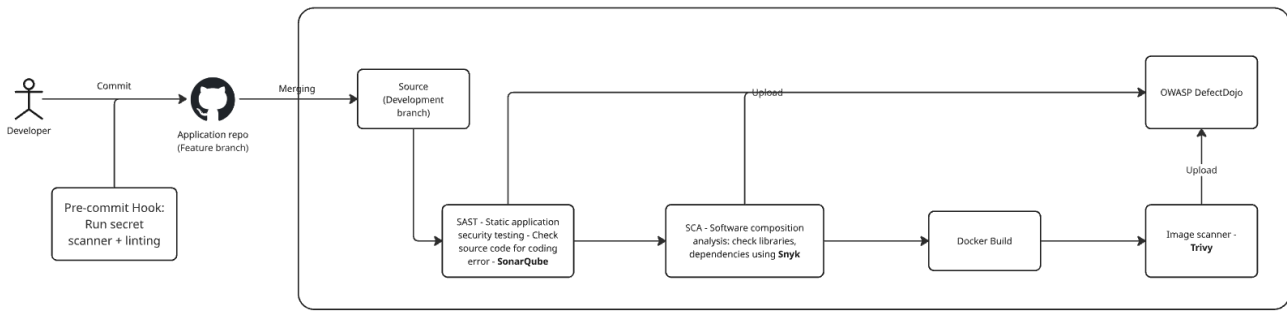
4. Quét bảo mật container và cấu hình với Trivy:

- **Trivy Image:** Kiểm tra CVE trong các gói OS, thư viện hệ thống.
- **Trivy Config:** Kiểm tra các sai sót bảo mật trong Dockerfile, docker-compose.yml.

5. Tập trung hóa kết quả với DefectDojo:

- Tự động import kết quả từ SonarQube, Snyk, Trivy vào DefectDojo thông qua API.
- Cho phép kiểm soát và theo dõi trạng thái lỗ hổng ở một nơi tập trung.

Sau khi pipeline chạy thành công, toàn bộ các bước được đánh dấu xanh lá. Developer có thể theo dõi chi tiết từng job (build, scan, import...) trong giao diện GitHub Actions.



Luồng thực thi CI pipeline từ Pull Request đến các bước kiểm tra bảo mật

Summary

Jobs

- ✓ SAST - SonarQube Testing
- ✓ SCA - Snyk Testing
- ✓ Build and Scan Docker Image wi...
- ✓ Upload Results to DefectDojo

Run details

Usage

Workflow file

Triggered via push 3 days ago

truongng201 pushed → 0c6adcc main

Status: **Success**

Total duration: **1m 15s**

Artifacts: **3**

security.yml

on: push

✓ SAST - SonarQube Testing 41s

✓ SCA - Snyk Testing 32s

✓ Build and Scan Docker Im... 42s

✓ Upload Results to Defect... 24s

Giao diện GitHub Actions thể hiện pipeline DevSecOps chạy thành công

3.2 Giới thiệu các công cụ

3.2.1 SonarQube – Kiểm tra chất lượng mã nguồn (Static Application Security Testing - SAST)

SAST (Static Application Security Testing) là kỹ thuật phân tích mã nguồn tĩnh mà không cần thực thi ứng dụng. Mục tiêu là phát hiện các lỗi lập trình và lỗ hổng bảo mật như: SQL Injection, Command Injection, Hardcoded credentials, hoặc các logic sai sót.

• Thành phần:

- SonarQube Server (web UI)
- PostgreSQL (lưu trữ kết quả)
- Sonar Scanner (thực hiện phân tích mã)

• Cấu hình:

- Tập cấu hình: `sonar-project.properties`
- Có thể tùy chỉnh phạm vi quét qua các biến như `sonar.sources`, `sonar.inclusions`, `sonar.exclusions` để giúp giảm thời gian chạy trên pipeline

• Chức năng nâng cao:

- Thiết lập **Quality Gate** – cho phép tự động chặn merge hoặc push nếu không đạt chuẩn.
- Ví dụ yêu cầu Quality Gate:
 - * `Bug = 0`
 - * `Vulnerabilities = 0` (tránh Critical/High)
 - * `Code Smells: Maintainability Rating = A`
- Cần sử dụng công cụ chuyển đổi (converter) để chuyển định dạng kết quả SonarQube sang JSON/XML tương thích với DefectDojo.

• Ví dụ:

- Dòng code Java sau sẽ bị SonarQube cảnh báo:

```
String query = "SELECT * FROM users WHERE id = " + userInput;
```

Đây là một trường hợp SQL Injection điển hình, vì đầu vào không được xử lý an toàn.

- **Lưu ý:**

- Phiên bản Community không hỗ trợ nhiều rule bảo mật nâng cao.
- Có thể đặt ngưỡng Quality Gate để CI pipeline tự động dừng nếu mã không đạt chuẩn.

3.2.2 Snyk – Quét lỗ hổng thư viện phụ thuộc (Software Composition Analysis - SCA)

SCA (Software Composition Analysis) là kỹ thuật kiểm tra các thành phần bên thứ ba (open-source libraries) trong ứng dụng để phát hiện lỗ hổng được công bố (CVE) và rủi ro cấp phép.

- **Thành phần:**

- Snyk CLI để quét thủ công hoặc tích hợp CI/CD
- Snyk API dùng để nhập dữ liệu vào dashboard hoặc công cụ quản lý lỗ hổng

- **Cấu hình:**

- Câu lệnh: `snyk test -file=requirements.txt, -file=package.json, v.v.`
- Thiết lập token truy cập qua `SNYK_TOKEN`

- **Ví dụ:**

- Một ứng dụng Node.js sử dụng thư viện `lodash@4.17.20`, bị ảnh hưởng bởi lỗ hổng CVE-2021-23337 (Prototype Pollution).
- Snyk sẽ cảnh báo và đề xuất cập nhật lên phiên bản an toàn hơn (`lodash@4.17.21`).

- **Lưu ý:**

- Có thể xảy ra **false positive** – lỗ hổng bị cảnh báo nhưng thực tế không ảnh hưởng.
- Cần kiểm tra lại thủ công trong DefectDojo với trường `Verified = True` để xác thực.
- **Transitive dependency** (phụ thuộc gián tiếp) có thể là nguyên nhân gây lỗi:
 - * `requests==2.21.0` phụ thuộc vào `urllib3==1.24.1`
 - * CVE-2019-11324: Lỗi CRLF Injection trong urllib3 cho phép chèn HTTP header độc hại.
- Cần xác định đúng file và ngôn ngữ của synk để quét hiệu quả.

- **Cách xử lý:**

- Cập nhật thư viện chính (`requests >= 2.22.0`)
- Ép dependency phụ thuộc vào phiên bản an toàn trong `requirements.txt`

3.2.3 Trivy – Quét lỗ hổng container và cấu hình (Container Scanning + IaC Misconfiguration)

Container Scanning là việc kiểm tra hình ảnh container để phát hiện lỗ hổng trong hệ điều hành, gói phần mềm và thư viện được cài đặt trong container.

IaC Misconfiguration Scanning (Infrastructure as Code) là việc quét các tập tin cấu hình như `Dockerfile`, `docker-compose.yml`, `Terraform`, v.v. để tìm lỗi cấu hình bảo mật (ví dụ: quyền quá cao, thiếu xác thực, cổng mở công khai).

- **Thành phần:**

- Trivy CLI
- Trivy Database (CVE database nội bộ)

- **Cấu hình:**

- Quét image: `trivy image myimage:tag`
- Quét cấu hình: `trivy config` .

- Ví dụ:

- Hình ảnh Docker chứa `openssl 1.1.1k` bị phát hiện lỗ hổng `CVE-2021-3711`.
- File `Dockerfile` ghi: `USER root` mà không hạ quyền → Trivy sẽ cảnh báo vì không tuân theo nguyên tắc Least Privilege.

- Lưu ý:

- Nên cập nhật database thường xuyên với `trivy -download-db-only`.
- Có thể tích hợp Trivy dưới dạng GitHub Action hoặc Docker runner.

- Triển khai thực tế:

- Sử dụng dạng binary cài trực tiếp trên máy hoặc trong runner.
- Để tối ưu hiệu suất, job scan Docker image được tách riêng và chạy trên runner phụ chuyên dụng.

3.2.4 DefectDojo – Tập trung hoá và quản lý lỗ hổng (Vulnerability Management Platform)

Vulnerability Management Platform là hệ thống tập trung để lưu trữ, xác minh, phân loại và theo dõi các lỗ hổng bảo mật được phát hiện bởi nhiều công cụ trong quy trình DevSecOps.

- Thành phần:

- Product (ứng dụng hoặc hệ thống)
- Engagement (đợt kiểm tra cụ thể)
- Test Type (SonarQube, Snyk, Trivy,...)

- Chức năng chính:

- Quản lý tập trung các findings, theo dõi trạng thái, mức độ nghiêm trọng.
- Hợp nhất báo cáo từ các công cụ khác nhau.
- Hỗ trợ đánh dấu `verified`, `false positive`, phân công xử lý và sinh báo cáo.

- Ví dụ:

- Import kết quả Trivy (JSON) vào Engagement → Gán severity và assign cho developer xử lý.
- Kết hợp kết quả từ SonarQube và Snyk để thấy lỗ hổng từ cả mã nguồn và thư viện.

- Lưu ý:

- Cần sử dụng định dạng JSON/XML đúng chuẩn tương ứng với parser.
- Có thể tích hợp CI để tự động đẩy kết quả lên DefectDojo thông qua API.

4 Mô phỏng và kết quả thực nghiệm

4.1 Môi trường thử nghiệm

Trong phần thực nghiệm, nhóm sử dụng một **project Python mẫu** có cấu trúc đơn giản để kiểm tra hiệu quả của DevSecOps pipeline. Các thành phần trong môi trường được thiết lập như sau:

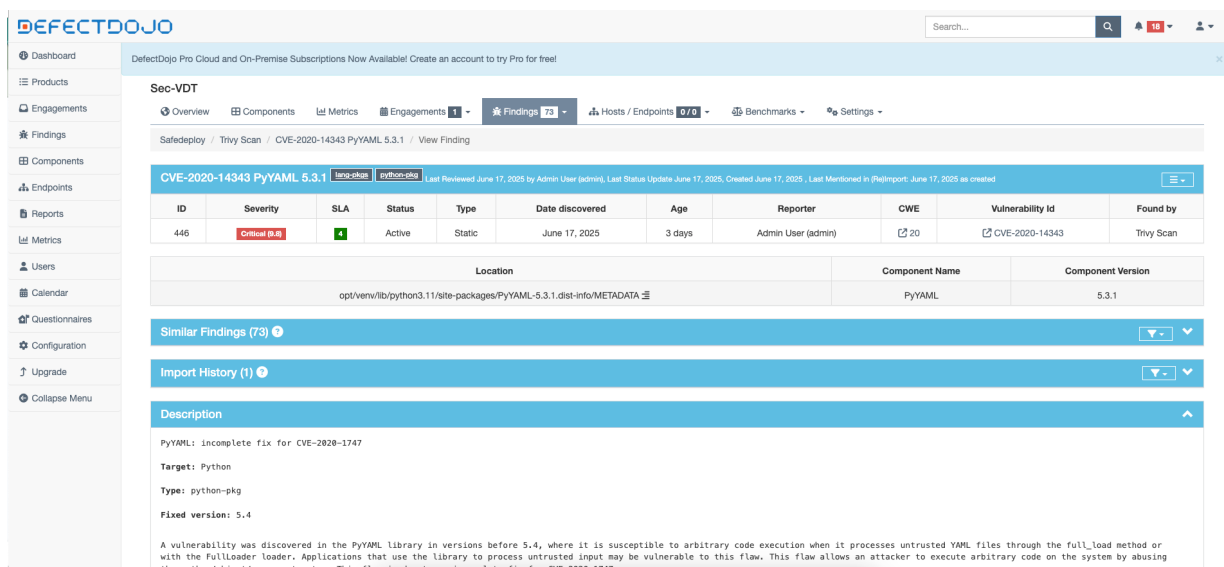
- **CI/CD Pipeline:** triển khai trên nền tảng **GitHub Actions**, được kích hoạt tự động khi tạo Pull Request.
- **SonarQube** và **DefectDojo** được triển khai cục bộ bằng `docker-compose`, thuận tiện cho việc tích hợp và theo dõi kết quả.
- **Các công cụ phân tích** bao gồm:
 - **SonarQube** – kiểm tra mã nguồn tĩnh (SAST).
 - **Snyk** – kiểm tra thư viện phụ thuộc (SCA).
 - **Trivy** – kiểm tra hình ảnh Docker và cấu hình (Container & IaC scanning).
 - **DefectDojo** – quản lý lỗ hổng tập trung.

4.2 Chi tiết lỗi phát hiện qua pipeline

4.2.1 Snyk – Quét thư viện phụ thuộc (SCA)

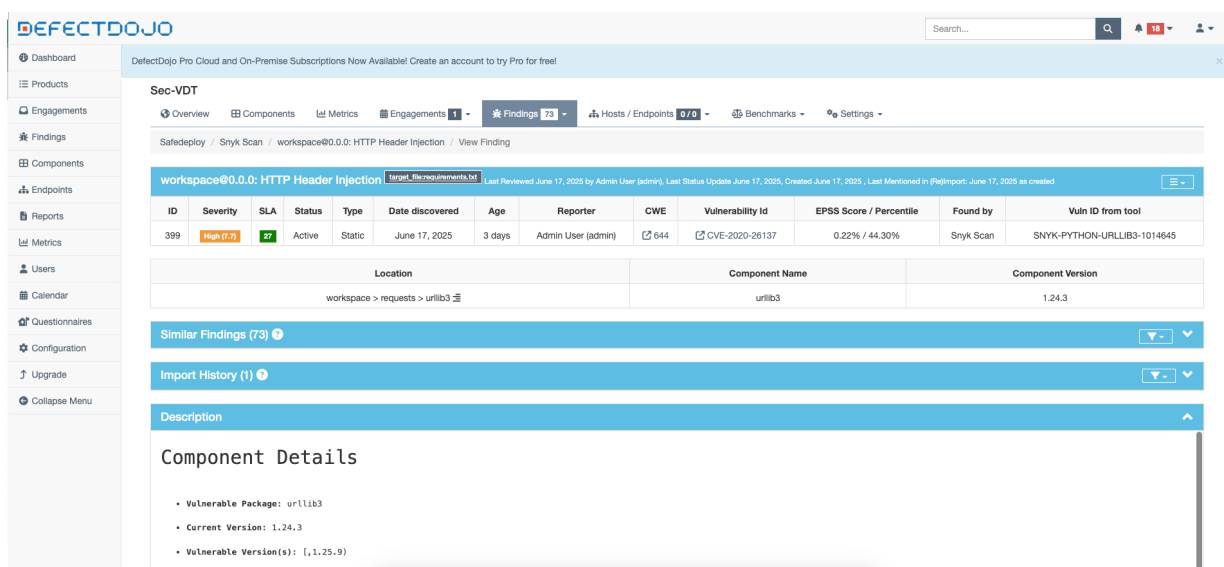
Snyk CLI được cấu hình để quét file `requirements.txt` trong project Python, phát hiện các lỗ hổng trong dependency trực tiếp và gián tiếp:

- `pyyaml==5.3.1` – Lỗ hổng trực tiếp (Direct CVE):
 - **CVE-2020-1747**: Lỗ hổng trong trình phân tích YAML cho phép thực thi mã tùy ý nếu xử lý dữ liệu không đáng tin cậy.
 - **Mức độ**: Critical
- `requests==2.21.0` – Lỗ hổng gián tiếp (Transitive):
 - Phụ thuộc vào `urllib3==1.24.1`, bị ảnh hưởng bởi **CVE-2019-11324** (HTTP header injection).
 - **Mức độ**: High



Giao diện

kết quả Snyk phát hiện lỗ hổng từ `pyyaml`



Giao diện

kết quả Snyk phát hiện lỗ hổng từ `request`

4.2.2 Trivy – Quét image Docker và cấu hình (Container & Config Scan)

Dockerfile sử dụng:

```
FROM python:3.11-alpine3.17
```

Trivy thực hiện phân tích image base và các gói hệ điều hành trong môi trường Alpine, phát hiện một số lỗ hổng bảo mật nghiêm trọng (CVEs) liên quan đến các thành phần hệ thống:

- **CVE-2023-25193** – ảnh hưởng đến `libexpat`, được sử dụng bởi nhiều thư viện phân tích XML.
 - *Mô tả:* Lỗi tràn số nguyên khi xử lý ký tự trong XML dẫn đến crash hoặc RCE.
 - *Mức độ:* High (CVSS 7.5)
- **CVE-2022-42898** – ảnh hưởng đến `krb5` (Kerberos).
 - *Mô tả:* Lỗi xử lý GSS message token có thể bị lợi dụng để vượt quyền trong quá trình xác thực.
 - *Mức độ:* Medium-High tùy theo context sử dụng.

Các lỗ hổng trên tồn tại trong image gốc `python:3.11-alpine3.17`, ngay cả khi chưa cài thêm gói nào từ ứng dụng, cho thấy tầm quan trọng của việc thường xuyên cập nhật và quét bảo mật định kỳ đối với base image.

Một vài lỗi

Trivy phân tích image `python:3.11-alpine3.17`

4.2.3 SonarQube – Quét mã nguồn (SAST)

SonarQube được cấu hình để quét toàn bộ thư mục mã nguồn Python. Một số lỗi được phát hiện bao gồm:

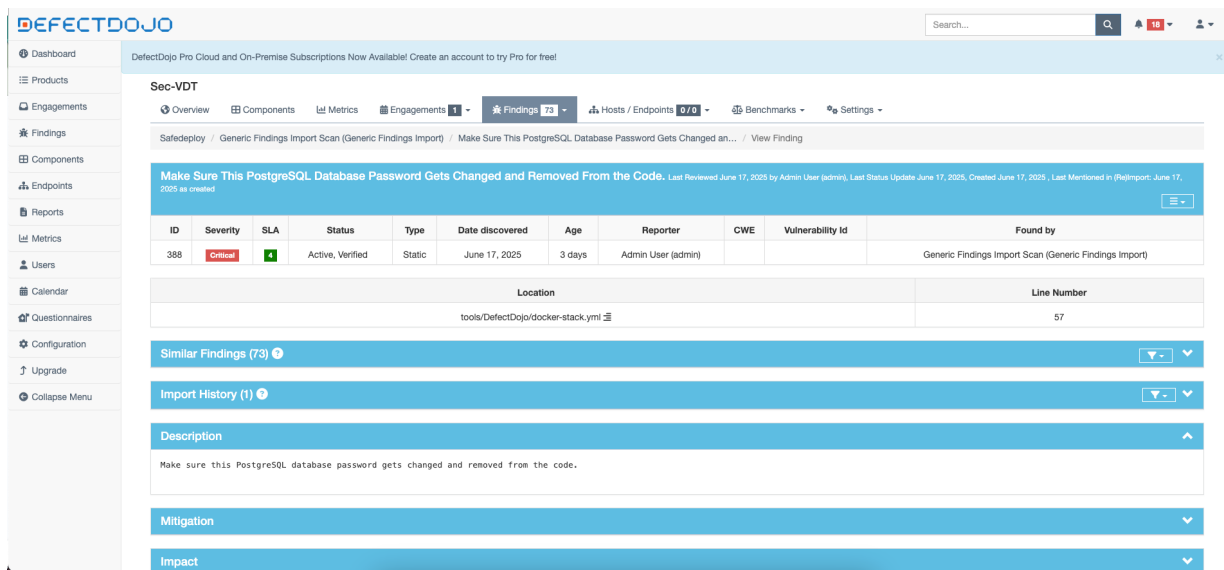
- **Hardcoded secret:**

```
DD_CREDENTIAL_AES_256_KEY = "91a*agLqesc*0DJ+2*bAbsUZfR*4nLw"
```

→ SonarQube cảnh báo đây là **thông tin nhạy cảm bị hardcoded**, vi phạm best practice bảo mật (should be stored in env variables or secret manager).

- **Code smell:**

- Hàm vượt quá 150 dòng, không tách thành các khối hợp lý.
- Biến đặt tên không rõ nghĩa (ví dụ: `x1`, `x2`, `tmp`).



The screenshot shows the DefectDojo interface. The left sidebar contains navigation links: Dashboard, Products, Engagements, Findings, Components, Endpoints, Reports, Metrics, Users, Calendar, Questionnaires, Configuration, Upgrade, and Collapse Menu. The main content area is titled 'Sec-VDT' and shows a finding with ID 388, Severity Critical, and Status Active, Verified. The finding description is 'Make sure this PostgreSQL database password gets changed and removed from the code.' The finding was discovered on June 17, 2025, by Admin User (admin). The finding is located in the file 'tools/DefectDojo/docker-stack.yml' at line 57. The interface also shows a table of similar findings and an import history.

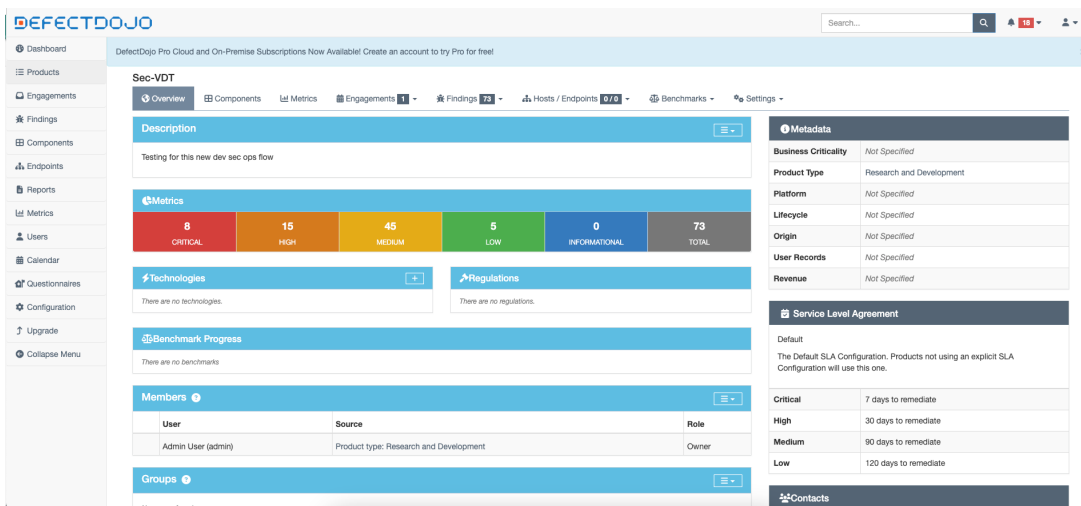
SonarQube cảnh báo hardcoded secret

Giao diện

4.2.4 DefectDojo – Tập trung hoá kết quả và theo dõi

Toàn bộ kết quả từ SonarQube, Snyk, Trivy được tích hợp qua API và import vào hệ thống **DefectDojo**, nơi cho phép theo dõi:

- Tổng số lỗi hỏng: **73 findings**
- Phân loại mức độ: **8 Critical, 15 High, 45 Medium, 5 Low**



The screenshot shows the DefectDojo interface with the Metrics section selected. The Metrics section displays a bar chart showing the distribution of findings by severity: 8 Critical, 15 High, 45 Medium, 5 Low, 0 Informational, and 73 Total. The interface also shows a table of findings with columns for ID, Severity, SLA, Status, Type, Date discovered, Age, Reporter, CWE, Vulnerability Id, and Found by. The table shows one finding with ID 388, Severity Critical, and Status Active, Verified. The finding was discovered on June 17, 2025, by Admin User (admin). The finding is located in the file 'tools/DefectDojo/docker-stack.yml' at line 57. The interface also shows a table of similar findings and an import history.

DefectDojo tập trung hoá kết quả quét

Bảng điều khiển

5 Tài liệu tham khảo

- SonarQube Documentation: <https://docs.sonarqube.org>
- Snyk CLI Documentation: <https://docs.snyk.io>
- Trivy Scanner by Aqua Security: <https://aquasecurity.github.io/trivy>
- OWASP DevSecOps Project: <https://owasp.org/www-project-devsecops>
- DefectDojo Documentation: <https://defectdojo.github.io>