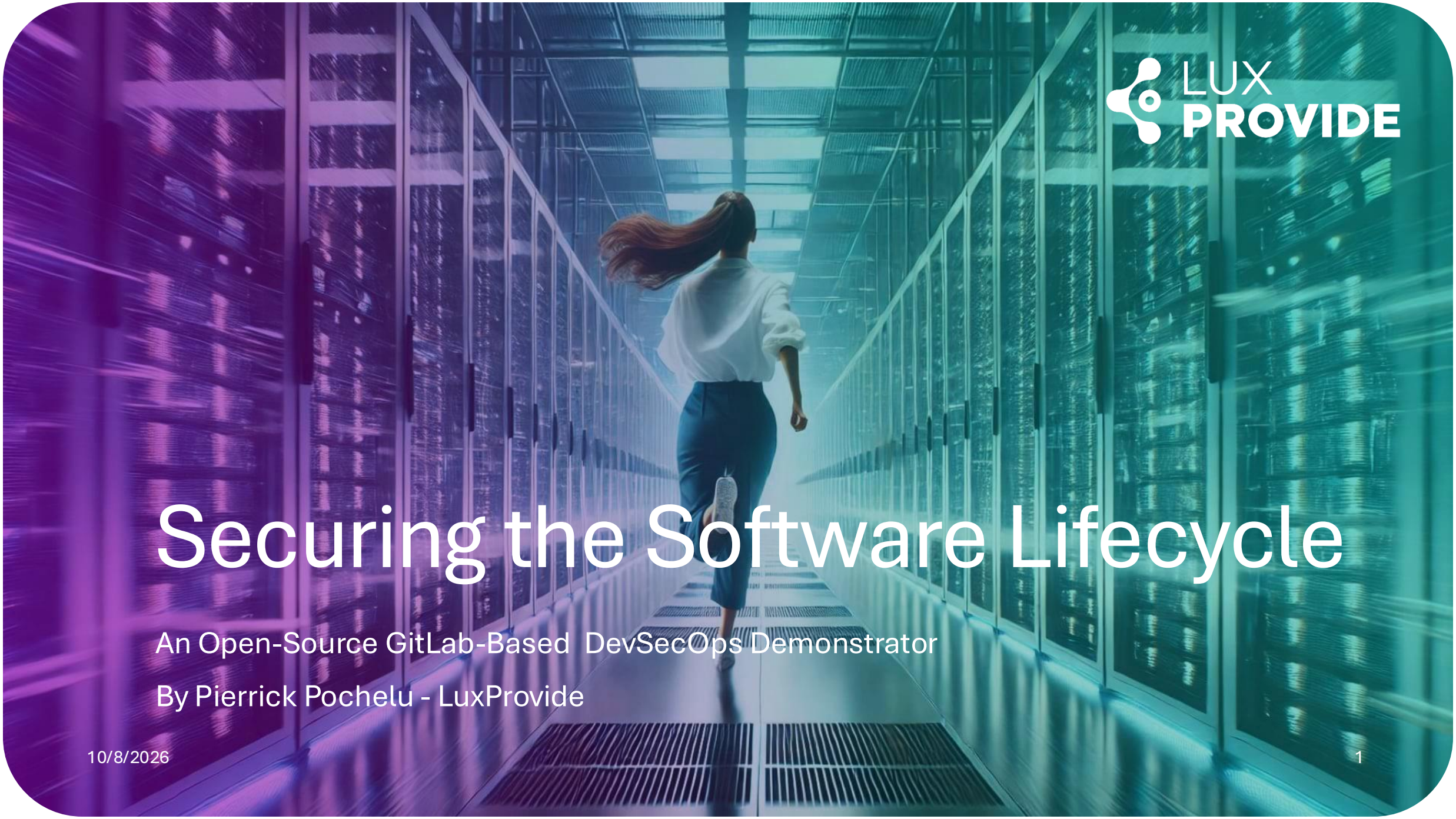


Securing the Software Lifecycle

An Open-Source GitLab-Based DevSecOps Demonstrator

By Pierrick Pochelu - LuxProvide

Summary

- Context
- Demonstrator presentation
- Coverage of Software Security Principles

Context of this work

SEANERGYS Project

- *Software for Efficient and Energy-Aware Supercomputers*
- Goal: Improve the efficiency and sustainability of HPC systems
- More information: https://seanergys.eu

Project Characteristics

- 16 partner organizations
- 6 countries
- 90+ GitLab contributors
- Technologies: Python, C/C++, Shell, container, ...

Challenges

- Consistent security practices across distributed teams
- DevSecOps tools scattered across multiple ecosystems
- Many proposed tools are obsolete, poorly maintained, or commercial

Proposed Solution

- GitLab-based DevSecOps demonstrator
- Aggregation of open-source security controls within CI/CD pipelines
- Automated security gates
- Validation of the security controls themselves

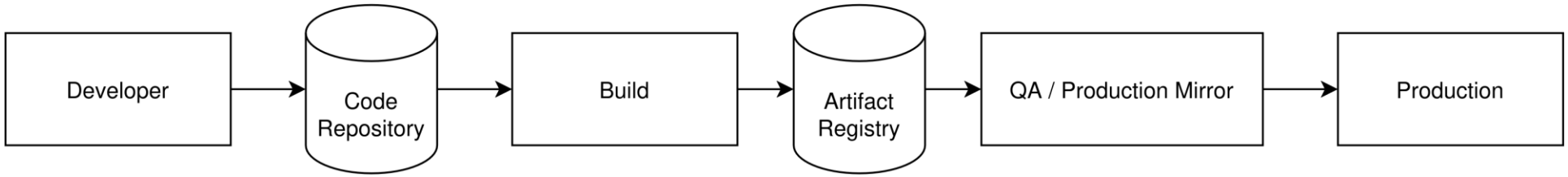
Disclaimer

- Illustrative and educational demonstrator
- Focus on concepts rather than promoting specific tools

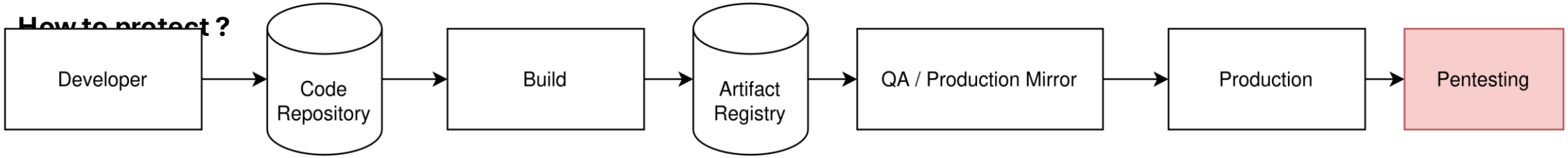
SEANERGYS
ENERGY EFFICIENT EXASCALE

From DevOps to Dev**Sec**Ops

Traditional DevOps focus on developing pipeline to software development lifecycle:



How to protect ?



Problem:

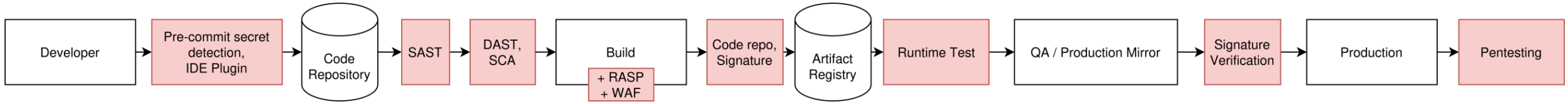
- Too late. Vulnerabity realsed to public !
- Slow feedback from pentesting to developers
- Non-informative: which step failed ?

Solution:

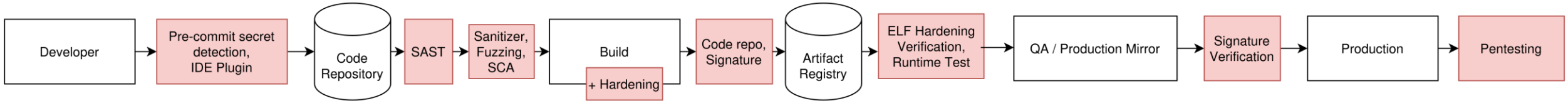
- "Shift Left": Quicker response to the developer
- Apply security controls at every stage

What is DevSecOps automatic pipeline ?

- **Example of a Web server secured pipeline:**



- **Example of a C program secured pipeline:**

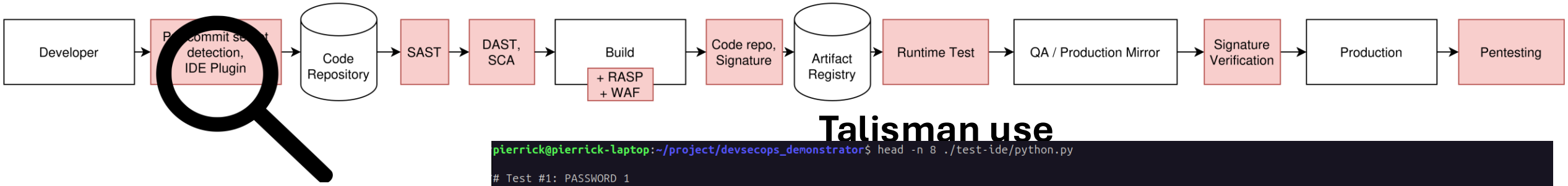


Four main stages:

- **Development phase security**
- **Build phase (Static, Dynamic, Composition) + Hardening practices**
- **Sign/Verify exchange with Repositories**
- **Production**

Demonstrator Presentation

Pre-commit secret detection



Installation of Talisman

2 lines installation:

- bash -c "$(curl --silent https://raw.githubusercontent.com/thoughtworks/talisman/main/install.sh)"
- echo "talisman -g pre-commit" >> .git/hooks/pre-commit

Talisman use

```
pierrick@pierrick-laptop:~/project/devsecops_demonstrator$ head -n 8 ./test-ide/python.py

# Test #1: PASSWORD 1

OPEN_API_KEY = "sk-prod-abcd1234xyz"

# Test #2: PASSWORD 2

PASSWORD = "sk-prod-abcd1234xyz"
pierrick@pierrick-laptop:~/project/devsecops_demonstrator$ git add ./test-ide/python.py
pierrick@pierrick-laptop:~/project/devsecops_demonstrator$ git commit -m "test talisman"
Talisman Scan: 21 / 21 <------------------------------------------------------------------------------------------------------------------------------------------------------> 100.00%

Talisman Report:
+--------------------+--------------------------------+----------+
|        FILE        |             ERRORS             | SEVERITY |
+--------------------+--------------------------------+----------+
| test-ide/python.py | Potential secret               | low      |
|                    | pattern : OPEN_API_KEY =       |          |
|                    | "sk-prod-abcd1234xyz"          |          |
+--------------------+--------------------------------+----------+
| test-ide/python.py | Potential secret               | low      |
|                    | pattern : PASSWORD =           |          |
|                    | "sk-prod-abcd1234xyz"          |          |
+--------------------+--------------------------------+----------+


If you are absolutely sure that you want to ignore the above files from talisman detectors, consider pasting the following format in .talismanrc file in the proje
ct root

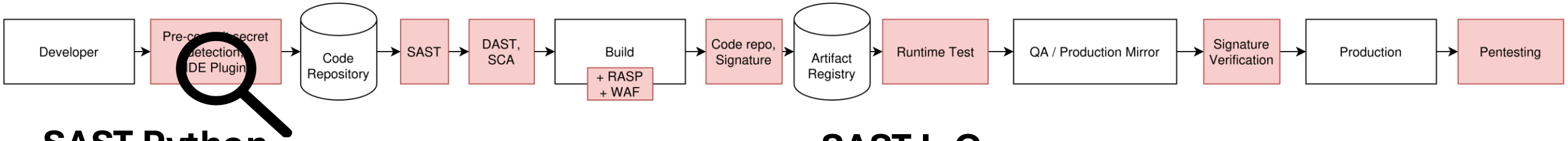
fileignoreconfig:
- filename: test-ide/python.py
  checksum: b87edc3a189c5629d538c709f30d0be8e97247db98e369062ec505291f0dc7df
version: "1.0"

Talisman done in 16.822735ms
pierrick@pierrick-laptop:~/project/devsecops_demonstrator$ git log
commit 87d1dce7ae1b5b6873c15cbb6f37b3ee88d52e20 (HEAD -> main, origin/main, origin/HEAD)
Author: Pierrick Pochelu <pierrick.pochelu@lxp.lu>
Date:   Sun Oct 4 19:03:35 2026 +0200

    pipeline optimization

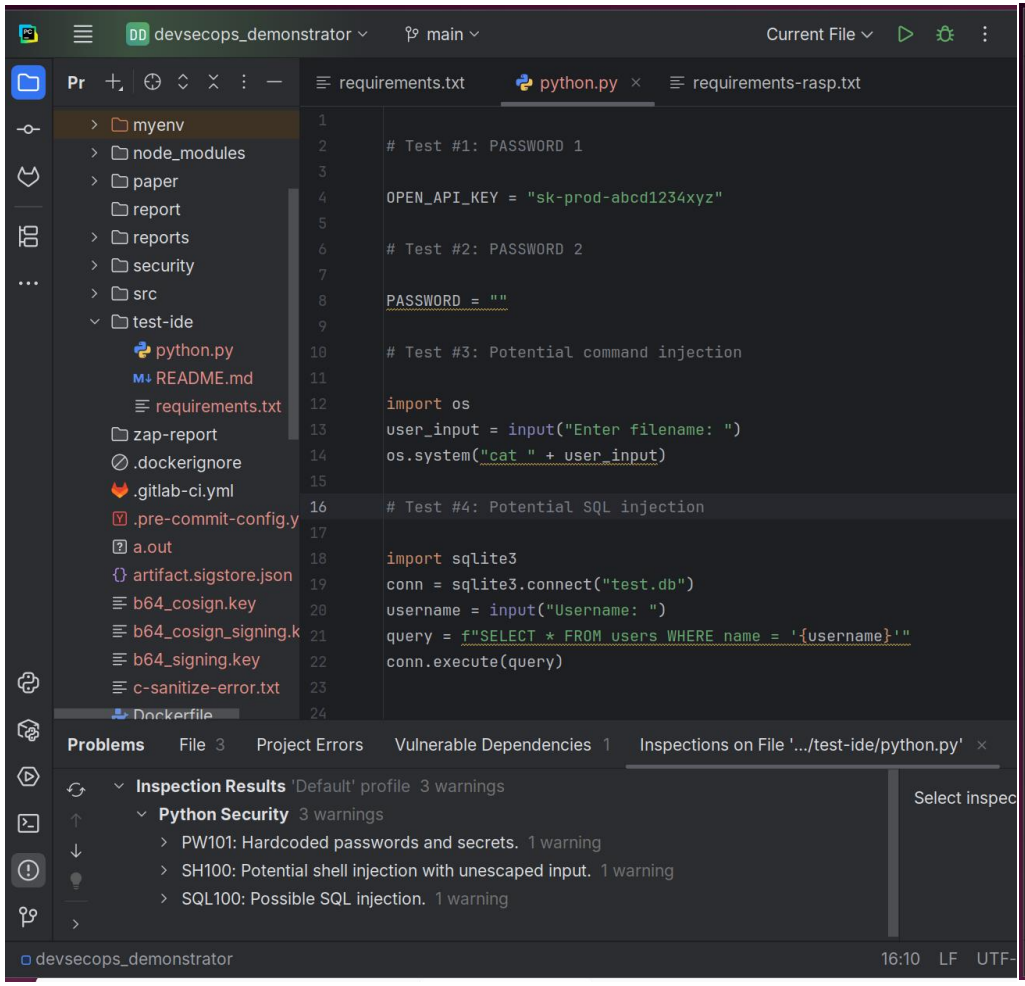
commit 1655e54f0df143e1f09ead55beb2c14e74a7d0c9 (tag: v1.0.0)
Author: Pierrick Pochelu <pierrick.pochelu@lxp.lu>
Date:   Sun Sep 20 17:33:03 2026 +0000
```

Python IDE



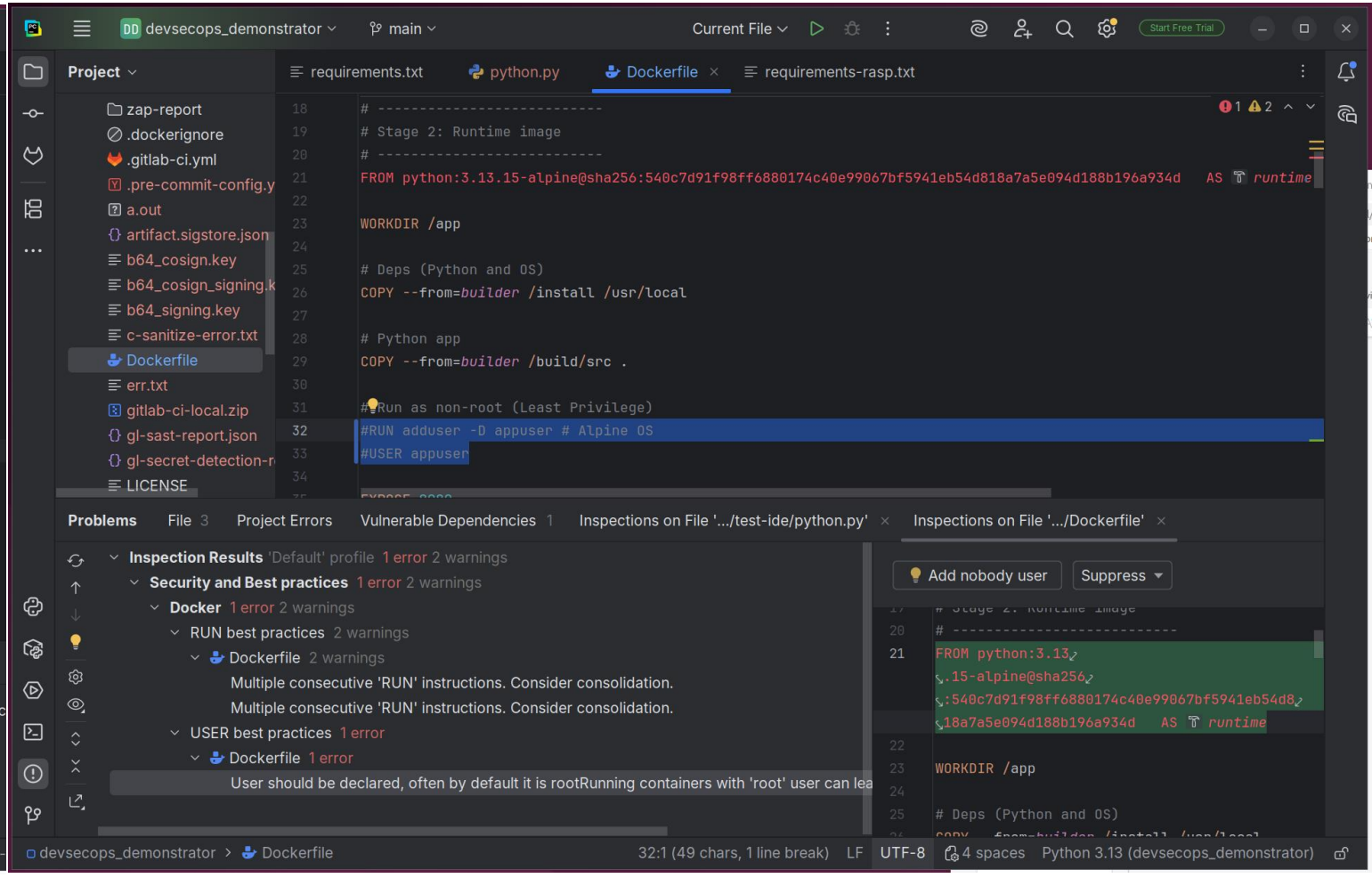
SAST Python

Python Security plugin on PyCharm

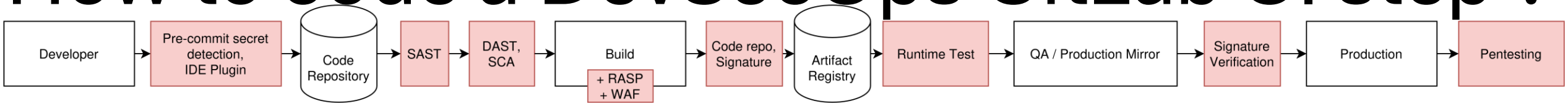


SAST IaC

Cloud IaC plugin on PyCharm



How to code a DevSecOps GitLab CI step ?



GitLab

CI/CD Integration

- Automatable in pipelines
- Tuning false positive vs false negative
- Open-source preferred when selecting security tools

Standardized Results

- Output format (ex: JSON, XML)
- Vulnerability reference (ex: CVE, CWE, CVSS)

Performance

- Fast developer feedback
- Conditional execution (`rules`)
- Parallel execution
- Tuning Performance vs. Scanning time

Supply Chain Security

- Minimal trusted images
  - Distroless when appropriate
- Version and image pinning
- Signature and provenance verification (Artifact download)

How to code a DevSecOps GitLab CI step ?

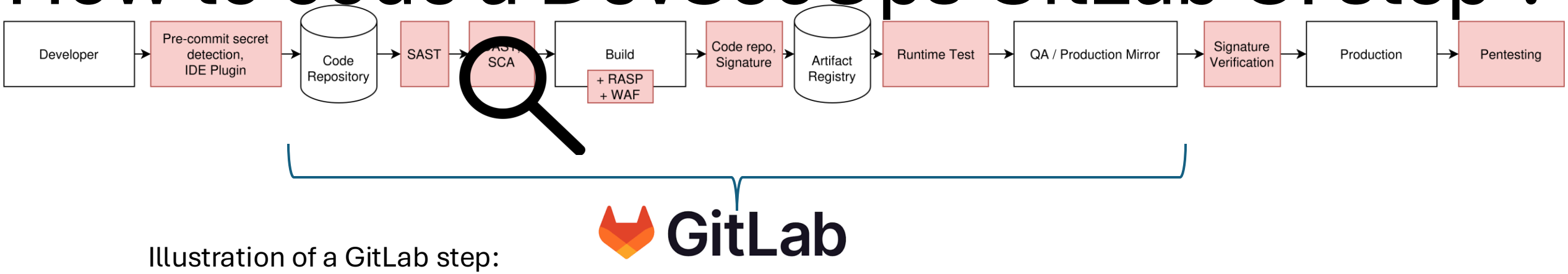
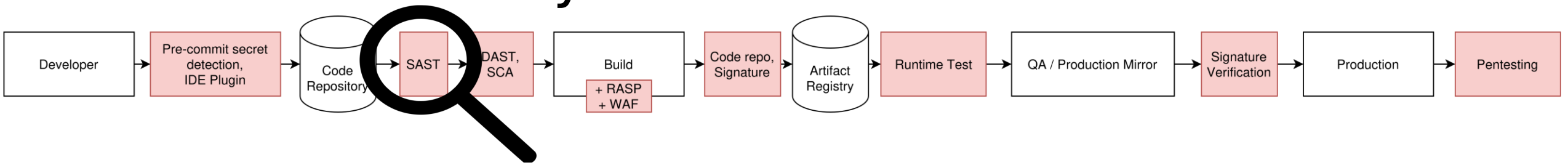


Illustration of a GitLab step:

```yaml
container_scanning:
  stage: py-test-build
  image:
    name: aquasec/trivy:0.74@sha256:62b1e65e8869bc4b4c6aa4fa2b21595256c7c2f6018a9d9ad61caf87187c1969
    entrypoint: [""]
    # override the image's default ENTRYPOINT with an empty one. So start a shell.
  rules:
  - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
  - changes:
    - "**/*.py"
    - "requirements.txt"
    - ".gitlab-ci.yml"
    - "Dockerfile"
  script:
    - trivy image --input myapp.tar --format json -o trivy-report.json
  artifacts:
    paths:
      - trivy-report.json
    expire_in: "7 days"
  needs:
    - py-build
```

Static Code Analysis



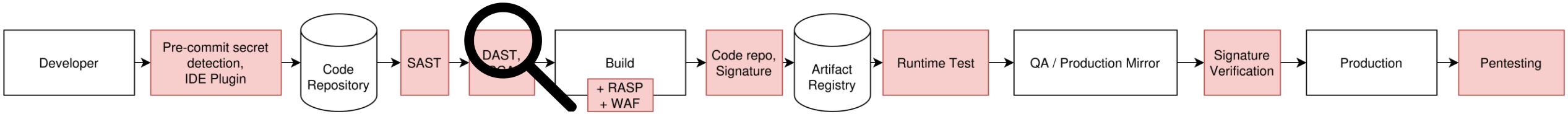
- Semgrep: standard program

(C/C++, Python, ...)

- Kics: Infrastructure-as-Code

(Dockerfile, Ansible, .gitlab-ci.yml, ...)

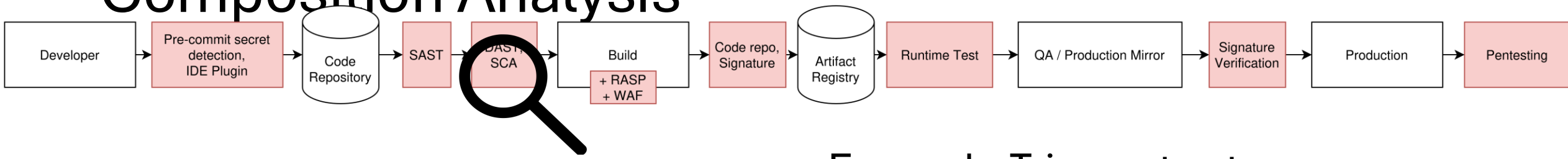
Dynamic Code Analysis



- Web-Application
  - OWASP ZAP (Black-box) attempts common injection inputs:
    - SQLi
    - XSS
    - CSRF
    - ...

- C/C++:
  - Tools:
    - Sanitization with GCC flags + Custom input test (Gray-box)
    - AFL++ (Gray-box). Generate inputs automatically
  - Detects:
    - Memory issues (E.g., Stack & Heap overflow, integer overflow, user-after-free, double free, ...)
    - Crashes
    - ...

Composition Analysis



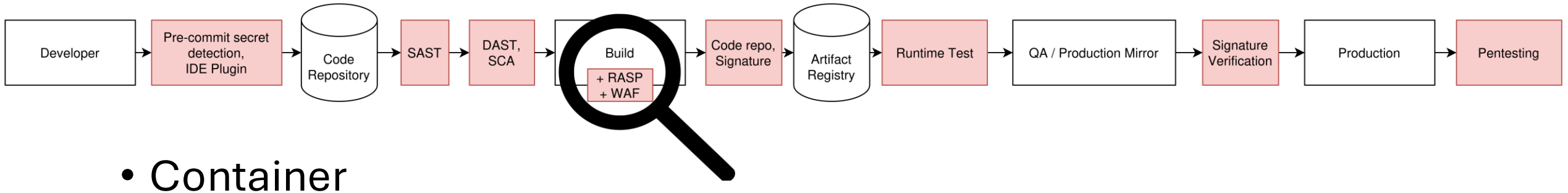
SCA

- Step 1 – Software Bill of Materials (SBOM) discovery
- Step 2 – Vulnerability matching (CVE lookup)

Example Trivy output

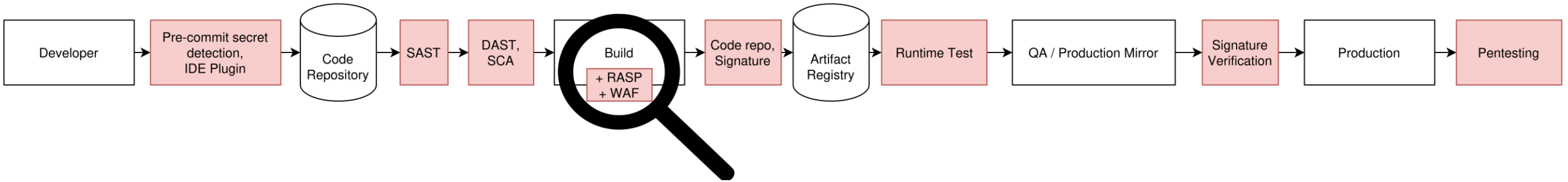
Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
jaraco.context (METADATA)	CVE-2026-23949	HIGH	fixed	5.3.0	6.1.0	jaraco.context: jaraco.context: Path traversal via malicious tar archives https://avd.aquasec.com/nvd/cve-2026-23949
msgpack	GHSA-6v7p-g79w-8964			1.1.2	1.2.1	MessagePack for Python: Out-of-bounds read / crash on Unpacker reuse after a... https://github.com/advisories/GHSA-6v7p-g79w-8964
setuptools (METADATA)	CVE-2026-59890	MEDIUM		80.9.0	83.0.0	setuptools: setuptools: MANIFEST.in exclusion bypass in sdist via Unicode normalization collision (NFC/NFD)... https://avd.aquasec.com/nvd/cve-2026-59890
setuptools	CVE-2025-47273	HIGH		70.3.0	78.1.1	setuptools: Path Traversal Vulnerability in setuptools PackageIndex https://avd.aquasec.com/nvd/cve-2025-47273
	CVE-2026-59890	MEDIUM			83.0.0	setuptools: setuptools: MANIFEST.in exclusion bypass in sdist via Unicode normalization collision (NFC/NFD)... https://avd.aquasec.com/nvd/cve-2026-59890
urllib3	CVE-2026-97687	HIGH		2.7.0	2.8.0	urllib3: urllib3: Traffic interception via HTTPS proxy TLS configuration override https://avd.aquasec.com/nvd/cve-2026-97687
	CVE-2026-97689					urllib3: urllib3: Denial of Service via unbounded memory allocation in chunk parser... https://avd.aquasec.com/nvd/cve-2026-97689
	CVE-2026-97688	MEDIUM				urllib3: urllib3: Denial of Service via infinite loop during chunked Deflate decoding... https://avd.aquasec.com/nvd/cve-2026-97688
wheel (METADATA)	CVE-2026-24049	HIGH		0.45.1	0.46.2	wheel: wheel: Privilege Escalation or Arbitrary Code Execution via malicious wheel file... https://avd.aquasec.com/nvd/cve-2026-24049

Hardening Build - Web App Protection Technics



- Container
  - Container hardening practices
    - Docker user permission (non-root)
    - Port white-list opening
    - Minimalist base image
  - Input protection:
    - Web Application Firewall (ex: nginx+ModSecurity+OWASP CRS):
      - Protection against common web attack (ex: OWASP Top 10)
    - RASP (ex: PyRasp):
      - Protection against common web attack (ex: OWASP Top 10)
      - LLM Protection (prompt injection, adversial inputs, ...)

Hardening Build – Web App Protection Validation



Test suite from "MITRE CWE Top 25 Most Dangerous Software Weaknesses":

- Legitimate use (*False Positive detection*)
- Cross-Site Scripting (XSS), CWE-79, ranked 1st.
- SQL Injection, CWE-89, ranked 2nd.
- Cross-Site Request Forgery (CSRF), CWE-352, ranked 3rd.
- Path Traversal, CWE-22, ranked 6th.
- Server-Side Request Forgery (SSRF), CWE-918, ranked 22nd.

```
# curl options:
# -s silent progress bar
# -o output redirect of the query response
# -w write out  HTTP header response variable

# Legitimate request must succeed
- URL="http://127.0.0.1:8088/?msg=hello"
- STATUS=$(curl -s -o /dev/null -w "%{http_code}" $URL)
- test "$STATUS" = "200" || { echo "Legitimate use should return HTTP 200"; exit 1; }

# XSS request must be rejected
- URL="http://127.0.0.1:8088/?msg=<script>alert(1)</script>"
- STATUS=$(curl -s -o /dev/null -w "%{http_code}" $URL)
- if [[ ! "$STATUS" =~ ^4 ]]; then echo "Expected WAF rejection, got $STATUS"; exit 1; fi

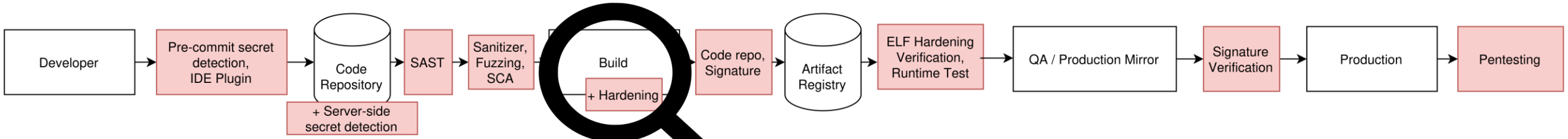
# SQL injection request must be rejected
- URL="http://127.0.0.1:8088/?msg=1%20UNION%20SELECT%201"
- STATUS=$(curl -s -o /dev/null -w "%{http_code}" $URL)
- if [[ ! "$STATUS" =~ ^4 ]]; then echo "Expected WAF rejection, got $STATUS"; exit 1; fi

# CSRF HTTP request must be rejected
- URL="http://127.0.0.1:8088/change-email"
- ATTRURL="http://evil.example.com"
- ATTMAIL="attacker@example.com"
- |
  STATUS=$(curl -s -o /dev/null -w "%{http_code}" -X POST "${URL}" \
  -H "Origin: $ATTRURL" -H "Cookie: session=test" -d "email=$ATTMAIL")
- if [[ ! "$STATUS" =~ ^4 ]]; then echo "Expected CSRF protection rejection, got $STATUS"; exit 1; fi

# SSRF
- URL="http://127.0.0.1:8088/fetch?url=http://169.254.169.254/latest/meta-data/"
- STATUS=$(curl -s -o /dev/null -w "%{http_code}" "$URL")
- if [[ ! "$STATUS" =~ ^4 ]]; then echo "Expected WAF rejection, got $STATUS"; exit 1; fi

# Path Traversal
- URL="http://127.0.0.1:8088/?file=../../etc/passwd"
- STATUS=$(curl -s -o /dev/null -w "%{http_code}" "$URL")
- if [[ ! "$STATUS" =~ ^4 ]]; then echo "Expected WAF rejection, got $STATUS"; exit 1; fi
```

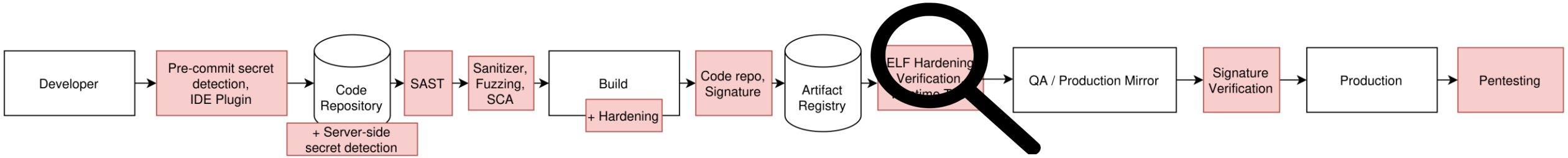
Hardening Build – ELF Binary Protection




```yaml
c-build:
  stage: c-build
  image: ubuntu:24.04@sha256:1e0a86e57d247923[illegible]75e0aaf48a1449cf8c543d51fb3e07a4a7d7bfa79316
  # info: https://github.com/ossf/wg-best-practi[illegible]-os-developers/blob/main/docs/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C%2B%2B.md
  before_script:
    - apt update
    - apt install -y gcc libc6-dev
  script:
    - ls src/
    - |
      gcc src/main.c \
      -O2 \
      -D_FORTIFY_SOURCE=3 \
      -fstack-protector-all \
      -fPIE \
      -pie \
      -fcf-protection=full \
      -Wl,-z,relro \
      -Wl,-z,now \
      -Wl,-z,noexecstack \
      -o secure
  artifacts:
    paths:
      - secure
    expire_in: "7 days"
  needs:
    - c-sast
#D_FORTIFY_SOURCE=3: compile-time and runtime checks for unsafe libc functions: obvious overflows at compile time, inserts runtime bounds checks, abort program when overflow detected. https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.html#-D_FORTIFY_SOURCE=3
#fstack-protector-strong: Add Stack Canary. It mitigates any stack smashing: Stack buffer overflows, Return-address overwrite attacks
#fPIE: Generate Position Independent Executable code (at compiler time). Support "Address Space Layout Randomization (ASLR)".
#pie: Link executable as a PIE binary. Linking randomization.
#fcf-protection=full: Intel CET (Control-flow Enforcement Technology). Protects against: Return-Oriented Programming, Jump-Oriented Programming.
#Wl,-z,relro: Protects against: Relocation Read-Only (RELRO). (linker time). Relocation Read-Only.
#Wl,-z,now: Program start. Prevents: dynamic-linker overwrite attacks.
#Wl,-z,noexecstack: Mark the stack as non-executable. (run time). Protect shellcode injection attacks.
```

"fortify"
"protection"
"enforcement"

Hardening Build – ELF Binary Protection Validation



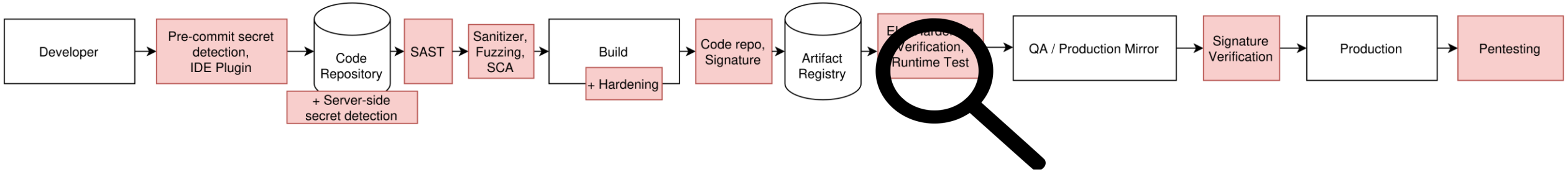
ELF Binary flag check:

```yaml
c-elf-check:
  stage: c-test-build
  image: ubuntu:24.04@sha256:1e0a86e57d247923571b75e0aaf48a1449cf8c543d51fb3e07a4a7d7bfa79316

  before_script:
    - apt-get update
    - apt-get install -y checksec binutils

  script:
    - readelf -d ./secure | grep -q BIND_NOW || { echo "RELRO missing"; exit 1; }
    - readelf -l ./secure | grep -q GNU_RELRO || { echo "RELRO missing"; exit 1; }
    - readelf -W -l ./secure | grep -q "GNU_STACK.*RW " || { echo "NX missing"; exit 1; }
    - readelf -s ./secure | grep -q __stack_chk_fail || { echo "Canary missing"; exit 1; }
    - readelf -s ./secure | grep -q __sprintf_chk || { echo "FORTIFY missing"; exit 1; }
    - readelf -s ./secure | grep -q __strcpy_chk || { echo "FORTIFY missing"; exit 1; }
    - readelf -h ./secure | grep -q "DYN" || { echo "PIE missing"; exit 1; }
    - readelf -n ./secure | grep -q "IBT" || { echo "IBT missing"; exit 1; }
  needs:
    - c-build
```

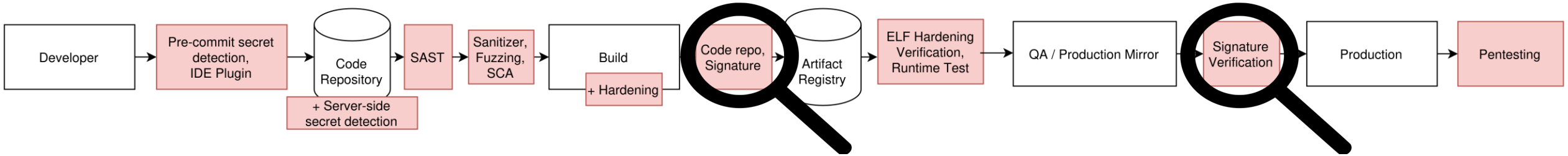
Hardening Build – ELF Binary Protection Validation



Test suite from "MITRE CWE Top 25 Most Dangerous Software Weaknesses":

- Legitimate use (*False Positive detection*)
- Use-After-Free, CWE-416, ranked 7th.
- Out-of-Bounds Read, CWE-125, ranked 8th.
- Integer Overflow, CWE-190.
- OS Command Injection, CWE-78, ranked 9th.
- Stack-Based Buffer Overflow, CWE-121, ranked 14th.
- Heap-Based Buffer Overflow, CWE-122, ranked 16th.
- Externally Controlled Format String, CWE-134.

Hardening Build – ELF Binary Protection Validation



Tools: Cosign or OpenSSL

Goals:

- Check binary stay consistent from Registry to Production
- Protection against spoofing the registry ("provenance checking")

Coverage of Software Security Principles

OWASP Secure Coding Principles

https://cheatsheetseries.owasp.org/cheatsheets/Secure_Product_Design_Cheat_Sheet.html

- Principle #1 Input validation
  - URL query (WAF+RASP)
- Principle #2 Error handling
- Principle #3 Authentication and Authorization
- Principle #4 Cryptography
  - In transit: signature, at rest: private key are hidden
- Principle #5 Least privilege
  - Minimal open port, adding Linux user in container
- Principle #6 Secure memory management
  - GCC Hardening Flags
- Principle #7 Avoiding hard coded secrets
  - Security
- Principle #8 Security testing
  - GitLab-based demonstrator
- Principle #9 Auditing and reviewing the code
- Principle #10 Keeping up-to-date
  - Automated vulnerability through Trivy, scanning and update rules (exampple: OWASP CRS)

Legend:

🔴 Red: Principles not demonstrated by the demonstrator requires human expertise

NIST Secure Software Development Framework (SSDF)

https://csrc.nist.gov/pubs/sp/800/218/final

SSDF Practice	Demonstrator Example
PO.3 – Implement Supporting Toolchains	Security controls are integrated directly into GitLab CI/CD through automated analysis, testing, artifact signing, signature verification, and runtime validation stages.
PO.4 – Define and Use Criteria for Software Security Checks	CI jobs implement objective pass/fail criteria based on security scan results, policy compliance, and artifact verification outcomes.
PO.5 – Implement and Maintain Secure Environments for Software Development	Development activities, source code, and pipeline configurations are version-controlled and executed within controlled CI/CD environments.
PS.1 – Protect All Forms of Code from Unauthorized Access and Tampering	Authentication mechanisms, cryptographically signed artifacts, protected branches, and protected CI/CD variables help prevent unauthorized modification.
PS.2 – Provide a Mechanism for Verifying Software Release Integrity	Cosign and OpenSSL signing demonstrate cryptographic verification of software artifacts before distribution and deployment.
PS.3 – Archive and Protect Each Software Release	Source code, pipeline definitions, build artifacts, execution logs, and security reports can be retained, audited, and reproduced through GitLab.
PW.1 – Design Software to Meet Security Requirements and Mitigate Security Risk	The demonstrator applies defense-in-depth principles through multiple complementary security controls. Project-specific security requirements can be enforced through CI/CD policy customization.
PW.3 – Verify Third-Party Software Complies with Security Requirements	Dependency scanning and container-image scanning identify known vulnerabilities in third-party libraries, packages, and container images.
PW.4 – Reuse Existing, Well-Secured Software When Feasible Instead of Duplicating Functionality	The demonstrator relies on established open-source security tools, pinned container images, and continuously scanned third-party components.
PW.5 – Create Source Code by Adhering to Secure Coding Practices	Secure coding weaknesses are identified through static analysis, secret detection, compiler-assisted analysis, and runtime sanitizers.
PW.6 – Configure the Compilation, Interpreter, and Build Processes to Improve Executable Security	Compiler and linker hardening mechanisms, including PIE, RELRO, NX, Stack Protector, FORTIFY, and CET, strengthen generated executables. Container hardening techniques further improve deployment security.
PW.7 – Review and/or Analyze Human-Readable Code to Identify Vulnerabilities and Verify Compliance with Security Requirements	Automated analysis produces standardized findings using identifiers such as CWE, CVE, OWASP Top 10, and ZAP Alert IDs, supporting review and remediation activities.
PW.8 – Test Executable Code to Help Identify Vulnerabilities and Verify Compliance with Security Requirements	Dynamic testing includes OWASP ZAP, AddressSanitizer (ASan), UndefinedBehaviorSanitizer (UBSan), AFL++, runtime attack simulations, and validation of runtime protection mechanisms.
RV.1 – Identify and Address Vulnerabilities on an Ongoing Basis	Continuous scanning, testing, and monitoring generate vulnerability reports that support prioritization, remediation, and verification of corrective actions.

Not all SSDF practices can be demonstrated through automation alone. The following practices primarily depend on human decision-making and are therefore outside the scope of this demonstrator:

- PO.1 Define Security Requirements for Software Development
- PO.2 Implement Roles and Responsibilities
- PW.2 Review the Software Design to Verify Compliance with Security Requirements and Risk Information
- PW.9 Configure Software to Have Secure Settings by Default

OWASP Software Assurance Maturity Model (SAMM) principles

https://owasp.org/www-project-samm/

Five business functions:

- Governance
- Design
- Implementation
- Verification
- Operation

SAMM Business Function	Demonstrator Example
Implementation	Secure coding and build practices are enforced through automated CI/CD security controls
Verification	Automated testing verifies both application vulnerabilities and the effectiveness of implemented security controls.
Operations	Runtime protections such as WAF and RASP help detect and block attacks during operation.

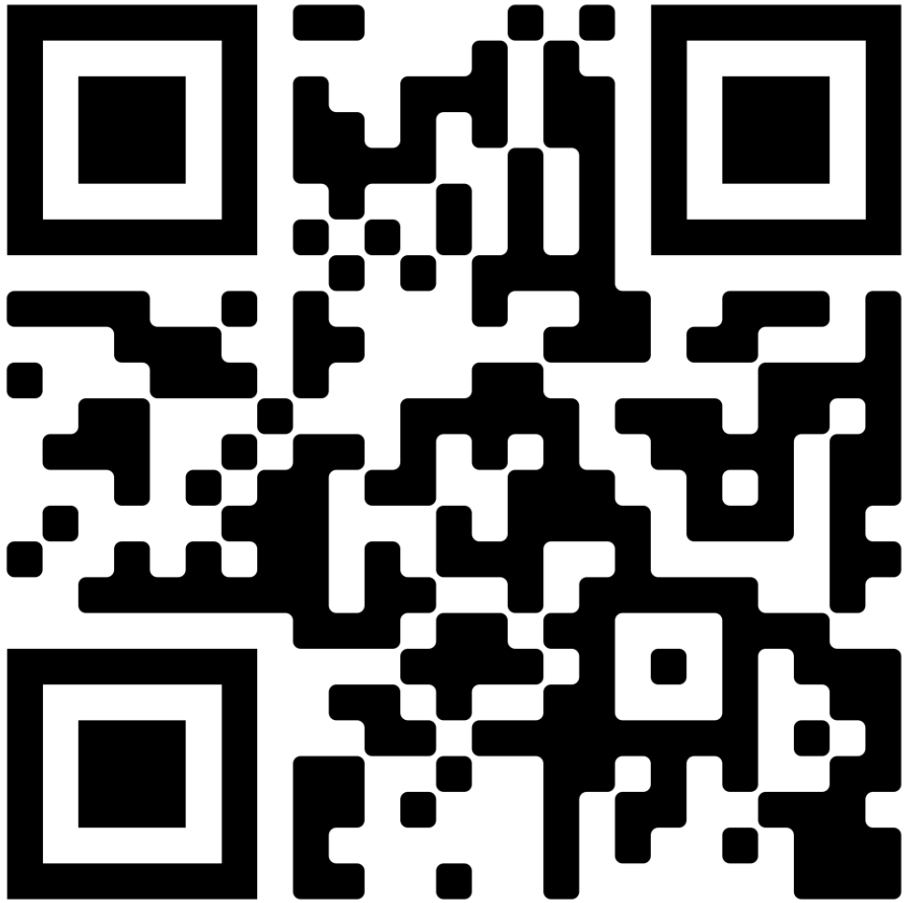
Conclusion

- **Full Open-source security scanners can coverage the Software Development Lifecycle**

- **Keep Humans in the Loop**
  - Evaluate risk and prioritize remediation
  - Continuously tune controls (Execution time, False positive, false negative)
  - Perform audits, architecture reviews, and penetration testing

Thank you

Pierrick Pochelu, LuxProvide

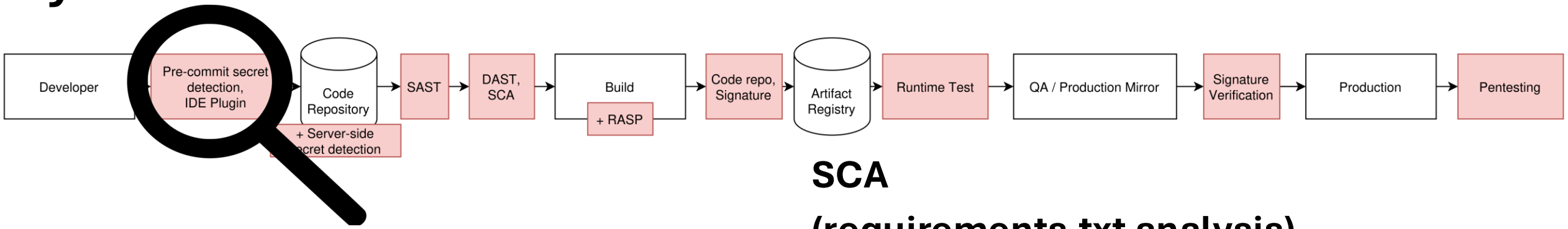


https://gitlab.seanergys.fz-juelich.de/pochelu1/devsecops_demonstrator

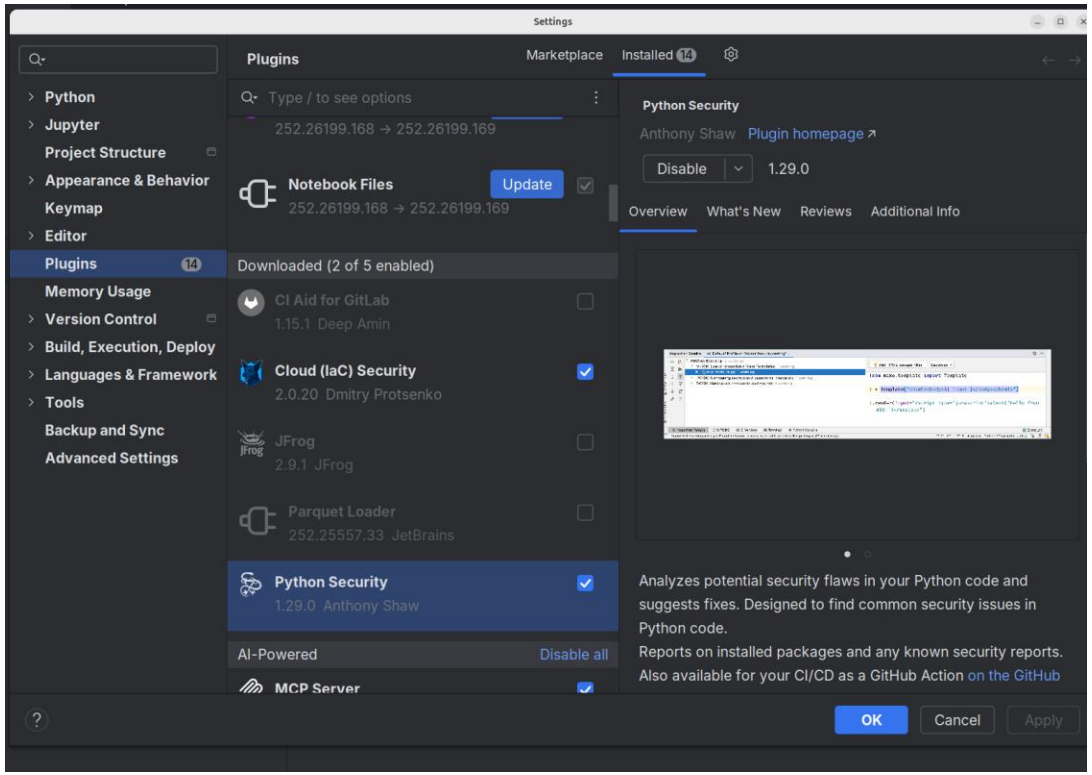
** QR available 15 days*

Back up slides

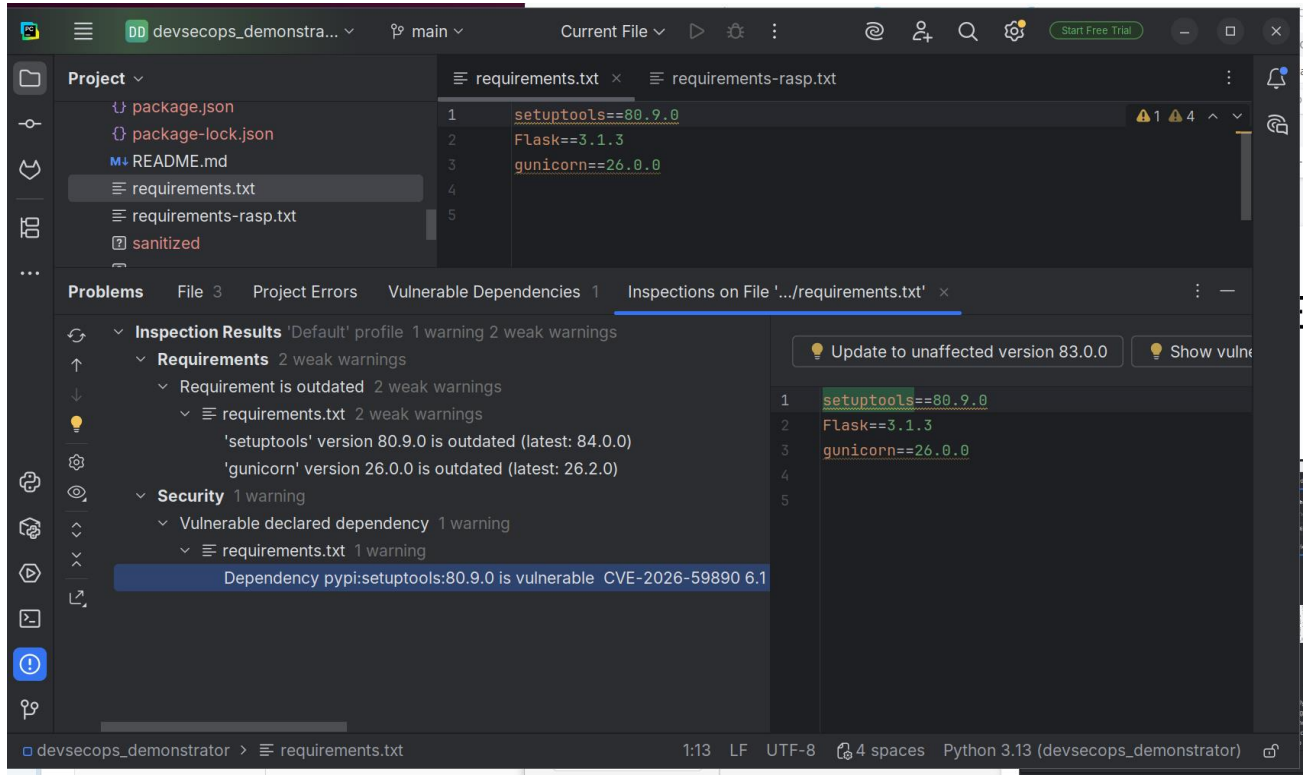
Python IDE



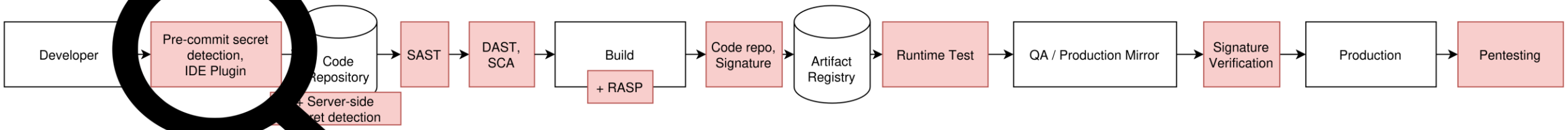
Plugins install



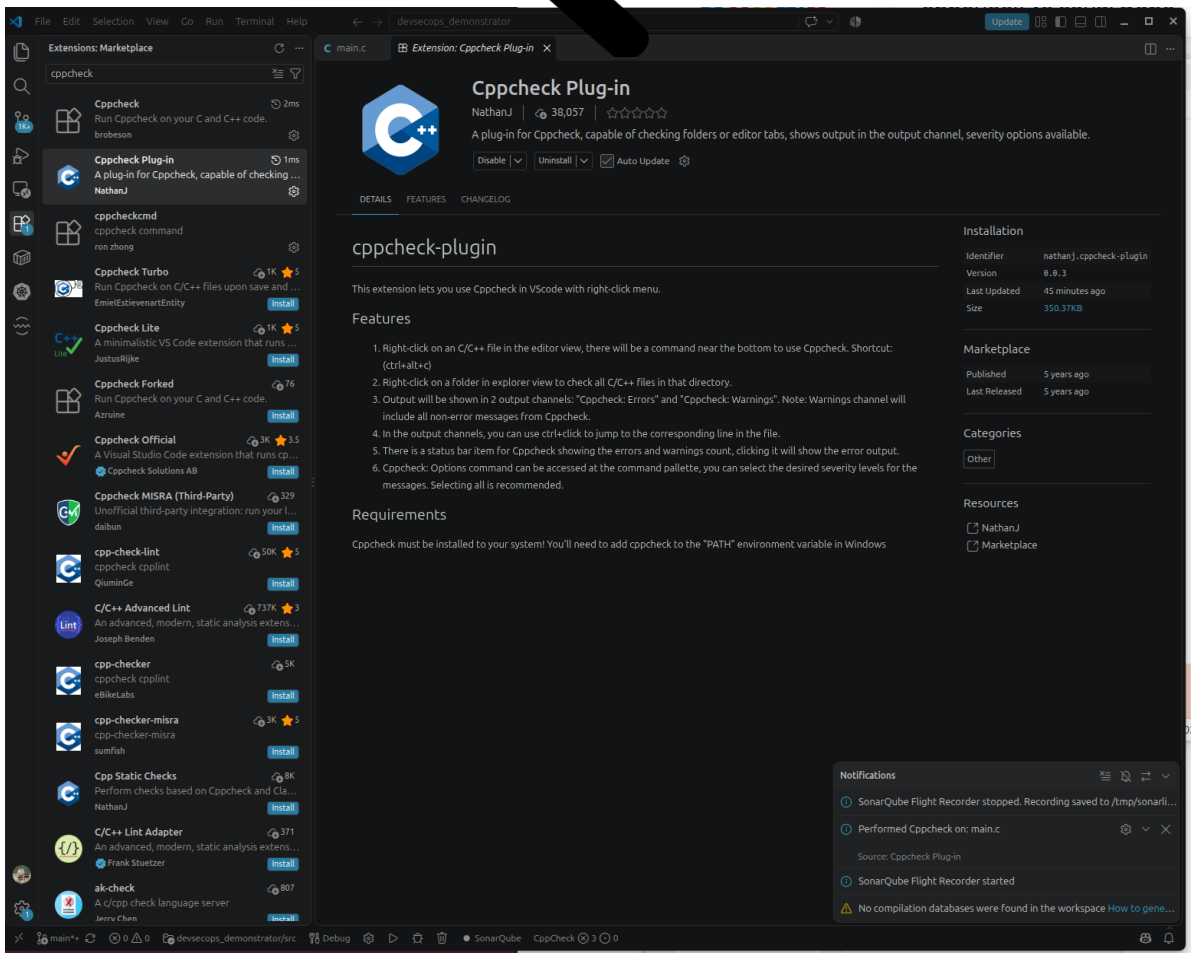
SCA (requirements.txt analysis)



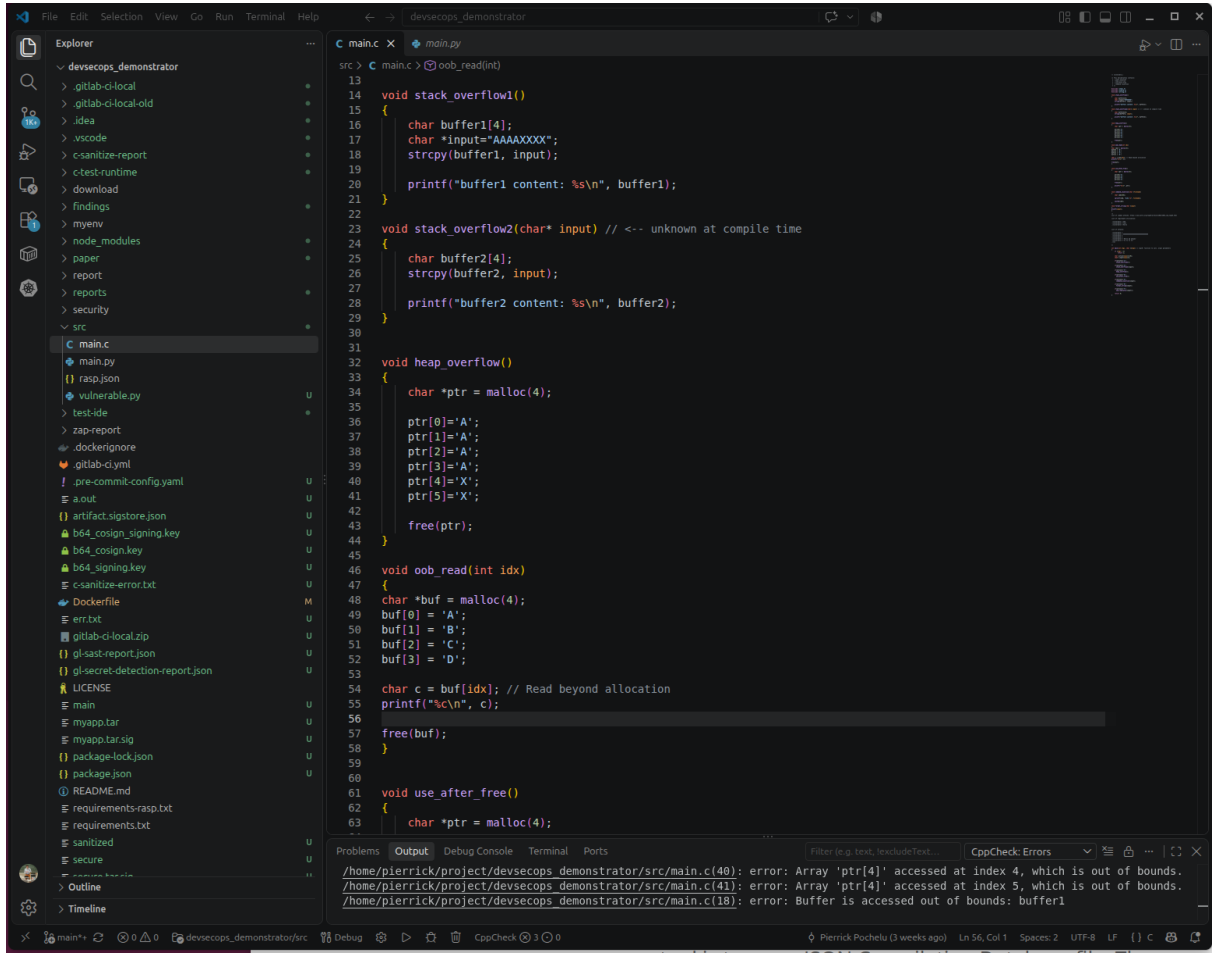
VScode / Cppcheck



Cppcheck plugin install



SAST C language



GitLab CI step (standard implementation)

```yaml
container_scanning:
  stage: py-test-build
  image:
    name: aquasec/trivy
    entrypoint: [""]
    # override the image's default ENTRYPOINT with an empty one. So start a shell.
  script:
    - trivy image --input myapp.tar --format json -o trivy-report.json
  artifacts:
    paths:
      - trivy-report.json
    expire_in: "7 days"
  needs:
    - py-build
```

GitLab CI step (advanced implementation)

```yaml
container_scanning:
  stage: py-test-build
  image:
    name: aquasec/trivy:0.74@sha256:62b1e65e8869bc4b4c6aa4fa2b21595256c7c2f6018a9d9ad61caf87187c1969
    entrypoint: [""]
    # override the image's default ENTRYPOINT with an empty one. So start a shell.
  rules:
  - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
  - changes:
    - "**/*.py"
    - "requirements.txt"
    - ".gitlab-ci.yml"
    - "Dockerfile"
  script:
    - trivy image --input myapp.tar --format json -o trivy-report.json
  artifacts:
    paths:
      - trivy-report.json
    expire_in: "7 days"
  needs:
    - py-build
```

GitLab CI step

Example output with "gcl" produces:

.gitlab-ci-local/

- artifacts/
  - build/
    - myapp.tar
  - container_scanning/
    - trivy-report.json
  - cosign-sign/
    - artifact.sigstore.json
  - ...
- output/
  - build.log
  - container_scanning.log
  - cosign-sign.log
  - cosign-verify.log
  - ...
- ...