Check for updates

SOFTWARE TOOL ARTICLE

Securing the Software Lifecycle: An Open-Source GitLab-Based DevSecOps Demonstrator

[version 1; peer review: awaiting peer review]

Pierrick Pochelu

LuxProvide, Bertrange, Luxembourg

v1 **First published:** 20 Oct 2026, **6**:437
https://doi.org/10.12688/openreseurope.25142.1
Latest published: 20 Oct 2026, **6**:437
https://doi.org/10.12688/openreseurope.25142.1

Abstract

Software development pipelines are increasingly targeted by attacks. These attacks are diverse, ranging from vulnerable code and exposed secrets to software supply chain compromise and runtime exploitation. While numerous security tools exist, they are often implemented as isolated controls, creating gaps in visibility and protection across the software development lifecycle.

This paper presents an open-source GitLab-based DevSecOps demonstrator that integrates 16 automated security controls spanning development, software supply chain security, and runtime protection. The architecture combines static and dynamic analysis, dependency and container scanning, artifact signing and verification, compiler hardening, fuzz testing, Web Application Firewall (WAF), and Runtime Application Self-Protection (RASP). The approach is demonstrated on two intentionally vulnerable applications implemented in Python and C and mapped to OWASP Secure Product Design Principles, NIST SSDF, and OWASP SAMM.

Experimental results show that all intentionally introduced vulnerabilities are detected, prevented, or mitigated by at least one security control. The demonstrator highlights the value of a defense-in-depth strategy in which complementary security mechanisms provide lifecycle-wide coverage while remaining fully open source, reproducible, and easily deployable within GitLab CI/CD environments.

Keywords

DevSecOps, CI/CD, SAST, DAST, Software Supply Chain, RASP, Defense-in-Depth.

Open Peer Review

Approval Status *AWAITING PEER REVIEW*

Any reports and responses or comments on the article can be found at the end of the article.

This article is included in the Horizon Europe gateway.

Corresponding author: Pierrick Pochelu (pierrick.pochelu@lxp.lu)

Author roles: Pochelu P: Conceptualization

Competing interests: No competing interests were disclosed.

Grant information: This project has received funding from the European Union's Horizon Europe research and innovation programme through the EuroHPC Joint Undertaking (JU) under the SEANERGYS project (Grant Agreement No. 101177590).
The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Copyright: © 2026 Pochelu P. This is an open access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

How to cite this article: Pochelu P. **Securing the Software Lifecycle: An Open-Source GitLab-Based DevSecOps Demonstrator [version 1; peer review: awaiting peer review]** Open Research Europe 2026, **6**:437 https://doi.org/10.12688/openreseurope.25142.1

First published: 20 Oct 2026, **6**:437 https://doi.org/10.12688/openreseurope.25142.1

Introduction

The software development lifecycle (SDLC) has become an increasingly attractive target for attackers. The 2026 Verizon Data Breach Investigations Report (DBIR),[1] based on more than 22,000 confirmed breaches worldwide, reports that software vulnerability exploitation accounts for 31% of breaches, up from 20% the previous year, surpassing credential abuse as the leading initial access vector for the first time in the report's history. Both the scientific literature[2,3] and industry guidance[4] recognize that point solutions alone are insufficient to address modern threats, motivating defense-in-depth strategies. Consequently, security controls are increasingly integrated and automated within CI/CD pipelines.[5,6] At the same time, these pipelines themselves have become high-value targets, making their protection a fundamental requirement for secure software delivery.[7,8]

DevSecOps has emerged as the dominant approach for integrating security into software delivery pipelines. However, existing security verification and repository management platforms typically provide only a limited subset of security capabilities, often focusing on specific stages of the software lifecycle, or specialized to cloud.[9] Examples include source code analysis during development,[10,11] software supply chain integrity and artifact verification,[8,12] and application protection at runtime.[13] In addition, many widely adopted solutions are delivered as commercial services, which can limit transparency, reproducibility, and accessibility for organizations seeking open and auditable security practices.

Therefore, the objective of this work is to provide a comprehensive open-source GitLab CI/CD demonstrator that consolidates security controls across the SDLC. The proposed demonstrator integrates complementary security mechanisms to implement a defense-in-depth strategy spanning source code, dependencies, build artifacts, deployment, and runtime environments. To illustrate the applicability of this approach, the demonstrator is evaluated using two representative projects implemented in Python and C.[14]

The main contributions of this work are:

- An open-source GitLab-based DevSecOps demonstrator covering development, software supply chain, and runtime security.
- Integration of 16 automated security controls within a reproducible CI/CD workflow.
- Demonstration on both interpreted (Python) and compiled (C) applications.
- Mapping of implemented controls to OWASP Secure Coding Principles, NIST SSDF, and OWASP SAMM.

The remainder of this paper is organized as follows. First, the security architecture is presented. Section II presents the security architecture of the demonstrator. Section III maps the implemented controls to relevant OWASP and NIST recommendations. Section IV describes the implementation details and demonstrates the practical execution of the proposed security pipeline.

Security architecture

Overview

The demonstrator is organized around three phases:

1. Development
2. Software Supply Chain Security
3. Runtime Security

To illustrate these phases, two intentionally vulnerable applications are provided:

- A containerized Python web service containing representative web application vulnerabilities.
- A C application containing representative memory-safety vulnerabilities and insecure coding patterns.

The objective is not to integrate every available security tool, but rather to demonstrate a representative set of complementary controls that collectively provide broad coverage across the software lifecycle. The selected tools were chosen based on their open-source availability, active maintenance, GitLab integration capabilities, and broad community adoption.

The implemented CI/CD workflows are summarized below.

Web Python application:

Secret Detection → SAST → DAST → Dependency and Container Scanning → Artifact Signing → Signature Verification → Container Hardening → WAF Protection → RASP Protection.

C language application:

Secret Detection → SAST → Dynamic Analysis (Sanitizer + test inputs) → Fuzz Testing → Build with Hardening Flags → Artifact Signing → Signature Verification → ELF Hardening Validation → Runtime Attack Simulation.

The characteristics of the implemented security controls are summarized in Table 1.

Coverage of software security principles

The goal of this demonstrator is to provide coverage of the major security principles. Security principles are commonly referenced by secure software development frameworks, including **OWASP Secure Product Design Principles**, **NIST SSDF**, **OWASP SAMM**.

Coverage of OWASP secure product coding principles

The OWASP Secure Product Design Cheat Sheet, chapter 4 code,[15] defines ten coding principles intended to reduce software vulnerabilities and improve overall security posture. This chapter maps these principles to the controls implemented within the demonstrator.

Principle #1 Input validation: Input validation ensures that unexpected or malicious inputs cannot trigger insecure application behavior.

Examples include:

- Web Application Firewall with Modsecurity and OWASP CRS rules. CRS includes rules capable of detecting suspicious input patterns.
- PyRASP validates incoming requests and blocks known attack payloads before application logic executes.

The automated tests contains one legitimate example (expected HTTP 200) and payload: “1 UNION SELECT 1”. (expected HTTP code 400) to demonstrate that malicious user input is rejected.

Principle #2 Error handling: Handle errors and exceptions in a secure manner, such as by logging them in a secure way and not disclosing sensitive information to an attacker.

Error handling depends of developers’s internal code and cannot be systematically demonstrated by the demonstrator.

Principle #3 Authentication and Authorization: Implement strong authentication and authorization mechanisms to ensure that only authorized users can access sensitive code and data. It should be implemented for the Supply Chain (GitLab multi factor authentication) or at runtime (authentication and authorization implemented by the developers).

The demonstrator has an automated nature and cannot demonstrate this principle alone without human intervention.

Principle #4 Cryptography: Cryptographic mechanisms are used throughout the demonstrator to protect the integrity, authenticity, and confidentiality of software artifacts and sensitive data, both *in transit* and *at rest.*

Table 1. Security controls integrated into the DevSecOps demonstrator and their corresponding phases, security tool type, inputs (file format or service), and outputs (reporting format or blocking effect).

CI Name	Phase	Type	Technology	Input	Output
secret_detection	Development	Secret Detection	GitLab Secret Detection	Git repositories, commits, merge requests	Secret findings (passwords, API keys, tokens, certificates)
semgrep-sast	Development	Source Code SAST	Semgrep	Python, C/C++, Java, and other source code	CWE, Bandit ID, OWASP Top 10 ID
kics	Development	IaC Static Analysis	KICS	Ansible, Terraform, Dockerfile, Kubernetes YAML, and other IaC files	CWE
container_scanning	Development	Dependency Scanning	Trivy	Source code dependencies, SBOMs, container images	CVE
owasp-zap	Development	Dynamic Application Security Testing (DAST)	OWASP ZAP	Running application (URL/IP: Port)	CWE, WASC ID, ZAP Alert ID, OWASP Top 10 ID
cosign-sign/cosign-verify	Supply Chain	Artifact Signing and Verification	Cosign[12]	Container images, OCI artifacts	Signature verification status (OK/FAIL)
test-rasp	Runtime	Runtime Application Self-Protection (RASP)	PyRASP[13]	Running Python application and attack traffic	HTTP 200 (Allowed) /HTTP 400 (Blocked)
test-waf	Runtime	Web Application Firewall (WAF)	Nginx + ModSecurity + OWASP CRS	HTTP requests to deployed application	HTTP 200 (Allowed) /HTTP 400 (Blocked)
c-sast	Development	Source Code SAST	GCC Static Analyzer (−fanalyzer)	C source code	Memory leaks, use-after-free, double-free, null dereference, tainted input, analyzer warnings
c-build	Development	Binary Hardening	GCC Hardening Flags	C source code	PIE,[16] RELRO,[17] NX, Stack Protector,[18,19] FORTIFY,[18,20] CET enabled[21]
openssl-sign/ openssl-verify	Supply Chain	Artifact Signing and Verification	OpenSSL	Artifacts, binaries, packages	Signature verification status (OK/FAIL)
c-sanitizer	Development	Dynamic Analysis	AddressSanitizer (Asan),[22] UndefinedBehaviorSanitizer (UBSan)[23] with test inputs	Executed binary and test inputs	Runtime memory and undefined behavior detection
c-fuzz	Development	Dynamic Analysis	AFL++[24] with representative inputs	Binary and input test cases	Crashes, hangs, coverage-guided vulnerability discovery
c-elf-check	Runtime	Hardening Validation	Checksec, Readelf	Compiled ELF binary	Verification of RELRO, NX, PIE, Canary, RPATH, RUNPATH
c-test-runtime	Runtime	Runtime Attack Simulation	Test inputs	Binary and input test cases	Process abort, security reports, non-zero exit status

In transit. During the software supply chain phase, build artifacts are digitally signed using Cosign and OpenSSL. Prior to execution, container images are verified to ensure that artifacts have not been modified and originate from a trusted source.

At rest. Cryptographic material, such as signing (or "private") keys, is not stored or visible in source code repositories. Instead, sensitive data is securely managed through GitLab protected variables.[25] Artifact verification is also performed before execution, ensuring that only trusted and cryptographically validated software components are used. Together, these controls demonstrate how cryptography can be applied to secure both software delivery and the protection of sensitive credentials within CI/CD environments.

Principle #5 Least privilege: The principle of least privilege requires that software components, services, users, and processes operate with only the minimum permissions necessary to perform their intended functions. Applying this principle reduces the attack surface and limits the potential impact of a successful compromise.

Within the demonstrator, least privilege is illustrated through several practical controls:

- Containerized applications execute with restricted privileges rather than full host access.
- Only the minimum network ports required by the applications are exposed.
- GitLab protected variables restrict access to sensitive credentials used during CI/CD execution.

Principle #6 Secure memory management: Memory-related vulnerabilities are common in low-level languages.

The C language shows principle to secure the memorizer.

- Sanitizer allow to detect based on AST.
- FORTIFY does not primarly aim this kind of attack, with Python was choosen has example. However, Semgrep contains default rules[26] and extended by open source community (such as Semgrep10K[27]) includes, for example C buffer overflow and use-after-free vulnerability rules.
- Runtime protection, abort the program

Language specific SAST rules reduces exposure to those language specific issues.

Principle #7 Avoiding hard coded secrets: Credentials and cryptographic materials should never be embedded directly within source code and git repository messages (git commits message, discussions, ….). Instead they should be stored as GitLab "secrets".

Examples include:

- GitLab Secret Detection automatically searches for exposed secrets
- Signing keys are injected through CI protected variables (or "secrets") rather than committed to repositories.

Principle #8 Security testing: GitLab-based demonstrator, allows continuous testing throughout development and deployment.

Principle #9 Auditing and reviewing the code: Security assessments should include both automated and human review processes.

The proposed architecture supports auditing through:

- Automated CI/CD security reports generated by tools such as SAST, dependency analysis, and secret scanning solutions.
- Findings uses standard identifiers (e.g., CVE, CWE, WASC ID, OWASP TOP10 ID, …).

The integration of scanners in GitLab provide a traceable history of detected issues and serve as evidence for human. Human reviewers can perform deeper analysis: prioritize remediation effort, assess business impact, identify false positives, and evaluate business security risks that automated tools may not detect.

The demonstrator has an automated nature and cannot demonstrate this principle alone without human intervention.

Principle #10 Keeping up-to-date: Software security depends on timely updates and vulnerability remediation.

Examples in the demonstrator include:

- Explicit versioning of container images.
- Digest-pinned dependencies.
- Automated vulnerability scanning through Trivy.
- Regular updates of rules (example: OWASP CRS) and security tools.

Mapping to NIST Secure Software Development Framework (SSDF)

The NIST Secure Software Development Framework (SSDF)[28] provides a set of recommended practices for reducing vulnerabilities throughout the software development lifecycle and improving software integrity, authenticity, and resilience. The framework is organized into four groups of practices: Prepare the Organization (PO), Protect the Software (PS), Produce Well-Secured Software (PW), and Respond to Vulnerabilities (RV). Table 2 summarizes the alignment of the demonstrator with selected NIST SSDF practices.

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

Coverage of OWASP SAMM principles

OWASP SAMM (Software Assurance Maturity Model)[29] provides a framework for evaluating and improving secure software development practices across five business functions: Governance, Design, Implementation, Verification, and Operations. Since the demonstrator focuses on technical controls integrated into CI/CD, the mapping is limited to the implementation, verification, and operations functions. The relationship between the demonstrator and OWASP SAMM business functions is shown in Table 3.

Framework coverage summary

An overall framework coverage summary is provided in Table 4.

Due to the CI/CD nature of the demonstrator and its use of generic example applications, some principles and practices are outside its scope.

Table 2. Mapping of OWASP SSDF to security controls implemented in the demonstrator.

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.

Software security principles not demonstrated:

- Error handling and exception management.
- Authentication and authorization mechanisms.
- Secure settings by default.

Table 3. Mapping of demonstrator security controls to OWASP SAMM pinciples.

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.

Table 4. Coverage of security frameworks and principles supported by the DevSecOps demonstrator.

Framework	Scope evaluated	Coverage
OWASP Secure Product Coding Principles	7 of 10 Principles Implemented or Demonstrated	70%
NIST SSDF Practices	13 of 17 Practices Supported	76%
OWASP SAMM Business Functions	3 of 5 Business Functions Covered	60%

Project and organizational practices not demonstrated:

- Security requirement definition.
- Roles and responsibilities assignment.
- Human review activities (e.g., design review, code review, threat modeling).
- Risk management and governance activities.

These topics are project- or organization-specific and cannot be fully demonstrated through generic CI/CD automation alone.

Project result

Running CI

While the architecture is hosted on GitLab, it can be downloaded and the CI can be executed locally to run DevSecOps easily on custom applications. Recommended tooling for local execution: **gcl**.[30]

CI Inspection

The result of GitLab CI pipeline contains:

- Artifacts. Including security reports generated by the CI stages, containers, binaries, signatures, …
- Logs generated by each stage. They allow to identify security controls, and if the security itself is well configured.

Specifically, the **gcl** demo produces the repository**.gitlab-ci-local/**containing:

- artifacts/
  - build/
    - myapp.tar
  - container_scanning/
    - trivy-report.txt

    - cosign-sign/
        - artifact.sigstore.json
    - …
- output/
    - build.log
    - container_scanning.log
    - cosign-sign.log
    - cosign-verify.log
    - …
- …

Experimental evaluation

The demonstrator includes intentionally vulnerable applications to validate that the integrated security controls are correctly configured and able to detect or prevent representative attack scenarios. Vulnerabilities were selected from the MITRE CWE Top 25 Most Dangerous Software Weaknesses[31] to cover both web application security and memory-safety issues.

Web Application attack suite:

- 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.

C Application attack suite:

- 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.

Other attack:

* Hard-Coded Credentials, CWE-798.

Web protection evaluation

The results of the web application protection evaluation are presented in Table 5. “Yes” indicates that the attack payload was detected and prevented by the corresponding protection mechanism, resulting in an HTTP error response to the client.

Table 5. Untime protection results for representative web application vulnerabilities using WAF (Nginx + ModSecurity + OWASP CRS) and PyRASP.

Vulnerability type	WAF	PyRASP
Cross-Site Scripting (CWE-79)	Yes	Yes
SQL Injection (CWE-89)	Yes	Yes
Cross-Site Request Forgery (CWE-352)	Yes	Yes
Path Traversal (CWE-22)	Yes	Yes
Server-Side Request Forgery	Yes	Yes

Table 6. Detection coverage of security controls against representative C-language vulnerabilities.

Vulnerability type	GCC static analyzer (-fsanitizer flag)	Semgrep	ASan/UBSan with AFL++ (10,000 iterations)	Runtime detection (hardened ELF)
Use-After-Free (CWE-416)	Yes	No	Yes	Yes
Out-of-Bounds Read (CWE-125)	Yes	No	Yes	Yes
Integer Overflow (CWE-190)	No	Yes	No	No
OS Command Injection (CWE-78)	No	Yes	No	No
Stack-Based Buffer Overflow (Fixed-Size Buffer*) (CWE-122)	Yes	Yes	Yes	Yes
Stack-Based Buffer Overflow (User-Controlled Input**)	No	Yes	Yes	Yes
Heap-Based Buffer Overflow (CWE-122)	Yes	Yes	Yes	Yes
Format String Vulnerability (CWE-134)	No	Yes	No	No

*Stack-Based Buffer Overflow (Fixed-Size Buffer) refers to an overflow caused by a programming error in a statically sized buffer.
**Stack-Based Buffer Overflow (User-Controlled Input) refers to an overflow triggered directly by attacker-controlled input.

C protection evaluation

Table 6 summarizes the detection and mitigation coverage achieved for representative C-language vulnerabilities. "Yes" indicates that the vulnerability was either detected during analysis or mitigated by runtime protection mechanisms.

Secrets

A hard-coded credential (CWE-798) is deliberately introduced to validate secret-detection capabilities. GitLab Secret Detection identifies secrets in repository content and commit messages, Semgrep detects secrets embedded in source code (Python and C), and KICS identifies secrets exposed in Infrastructure-as-Code (YAML and Dockerfile).

Summary

The results highlight the complementary nature of the security controls. No individual control detects all vulnerability classes. Instead, multiple independent layers provide coverage across the possible vulnerabilities. Static analysis identifies weaknesses before deployment, dynamic analysis detects vulnerabilities during testing, and runtime protections or hardening mechanisms provide an additional layer of defense by blocking attacks or terminating execution when unsafe behavior is detected. The evaluation demonstrates that every common vulnerability intentionally included in the demonstrator is detected, prevented, or mitigated by at least one security.

Conclusion

DevSecOps tooling are often fragmented, with individual solutions typically addressing only specific phases of the software lifecycle. This work presents an open-source GitLab-based DevSecOps demonstrator that integrates security tools across development, software supply chain, and runtime phases within a single reproducible architecture. To cover

the reality of software, we apply it to 2 popular and complementary languages: Python and C. To cover different programming ecosystems and demonstrate a wider range of security tools, two intentionally vulnerable applications are used: one written in Python and one in C.

The demonstrator illustrates how defense-in-depth principles can be applied across the software delivery lifecycle while remaining aligned with established frameworks such as NIST SSDF, OWASP SAMM, and OWASP Secure Product Design Principles.

Future work will extend the demonstrator with additional security gates, applications, and attack suites. It will also investigate larger-scale deployments, including cross-GitLab security management and security governance across multiple projects.

Ethics and consent

Ethical approval and consent were not required.

Data and software availability

Data

CI/CD artifacts and execution outputs are available from: Zenodo: DevSecOps Demonstrator https://doi.org/10.5281/zenodo.22862862,[32] in the file *gitlab-ci-local.zip*.

Software

Code repository URL:

- Software available[32] from: https://gitlab.seanergys.fz-juelich.de/pochelu1/devsecops_demonstrator
- Source code available from: https://gitlab.seanergys.fz-juelich.de/pochelu1/devsecops_demonstrator
- Archived source code at time of publication: https://doi.org/10.5281/zenodo.22862862 (file *devsecops_demonstrator-v1.0.0.zip*)
- Demonstrator execution at time of publication:
  - Pipeline: https://gitlab.seanergys.fz-juelich.de/pochelu1/devsecops_demonstrator/-/pipelines/3980
  - Terminal recording: https://asciinema.org/a/1265349
- License: Apache License 2.0

References

1. Verizon Business: *2025 Data Breach Investigations Report (DBIR).* Verizon Business; 2025.
Reference Source
2. Rajapakse RN, Zahedi M, Babar MA, *et al.*: **Challenges and solutions when adopting DevSecOps: A systematic review.** *Inf. Softw. Technol.* 2022; **141**: 106700.
Publisher Full Text
3. Mohammed KI, Shanmugam B, El-Den J: **Evolution of DevSecOps and Its Influence on Application Security: A Systematic Literature Review.** *Technologies.* 2025; **13**(12): 548.
Publisher Full Text
4. National Institute of Standards and Technology (NIST): *Secure Software Development, Security, and Operations (DevSecOps) Practices. NIST Special Publication 1800–44.* Gaithersburg, MD, USA: NIST; 2025.
5. Cyril HP, Kumara S: **DevSecOps-Driven Security Integration in the Software Development Lifecycle Using CI/CD Pipelines.** *Proceedings of the 2026 IEEE 5th International Conference on AI in Cybersecurity (ICAIC).* IEEE; 2026; pp. 1–6.
Publisher Full Text
6. Lella I, Theocharidou M, Tsekmezoglou E, *et al.*: *ENISA Threat Landscape for Supply Chain Attacks.* European Union Agency for Cybersecurity (ENISA); 2021.
Publisher Full Text
7. OWASP Foundation: **CI/CD Security Cheat Sheet. OWASP Cheat Sheet Series.** Accessed September 10, 2026.
Reference Source
8. Torres-Arias S, Afzali H, Kuppusamy TK, *et al.*: **In-toto: Providing Farm-to-Table Guarantees for Bits and Bytes.** *Proceedings of the 28th USENIX Security Symposium.* USENIX Association; 2019; pp. 1393–1410.
9. Mittal A: **AI-Augmented DevSecOps Pipelines for Secure and Scalable Service-Oriented Architectures in Cloud-Native Systems.** *Proceedings of the 2025 IEEE International Conference on Service-Oriented System Engineering (SOSE).* IEEE; 2025; pp. 79–84.
Publisher Full Text
10. OWASP Foundation: **Source Code Analysis Tools.** Accessed September 7, 2026.
Reference Source
11. OWASP Foundation: **Vulnerability Scanning Tools.** Accessed September 7, 2026.
Reference Source
12. Newman Z, Meyers JS, Torres-Arias S: **Sigstore: Software Signing for Everybody.** *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22).* ACM; 2022; pp. 2353–2367.
Publisher Full Text

13. ParaCyberBellum: **PyRASP Resources.** Accessed September 10, 2026.
Reference Source

14. TIOBE Software: **TIOBE Programming Community Index.** September 2026 Edition.
Reference Source

15. OWASP Foundation: **Secure Product Design Cheat Sheet.**
Reference Source

16. Shacham H, Page M, Pfaff B, *et al.*: **On the Effectiveness of Address-Space Randomization.** *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04).* ACM; 2004; pp. 298–307.
Publisher Full Text

17. Sidhpurwala H: *Hardening ELF binaries using Relocation Read-Only (RELRO).* Red Hat Blog; 2019.
Reference Source

18. Shacham H: **The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86).** *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS).* ACM; 2007.

19. Cowan C, Pu C, Maier D, *et al.*: **StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.** *Proceedings of the 7th USENIX Security Symposium.* USENIX Association; 1998.

20. Moghadam VE, Serra G, Aromolo F, *et al.*: **Memory Integrity Techniques for Memory-Unsafe Languages: A Survey.** *IEEE Access.* 2024; **12**: 43201–43221.
Publisher Full Text

21. Abadi M, Budiu M, Erlingsson U, *et al.*: **Control-Flow Integrity.** *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05).* ACM; 2005; pp. 340–353.
Publisher Full Text

22. Serebryany K, Bruening D, Potapenko A, *et al.*: **AddressSanitizer: A Fast Address Sanity Checker.** *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12).* USENIX Association; 2012; pp. 309–318.

23. Hathhorn C, Ellison C, Roşu G: **Defining the Undefinedness of C.** *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM; 2015; pp. 336–345.

24. Fioraldi A, Maier D, Eißfeldt H, *et al.*: **AFL++: Combining Incremental Steps of Fuzzing Research.** *14th USENIX Workshop on Offensive Technologies (WOOT '20).* USENIX Association; 2020.

25. GitLab Inc: **CI/CD Variables Documentation.**
Reference Source

26. Semgrep: **Semgrep Rules Repository.**
Reference Source

27. 0xBugatti: **SEMGREP10K.**
Reference Source

28. National Institute of Standards and Technology (NIST): **Secure Software Development Framework (SSDF), SP 800–218.**
Reference Source

29. OWASP Foundation: **OWASP Software Assurance Maturity Model (SAMM).**
Reference Source

30. Firecow: **gitlab-ci-local.**
Reference Source

31. MITRE Corporation: **2025 CWE Top 25 Most Dangerous Software Weaknesses.**
Reference Source

32. Pochelu P: **DevSecOps Demonstrator (Version v1.0.0) [Computer software].** *Zenodo.* 2026.
Publisher Full Text