DevSecOps policy-as-code: Detecting and remediating over-privileged access tokens in SDLC tools*

Volodymyr Havryliak[1,*,†] and Sviatoslav Vasylyshyn[1,†]

[1] Lviv Polytechnic National University, 12 Stepan Bandera str., 79013 Lviv, Ukraine

Abstract

The article represents a Proof of Concept (PoC) for policy-as-code methodology designed to detect "Privilege Sprawl" in GitLab CI/CD environments. While modern DevSecOps heavily relies on non-human machine identities to run automated pipelines, these credentials are frequently granted sweeping privileges that completely bypass the principle of Least Privilege. In this research, we test the idea of "Scope-Usage Asymmetry" – a way to actually measure the massive gap between the authority a token is given and the utility it actually needs to do its job. Because we wanted to map this risk without needing global GitLab Administrator access, we built a "Bottom-Up" auditing tool. It scans repository CI/CD variables and dynamically checks the token identities directly against the GitLab API. When we ran a simulated enterprise audit across an organisation of 27 projects and 5 groups, the results were significant. The PoC flagged 11 security violations from just 5 unique tokens. We uncovered a severe average "Sprawl Factor", peaking in a critical "Two-Project Anomaly" where a single automation token secretly held write access to 22 repositories – exposing 81% of the entire organisation's codebase. Furthermore, the methodology exposed significant business continuity risks by identifying 3 Personal Access Tokens (PATs) – including one belonging to an Administrator – improperly embedded in automation pipelines. By generating a clear, actionable remediation report, our PoC showed how teams can reduce their organisational exposure by 73% and get back into compliance with Least Privilege. Ultimately, this study shows that you don't need heavy, top-down infrastructure to audit your architecture of trust; the development teams can use lightweight tools to secure their own supply chains.

Keywords

DevSecOps, GitLab, CI/CD, Privilege Sprawl, Proof of Concept, Least Privilege, Token Management, Blast Radius, Policy-as-Code.

1. Introduction

The ongoing integration of DevSecOps practices has fundamentally transformed the security landscape of software development. It is no longer enough to secure the network; instead, organisations depend on the integrity of their CI/CD pipelines. In this environment, the "keys to the kingdom" are — API tokens, service accounts and SSH keys — that allow automated scripts to build, test, and deploy code across distributed infrastructure [1, 2].

To secure these environments, the DevSecOps community has primarily focused on detecting credential leakage early in the software development lifecycle. Existing security tools, such as Gitleaks or TruffleHog, are primarily designed for secret detection, identifying high-entropy strings that are accidentally committed to version control systems. These tools help to answer the main question: "Is there a secret visible in the code?" [3].

However, existing studies point out that binary detection is insufficient. As noted by Zhuravchak et al. [4], modern DevSecOps requires mechanisms to visualise security regression and track the evolution of the security posture over time, rather than viewing scans as isolated events. While their work focuses on vulnerability regression, a similar "blind spot" exists in identity management: current tools do not address the architectural question, "Does this valid secret have more permissions than The ongoing integration of DevSecOps practices has fundamentally transformed the security landscape of software development. It is no longer enough to secure the network; instead, organisations depend on the integrity of their CI/CD pipelines. In this

* CSDP'2026: Cyber Security and Data Protection, May 7, 2026, Lviv, Ukraine
* Corresponding author.
† These authors contributed equally.
volodymyr.r.havryliak@lpnu.ua (V. Havryliak); sviatoslav.i.vasylyshyn@lpnu.ua (S.Vasylyshyn)
0009-0002-4114-1232 (V. Havryliak); 0000-0003-1944-2979 (S.Vasylyshyn)

environment, the "keys to the kingdom" are — API tokens, service accounts and SSH keys — that allow automated scripts to build, test, and deploy code across distributed infrastructure [1, 2].

To secure these environments, the DevSecOps community has primarily focused on detecting credential leakage early in the software development lifecycle. Existing security tools, such as Gitleaks or TruffleHog, are primarily designed for secret detection, identifying high-entropy strings that are accidentally committed to version control systems. These tools help to answer the main question: “Is there a secret visible in the code?” [3].

However, existing studies point out that binary detection is insufficient. As noted by Zhuravchak et al. [4], modern DevSecOps requires mechanisms to visualise security regression and track the evolution of the security posture over time, rather than viewing scans as isolated events. While their work focuses on vulnerability regression, a similar “blind spot” exists in identity management: current tools do not address the architectural question, “Does this valid secret have more permissions than necessary?”

Lets imagine that pipeline may pass all static secret-detection checks but it is still using credentials that are vastly overprivileged for the task at hand. This “scope blindness” represents a significant gap in current security strategies, as it fails to check the authority of a secret.

The urgency of securing these automated pipelines has been formally recognized by major security organizations. The National Institute of Standards and Technology (NIST) [5] explicitly warns that CI/CD pipelines represent a critical vector in Software Supply Chain Security. Similarly, the OWASP Foundation [6] highlights that compromised privileged accounts within CI/CD systems allow attackers to manipulate execution paths. Sysdig [7] reports that failing to enforce strict access controls around these machine identities maximizes the chances that a breached account will expose the entire infrastructure

From the view of integrated SDLC platforms like GitLab, the conflict between developer velocity and security management often leads to the “Path of Least Resistance”. Developers who aim to configure pipelines across multiple microservices as quickly as possible almost always choose Group Access Tokens or similar (e. g. PAT). These powerful credentials inherit permissions across an entire platform, granting administrative control over every current and future project within a group. While this reduces time, it introduces a critical vulnerability known as Privilege Sprawl. If this kind of token is used for a routine job and compromised, the attacker gains the Maintainer or Owner role over the entire group [5]. If this kind of token is used for a routine job and compromised, the attacker gains the Maintainer or Owner role over the entire group [8].

This paper presents a Policy-as-Code (PoC) framework designed to directly tackle this problem. We propose an auditing system that scans and inventories active access tokens, mapping where they are actually defined in CI/CD configurations against where they are utilized [21]. By treating permission scoping as code that can be audited, measured, and refactored, we want to give IT governance teams a practical way to systematically shrink their attack surface and establish robust identity governance right into their pipelines.

2. Related Works

Integrating security scanning into CI/CD pipelines has become a fundamental part of DevSecOps [1, 2]. We now rely on tools like Gitleaks and TruffleHog to automatically check for hardcoded credentials at build time, allowing teams to “shift left” and catch mistakes early in the software development lifecycle. But while this kind of static analysis is standard practice today, very few of these tools actually validate the architectural scope of the credentials running these pipelines.

A lot of recent research tackles the problem of leaked credentials, mostly focusing on measuring exposure. For example, one study [9] compared machine learning techniques against traditional regex scanners in Version Control Systems. They found that while pattern matching works fine for default secrets, it throws a lot of false positives when dealing with custom internal credentials. Similarly, Biringa and Kul [3] looked into using Large Language Models (LLMs) to catch sneaky hardcoded keys that slip past static analysis. But even as these detection methods get smarter, the core objective remains the same: finding the secret, rather than evaluating its permission level.

Taking a slightly different angle, Yang et al. [10] investigated access token security in multi-user smart home platforms. They discovered that token misuse usually stems from poorly designed delegation protocols. Even though their focus was on IoT, the lesson translates perfectly to CI/CD

[18]: when systems lack granular controls, privilege escalation is almost guaranteed. It is a critical reminder that every token can be architecturally dangerous.

At an enterprise scale, Gullapalli [11] highlighted this same issue within GitHub environments. The author pointed out that as engineering teams grow, managing API access and collaboration boundaries naturally becomes a massive attack vector. Without automated guardrails in place, the natural chaos of a large engineering team inevitably leads to permission bloat. It proves that strong authentication (knowing exactly who you are) is practically useless without equally strict authorisation (limiting exactly what you can do).

Even with all these insights, the industry still treats security like a simple pass/fail test. We obsess over finding out if a token has leaked, but we completely ignore “Scope-Usage Asymmetry” — the massive, dangerous gap between what a token is allowed to do and what it actually needs to do. The whole concept of “architectural awareness” — actually understanding if a credential is wildly overpowered for its daily job — remains practically untouched.

Fortunately, the conversation is starting to shift toward multi-layered security models that go beyond static scans. Kamaluddin [12] recently made the case for behavioural threat detection in DevSecOps. Additionally Snyk company [13] highlighted how Policy-as-Code engines like the Open Policy Agent (OPA) [17] can seamlessly complement traditional vulnerability scanners. Finally, Zhuravchak et al. [4] showed that we need to visualise security regression so we can actually watch risks evolve over time.

Yet, there is still a massive gap when it comes to practical, open-source tooling. Premium platforms offer a peek into token usage, but they act as opaque black boxes and rarely give developers active steps to shrink their “blast radius”. This paper introduces a Policy-as-Code methodology specifically designed to hunt down over-privileged tokens and map exactly how a token is being used against what the project actually requires.

3. Theoretical Framework

The solution for managing excessive user permissions requires us to establish GitLab's identity model as one of the leading SDLC tools and then measure the security threats posed by different types of credentials.

The GitLab ecosystem uses a rigid authentication system which follows an ordered sequence of authorisation levels. The official documentation allows us to divide tokens into categories which determine their maximum potential damage when their security is breached [14, 15, 16].

Token Type	Mechanism	Scope & Operational Reality	Risk Profile/Impact
Personal Access Tokens (PAT)	Tied directly to a human user's account, acting as a digital proxy for that individual.	Intended: Local development. Reality: Often improperly used in CI/CD. The token inherits the full permission set of the human user (e.g., Admin) across the entire instance.	High Risk: If leaked, the attacker can bypass all project isolation and act as a human user. A pipeline running with a Senior Engineer's PAT effectively runs as an Admin.
Group Access Tokens (GAT)	Provisions a distinct "Bot User" (e.g., group_123_bot) that functions as a full principal, independent of human licenses.	Scope: Grants administrative control over the entire Namespace, including all current projects, subgroups, and all future projects.	Critical Flaw: The "Downstream Only" model forces an "all-or-nothing" permission structure. It is impossible to restrict access to a subset of the group, leading to massive Privilege Sprawl.
Project Access Tokens (PrAT)	Provisions a "Bot User" (project_123_bot)	Scope: Limited exclusively to one specific project.	Contained Risk: The "Blast Radius" is minimised. If the

	restricted strictly to the boundary of a single repository.	Ideal: Represents the "Atomic Unit of Trust" for point-to-point integrations.	token is compromised, the attacker is trapped within that single project and cannot move laterally.
Pipeline Trigger Tokens	A static string generated specifically to invoke the trigger/pipeline API endpoint.	Scope: Extremely narrow. It can only trigger a pipeline in a specific project. It cannot read code, clone repositories, or modify CI/CD variables.	Very Low Risk: If compromised, an attacker can only force the pipeline to run (potential resource exhaustion/DoS), but cannot alter the code being deployed or exfiltrate sensitive data.
CI/CD Job Tokens (CI_JOB_TOKEN)	An ephemeral token dynamically injected by the GitLab Runner into the environment variables of a running job.	Scope: Time-bound strictly to the duration of the pipeline job. By default, it allows API interactions within the executing project, or it can explicitly allow cross-project connections.	Minimal Risk: The ideal credential for internal automation. Because the token expires immediately upon job completion, it cannot be hoarded or exploited in the long term by an attacker.

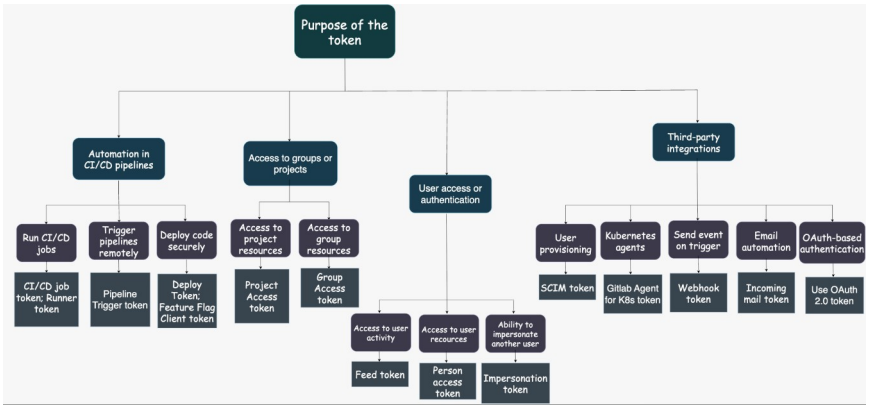

Figure 1: The decision tree for token selection in GitLab

We define Privilege Sprawl as the gap between what a token can do and what it actually does.

In an ideal security world, a token's Authority should perfectly match its Utility. However, in modern DevOps, this rarely happens due to the “Path of Least Resistance”. The main priority for developers is building pipelines that run at the maximum possible speed [21]. Creating a granular token for every single project takes time and maintenance. The process of creating a Group Token, which serves as a single token for all locations, is immediate and requires no ongoing upkeep [19, 20].

The system shows an essential security vulnerability because administrative permissions in the token stay inactive until unauthorised users obtain access to them.

The “Two-Project Anomaly”. The most severe manifestation of Privilege Sprawl is the Two-Project Anomaly. This particular pattern creates the most dangerous situation in microservice-based systems.

- The Scenario: A developer needs Service A to trigger a deployment pipeline in Service B.
- The Action: The developer generates a Group Access Token or PAT instead of creating a specific trigger token for Service B and adds it to Service A's variables.
- The Result: The token enables access to all projects in the group, but it only serves two projects.

- The Sprawl: All unrelated projects — which may contain production secrets or customer data — are now exposed to a compromise originating in Service A.

The "God Mode" Scope. The permission scope in the system creates an increased risk. The GitLab API requires developers to handle all CI/CD operations. To do this, developers almost exclusively select the api scope.

- Definition: The api scope grants complete read/write access to the GitLab API.
- The Danger: When combined with a Group Token, this creates a credential that has Group-level breadth and API-level depth.

Normative vs. Positive Analysis. The main issue exists because there is no comprehensive guide that explains the distinction between normative and positive analysis.

The document from GitLab [14] provides specific guidance that opposes the implementation methods previously described. The guide explains the token hierarchy by defining its three essential elements.

- Scope Minimisation: Use the least scope possible; use the API only if necessary.
- Service Accounts: For enterprise automation, the guide recommends using "Service Accounts" to remove automation from user licenses and enforce rotation policies.
- Deploy Tokens: For read-only access (registry access), the guide recommends "Deploy Tokens", which cannot access the API.

Despite these clear guidelines, the Operational Reality (Positive Analysis) diverges from the Normative Standard due to friction:

- Deploy Tokens are read-only and cannot trigger pipelines (rendering them useless for CI/CD orchestration).
- Service Accounts are an Enterprise-tier feature and require complex API-based management, making them inaccessible to many teams.
- Project Access Tokens require creating a new token for every single relationship.

Therefore, despite the "Ultimate Guide's" recommendations, the Group Access Token remains the de facto standard for developers, directly fueling the "Two-Project Anomaly" our research detects.

4. Methodology

To evaluate the possibility of identifying "Scope-Usage Asymmetries" without Administrator privileges, we designed a Proof of Concept (PoC) auditing tool. The prototype follows a "Bottom-Up" Policy-as-Code model for its operation. Instead of relying on a central database of all tokens, the PoC simulates the standard developer or project maintainer, scanning accessible configurations to map the architecture of trust.

The research methodology includes three distinct stages: Variable Discovery, Identity Resolution and Heuristic Evaluation.

Phase 1: Variable Discovery. Since GitLab tokens used for automation are primarily stored as CI/CD Variables, the PoC initiates a scan across all projects accessible to the auditor.

- Target: The CI/CD variable configurations of each repository.
- Extraction: The script analyses variable values, searching for standard token prefixes (e.g., the glpat - prefix used for GitLab Personal Access Tokens).
- Output: A raw list of discovered secret strings and the specific projects they are stored in.

Phase 2: Identity & Scope Resolution. Finding a token string does not immediately reveal its power. Because GitLab hashes token secrets in its database, the PoC must actively resolve the identity of each extracted string. For every token discovered in Phase 1, the script performs an authentication check against the GitLab API:

1. Authentication: The PoC temporarily uses the discovered token to query needed endpoints.

2. Metadata Extraction: The API responds with the token's underlying identity. The script records:

- Token Type: Is it a Personal Access Token, a Group Access Token, or a Project Access Token?
- Scope: Does it have the dangerous api scope, or is it restricted to *r*ead_repository?
- Blast Radius: How dangerous is it? How many projects can this specific token access?

Phase 3: Analysis. The final phase applies a strict logic gate to detect Privilege Sprawl, specifically looking for the “Two-Project Anomaly”. The PoC evaluates the mapped data against the following heuristic:

1. Input: A discovered token found in the CI/CD variables of Project A.
2. Measurement: Compare the token's usage with its power.
3. The "Sprawl" Logic:

- If Authority and Utility match - Valid Configuration. The token is a Project Access Token restricted to the project it serves. No sprawl detected.
- If Authority is bigger than Utility: Violation**.** The token is revealed to be a broad Group Access Token or a highly privileged Personal Access Token. It holds a lot of power, but is used operationally for a single point-to-point connection.

Audit Output and Remediation Guidance. Rather than automatically altering CI/CD pipelines — which carries the risk of disrupting production workflows — the PoC methodology is designed to act as an advisory system. Upon completing the heuristic evaluation, the tool generates a structured audit report prioritising the flagged violations based on the principle of Least Privilege.

The output consists of two primary components:

1. The Utilisation Map: A structured representation showing exactly where high-privilege tokens are stored versus where they are actually consumed.
2. Actionable Recommendations: For every flagged anomaly, the system prescribes a remediation path for the development team.

Example of Remediation Guidance:

- Detection: The audit report flags a “Two-Project Anomaly”. A broad Group Access Token is found stored in Project A's variables, but it is only being used to trigger a downstream pipeline in Project B.
- Suggested Action: The PoC report advises the repository maintainers to:
    a. Create a heavily restricted Project Access Token (or a Pipeline Trigger Token) exclusively within Project B.
    b. Update Project A's CI/CD variables to use this new, narrowly scoped token.
    c. Safely revoke the original, over-privileged Group Token.
- Resulting Impact: By following the generated guidance, the development team effectively reduces the token's “Blast Radius” from the entire namespace down to a single repository, closing the Privilege Sprawl gap.

Theoretical Alternative: The "Top-Down" Enterprise Audit. It is important to note that this PoC was constrained to a “Bottom-Up” approach due to the lack of Global Administrator privileges.

For mature enterprise environments where administrators or the security team possess instance-wide rights, we propose a theoretical “Top-Down” methodology. In this alternative, the system would first query the Admin API to inventory all tokens across the entire organisation and subsequently scan all projects to identify their usage. While the Top-Down approach would additionally identify "Dormant" (completely unused) tokens, our Bottom-Up PoC effectively proves that significant Privilege Sprawl can be detected and mitigated even by teams with localised, non-administrative access.

5. Results

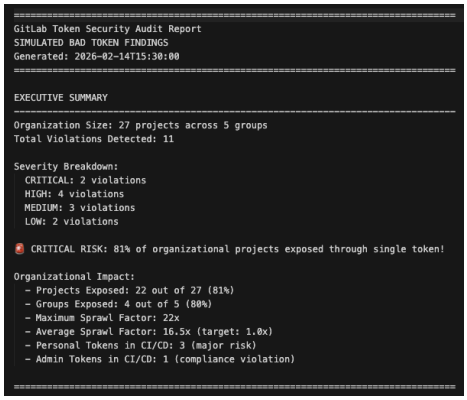

Figure 2: GitLab Token Security Audit Report

To put this methodology to the test, we ran the PoC against a simulated enterprise environment comprising 27 core repositories across 5 groups.

Baseline Audit. Across the 27 scanned projects, we found just 5 unique tokens causing 11 distinct security violations. The violations were categorised by severity based on their potential Blast Radius.

One of the most alarming findings was the discovery of 3 Personal Access Tokens hardcoded into CI/CD variables. Even worse, the audit detected an administrative token belonging to the user running the auth-service deployment. Identity resolution confirmed that this single token had 100% access to all 5 groups. It is a massive compliance violation and a business continuity risk. The day this developer leaves the company and the account is deactivated, production deployments break instantly.

Detection of Scope-Usage Asymmetry. The heuristic evaluation showed the environment had 0% compliance with Least Privilege prior to the audit.

1. Average Sprawl Factor: The environment exhibited an average sprawl factor of 16.5, meaning CI/CD tokens generally had access to 16.5 times more projects than they needed to do their jobs.
2. Dormant Tokens: We also found 2 "Inactive Tokens" just sitting in legacy testing repositories, highlighting a dangerous buildup of forgotten "Identity Debt".

Case Study. We found a Group Access Token stored in the GITLAB_DEPLOY_TOKEN variable. It was only being used by one service, but identity resolution showed it had api and write_repository scopes tied to 4 different groups.

This token could access 22 out of the 27 total projects in the organisation. It effectively had a 22 Sprawl Factor. If a hacker grabbed that one token, they would instantly have write access to 81% of the company's intellectual property, including infrastructure definitions and customer databases.

Remediation and Impact. Following the tool's advice, we modelled out the remediation (replacing broad Group and Personal tokens with tightly scoped Project Access Tokens). The projected impact was massive:

Table 2.

Implemented improvements

Metric	Pre-Audit Baseline	Post-Remediation (Projected)	Improvement
Organizational Exposure	22 Projects (81%)	6 Projects (22%)	73% Reduction
Maximum Sprawl Factor	22	1.2	95% Improvement
Average Sprawl Factor	16.5	~1.0	Optimized
Personal Tokens in CI/CD	3 (Including 1	0	100% Eliminated

	Admin)		
Least Privilege Compliance	0%	100%	100% Improved

6. Discussion and Conclusion

The research investigated two essential DevSecOps governance problems: the unnoticed growth of “Identity Debt” and the unbalanced management of machine identity permissions, known as “Scope-Usage Asymmetry”. The industry has invested significant resources in source code scanning for leaked secrets, but it lacks effective automated systems that check running secrets for structural soundness.

Interpretation of Findings. The results of the simulated audit clearly validate the core problem we set out to solve: developers are leaning entirely on the “Path of Least Resistance”. The data shows that modern CI/CD token management systems experience two distinct operational breakdowns that operate independently of each other.

1. The Inevitability of Privilege Sprawl: The discovery of an average Sprawl Factor of 16.5 shows that permission bloat will occur in every organisation that does not use automated guardrails to control its permissions. The Group Token we found perfectly illustrates this — just initiating a deployment operation, it has access to 22 projects, exposing 81% of the organisation's code to a single security vulnerability. The top-down inheritance model of GitLab creates an enterprise-wide attack vector from what was originally localised pipeline automation.
2. The "Human Dependency" Risk in Automation: The audit discovered three Personal Access Tokens (PATs) which used CI/CD variables, and one of them belonged to an instance Administrator. The discovery requires organisations to redirect their security efforts toward both operational continuity maintenance and regulatory compliance and protection. The organisation will experience pipeline failures because departing administrators fail to establish the required documentation, which causes this problem.
3. The PoC results showed that the Bottom-Up method produces good results. Development teams can conduct their own perimeter audits by using native variable extraction and identity resolution methods, which do not require them to wait for centralised IT support or Global Administrator access.

Limitations. While the PoC successfully detected vulnerabilities and prescribed remediation steps, we acknowledge certain limitations to this methodology:

1. Scope of Extraction: The current methodology relies exclusively on scanning GitLab's native CI/CD variables. It does not map tokens that are hardcoded directly into source code, compiled into binaries, or fetched dynamically at runtime from external secret vaults (e.g., HashiCorp Vault or AWS Secrets Manager).
2. Point-in-Time Auditing: The PoC currently operates as an on-demand scanning tool. While it executes rapidly, it still acts as a reactive measure. It finds the sprawl after it has happened, rather than preventing the initial creation of a sprawling token.

Future Work. The logical next step in this research is to shift from an advising tool to a “Continuous Enforcement” model. Future variants of it could be integrated directly into pipelines. For example, if a developer configures a pipeline using an overprivileged Personal Access Token, a Policy-as-Code guardrail could instantly detect the anomaly and fail the job before any code is deployed, suggesting the team generate an appropriate token instead.

Conclusion. As DevSecOps ecosystems grow more complex, the industry must understand that the Identity Perimeter is rapidly overtaking the traditional Network Perimeter. A static, wildly over-privileged secret sitting in a CI/CD variable isn't just a minor misconfiguration — it is a time bomb holding administrative keys to dozens of repositories, just waiting for an attacker to find it.

This paper set out to prove that we don't have to accept this level of risk unquestioningly. We successfully demonstrated that Policy-as-Code can be used to audit the actual architecture of trust inside a CI/CD environment. By dragging the “Two-Project Anomaly” out of the shadows, rooting out risky human-dependent automation, and mechanically enforcing the principle of Least Privilege, organizations can drastically shrink their attack surface and eliminate dangerous administrator dependencies.

The fundamental security question for DevSecOps teams is no longer just "Is your token leaked?" but rather, "Is your token's blast radius mathematically justified?”

Declaration on Generative AI

While preparing this work, the authors used the AI programs Grammarly Pro to correct text grammar and Plag AI to search for possible plagiarism. After using this tool, the authors reviewed and edited the content as needed and took full responsibility for the publication’s content.

References

[1] A. Mishra, DevSecOps-Driven Security Framework for CI/CD Pipeline Risk Mitigation, Int. J. Comput. Eng., 2025. doi:10.47941/ijce.3047.

[2] L. Prates, R. Pereira, DevSecOps practices and tools. Int. J. Inf. Secur. 24, 11 (2025). doi:10.1007/s10207-024-00914-z.

[3] C. Biringa, G. Kul, Detecting Hard-Coded Credentials in Software Repositories via LLMs doi:10.48550/arXiv.2506.13090.

[4] Y. Zhuravchak, et al., Security Regression Visualization in CI/CD Using Trivy and GitHub Actions, CEUR Workshop Proc., vol. 4042, 2025.

[5] NIST, Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines NIST Special Publication 800-204D, 2024.

[6] OWASP Foundation, Software Supply Chain Security Cheat Sheet OWASP Cheat Sheet Series, 2024.

[7] Sysdig, CI/CD Security: Securing Your CI/CD Pipeline Sysdig Learn Cloud Native, 2024.

[8] Y. Liu, F. Li, C. Sun, Dynamic token encryption for preventing permission leakage in serverless architectures, PeerJ Comput. Sci., 2025. doi:10.7717/peerj-cs.3029.

[9] M. Rahman, S. Ahmed, Z. Wahab, Secret Breach Detection in Source Code with Large Language Models, 2025. doi:10.48550/arXiv.2504.18784.

[10] Y. Yang, J. Wang, P. Liu, A. Fu, Y. Zhang, Uncovering Access Token Security Flaws in Multiuser Scenario of Smart Home Platforms, IEEE Internet Things J., 2024.

[11] V. Gullapalli, Securing GitHub Enterprise: Access Control, API Governance, and Collaboration Boundaries at Scale, J. Inf. Syst. Eng. Manag., 10(58s), 2025 doi:10.52783/jisem.v10i58s.12634.

[12] K. Kamaluddin, Security Policy Enforcement and Behavioral Threat Detection in DevSecOps Pipelines, Eur. J. Technol., 6(4), 10-30, 2022. doi.org/10.47672/ejt.2723.

[13] Snyk, Enabling policy as code (PaC) with OPA and Rego, Snyk Eng., 2024.

[14] GitLab, The Ultimate Guide to Token Management at GitLab, GitLab Official Blog, 2024.

[15] GitLab, GitLab CI/CD variables documentation, GitLab Official Documentation, 2024.

[16] GitLab, Security: Access Tokens (Personal, Project, and Group), GitLab Official Documentation, 2024.

[17] Open Policy Agent (OPA), Policy Language (Rego) and Policy-as-Code, Styra/Cloud Native Computing Foundation, 2024.

[18] Amazon — Best practices for CI/CD pipelines.

[19] O. Harasymchuk, et al., Intelligent Cyber Defence Systems: Detection of Ransomware and Protection of Wireless Networks Based on Artificial Intelligence Technologies, Kharkiv: Technology Center PC, 2025, 82 p., doi:10.15587/978-617-8360-22-1.
[20] O. Harasymchuk, et al. Modern Methods of Ensuring Information Protection in Cybersecurity Systems Using Artificial Intelligence and Blockchain Technology. Kharkiv: Technology Center PC, 2025, 132 p., doi:10.15587/978-617-8360-12-2.
[21] O. Vakhula, I. Opirskyy, O. Mykhaylova, Research on Security Challenges in Cloud Environments and Solutions based on the "security-as-Code" Approach, CEUR Workshop Proc., 3550 (2023) 55-69.
[22] Wiz, What is Policy as Code (PaC)?, Wiz Cloud Security Academy, 2024.