2025 State of Application Risk

An ASPM view of the security of software factories

LEGIT

Contents

Executive Summary

The Legit research team analyzed the data uncovered by the Legit Application Security Posture Management (ASPM) platform over the past 18 months.

Because our platform discovers and visualizes all aspects of both applications and the software factory producing these assets, plus all security controls and gaps, Legit is in a unique position to offer a snapshot of common areas of AppSec posture risk.

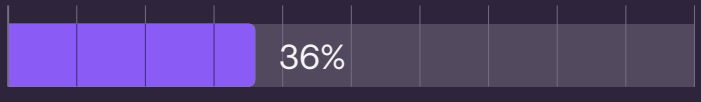
KEY DATA POINTS INCLUDE:

The state of organizations' application risk is troubling

Organizations that have high or critical risks in their development environments

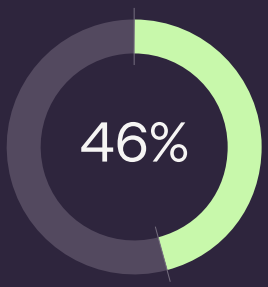
Secrets exposure is pervasive and widespread

Organizations that have high or critical exposed secrets (secret issues rated by Legit as high or critical severity based on their exploitability and impact)



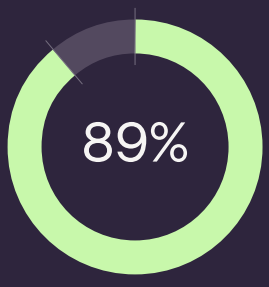
Secrets found outside of source code (tickets, logs, artifacts, etc.)

AI is emerging as a threat



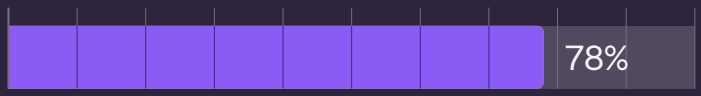
Organizations that use AI models in source code in a risky way

Misconfigurations are rampant

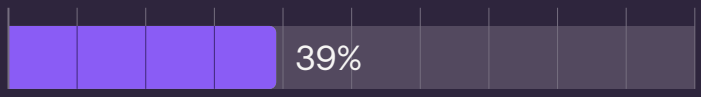


Organizations with pipeline misconfiguration issues

AppSec scanning is inefficient

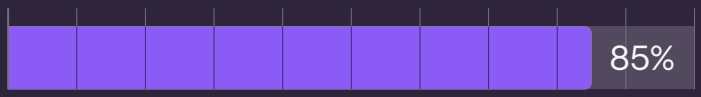


Organizations with duplicate SCA scanners



Organizations with duplicate SAST scanners

Development teams are over-permissioned



Organizations with least privilege violations

Repositories across organizations in which external collaborators with admin privileges can access pipelines with critical and high misconfigurations

This report is based on our initial assessments of enterprises' software factories in the past 18 months.

The data represents a wide range of industries and company sizes – from organizations with fewer than 100 developers to those with thousands. Some of the enterprises had hundreds of code repositories, some had tens of thousands.

Introduction

Application security is not just about finding vulnerabilities in source code anymore.

With software development that is faster, more automated, more dynamic, and highly reliant on third parties, new opportunities to introduce risk abound. From vulnerabilities in applications to misconfigured build servers, exposed secrets in Jira tickets, and more, the attack surface has grown and diversified.

In addition, identifying application risk and remediating it are now complicated by lack of visibility into the full software factory, and of correlation among types of risk — such as cloud, app, supply chain.

Effective AppSec today requires a robust foundation of visibility and insights.

The Legit ASPM platform discovers and visualizes all aspects of applications and the associated software factory, including assets, dependencies, misconfigurations, and shadow IT. It also provides the context teams need to correlate, dedupe, and prioritize vulnerabilities across the SDLC, and then automates triage and remediation.

In this 2025 State of Application Risk report, we leveraged our powerful visibility capabilities to highlight our 2024 risk findings.

Our data set gives a snapshot of where application security risk lives in the modern development environment. Read on to get answers to questions like:

- → What are the most common toxic combinations increasing enterprises' application risk?
- → Where are secrets most often exposed?
- → What types of SDLC misconfigurations are most common, and which are the most risky?
- → What are common AppSec testing inefficiencies?

Ultimately, we hope you'll use this report to understand where the greatest application risks now lie, and to prioritize your own application security efforts.

In this section, we provide a high-level snapshot of the state of application risk, courtesy of the Legit ASPM platform's view of the entire software development lifecycle — from assets to dependencies, misconfigurations, GenAI code, and shadow IT — plus its security controls and gaps.

Overall State of Application Risk

Extent of Application Risk

Bottom line: There is a significant amount of risk throughout the application development process, including exposed secrets, risky misconfigurations, and dangerous vulnerabilities.

> We found that 100% of organizations have high or critical application risks in their environments.

On a more positive note, the percentages decrease for those with risks that are public (Figure 1).

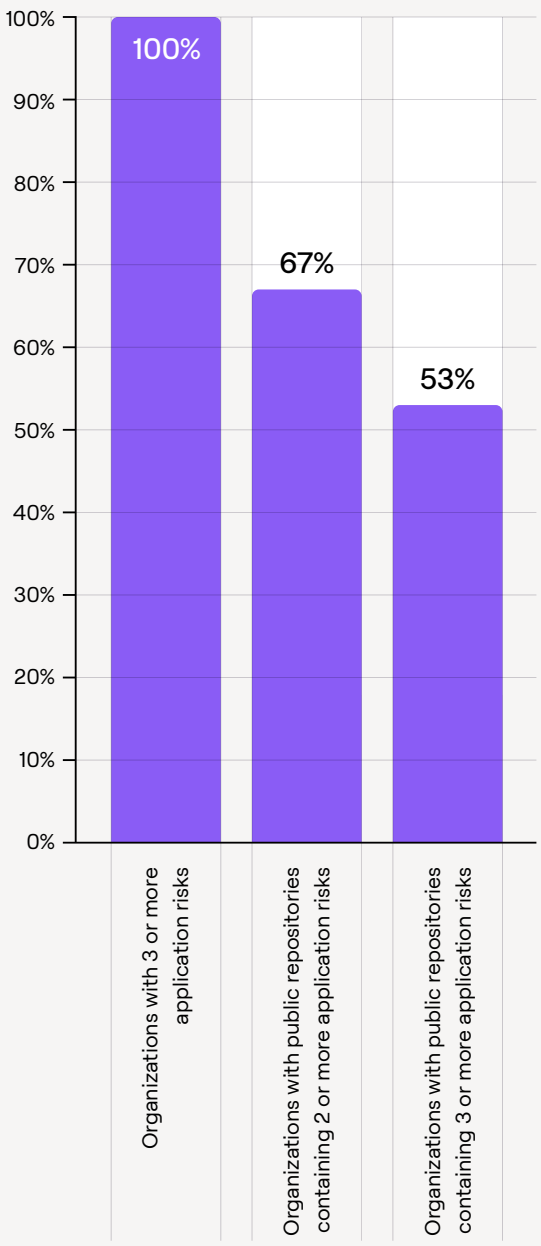


Figure 1: Extent of application risk

Adherence to the Application Security Requirements of Cybersecurity Regulations

When we looked at adherence to the application security requirements of cybersecurity regulations, the results were mixed. For instance, when we looked at average rate of regulation compliance per organization, there was 42% compliance with the NIST Secure Software Development Framework (SSDF), but 76% with ISO 27001 (Table 1).

It's not a coincidence that the regulations with higher compliance percentages are also those that are measured by auditors. However, it's noteworthy that even the auditors couldn't incent 100% compliance, and the regulations not measured by auditors are hovering around the 50% mark.

Clearly, organizations have some work to do to improve compliance with cybersecurity regulations.

Framework	Average Rate of Compliance
CISA	47%
FedRamp	64%
ISO	76%
PCI DSS	62%
SOC2	71%
SSDF	42%
OWASP CI/CD	33%

Table 1: Adherence to application security requirements of cybersecurity regulations

AppSec Testing Inefficiencies

The extent of the risk we have uncovered in this report results in part from an inefficient and ineffective process for assessing risk. Not only do application security testing tools like static application security testing (SAST) and software composition analysis (SCA) look exclusively for known vulnerabilities in source code, but they also return results without the context needed to make informed decisions about remediation.

In addition, we have also found that a significant number of organizations have duplicate scanners producing duplicate results. As shown in Figure 2, a whopping 78% have duplicate SCA scanners; 39% have duplicate SAST scanners.

The high SCA duplication numbers are likely a reflection of developers in different business units downloading different free versions of SCA scanners, like GitHub Dependabot or OWASP Dependency-Check. M&A would exacerbate this redundancy and overlap.

There are fewer free versions of SAST, and it is more challenging solution to implement, so the 39% SAST duplication number is most likely related to M&A. Teams get merged, each with their existing scanners, and no one is keeping track of what scanners are in use where.

The problem with the duplication of both these tools is that teams will end up with duplicate vulnerability findings, and duplicate or (often) contradictory remediation advice. Some AppSec scanners offer more solid, robust remediation advice based on their ability to view dependencies and connections, and some offer more basic, less custom remediation advice. When one scanner is telling the team that a finding is not worth remediating, and another scanner is giving them details on how to remediate it, confusion, and likely inaction, ensue.

In the end, you have security and development teams overwhelmed with findings and advice — some duplicated, some contradictory — and all leading to a mountain of security debt.

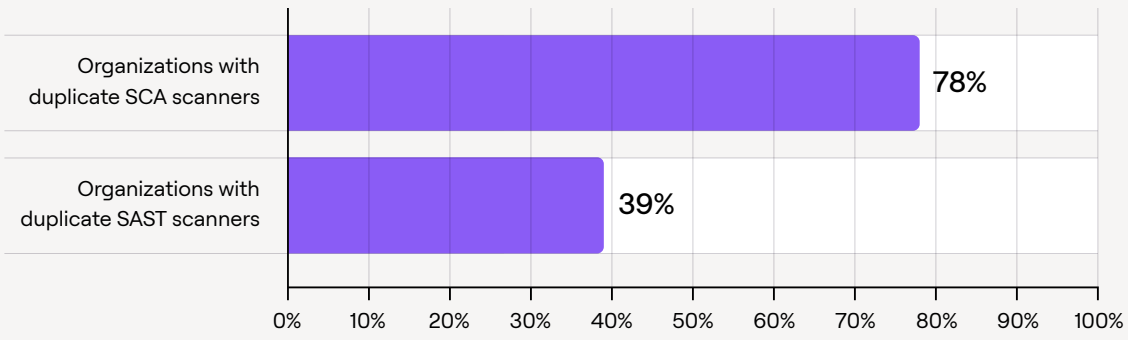


Figure 2: Duplicate AppSec testing tools

In the following pages, we highlight and detail four application risks we see frequently:

01. Secrets Exposure
02. AI Risks
03. SDLC Misconfigurations
04. Software Supply Chain Issues

Detailing the Most Common Application Risks

01 Secrets Exposure

Secrets are extremely pervasive in software development environments, and their exposure is one of the most common risks unearthed by the Legit platform. This is troubling because secrets are often the first foothold that attackers leverage to mount much larger attacks.

The types of exposed secrets we find most often include:

- → Cloud keys (AWS/GCP)
- → GitHub personal access tokens and CI/CD server keys (Jenkins, ADO)
- → PII, such as Social Security and credit card numbers

Why are exposed secrets so common in software development environments today?

Partly because modern development requires lots of different tool sets that need to integrate with each other, and modern apps require hundreds of secrets to function (API keys, third-party tokens, cloud credentials, etc.).

At the same time, developers are pushed to innovate and develop code as fast as possible, frequently leading to shortcuts intended to drive efficiency and speed. One of those shortcuts is using secrets in development to accelerate testing and QA. The problem is that it's very easy for these secrets to remain exposed.

And exposed secrets are not just a hypothetical risk. In a recent survey of 350 IT and cybersecurity professionals and application developers, TechTarget's Enterprise Strategy Group (ESG) found that the top cybersecurity incident (related to internally developed cloud-native apps) experienced by survey respondents in the previous 12 months was secrets stolen from a source code repository.

In addition, recent breaches, such as those at Sisense and Toyota, were caused by secrets exposure.

Secrets Exposure Data

Not surprisingly (but disconcerting), we found exposed secrets in 100% of organizations. The numbers drop to 53% for exposed secrets in public assets, and 35% for exposed secrets deployed to the cloud, but they're still alarming numbers (Figure 3 on the following page).

Secrets exposure is clearly a huge problem. How huge exactly? We found, on average, 33% of repos within organizations contain exposed secrets, and an average of 17% of repos with high or critical secrets. Within organizations, the average number of repositories with the toxic combination of exposed secrets and branch protection issues is about 1 out of 3 (30%).

Figure 3: Extent of secrets exposure

*Secret issues rated by Legit as high or critical severity based on their exploitability and impact

DEFINITIONS

Active vs. inactive development

Distinguishing between a security issue in active or inactive development is an important way to prioritize risk. Projects in active development will always be more business critical and should be addressed first.

Inactive development refers to a project that is archived, or hasn't been changed for more than a year. Misconfigurations in inactive development would be an especially low-priority fix since you need activity to exploit a misconfiguration.

Public assets

A public asset refers to a repository anyone can search for and find on the Internet. If you have a secret on a public repository, it would be accessible to anyone on the Internet. Security findings in public assets should be prioritized.

Deployed to cloud

This refers to an application hosted on some form of cloud infrastructure, even a private cloud. This is likely accessible through the Internet, and also could be compounded by misconfigurations inside the cloud environment. Security findings deployed to cloud should be prioritized.

External collaborators

This term refers to developers who are not part of the organization, such as contractors. The XZ backdoor was a good example of the dangers of external collaborators. Malicious external collaborators worked on an open source project for years without issue, then mounted an attack. Lax security combined with external collaborator access creates a risky situation.

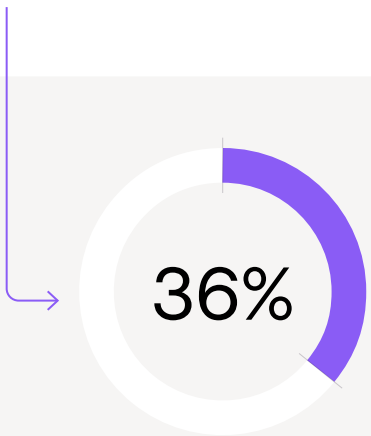
Toxic combination

This refers to the ability to tie different types of risks together in a way that creates an attack path or an elevated combined risk.

We regularly find exposed secrets in source code, which can be accessed by any user with access to the repository.

But increasingly, we are finding exposed secrets in many other places as well — like yaml files, build logs, containers, bash scripts, artifacts, containers, Jira, Confluence, Slack, and more.

In fact, 36% of the secrets we found were outside source code.



Secrets found in places like knowledge pages (i.e., Confluence), ticketing systems (i.e., Jira), and containers

Figure 4: How many secrets are found outside of source code?

Preventing Exposed Secrets

To prevent exposed secrets, focus first on SaaS services keys (e.g., AWS access keys), since if code is leaked, credentials to SaaS services are immediately usable if they are valid, whereas internal credentials require attackers to also have network connectivity.

Our recommended best practices include:

- Avoid committing secrets to any Git repository. Once in the Git history, remediation steps are lengthy.
  - Avoid git add * commands.
  - Name sensitive files in .gitignore.
  - Don't rely on code reviews to discover secrets.
  - Use automated secrets scanning on repositories.
  - Use CLI or pre-hook commit tools when able to catch secrets before they get to your Git repository.
- Change the source code to not rely on hard-coded secrets by using a password manager, environment variable, etc. Then revoke the sensitive data.
- Use encryption to store secrets within Git repositories.
- Use local environment variables, when feasible.
- Use secrets management as a service solutions (e.g., Hashicorp Vault, CyberArk Conjure, etc.).
- Avoid secrets within build logs and sharing secrets via messaging services.
- Reduce AuthZ and Admin credentials to least privileged.

02 AI Risks

GenAI has recently emerged as an additional risk we uncover. Although it gives developers an easier way to produce code at scale, it also adds risk.

We often discover that security teams first don't know where AI is in use, and then find out it's used in a location that isn't configured securely. For instance, a developer is using AI and generating code on a repository that doesn't have a code review step. This could, for instance, allow for licensed code to enter the product, exposing the organization to legal or copyright issues.

We also often detect low-reputation LLMs in use, which could contain malicious code or payloads, or exfiltrate data sent to them.

Preventing AI Risk

Security teams need to know (and typically don't) who's using AI, where they are using it, and if there are guardrails in place in those areas.

Best practices include:

- → Threat modeling the impact of AI-specific threats
- → Beyond functionality and performance, considering security when selecting AI models
- → Employing tools to get visibility into AI use across your development environment
- → Protecting AI models from direct or indirect access

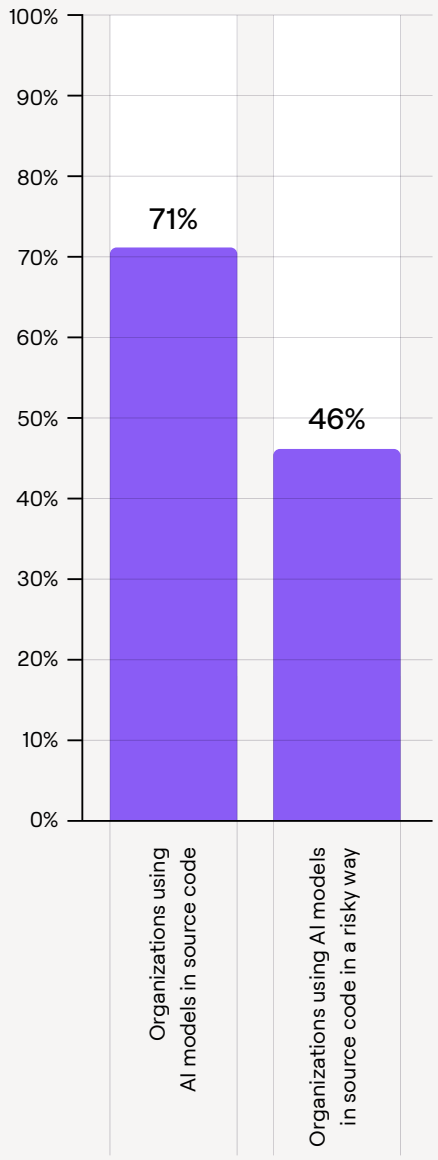


Figure 5: Extent of AI risk

03 SDLC Misconfigurations

Misconfigured SDLC assets, such as SCMs, build servers, and artifact repositories, provide an opportunity for threat actors to gain access to systems and then move laterally within an organization. These assets all provide configuration mechanisms to prevent this, but they need to be used and continually monitored for proper use.

That use and monitoring are clearly not consistent, as seen in the stats below. Our data finds that 100% of organizations have SDLC misconfiguration issues, and some of the most common are shown in Table 2.

Legit Policies	Percent of Repos Violating this Policy
Unsigned commits are allowed in the repository A signed commit verifies the authenticity and origin of a code change. Allowing only signed commits makes it harder for malicious actors to tamper with the source code stored in the repository, and helps ensure all updates to the code are only done by approved individuals.	56%
Repository doesn't have defined default reviewers When you add default reviewers, they are automatically assigned to review pull requests.	33%

Table 2: Misconfiguration policy violations (continued on next page)

(Table 2 continued)

Legit Policies	Percent of Repos Violating this Policy
Default repository branch is not protected A protected branch is a branch that contains restrictions on who can modify it and how. Without this protection, one compromised account could give an attacker the ability to cause widespread destruction. For instance, an attacker could push a malicious backdoor if they hacked a privileged user who can push code directly to the main branch without a proper default branch protection mechanism.	48%
Code review by at least one reviewer is not enforced	48%
Code review by at least two reviewers is not enforced	36%
It is possible to merge code without the approval of all reviewers	34%
Jenkins server has non-pipeline jobs Non pipeline jobs are prone to bypassing security guardrails, are inconsistent, and are difficult to reproduce, raising security concerns over the created artifacts, as opposed to Configuration-as-Code pipelines.	33%

These policies require that pull requests are reviewed, at varying levels. This ensures the code is reviewed before it's merged into the main codebase.

In the following pages, we highlight several SDLC misconfigurations that we encounter frequently and that create significant risk:

→ Pipeline misconfigurations

→ Branch protection issues

→ Developer permission issues

Pipeline Misconfigurations

A pipeline misconfiguration is one in a pipeline platform such as Jenkins, GitHub Actions, etc. When we first start working with an enterprise, we often discover misconfigured build servers in their environment. This is a common problem, but also one that creates significant vulnerabilities. Build systems are essentially automated, implicitly trusted pathways straight to the cloud, yet most aren't treated as critical from a security perspective. In many cases, these systems — like Jenkins, for example — are misconfigured or otherwise vulnerable and unpatched.

> Working with one large enterprise, the Legit team found an exposed Jenkins server with access to the public Internet. The Legit research team was able to access proprietary code via the Internet through that Jenkins server.

We also often find Jenkins servers that have unnecessary access to many S3 buckets. An attacker who breaches a server with this kind of access has a treasure trove of data to pursue.

Figure 6 highlights our findings on pipeline misconfigurations. 89% had pipeline misconfiguration issues, with 64% of those in active development. Also noteworthy that 25% have the toxic combination of external collaborators in repos with pipeline misconfigurations.

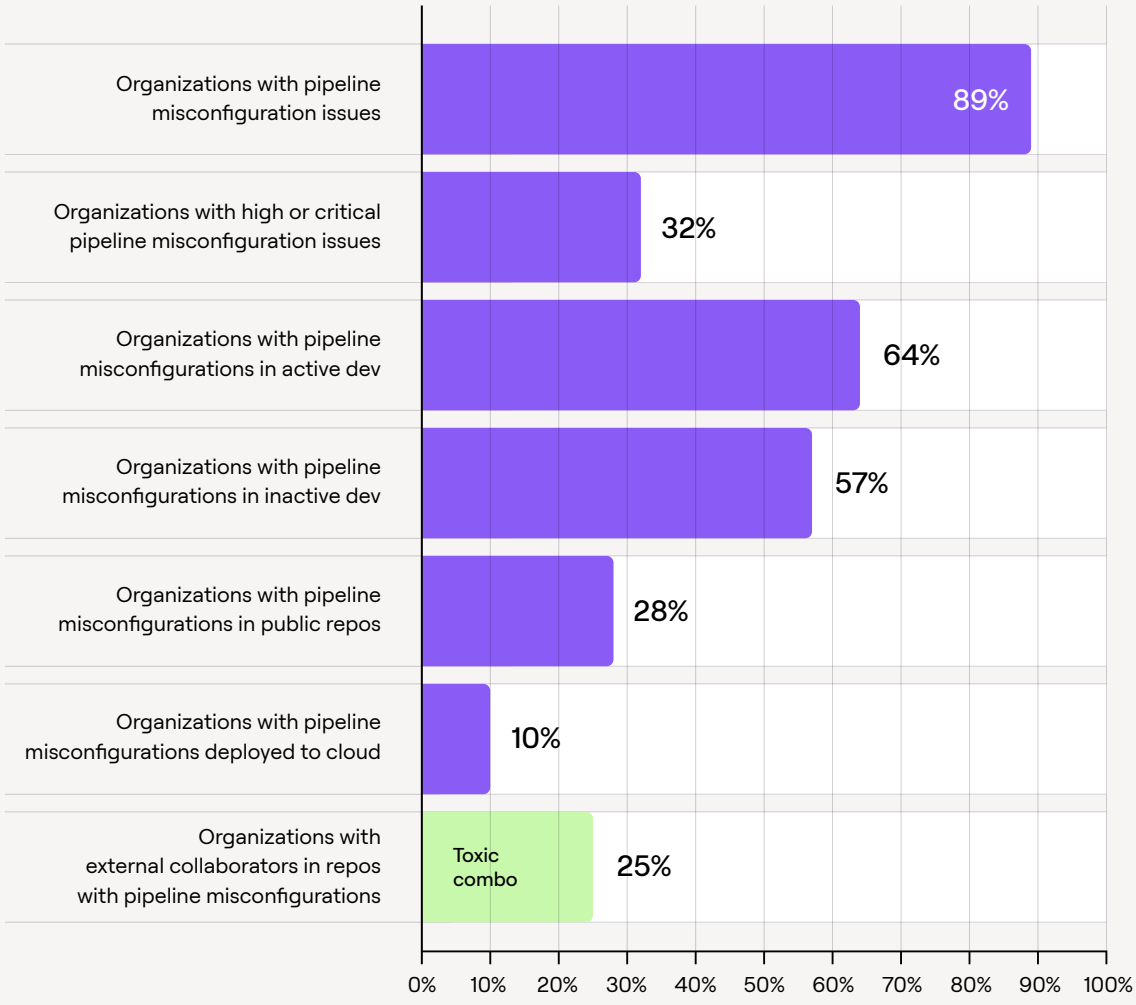
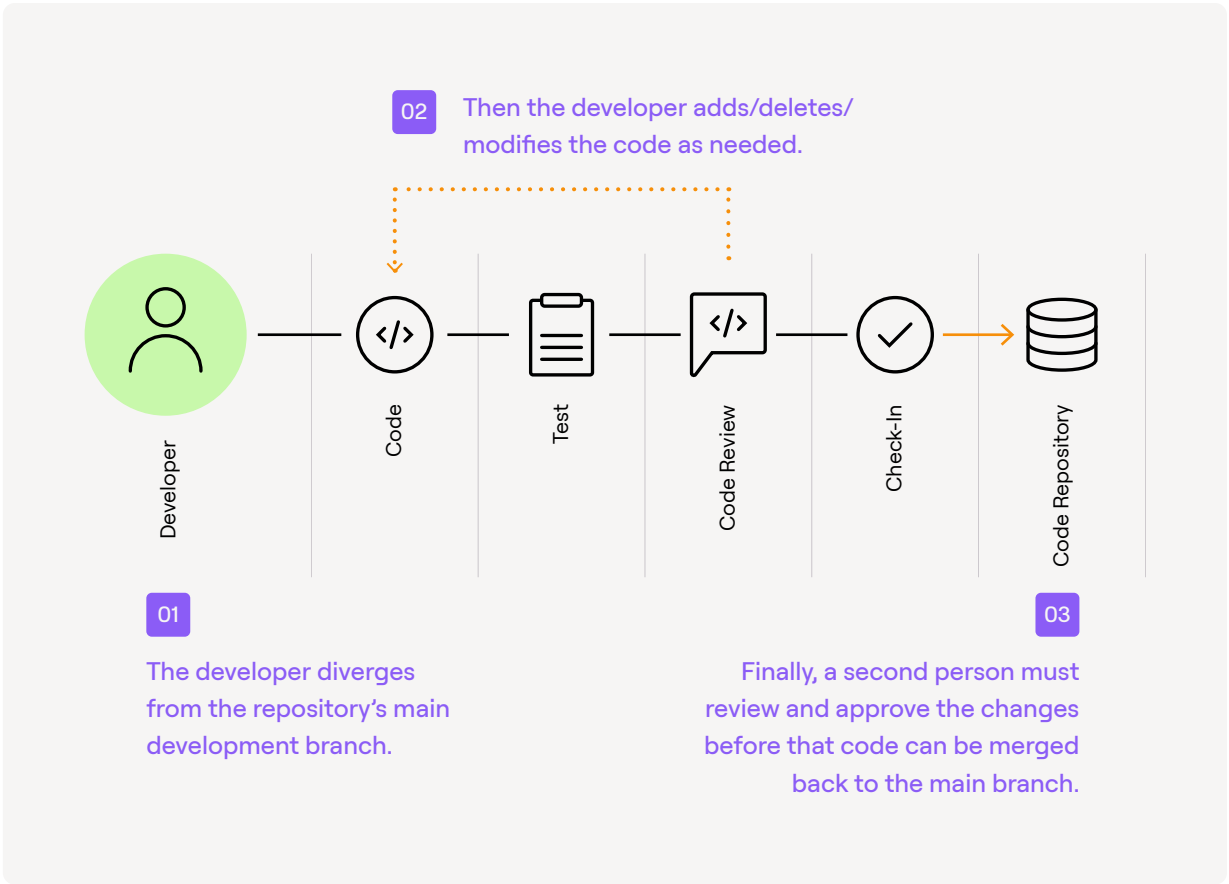


Figure 6: Extent of pipeline misconfigurations

Branch Protection Issues

A typical development workflow looks like the following:



To support this common workflow, GitHub introduced the concept of Protected Branches. A protected branch is a branch that contains restrictions on who can modify it and how. Without this protection, one compromised account could give an attacker the ability to cause widespread destruction.

For instance, an attacker could push a malicious backdoor if they hacked a privileged user who can push code directly to the main branch without a proper default branch protection mechanism.

100% (!) of the organizations in our dataset had branch protection issues. Of those, 89% had high or critical branch protection issues in active development.

Finally, we found that almost one in five repos within each organization had the toxic combination of developers using GenAI tools plus a lack of branch protection and code review. See Figure 7 for more details.

We found branch protection issues in 100% of organizations.

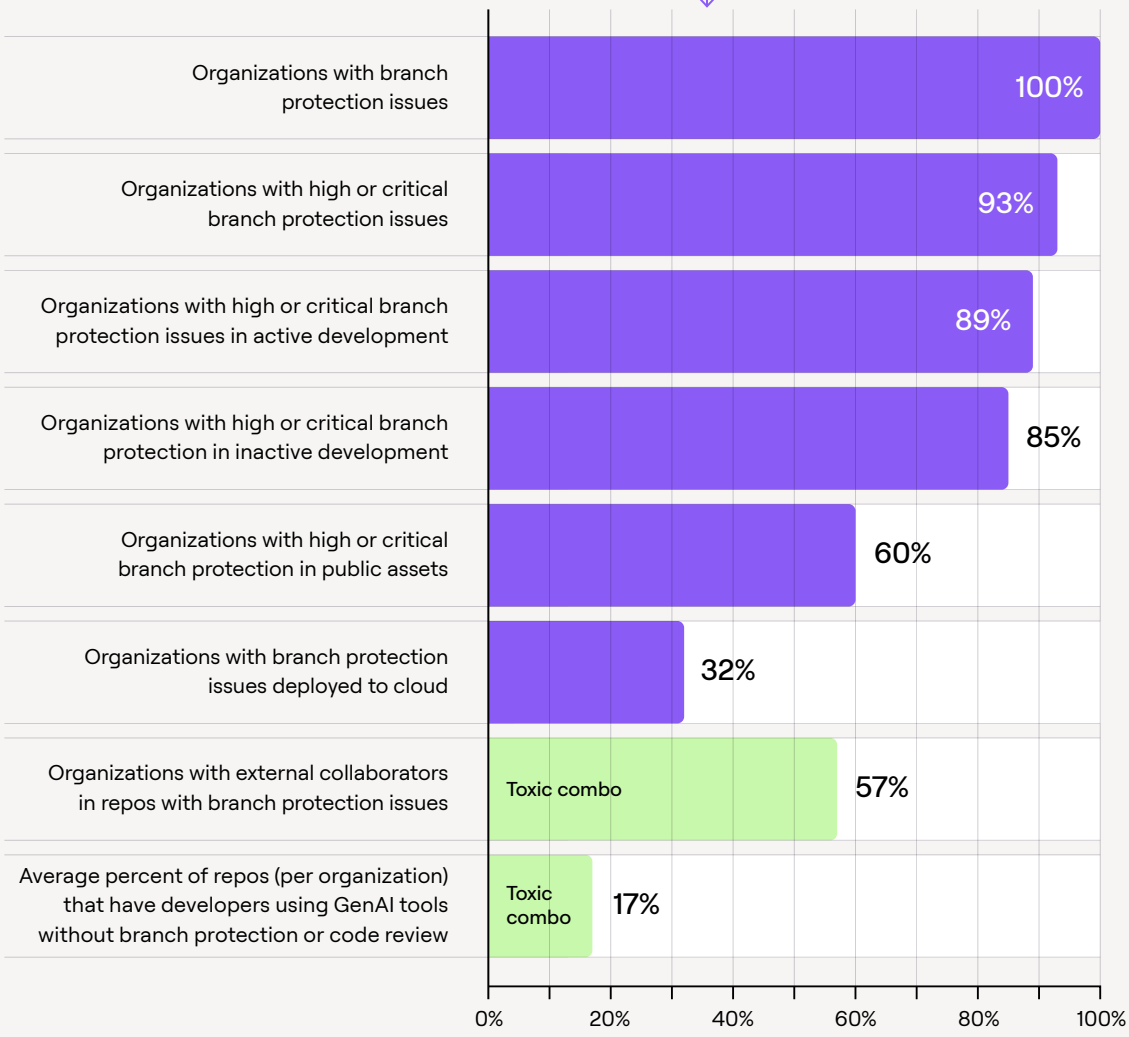


Figure 7: Extent of branch protection issues

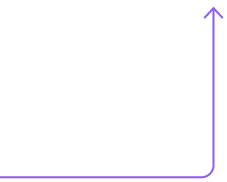
PREVENTING BRANCH PROTECTION ISSUES

Enable branch protection for your default main branch. Once enabled, you'll be able to enforce the following GitHub repository best practices.

The GitHub best practices include:

- Prevent force pushes and deletions.
- Require a pull request before merging.
  - Define a minimal number of required approvals before merging.
  - Dismiss stale pull request approvals when new commits are pushed.
  - Require review from code owners (defined in a dedicated file inside the repository).
  - Restrict who can dismiss pull request reviews.
- Require a branch to be up to date and pass predefined status checks before merging.
- Require conversation resolution before merging.
- Require signed commits.
- Require deployments to succeed before merging.
- Enforce restrictions for administrators as well.

Keep in mind: It's important to have continual monitoring and verification of configurations. We recently partnered with a security team at a large enterprise; the team would enable branch protection, and then their development team would disable it. Obviously, developer/security communication and collaboration go a long way here, but also continual monitoring of a constantly changing environment.



Developer Permissions Issues

Mishandled developer permissions is a pervasive issue. When we first start working with enterprises, we almost always find overly and/or incorrectly permissioned development teams.

The LastPass attack was a good reminder that it only takes one compromised account for a malicious actor to gain access to the entire SDLC.

STALE COLLABORATORS

A "stale collaborator" is a developer account that hasn't been used in six months, but still has active permissions. For instance, if a developer leaves an organization and his or her permissions are not revoked, that would become a stale collaborator after six months.

Bottom line: Stale collaborators expand your attack surface unnecessarily. Any developers' credentials could be compromised at any time, so limiting the amount of valid credentials to only those absolutely necessary shrinks your attack surface.

We found 85% of organizations had stale collaborators, and 64% had stale collaborators in active development (Figure 8).

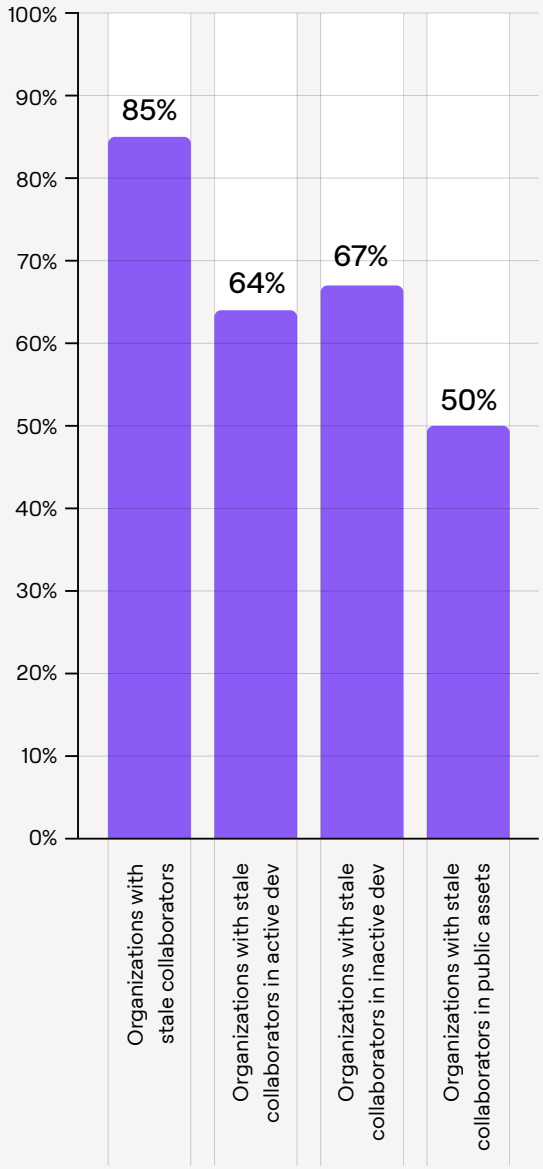


Figure 8: Extent of stale collaborators

LEAST PRIVILEGE ISSUES

Developer permissions sprawl is a significant issue that arises when developers are granted excessive access rights across various systems without proper oversight. This challenge is common in large enterprises, where organizations often provide admin access to every repository by default during onboarding. If a developer's credentials are compromised, an attacker could gain access to the entire system.

Why is this a significant threat?

A wide range of people have permissions to access the source code management systems, CI/CD systems, and artifact registries that make up your software development processes.

Each system follows a different permission model, but they all operate together. Given the complex infrastructure that underpins the SDLC, this makes it difficult to manage all the permissions a user may have. Complicating matters further, many organizations have a poor security habit of sharing credentials and secrets across systems. As a result, much of your development team may have extensive access to large parts of your SDLC.

This multifaceted and untamed web of permissions creates an attractive target for attackers. All they need to do is gain access to the right developer or account — whether active or dormant — and obtain broad and powerful access to your build environment. Once inside, they can wreak havoc and steal intellectual property and inflict serious downstream damage to your customers.

Least privilege data

Our data shows that most organizations (85%) do not have least-privilege set up properly — meaning, developers have unnecessary access that would needlessly give an attacker extra access if those credentials are compromised.

On the good news front, the percentages are much lower for those with least-privilege issues in public assets (25%).

However, one-quarter have the toxic combination of external collaborators in a repo with least-privilege issues, and 23% have the toxic combination of external collaborators with admin privileges in pipelines with critical and high misconfigurations (Figure 9 on the following page).

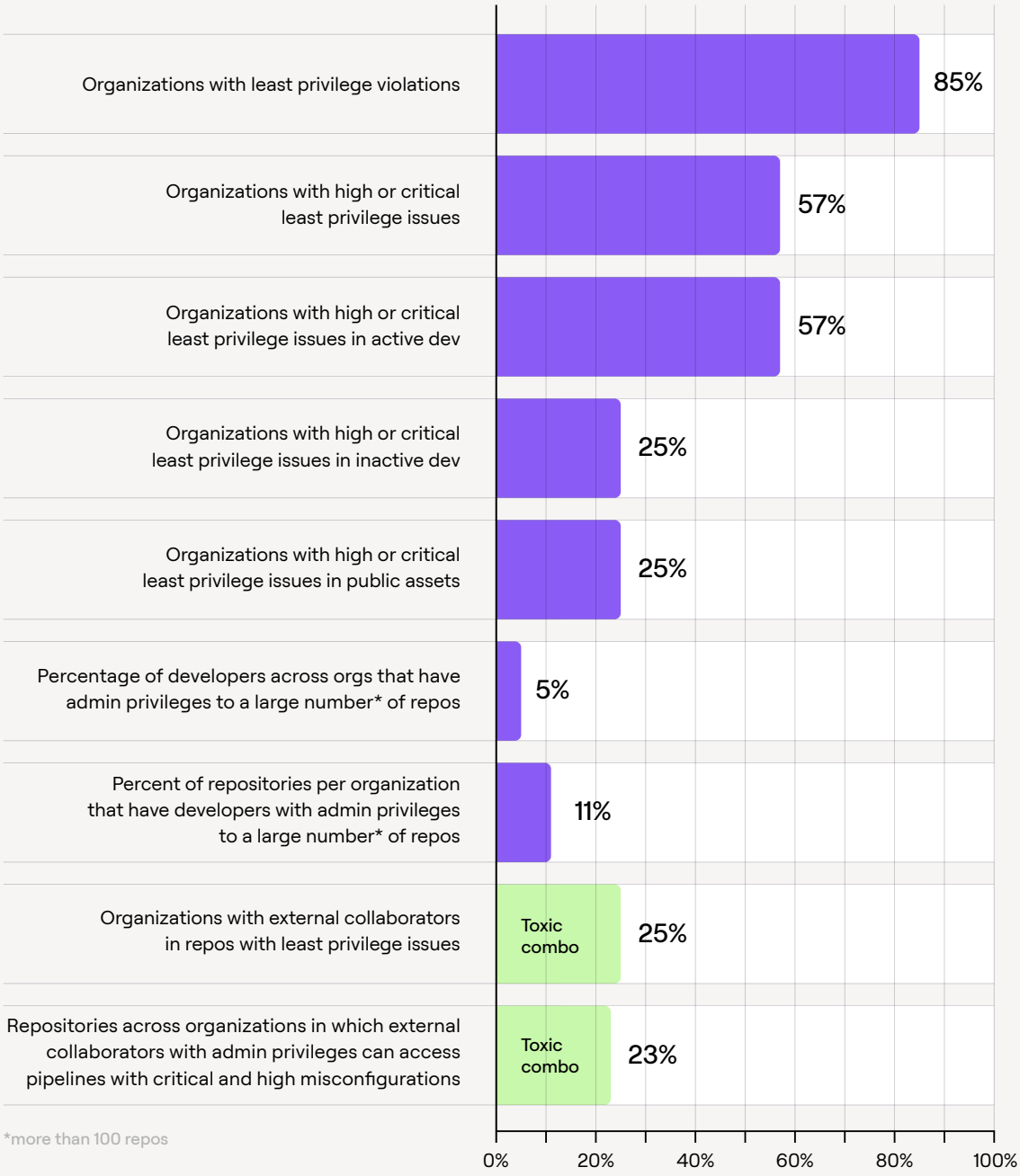


Figure 9: Extent of least privilege violations

RECOMMENDATIONS TO ADDRESS PERMISSIONS SPRAWL

→ Implement role-based access control (RBAC)

Utilize RBAC to provision permissions based on job roles rather than individual users. This approach makes onboarding, offboarding, and permission management more scalable and consistent.

→ Establish and enforce least privilege

Apply the principle of least privilege by granting developers only the minimum permissions necessary to perform their specific job functions. This reduces the risk of unauthorized access and potential security breaches.

→ Conduct regular permission audits

Conduct periodic reviews of user permissions to identify and revoke unnecessary access. This helps maintain the principle of least privilege and ensures that permissions align with current job responsibilities.

→ Automate permission management

Implement automated tools and processes for permission assignment and review. This can help maintain scalability and reduce the risk of human error in managing access rights.

→ Educate developers

Train developers on the importance of access control and the principle of least privilege. This can help create a security-conscious culture and reduce the tendency to request or retain excessive permissions.

04 Software Supply Chain Issues

Software supply chain issues include the following:

→ **Unsafe cross-workflow actions**

For example, moving an artifact from one stage of the build process to another without validating that it hasn't been maliciously tainted.

→ **Build actions utilizing mutable images**

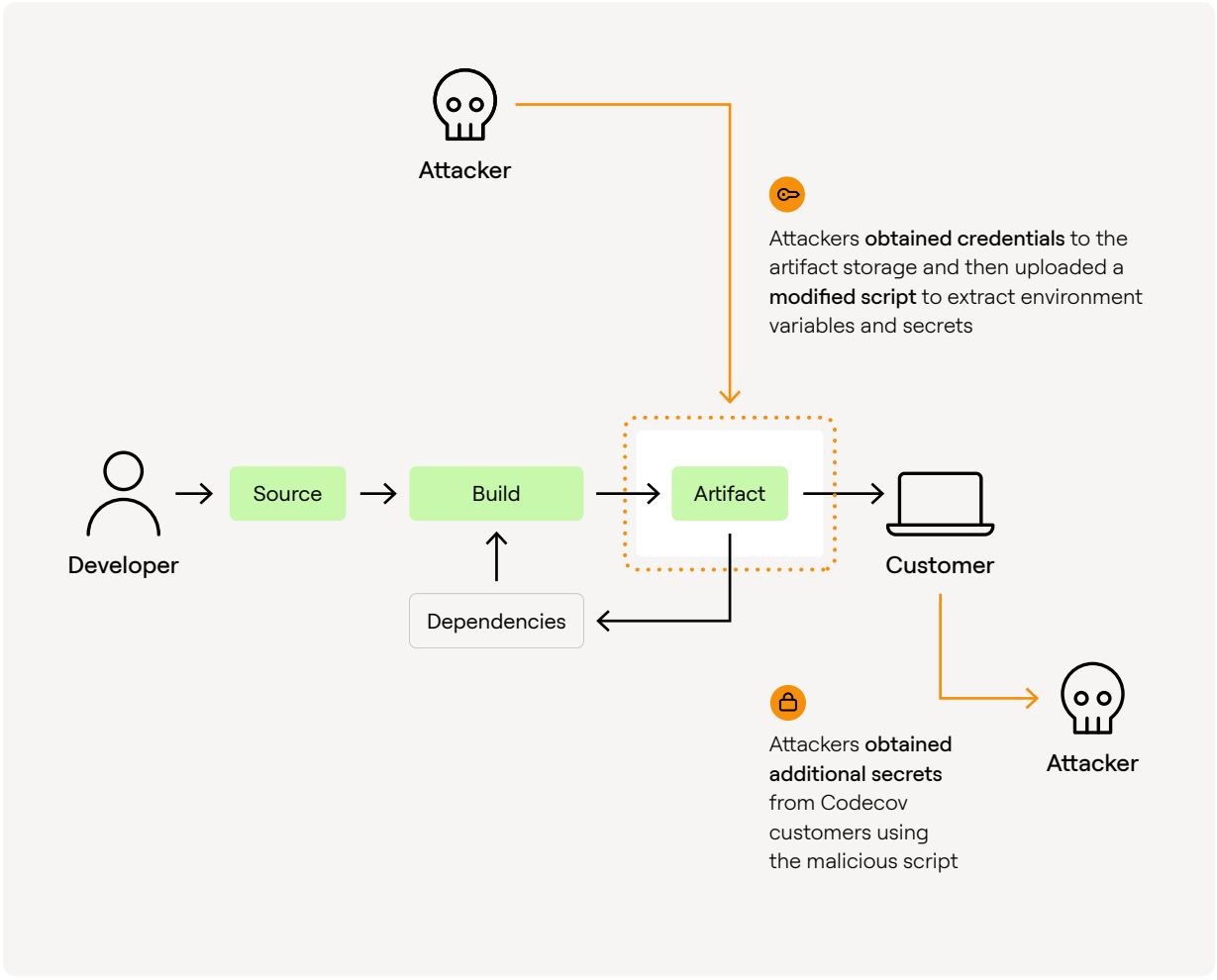
If you are pulling containers for open source repositories into your build process, you should specify the version, rather than just "latest version." That latest version could have a malicious backdoor, for example.

→ **Privileged pipeline checking out insecure code**

A pipeline with extensive privileges, such as the ability to auto-deploy, should not be able to pick up a package from a third-party repository without validating the code. This vulnerability could lead to dependency confusion, where the pipeline is picking up a malicious package from an open source repository because an attacker has renamed it to something extremely similar to a legitimate package.

An example attack leveraging these types of issues is the Codecov attack (shown below).

Attackers used an unpinned Docker image to alter the Bash Uploader script. This modification enabled them to steal sensitive data from many of Codecov's clients, highlighting the risks of not locking Docker image versions in CI/CD pipelines.



Software Supply Chain Issues Data

Another 100% stat here for organizations with high or critical software supply chain issues. That number drops to 46% for high or critical issues in public assets. Within organizations, we see an average of almost a quarter of repos with software supply chain issues, and 23% with the toxic combination of software supply chain issues plus branch protection issues (Figure 10).

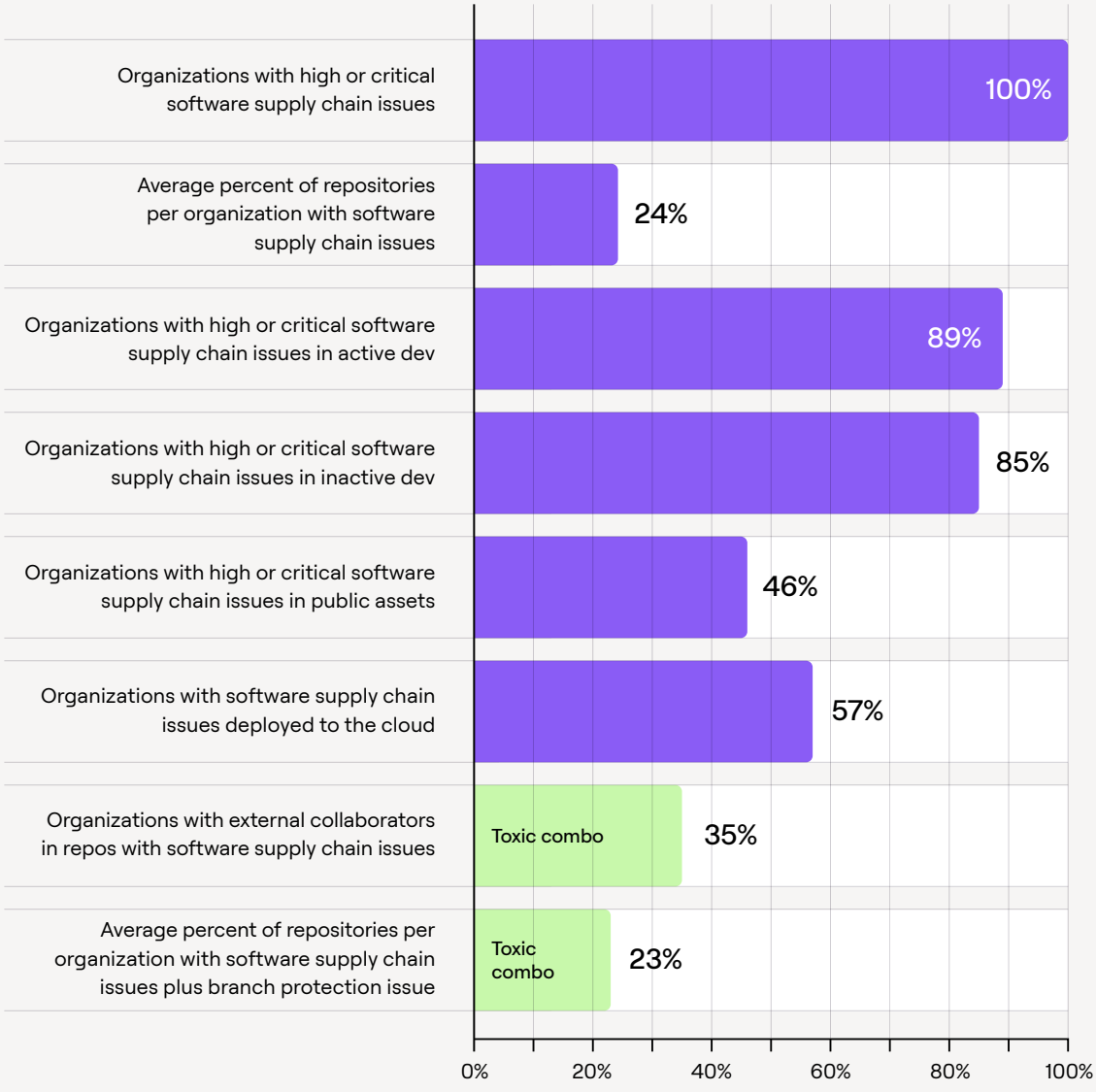


Figure 10: Extent of software supply chain issues

Preventing Software Supply Chain Issues

PREVENT UNSAFE CROSS-WORKFLOW ACTIONS

→ Limit workflow permissions

Ensure that workflows are scoped with minimal privileges.

→ Use job dependencies wisely

Define clear job dependencies to control which jobs can access or trigger others.

→ Audit and monitor workflow activity

Implement logging and monitoring to detect and alert on suspicious cross-workflow actions.

PREVENT BUILD ACTIONS USING NON-VERSIONED MUTABLE IMAGES

→ Use immutable, versioned container images

Always reference container images by specific version tags.

→ Use container scanning tools

Scan container images for vulnerabilities before they are used in builds.

→ Establish a secure image registry

Maintain your own private registry with strict policies around how images are pushed and pulled. Additionally, enable signing of images to ensure integrity and authenticity.

→ Automate image promotion

Set up a pipeline where images must pass tests and security scans before they can be promoted through environments.

PREVENT PRIVILEGED PIPELINES CHECKING OUT INSECURE CODE

→ Enforce code reviews and approvals

Require at least one peer review before code can be merged into branches that trigger production workflows.

→ Use role-based access control (RBAC)

Limit who can trigger privileged builds.

→ Use static code analysis

Integrate static analysis tools into your pipeline to scan for insecure code patterns, vulnerable dependencies, and compliance issues.

→ Leverage pull request (PR) checks

Configure pipeline checks to block the merge of pull requests unless specific security conditions are met.

→ Limit privileged actions in CI/CD

Run builds and deployments in non-privileged contexts as much as possible. Use sandboxed or isolated environments (e.g., Docker containers) to reduce the potential impact of security issues.

→ Use secrets management tools

Ensure that sensitive data (e.g., API keys, tokens) is stored securely.

The application risk landscape has changed, and traditional application security approaches are no longer sufficient.

This report highlights the risks that are embedded and enmeshed across the software factory, well beyond vulnerabilities in code.

Key Takeaways

The good news is that with the right visibility, an emphasis on developer-security collaboration, and a focus on some established best practices, you can make a significant dent in your level of application risk.

THOSE BEST PRACTICES INCLUDE:

Scan for secrets exposure.

- Avoid committing secrets to any Git repository.
- Change the source code to not rely on hard-coded secrets by using a password manager, environment variable, etc.
- Avoid secrets within build logs and sharing secrets via messaging services.
- Use encryption to store secrets within Git repositories.
- Use local environment variables, when feasible.

Manage AI risk.

- Employ tools to get visibility into AI use across your development environment.
- Threat model the impact of AI-specific threats.
- Consider security when selecting AI models.

Enable branch protection for your default main branch.

Tighten up permissions.

- Implement role-based access control.
- Establish and enforce least privilege.
- Automate permissions management.

Streamline AppSec scanning tools.

Avoid contributing to alert fatigue and security debt by getting clarity on what AppSec testing tools you have where, then consolidating and eliminating tools as needed.

Continually monitor and verify configurations (including permissions).

- Create paved pathways and standardized pipelines for attack surface reduction and configuration drift.
- Validate controls within these pipelines on an ongoing basis.
- Set up secure by design defaults for developers to make it easy for them to spin up new secure infrastructure when needed.

Legit is a new way to manage your application security posture for security, product and compliance teams. With Legit, enterprises get a cleaner, easier way to manage and scale application security, and address risks from code to cloud.

Built for the modern SDLC, Legit tackles the toughest problems facing security teams, including GenAI usage, proliferation of secrets and an uncontrolled dev environment. Fast to implement and easy to use, Legit lets security teams protect their software factory from end to end, gives developers guardrails that let them do their best work safely, and delivers metrics that prove the success of the security program. This new approach means teams can control risk across the business — and prove it.

legitsecurity.com