XLAB Steampunk

E-BOOK

Securing Ansible Playbooks and Infrastructure

How to Embed Security Early in Ansible Automation Lifecycle

steampunk@xlab.si | +386 1 244 77 56 | steampunk.si

Introduction

Ansible Playbooks can configure hundreds or thousands of systems in a single run. That power also means that small mistakes – such as hard-coded secrets, unvalidated variables, or excessive privileges—can quickly lead to large-scale security incidents.

Securing Ansible goes beyond writing safe playbooks. It requires visibility and governance across the entire automation stack, including collections, dependencies, execution environments, and the infrastructure being managed. Security must be built in early and enforced consistently, not added as a last-minute check.

This e-book takes a practical approach to securing Ansible automation. You will learn why security must start early in the development lifecycle, how to identify and manage risks across the full dependency chain, and the most common security pitfalls in playbooks and infrastructure. We'll also cover best practices for securing automation and show how to build governance directly into your Ansible workflows, so your automation remains secure, compliant, and reliable at scale.

Contents

XLAB Steampunk

Why security must start early in Ansible automation?

Insecure Ansible Playbooks inevitably introduce risks into your environment. Even minor mistakes, such as hard-coded secrets, unchecked inputs, or excessive privileges, can affect thousands of systems, leading to downtime, compliance breaches, or security incidents. True security in Ansible automation extends beyond the playbook itself; it requires evaluating every layer of the software supply chain. Every playbook – whether created by AI, a junior engineer, or an experienced developer – should follow consistent practices and undergo automated checks against internal or standardized benchmarks.

While runtime monitoring and security validation in production are essential since systems can behave unpredictably and addressing issues after deployment is costly and disruptive. Shifting security left, by embedding it into the design, development, and testing phases helps prevent problems before they reach production.

Shift left practices also address a common enterprise challenge: governance. Manual security reviews are slow, inconsistent, and difficult to scale. By automating governance, maintaining clear audit trails, and enforcing policies across teams, organizations gain full visibility into their Ansible environment. This ensures that new users can onboard safely, automation can scale effectively, and new automation can be deployed without putting production systems at risk.

1. What does “shift left” mean in Ansible automation?

“Shift left” is a concept from software development that moves security and quality checks earlier in the development lifecycle – toward design, coding, and testing – instead of waiting until production. In traditional workflows, security often occurs late, during deployment, or after an incident. In Ansible automation, this delay is especially risky because playbooks can affect thousands of systems at once, often with high privileges, amplifying any mistake.

Shifting left in Ansible means embedding security into every stage of playbook development:

- **Early Scanning**: Analyze playbooks and their dependencies before execution to catch misconfigurations or vulnerable modules.
- **Input validation**: Treat all variables as untrusted and validate them during design and testing to prevent command injection or unintended actions.
- **Policy enforcement**: Apply security policies and benchmarks automatically, ensuring every playbook follows organizational or regulatory standards.
- **Governance integration**: Automate audits, track rule changes, and maintain full visibility across the Ansible environment, enabling teams to scale safely.

2. **Securing Ansible workflows early**: why it is critical

Small mistakes scale quickly
A ten-line playbook with root privileges can impact thousands of systems in seconds. Catching errors early prevents widespread issues.

Infrastructure is critical and complex
Playbooks interact with operating systems, cloud APIs, third-party modules, and execution environments. Each layer is a potential attack surface that must be validated before deployment.

Manual checks do not scale
Manual code reviews cannot keep up with frequent playbook updates, playbooks at scale or the speed of modern automation. Automated code reviews with integrated security and governance is essential for consistency and reliability.

Governance enables safe scaling
Automating compliance and maintaining audit trails ensures consistent application of policies, accelerates onboarding, and allows organizations to expand automation use cases safely.

“Small mistakes scale instantly”
A ten line playbook with root privileges can impact thousands of systems in seconds. Small mistakes do not stay small. They scale.

3. **Shift left in action:** a practical example

A playbook that accepts a cmd variable and executes it on multiple servers illustrates the risks:

```
name: Execute user command
hosts: all
tasks:
      name: Run command
      shell: “{{ cmd }}”
      become: true
```

Technically valid, this playbook is risky because:

- Any user can execute dangerous commands.
- Global become: true increases the impact of mistakes.
- Dependencies and modules may contain vulnerabilities.

Applying a shift left approach:

- Validate inputs against allowed commands.
- Enforce least-privilege execution.
- Scan all modules and dependencies for security issues.
- Apply automated governance and policy checks before deployment.

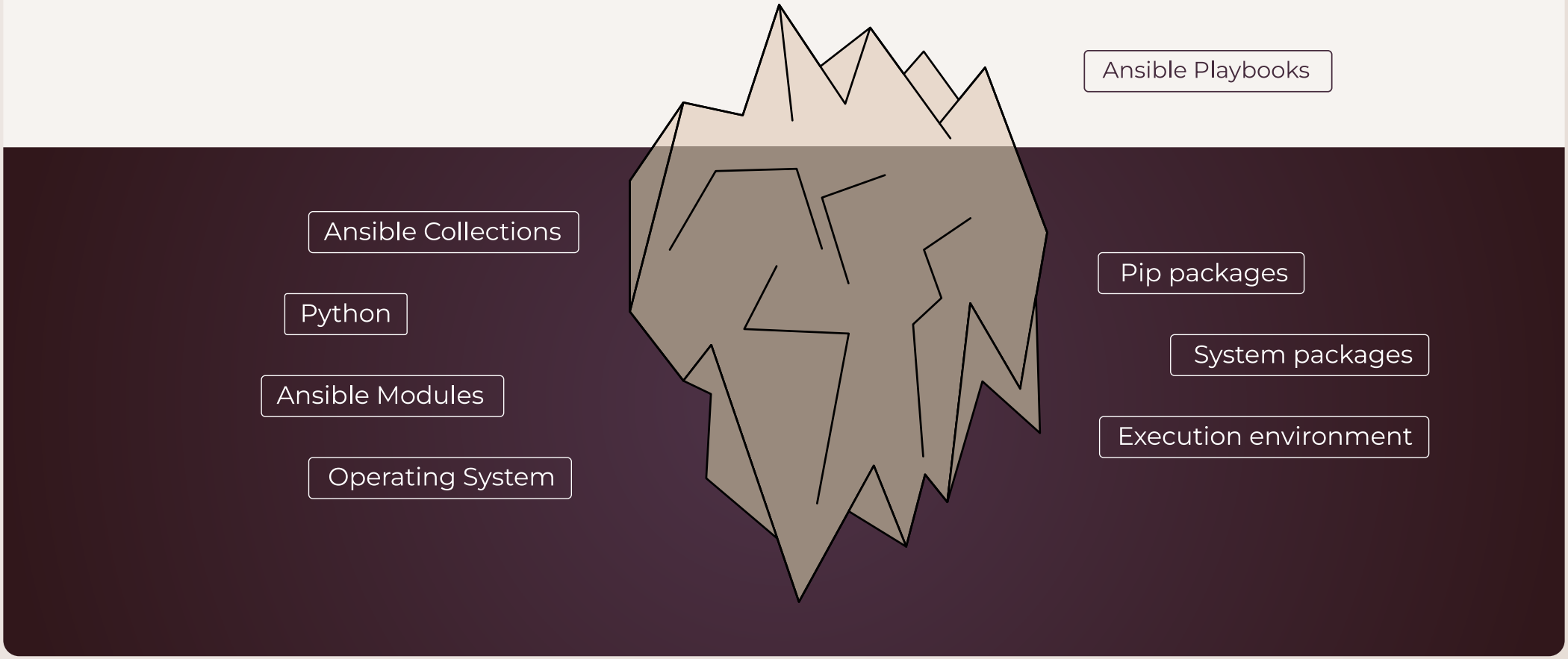
XLAB Steampunk

Managing security risk across the Ansible dependency chain

Ansible Playbooks are just the tip of the iceberg. While the playbook code is visible, the full automation execution relies on a software supply chain that extends far beyond the playbook itself. Most security risks lie in the hidden layers, that include:

- **Ansible collections**: Bundles of modules and roles that may come from public sources or internal repositories. Vulnerabilities here are inherited by any playbook that uses them.
- **Ansible modules and roles**: Individual modules or task sets written in Python, PowerShell, or shell. Weaknesses or insecure defaults can propagate into your automation.
- **Python and system packages**: Libraries or packages required by modules and roles. Outdated or vulnerable packages introduce hidden risks.
- **Execution environment and operating system**: The runtime, including OS-level settings and interpreter versions, can carry vulnerabilities that affect all playbook executions.

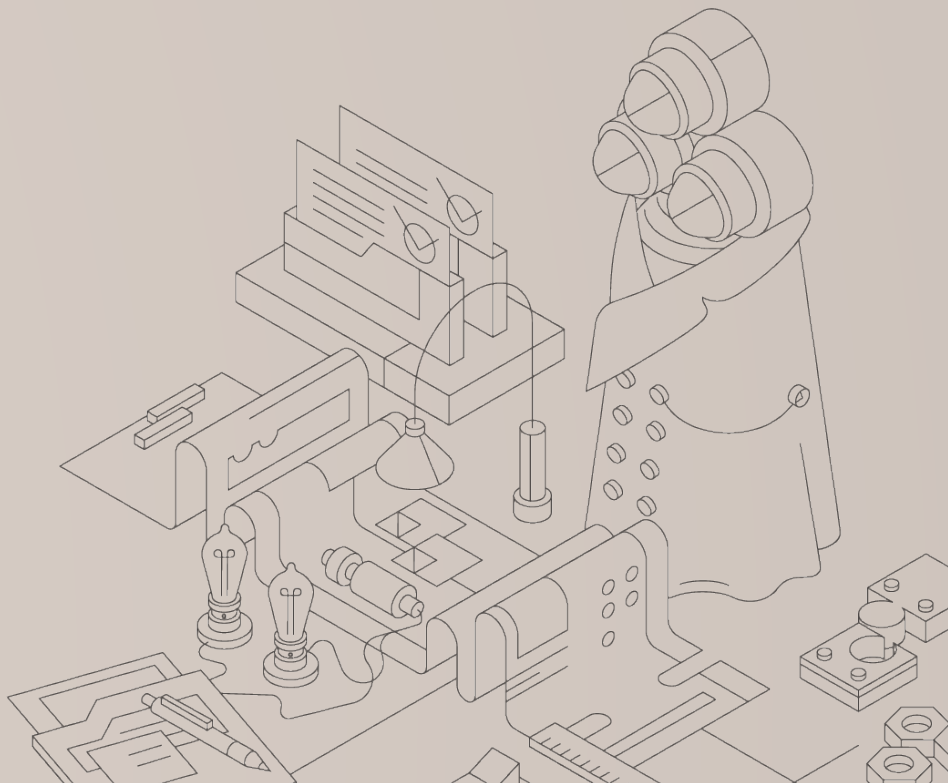
Together, these layers form the “underwater” part of the iceberg where risks can hide, while the playbook itself is just the visible tip. Even small mistakes or vulnerabilities in these dependencies can scale across thousands of systems, leading to downtime, compliance issues, or security incidents. By understanding the full dependency chain and applying shift left practices—scanning dependencies early, validating inputs, and enforcing automated governance—you can secure not just the playbook, but the entire Ansible ecosystem before it reaches production.



XLAB Steampunk

Common security risks in Ansible Playbooks

1. Hard-coded secrets and sensitive data
2. Trust in unvalidated variables
3. Missing or weak input validation
4. Excessive privileges
5. Insecure communication channels
6. Unrestricted command execution



Here are some of the most common risks in Ansible Playbooks we see in practice:

1. Hard-coded secrets and sensitive data

Passing passwords, API keys, or tokens directly in playbooks is a frequent mistake. Even when playbooks are valid, logs can unintentionally capture sensitive information if tasks are not configured correctly.

> **Best practices:** Use Ansible Vault or a dedicated secret management solution to store sensitive data securely. Avoid exposing secrets in logs, variables, or code repositories.

2. Trust in unvalidated variables

A single variable from a vars file or unchecked input can turn a valid playbook into a destructive one. Assumptions about “trusted users” or “safe defaults” often lead to problems, exposing sensitive data or causing full-scale system failures.

```
---
- hosts: localhost
  tasks:
      - name: execute any command, given by -e switch ansible.builtin.command: “{{ cmd }}”
```

> **Best practices**: Always validate and sanitize all inputs. Treat every variable as untrusted until verified, regardless of its source.

3. Missing or weak input validation

Playbooks that accept variables without proper checks can lead to command injection or unintended execution. Even trusted sources can provide unexpected or malicious input.

> **Best practices**: Apply strict validation rules and type checks during development. Ensure all variables conform to expected formats and ranges.

4. Excessive privileges

Running playbooks with become: true or root privileges everywhere may seem convenient, but it multiplies the potential impact of mistakes or malicious commands.

> **Best practices**: Apply the principle of least privilege. Only grant the access necessary for each task to reduce risk.

5. Insecure communication channels

Downloading packages or interacting with systems over insecure channels can allow interception, tampering, or man-in-the-middle attacks.

> **Best practices**: Use HTTPS, SSH, and verified repositories. Ensure that all communication with remote systems is encrypted and authenticated.

6. Unrestricted command execution

Playbooks that execute raw commands from variables, such as a CMD variable, are particularly dangerous. They may work as intended, but users can pass anything, and assumptions about safe usage often fail.

> **Best practices**: Define allowed actions and sanitize inputs. Validate commands at design time to prevent misuse or accidental destruction.

XLAB Steampunk

Security risks beyond playbooks: the Ansible infrastructure layer

1. Vulnerable collections and third-party dependencies
2. Unvalidated variables passed to external modules
3. Cloud misconfigurations at scale
4. Secret exposure beyond the playbook
5. Shell injection and unsafe module use
6. Privilege escalation at the infrastructure level

Running an Ansible Playbook activates an entire software supply chain. Collections, roles, modules, interpreters, third-party packages, and the operating system all work together to enable automation. Each layer introduces potential vulnerabilities, some of which developers may not be aware of. Ansible infrastructure goes beyond playbook logic – it requires understanding and managing risks across the entire automation stack.

Here are the most common infrastructure-level risks:

1. Vulnerable collections and third-party dependencies

Ansible collections and roles often rely on external modules and packages. If any of these dependencies have **known vulnerabilities**, your automation inherits those risks automatically.

Example: A playbook installs software using a third-party Python package that contains a remote code execution vulnerability. Even if your playbook appears safe, running it in production could compromise your systems.

> **Best Practice**: Maintain an up-to-date inventory of all dependencies and scan them regularly against vulnerability databases (CVEs and CWEs).

2. Unvalidated variables passed to external modules

Variables that are not properly validated at the playbook level can propagate to modules and libraries deeper in your infrastructure. This can lead to **unexpected behavior or command injection.**

Example: Passing unchecked user input to a cloud module might trigger unintended changes in your cloud environment, such as creating resources in the wrong region or deleting critical assets.

> **Best Practice**: Treat every variable as untrusted and validate inputs rigorously before execution.

3. Cloud misconfigurations at scale

Infrastructure automation often manages cloud resources. Misconfigured settings –such as overly permissive IAM roles, exposed storage buckets, or unsecured networking rules –can create **large-scale security gaps**.

Example: A playbook deploys hundreds of virtual machines with default security groups allowing public SSH access. One misstep could expose your entire environment to attackers.

> **Best Practice**: Implement policy-as-code checks for cloud configurations and integrate them into your CI/CD pipeline.

4. Secret exposure beyond the playbook

Infrastructure-level secrets include API keys, credentials for cloud services, and database passwords. Unlike playbook-level secrets, these often interact with modules, interpreters, and external systems, increasing exposure risk.

Example: Secrets stored in environment variables or logs without encryption can be accessed by anyone with playbook execution privileges.

> **Best Practice**: Use centralized secret management and audit secret usage across the entire automation stack.

5. Shell injection and unsafe module use

High-privilege modules or shell commands executed by Ansible can be **exploited if inputs are untrusted or improperly sanitized**. This is the infrastructure-level equivalent of the playbook-level command injection risk.

Example: A playbook executes a shell module on thousands of servers using a variable for commands. If an attacker can control that variable, they could execute arbitrary commands across your entire fleet.

> **Best Practice**: Prefer dedicated Ansible modules over shell commands, sanitize all inputs, and avoid running commands as root unless necessary.

6. Privilege escalation at the infrastructure level

Running automation with high privileges across your infrastructure amplifies mistakes. A playbook might behave correctly in testing but cause unintended disruption at scale in production.

Example: A collection updates system packages on all servers using root privileges. A minor misconfiguration could break critical services everywhere.

```
---
- hosts: localhost
  become: true
  vars:
     working_file: /tmp/playbook.zip
     working_folder: /tmp/playbook_folder
  tasks:
     - servicenow.itsm.incident_info:
          instance: “{{ sn_instance }}”
          query:
```

```
        -state: = new
    register: incident_result

- ansible.builtin.set_fact:
      ones_with_attachments: “{{ incident_result.records | list }}”

- ansible.builtin.include_tasks: act.yml
```

> **Best Practice**: Apply the principle of least privilege, limit root-level tasks, and segment execution environments to reduce blast radius.

XLAB Steampunk

Best practices for securing the Ansible supply chain

1. Automate security controls
2. Build a Software Bill of Materials (SBOM)
3. Implement Policy as code
4. Apply least privilege principles
5. Validate inputs and sanitize commands
6. Leverage industry benchmarks
7. Continuous monitoring and governance

Securing Ansible automation involves more than protecting individuals – it requires safeguarding the entire Ansible supply chain, including collections, modules, third-party packages, and execution environments. Strong governance ensures consistent enforcement of security policies, reducing the risk of vulnerabilities across playbooks and infrastructure. To help you get started, here are some key best practices for securing Ansible automation effectively:

Automate security controls

Manual reviews are insufficient. Integrate automated security scanning into CI/CD pipelines to detect misconfigurations and vulnerabilities early in the development lifecycle. Automation enforces consistent security policies across all playbooks and environments, minimizing human error and accelerating delivery.

Build a Software Bill of Materials (SBOM)

Maintain a detailed inventory of every component in your Ansible automation – collections, modules, third-party dependencies, and execution environments. An SBOM enables continuous monitoring for known vulnerabilities and ensures compliance with security standards before components reach production.

Implement Policy as code

Encode security policies directly into your automation platform using policy-as-code frameworks. This guarantees that all playbooks and underlying infrastructure comply with your organization's requirements, eliminating reliance on manual interpretation of documentation.

Apply least privilege principles

Limit the use of root or high-privilege access wherever possible. Ensure playbooks and infrastructure components run with only the permissions necessary for their tasks, reducing the impact of errors or attacks.

Validate inputs and sanitize commands

Treat every variable and input as untrusted. Validate data at the playbook and infrastructure levels and avoid executing raw shell commands when possible. Use dedicated Ansible modules that handle inputs safely, preventing injection or unintended actions.

Leverage industry benchmarks

Use widely accepted frameworks such as CIS and NIST to validate playbooks and automation workflows. Aligning with these benchmarks ensures that your Ansible automation adheres to recognized security and compliance standards.

Continuous monutoring and governance

Regularly monitor the entire Ansible supply chain, including collections, third-party dependencies, and execution environments. Automated governance detects and remediates vulnerabilities, misconfigurations, and policy violations before deployment, keeping automation secure and compliant.

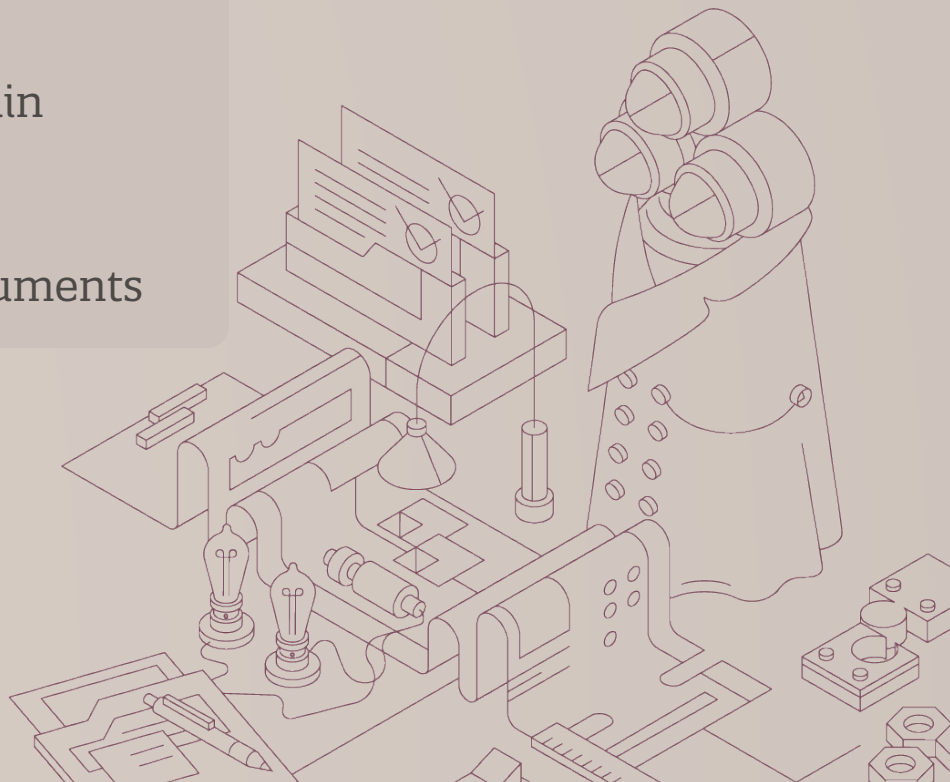
Key Takeaway

Securing Ansible automation requires a holistic, proactive approach. By combining automated scanning, governance, SBOMs, policy-as-code, least-privilege principles, and industry benchmarks, organizations can enforce consistent security standards across playbooks and infrastructure—reducing risk and enabling safe, scalable automation.

XLAB Steampunk

A secure Ansible workflow in practice

1. Manual reviews only
2. Security checked too late
3. Focus on the playbook, not the full supply chain
4. Ad hoc and nonrepeatable fixes
5. Manual upgrades can reintroduce risk
6. Requirements buried in human-readable documents



In most organizations, resolving security issues in Ansible automation relies heavily on manual processes. Manual code reviews catch some problems, but security is often checked late, sometimes only after an incident occurs in production. This reactive approach leaves gaps that can compromise both playbooks and the underlying infrastructure.

Limitations of traditional security practice:

1. Manual reviews only

Most potential issues in playbooks are identified through human code reviews. While reviewers can spot obvious mistakes, this method is time-consuming, error-prone, and difficult to scale.

2. Security checked too late

Security is often an afterthought. Without automated checks integrated into the development process, vulnerabilities may only surface when a failure occurs in production.

3. Focus on the playbook, not the full supply chain

Organizations tend to focus on the visible playbook layer, ignoring the broader automation stack. Playbooks rely on collections, roles, modules, interpreters, and external dependencies. Each of these layers can introduce vulnerabilities that are missed if security reviews only consider the playbook itself.

4. Ad hoc and nonrepeatable fixes

Even when a problem is identified and corrected, there is often no standardized way to prevent it from recurring. Manual fixes are not encoded into the automation environment, so the same mistakes may happen again.

Manual upgrades can reintroduce risk

Manually updating playbooks or dependencies can inadvertently introduce new risks. Without automated checks, there is no systematic way to detect what could break during upgrades. At the same time, delaying upgrades to avoid these risks can create security vulnerabilities and accumulate technical debt.

6. Requirements buried in human-readable documents

Organizations often rely on lengthy documents to define security requirements. These are left to human interpretation and are rarely encoded into the automation process, creating inconsistencies and misunderstandings.

XLAB Steampunk

Building a secure Ansible Workflow

1. Define security and compliance requirements early
2. Develop with guardrails in place
3. Automated scanning and validation
4. Full supply chain visibility
5. Policy enforcement and governance
6. Safe testing and controlled execution
7. Continuous feedback and improvement

A secure Ansible workflow replaces manual reviews, late security checks, and ad hoc fixes with automated, repeatable controls. Security is no longer something that happens after a problem occurs; it becomes part of how automation is designed, tested, and executed from the start.

Instead of relying on human interpretation and lengthy documents, security requirements are encoded directly into the workflow and enforced consistently at every stage.

1. Define security and compliance requirements early

Most potential issues in playbooks are identified through human code reviews. While reviewers can spot obvious mistakes, this method is time-consuming, error-prone, and difficult to scale.

2. Develop with guardrails in place

Playbooks, roles, and collections are written with built-in guardrails. Secrets are never hard-coded, variables are validated, and privileges are minimized by default. Developers receive immediate feedback when something violates a policy.

3. Automated scanning and validation

Security checks run automatically during development and in CI pipelines, acting as a gatekeeper before changes move forward. Playbooks, collections, and dependencies are scanned for common risks, misconfigurations, and known vulnerabilities before they reach production.

4. Full supply chain visibility

Security checks extend beyond the playbook. Collections, modules, interpreters, third-party packages, execution environments, and operating systems are all part of the review process. This prevents hidden risks from entering the workflow unnoticed.

5. Policy enforcement and governance

Organizational requirements are enforced automatically. Instead of fixing the same issues repeatedly, policies ensure problems are prevented consistently across all automation.

6. Safe testing and controlled execution

Playbooks are executed in controlled environments that mirror production. This allows teams to detect unintended behavior early and reduce the impact of mistakes.

7. Continuous feedback and improvement

Findings from scans, testing, and production execution feed back into the workflow. Policies and checks evolve as automation grows, keeping security aligned with scale.

XLAB Steampunk

Building governance into the Ansible workflow

1. Visibility
2. Consistency
3. Policy as code
4. Feedback loops

Governance is effective only integrated into the workflow, not added as an afterthought. In Ansible automation, this means moving beyond manual reviews and documentation to enforce expectations through automation itself. A governed Ansible workflow is built on four foundational pillars: **visibility, consistency, policy code, and feedback loops**.

Together, these pillars ensure that automation remains secure, compliant, and scalable as environments grow.

Governance is built on four pillars:

Visibility **Consistency** **Policy as Code** **Feedback Loops**

1. Visibility

You cannot govern what you cannot see. In Ansible, visibility means understanding what is actually executed, not just what is written in a playbook. Playbooks are only one part of the automation stack. Collections, roles, modules, dependencies, interpreters, and execution environments all influence the final outcome. Without visibility into these layers, organizations develop blind spots where vulnerabilities can hide.

Effective visibility covers:

- **Covers the full execution context, not just playbooks**
- **Includes dependencies and execution environments**
- **Eliminates unknowns in what runs in production**

Visibility is the foundation on which all other governance controls depend.

2. Consistency

Manual processes create exceptions. Exceptions create risks. Consistency means applying the same rules across all teams, playbooks, and environments. There are no special cases and no reliance on tribal knowledge or individual expertise. When governance is inconsistent, the same mistake is fixed repeatedly, and security depends on who reviewed the code rather than what the code does.

A consistent governance model:

- **Applies the same rules to every team**
- **Eliminates one-off decisions**
- **Ensures predictable behavior across environments**

Consistency turns security from a best effort into a guarantee.

3. Policy as code

Policy as code means translating organizational requirements into machine-readable rules that are automatically enforced. Expectations such as how secrets are handled, which modules are allowed, or how privileges are managed must be encoded directly into the workflow. As automation evolves, policies evolve with it. This ensures that governance keeps pace with change instead of becoming outdated.

Policy as Code:

- **Makes expectations explicit and enforceable**
- **Removes ambiguity and interpretation**
- **Evolves alongside the automation platform**

This is where governance becomes reliable rather than aspirational.

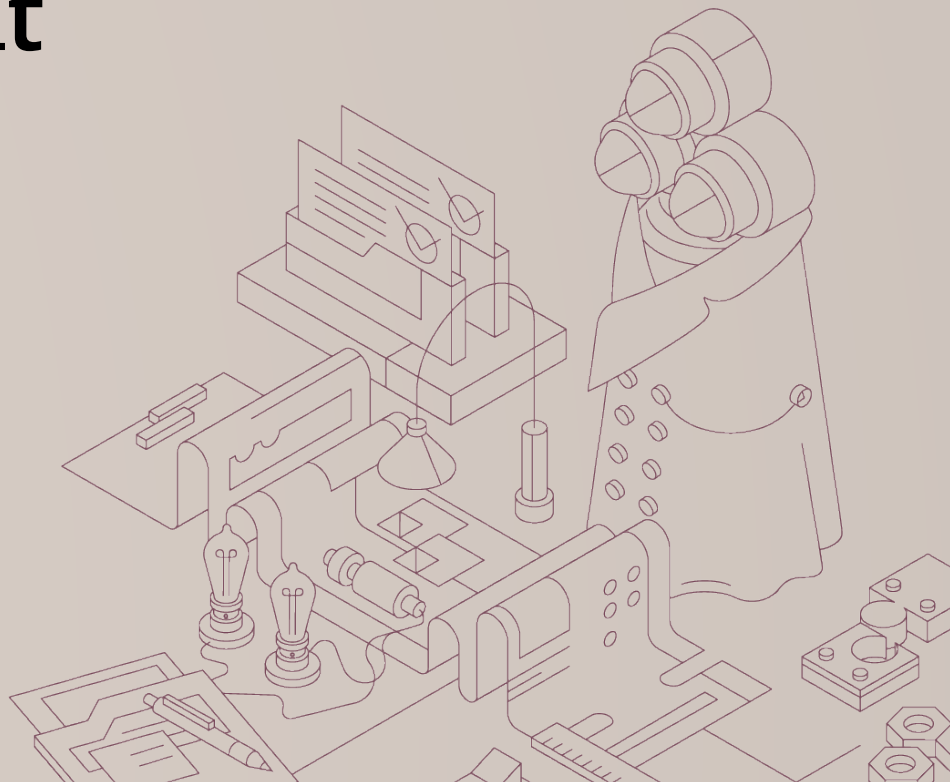
When these four pillars are in place, governance stops being a bottleneck. It becomes an enabler for safe scaling.

Teams gain confidence that automation follows shared standards. Security issues are prevented rather than discovered. Upgrades introduce less risk, and automation can grow without sacrificing control.

Governance, when built into the Ansible workflow, transforms automation from a powerful tool into a dependable operating model.

XLAB Steampunk

Tools and technology for shifting security left in Ansible development



Shifting security left means catching risks early—before automation reaches production. **Steampunk Spotter** helps teams identify risks, enforce compliance, and ensure secure, reliable automation. Key features include:

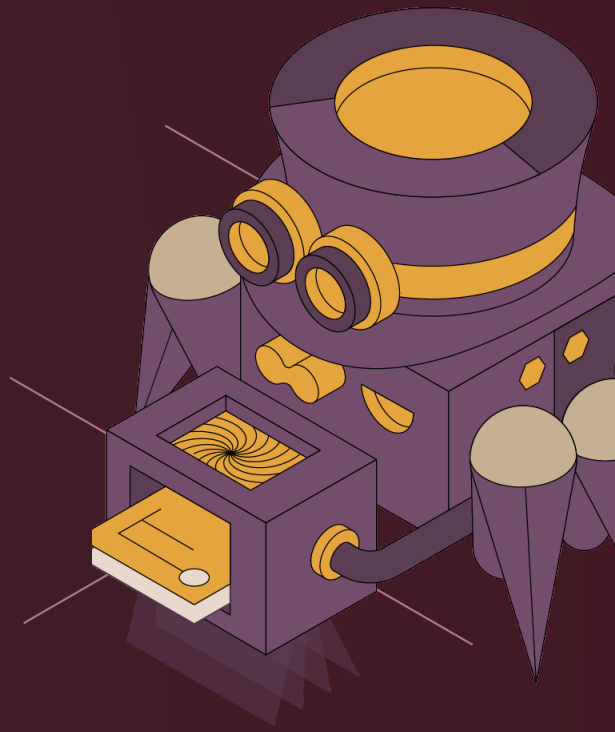
Feature	What it does	Benefit/Value
Playbook Scanning	Flags risky tasks and misconfigurations.	Catch hard-to-spot errors, align playbooks with best practices, and detect high-risk issues before they reach production.
Custom Security Policies	Integrates custom and CIS benchmark policies, with AI-assisted REGO policy creation from plain-language requirements.	Define, enforce, and scale security and compliance rules; simplify policy management.
Software Bill of Materials (SBOM)	Generates a complete inventory of roles, collections, and plugins per automation run.	Tracks dependencies and identifies outdated components.
CVE Analysis & Security Reports	Integrates vulnerability intelligence from CVE/CWE databases.	Highlights high-risk dependencies for proactive remediation.
Ansible Supply Chain Security	Tracks origin and usage of external roles, collections, and plugins.	Ensures only trusted components are deployed.

Sign up for a free playbook assessment

“I can’t review every line of Ansible code, so having a tool that scans my playbooks against our base security requirements is a huge safety net and something I could never do manually. The assessment clearly showed how Spotter helps us stay on top of security standards.”

- Senior System Administrator, Fortune 500

Apply now



XLAB Steampunk

Final thoughts

Ansible automation delivers enormous power and efficiency, but that power comes with responsibility. When automation can configure thousands of systems in seconds, security can no longer be reactive, manual, or optional. Small oversights do not stay small; they scale.

Securing Ansible is not about writing “perfect” playbooks or adding another checklist at the end of a release. It is about treating automation as a first-class part of your infrastructure and governing it accordingly. Playbooks, collections, dependencies, execution environments, and the underlying systems all contribute to the final outcome. Ignoring any layer leaves blind spots where risk can hide.
Shift left security changes this dynamic. By embedding validation, policy enforcement, and governance directly into the automation lifecycle, teams catch issues when they are easiest to fix and least expensive to address. Security becomes part of how automation is built and evolved; not something added after problems appear.

Governance, when done right, does not slow teams down. Visibility, consistency, policy as a code, and fast feedback loops give teams confidence to move faster, onboard new users safely, and scale automation without increasing risk. Standards are enforced automatically; audits become easier, and upgrades introduce fewer surprises.

The goal is not to restrict automation, but to make it dependable. Secure, governed automation enables organizations to trust what runs in production, even as environments become more complex and automation becomes more critical.

When security and governance are integrated into Ansible workflows from the beginning, automation ceases to be a potential liability and becomes a durable foundation for scale, reliability, and long-term operational confidence.

XLAB Steampunk

Visit our page

steampunk.si/spotter/

Follow us

Talk to us

steampunk@xlab.si