

ShopNow

Building a better tomorrow for buyers and vendors
(especially vendors)

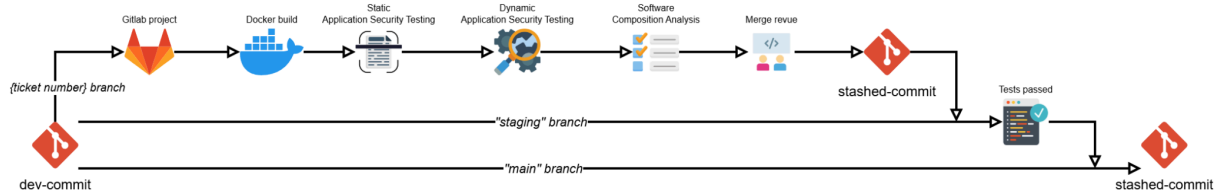


DevSecOps CI/CD pipeline.....	2
Pipeline definition/scope.....	2
Security test end-to-end.....	3
Secrets management.....	3
Per-project secrets within Gitlab.....	3
Dedicated VAULT instance.....	3
“Security gates” policy.....	5
Build state in case of vulnerability.....	5
State of deployment with security failures.....	7
Velocity impact.....	7
Immediate controls to undertake.....	8
Progressive controls to implement.....	8
Global thinking about security.....	9
Exercise 10 - Continuous improvement and security indicators.....	10
1. Security Key Performance Indicators (KPIs).....	10
2. Continuous Improvement Cycle (PDCA).....	11
3. Link with Previous Exercises.....	11
4. Governance & Roles.....	11
5. Ensuring Security Remains Alive Over Time.....	12
6. Priority KPIs at Launch.....	12

DevSecOps CI/CD pipeline

Pipeline definition/scope

A Continuous Integration / Continuous Deployment (CI/CD) requires a strong stack with redundancy and containerization environment. The urgency to go from an on-premise technical stack to an hybrid stack is of a capital importance to address scalability and resilience, as well as reduce to a mere minimum the technical debt.



A complete Pipeline includes, in this order :

1. The presence of a RUN ticket with a unique number
2. The creation of a branch with this very ticket number as name to track modifications in a gist
3. A `git push origin ticket-number-branch` will automatically build the project and launch the tests
 - a. SAST testing for “static” errors (hard-coded secrets, leaked tokens or endpoints in logs, etc)
 - b. End-to-End test (charge load, requests, endpoints accessibility, etc)
 - c. DAST testing for “dynamic” errors (running app leaking info in logs, bad requests and/or answers, broken/missing tokens/headers, etc)
 - d. SCA testing for dependencies (libraries analysing, checking versions and cross-referencing CVE databases, etc)
4. Merge revue (it may appear counter-intuitive, but a merge revue must be conducted AFTER the initial testing, as a secure backend is a first priority BEFORE any code architecture and good redaction practices)
5. Merge Revue approbation, and merging into “staging” branch to test within a production-ready sandboxing environment
6. Second test round, usually with new bugs and errors to fix from a new branch (likely `ticket-number-branch-2`)
7. Final validation and merging into the “main” branch.

Security test end-to-end

PS from Gauvain : again, as I don't know every tool nor the people involved, and because search engines are doomed these days, I took the shortcut of an AI to help me here.

Security testing	Step in CI/CD	Example Tools	Security Controls	STRIDE	Main Actors	Terms of Success
SAST	Commit / Build	SonarQube Checkmarx	Secure coding standards enforcement, hardcoded secret detection, syntax and logic flaw detection	<i>Information Disclosure</i> <i>Tampering</i>	Developers AppSec Engineers	Zero critical vulnerabilities
DAST	Test / Staging (Continuous Delivery)	OWASP ZAP Burp Suite	Runtime vulnerability scanning, session management verification, input fuzzing	<i>Spoofing</i> <i>Elevation of Privilege</i> <i>Denial of Service</i>	QA Testers DevSecOps Pen Testers	No exploitable vulnerabilities found All simulated attacks fail
SCA	Build / Package (Continuous Integration)	Snyk Black Duck	Known CVE detection in third-party libraries, SBOM generation, license compliance	<i>Tampering</i> <i>Elevation of Privilege</i>	DevOps Developers Compliance Legal	No dependencies with unpatched critical CVEs All licenses approved

Secrets management

There are lots of ways to handle secrets from a safe perspective. When correctly configured, handling secrets from a local .env file could be enough for small projects that doesn't require workers nor resources scalability.

Best is to centralize them in a dedicated hosting service with efficient security level (including local encryption and strong authentication to the service). Considering the future presence of a self-hosted Gitlab Instance on-premise, two solutions comes to mind :

Per-project secrets within Gitlab

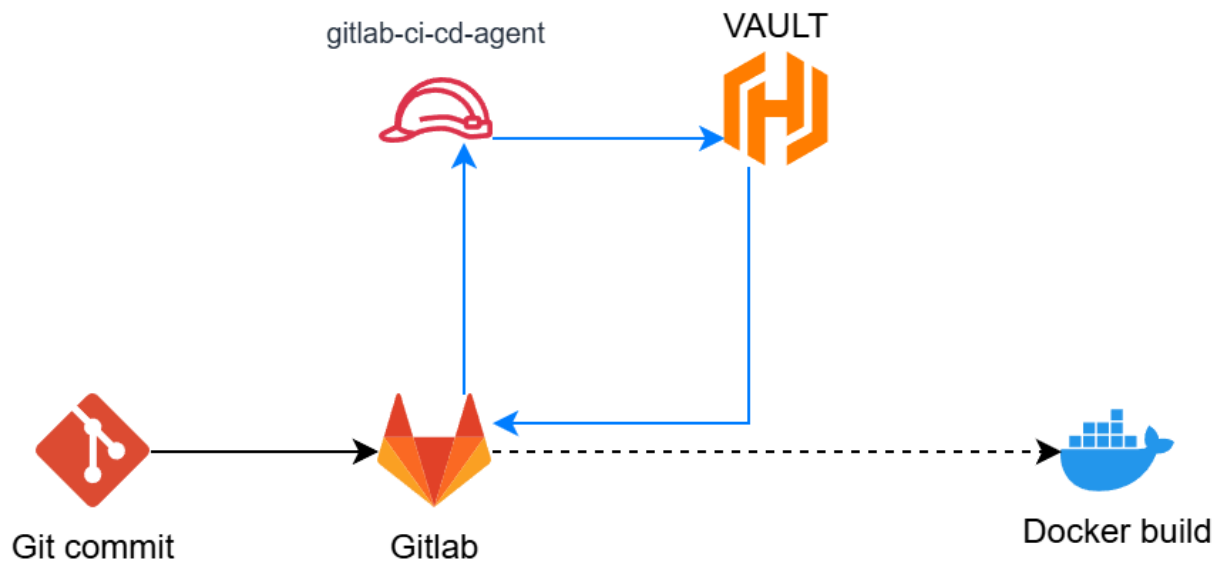
Gitlab let secrets be uploaded as global variables, or per groups or even projects. This solution is good for limited teams and scopes, but is not scalable per se as the variables are mostly obtained via CI/CD code, and not during the build.

The solution still remains efficient, secure and without any additional cost of IT administration.

Dedicated VAULT instance

Best solution for scalability, resilience and security is to deploy a dedicated HashiCorp VAULT instance. The solution is tailor-built to hold secret in a tree arborescence.

Secrets are accessed by a service account which role is “assumed” by Gitlab via an IAM policy :



A functional Terraform code would look like this :

```
data "vault_generic_secret" "db_customers" {
  path = "secret/backend_api/db_customers"
}

resource "aws_iam_role" "rds_role" {
  name = "rds_write_role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Resources = "gitlab-instance-ARN"
        }
      }
    ]
  })
}

data "aws_iam_policy_document" "rds_write" {
  statement {
    effect = "Allow"
    actions = [
      "rds-data:ExecuteStatement",
      "rds-data:BatchExecuteStatement"
    ]
    resources = [
      aws_db_instance.this.arn
    ]
  }
}
```

```
resource "aws_iam_policy" "rds_write" {
  name      = "rds_write_policy"
  policy    = data.aws_iam_policy_document.rds_write.json
}

resource "aws_iam_role_policy_attachment" "rds_attach" {
  role       = aws_iam_role.rds_role.name
  policy_arn = aws_iam_policy.rds_write.arn
}

resource "aws_db_instance" "this" {
  allocated_storage = 20
  engine            = "postgres"
  engine_version    = "15.4"
  instance_class     = "db.t3.micro"
  identifier        = "my-short-rds"
  username          = "dbadmin"
  password          = data.vault_generic_secret.db_customers.data["password"]
  skip_final_snapshot = true
}
```

This method let password be deployed dynamically, with only the deployment of secrets being manual, and are accessed without any leak or real exposition during the pipeline.

“Security gates” policy

A CI/CD pipeline let us decide whenever we’re ready to deploy something, as every step can be monitored and raise errors according to the inbuilt parts used or defined by our own code. Errors should always be made verbose and return specific error codes to allow for a thorough inspection.

Build state in case of vulnerability

When the code is being compiled, or built and run in a testing environment, SAST, DAST and SCA steps are made to raise alarms for each vulnerability they may find. Gitlab CI/CD is designed to raise a “failed” state when an exit code of 1 is detected and stop the remaining pipeline to avoid resource consumption.

So, there’s no policy to deploy *per se*, as every measure made is already built-in.

Yet, a production error may break a pipeline, but a bad implementation of SAST/DAST/SCA tests may raise alarms without blocking efficiently the deployment process. This is up to DevOps or GitOps in charge of the pipelines to define blocking rules.

Considering current severity classifications :

Severity	Severity Score Range
None*	0.0
Low	0.1-3.9
Medium	4.0-6.9
High	7.0-8.9
Critical	9.0-10.0

A rule letting people work without major interference could be to only block pipelines with **a severity over 5.0** so minor inconveniences will not require an immediate patching. Then, it's up to the IT director to state the balance between “acceptable risks / protection against attacks” and define the maximum severity threshold.

A quick solution to avoid CVE in the pipeline could be the use of “hardened images” either from [ChainGuard](#) or [Docker Hardened](#) to significantly reduce surface attacks of in-house images, for a fee to the business department discretion.

[A comparison made by french DevSecOps Stéphane Robert](#) states for an NGINX image :

```
# Traditional nginx:latest : list installed packages
docker run --rm nginx:latest dpkg -l | wc -l
# 152

docker images nginx:latest
REPOSITORY TAG IMAGE ID CREATED SIZE
nginx latest 60adc2e137e7 2 weeks ago 152MB
```

Target	Type	Vulnerabilities	Secrets
nginx:latest (debian 13.2)	debian	4	-

```
# Chainguard nginx:latest : list installed packages
docker run --rm cgr.dev/chainguard/nginx:latest dpkg -l | wc -l
# 16

docker images nginx:latest
REPOSITORY TAG IMAGE ID CREATED SIZE
cgr.dev/chainguard/nginx latest 24a61bdb908b 2 weeks ago 16.9MB
```

Target	Type	Vulnerabilities	Secrets
nginx:latest (wolfi 20230201)	wolfi	0	-

The investment can be made for a fee per image used and allow specific version usage, for a security hardening that requires less human-monitoring and let specifically trained teams purge attack vectors from vulnerabilities.

State of deployment with security failures

As mentioned before, deployment can be blocked if an error was raised. The deployment is already undergoing many steps to avoid any error with Static and Dynamic testing, made to scan first for hard-coded and “cold” errors and security failures.

A good practice is to make deployment automatically fail if a DAST scan detects something (leaked secret, too verbose log, plain string containing PII, etc) and fix these in a new branch if a first merging occurred from the “dev” to the “staging” branch. This way, a complete deployment requires a sanitized (within humanely defined margin of errors and failures detected as mentioned in the severity minimum definition from the IT director).

Bear in mind : an objective of 0 CVE is not obtainable. Never, ever, ever. So don't try catching a unicorn here, and instead evaluate each raised severity with these questions :

- Is this vulnerability exploitable in my context ?
 - You may have an exposed port, but it would require the hacker to already be in your private network, so basically infiltrated in an AWS/GCP/Scaleway datacenter, and from this perspective, maybe it's a risk worth taking.
- Is this vulnerability really exploited ?
 - It may raise hell in your monitor, but so far nobody was ever able to correctly extract info with it. Maybe the hacker will know about your postgresql engine version. Great. But no data will ever be stolen anyway. So, whatever.
- Is this fixable ?
 - Sometimes, a CVE is tied with a specific and non-replaceable feature. Would you rather let go of your business solution and try to strengthen your stack elsewhere or patch it anyway with the risk of breaking any dependency and make your whole stack unexploitable ?

Velocity impact

Strong scanning is always better than none at all, yet think about the consequences of a powerful solution :

- Each error/vulnerability raised will stop the pipeline entirely. The “Stop-and-fix” philosophy will make your devs stop everything currently ongoing to fix errors, and slow down the entire feature releases.
- False positives will grow as your security is risen. Some CVE are not as dangerous as the reports indicate, some behaviors may sound phony and are yet legitimate. Whenever a blocking occurs, the menace must be addressed humanely and securities changed accordingly.
- The more security test you have, the more slower the whole deployment of a single feature gets. Even a change in a front-end form can lead to a whole 2 hours wasted. Again it's up to the IT director (or security manager) to balance things between

acceptable risks and productivity loss acceptance. And if any error raise at the end of the pipeline, time wasted will be of significant scale.

To address velocity, **best is to set up testing (SAST/DAST/SCA) per project or even per feature**, as to better target failing points and not waste any time without security compromise.

Immediate controls to undertake

Pre-requisites :

1. Deploy a self-hosted Gitlab instance to keep the codebase internal. Best would be an on-premise deployment behind a VPN wall.
2. Deploy a VAULT instance (or any dedicated secret manager with strong local encryption) and declare any secret as a dynamic variable.
3. Define Dev and DevOps (infrastructure) groups. The first making applications, the second making infrastructure as code to deploy it.
4. Define a CI/CD pipeline within each project you may have, tailor-made for its needs.

Therefore :

1. Implement Secret Scanning (basic SAST per project) from the Gitlab templates
2. Basic Software Composition Analysis (SCA per project) from the Gitlab templates
3. IaC scanning (to scan for errors in Terraform/Kubernetes files to prevent leaks) from the Gitlab templates

Progressive controls to implement

1. Enforce strict SAST and Code Quality testing (for deep scanning and tech debt evaluation)
2. Make advanced SCA and compliance testing (to look beyond CVE and evaluate overall health of the stack, as well as legal status of licences used)
3. Implement DAST scanning for the end of “dev” branches and “staging” branch (to simulate hacking attacks in a sandboxing environment)
4. Make API testing (security leaks, requests rate enforcement, massive malformed data injections, etc)
5. Use [Sigstore binary validation](#), as to make sure the code deployed has not been tampered with by a man-in-the-middle.

Global thinking about security

As strong scans are implemented, now think outside CI/CD logic and take a holistic approach :

1. Adopt a “Shift-Left” strategy. Running small and fast scans locally by each dev in it's IDE before pushing code will remove unnecessary scanning time and catch bug at minute 1.
2. Adopt a “Gradual Enforcement” strategy. Implement a two-times scanning, first with a “Warning only” to fix small bugs, then pass on a “Hard Block” scanning pass to raise critical issues.
3. Make “Tiered scanning”. May sound redundant with the previous point, but no. Idea is to run fast and light scans on each commits, and let slow, complete scans run outside production hours (at night for instance) before any major release to provide a complete in-depth report the morning after.
4. Think about “Exceptions”. Again, talk about the IT director/security manager about acceptable risks to not block every pipeline whenever a dependency sneeze without saying “Bless you”.

Exercise 10 - Continuous improvement and security indicators

1. Security Key Performance Indicators (KPIs)

To measure the effectiveness of our security posture, we track 10 essential KPIs. These metrics allow us to verify our alignment with Zero Trust and the mitigation of STRIDE threats.

KPI	Description	Data Source	Target	Scope
MTTD	Mean Time To Detect an incident	SIEM Logs	< 2 hours	All
MTTR	Mean Time To Respond	Incident Reports	< 4 hours	All
Critical Vulns	Number of open critical vulnerabilities	SAST/DAST Tools	0	Code/Infra
MFA Success	% of successful MFA logins for Admins	Auth Logs	100%	Admin (A2)
Blocked DoS	Rate of DoS attacks successfully mitigated	WAF/CDN	> 99%	C2/C5
Test Coverage	% of security requirements covered by tests	CI/CD Pipeline	> 90%	Global Flow
Secret Leaks	Number of secrets detected in Git repositories	Secret Scanner	0	Dev Repos
BOLA/IDOR	Unauthorized access attempts to other user records	API Gateway	< 1 month	Customer Data (D1)
Patch Latency	Avg time to apply a critical security patch	Ops dashboard	< 24 hours	Servers (C2, C3)
Training Rate	% of staff who completed security awareness	HR System	100%	All Personnel

*IDOR (Insecure Direct Object Reference), BOLA (Broken Object Level Authorization)

2. Continuous Improvement Cycle (PDCA)

To implement a continuous improvement framework, ShopNow will apply the Plan-Do-Check-Act(PDCA) methodology:

- **Plan:** Define the platform's security objectives based on the STRIDE threat model and establish the target KPIs.
- **Do:** Implement the technical and organizational security measures, such as the DevSecOps pipeline and Zero Trust access controls.
- **Check:** Measure the effectiveness of these measures by constantly monitoring the defined KPIs.
- **Act:** Adjust the security architecture, update requirements, and refine test scenarios based on the gaps identified during the 'Check' phase to prevent recurring incidents.

3. Link with Previous Exercises

The KPIs defined above are essential to validate and improve the work done in all previous phases:

- **Verify Requirements (Ex. 3):** Tracking the "Admin MFA Success" KPI proves that the requirement S2 (Mandatory MFA) is actively enforced.
- **Validate Zero Trust (Ex. 4):** Monitoring "Blocked Credential Stuffing" and strict access metrics validates that the "Never Trust, Always Verify" architecture is functioning properly.
- **Adjusts Test (Ex. 5):** If the "Open Critical Vulns" KPI rises in production, it means the "Security Test Coverage" in the DevSecOps pipeline must be adjusted to catch these flaws earlier.
- **Improve Detection (Ex. 6):** Measuring **MTTD** and **MTTR** directly evaluates the efficiency of our SIEM rules and Incident Response Playbooks. High times indicate a need to tune alerting rules.

4. Governance & Roles

Security is a shared responsibility. The following governance structure distributes the roles across ShopNow's teams:

- **CTO:** Defines the overarching security vision, allocates budget, and ensures alignment with business goals.
- **Product Team:** Responsible for risk arbitration, balancing the roadmap between new feature development and security debt remediation.
- **Security Team (SecOps):** In charge of threat detection, real-time incident response, SIEM management, and KPI tracking.
- **Developers:** Responsible for writing secure code, integrating security testing in their daily workflows, and fixing vulnerabilities identified by SAST/SCA.
- **Ops:** Manages secure infrastructure provisioning, system monitoring, and the application of security patches.

5. Ensuring Security Remains Alive Over Time

To guarantee that security does not become a forgotten “one-off project”, ShopNow must:

- **Automate Security:** Integrate security checks deeply into the CI/CD pipeline so they run automatically without manual effort.
- **Regular Security Reviews:** Conduct quarterly reviews of the KPIs to identify negative trends and update the threat model.
- **Continuous Training:** Provide regular secure coding training for developers to build a strong security culture.
- **Red Teaming / Pentesting:** Perform annual external penetration test to challenge the existing defenses and uncover blind spots.

6. Priority KPIs at Launch

When launching the platform, the operational focus should be narrow and highly impactful. The following KPIs must be tracked with absolute priority:

1. **Open Critical Vulnerabilities:** To ensure the platform does not go live with known fatal flaws (Zero-day prevention).
2. **Admin MFA Success Rate:** To protect the most critical asset-administrative access-from day one.
3. **MTTD & MTTR:** To ensure that if an incident occurs during the chaotic launch period, the SecOps team is capable of detecting and stopping it immediately.