GitGuardian

Secrets Management Maturity Model

As organizations are looking to develop secure digital services faster, the DevSecOps movement has seen its popularity soar, with the promise of breaking the silo between development, operations, and security. Although many tools and practices have emerged to support the development of "secure by default" applications for the cloud, the matter is still far from resolved. Secrets management, in particular, remains a thorny issue even for the most mature organizations. With hyperconnected systems, secrets have become omnipresent along the software development cycle, making the legacy security perimeter obsolete.

With this document, we wish to contribute to the consolidation of knowledge around DevSecOps practices by introducing a secrets management maturity model.

Information security leaders who would like to start by doing a quick assessment of their security posture can take the following five-minute questionnaire (it's completely anonymous):
Secrets Management Maturity Questionnaire
The result chart at the end will provide an overview of their current level of secrets management maturity, complete with weaknesses and strengths, and invite them to jump directly to the most relevant part of this white paper.

A maturity model is a tool that helps people assess the current effectiveness of a person or group and supports figuring out what capabilities they need to acquire next in order to improve their performance.

- Martin Fowler

01

A DevSecOps Challenge

A modern application uses many external resources (databases, third-party SaaS applications, other microservices) that require credentials. Not only that, but the development cycle itself sometimes requires the interaction of many entities and services which can be running on the same machine or on the other side of the world (eg. a local development instance of a microservice using a remote database, a testing CI pipeline querying a third party API). All these interactions rely on secrets to work as intended, and only as intended.

A cornerstone of security in general, secrets are essential in keeping information and compute resources secure and out of threat actors' reach. Well-kept secrets, such as passwords and other authentication credentials, ideally allow the right identities (human or machine) to access the right thing at the right time.

The consequence of leaked credentials can be devastating. Exposed secrets are the low-hanging fruits hackers always look for to gain initial access or privilege escalation, as recently again demonstrated by major attacks against Uber (2022) or Codecov (2021).

For organizations, the adoption of DevOps and agile practices completely reshuffled the cards when it comes to the notion of a "security perimeter". Today's hyperconnected systems bring an immense challenge to security in general and secrets management in particular since the number of credentials in use has grown exponentially.

Developers, operations, and application security engineers are all faced with this increasing complexity and try their best to balance the risks and rewards of this process across the development cycle. Managing secrets is easier said than done, and there is no one-size-fits-all due to the diversity of use cases. Hence the importance of a maturity model.

02

No Silver bullet

Between development secrets used by developers, machine authentication at the build and testing phase, application secrets, infrastructure secrets, etc., secrets can vary widely in origin, nature, and function. Their sheer number in software companies means that there simply cannot be a one-size-fits-all solution for managing them.

Even for the most mature DevSecOps organizations or teams, secrets management is very difficult to do well, because it is a matter of striking a subtle balance between security and flexibility. This second point is very important for one simple reason: in modern development teams, secrets are used by all people & assets.

Making it hard to use secrets will inevitably lead to the bypassing of the protective layers in place.

On the ground, it is easy to see that there is a gap between theory and practice when it comes to handling and sharing credentials in a team, a department, or an organization. For example, the organization may pay for a cloud-based secrets manager, a vault, or maybe even for a dedicated team to administer these tools, which makes it falsely think it has solved this problem. But under further scrutiny, it would realize that the long-lived credentials are also stored on the devs' local machines for convenience.

Example: Uber (2022)

In this attack scenario, the attacker found a Thycotic admin username and password hard-coded in a Powershell script on a network drive. Thycotic is a Privilege Access Management solution that stores secrets for accounts with high privileges for a multitude of assets: in that case, AWS, GCP, GSuite, Slack, SentinelOne, HackerOne, and more. A single hardcoded credential was enough to obtain valid accounts for all these systems.

This is why having a holistic approach to secrets is essential to account for their double-edged nature: secrets are at the same time a major risk that needs to be closely monitored, but also an indispensable commodity to spin the famous DevOps infinity symbol (∞) without friction.

03

A Quick Look at the Secrets Management Panorama

In the secrets management model, we refer to the various ways secrets can be made available from the most rudimental to the most advanced:

- Hardcoded in source code and templates
- Grouped in common, unencrypted files, such as .env, outside of the git repository
- Encrypted in a GitOps or sealed secrets approach, with decryption key stored in a vault
- Stored in a vault and distributed through a secrets management service
- Generated just-in-time and ephemeral, through a complex secrets infrastructure

All the major VCS platforms (GitHub, GitLab, BitBucket), CI/CD vendors (CircleCI, Jenkins, TravisCI, etc.), and cloud providers (AWS, GCP, and Azure) come with their own secrets management facility. More precisely, they offer a mechanism for storing sensitive values (vault service), and a mechanism to inject and serve these credentials when required (secrets management service), although their exact features can vary.

Side-note

In this paper, we have chosen to distinguish between three functions of secret management tools, which are often overlapping in commercial offerings:

- A **vault** service offers a secure, centralized key store with tight access controls.
- A **secrets management** service allows to create, serve and rotate credentials (notably, dynamic or ephemeral secrets).
- A **key management system (KMS)** service allows the management of cryptographic keys.

Popular options to store, manage and synchronize secrets across environments include HashiCorp Vault, Akeyless, and Doppler. They offer the advantage of being an external single source of truth for all the aforementioned services while implementing state-of-the-art encryption and access controls.

But again, the variety of use cases for secrets makes it very difficult to make all the DevOps cycle phases dependent on one single service. The state of secrets management in modern software development shops is always a mixture of ad-hoc solutions, depending on which link of the chain is considered and the level of maturity.

One common flaw of secrets management, in general, is that it fails to account for human error. A developer hard-coding an API key to test a program on their local machine and then accidentally committing this change to a code repository would have a very difficult time figuring out:

1. he leaked a secret
2. how to revoke that secret
3. how rotation might impact downstream services dependent on that credential.

Multiple back-and-forths with CloudOps or SRE and AppSec engineers would be needed.

No matter the maturity of processes, seniority of engineers, the rigidity of policies, or the sophistication of tools, **secrets will be hardcoded somewhere at some point.**

In other words, failure modes should be an integral part of secrets management thinking. Let's look at the details of what this means.

04

Key considerations for secrets management

First, we need to define what we consider to be part of secrets management and what we consider to be out of scope, even if related to the domain.

In a holistic approach to secrets, we consider essential to have visibility over an organization's perimeter. Therefore, being able to monitor secrets not only where they are supposed to be, but (more importantly) where they really are is a must-have.

In other words, if secrets management is about protecting your secrets, it cannot go without secrets detection.

You just can't protect what you don't see. In the previous example, it is easy to understand that despite having a solution to manage secrets across the organization, the lack of secrets detection is defeating the very purpose of this solution in the first place. While the two tools have distinct usage, they share the same objective: keeping secrets secret.

From the same perspective, we should remember that managing secrets is a security feature. Since security can't be achieved without incident response capabilities,

remediation is also a core aspect of sane credentials management.

Failing to acknowledge errors and misbehaviors is a recipe for failure in any security context. Secrets are no different and taking into account the human factor is essential: the most recognized industry reports[1,2], as well as our own data, consistently point at human errors as the number one source of breaches and leaks. Preparing for when (not if) leaks happen must be part of the secrets management process.

[1] Cost of a data breach 2022, IBM
[2] 2022 Data Breach Investigations Report, Verizon

To sum up,

our model considers that secrets detection and remediation are unvoidable aspects of secrets management. It should help teams make their goals more explicit when it comes to:

Secret lifecycle

Creation, lifetime (for how long is it valid?), regular rotation.

Secrets scope

What kind of action is allowed by a certain secret?

Secrets detection

How do we make sure we know when a leak happens?

Remediation procedures

What happens after a secret is leaked?

05

The Model

We propose a 5 level maturity model for secrets management, detection, and remediation in 4 main areas of the software development cycle:

- Developer environment
- Source code management
- CI/CD pipeline & artifacts
- Runtime environments

Splitting the DevOps cycle into distinct phases is necessarily an arbitrary decision, and even more so when talking about something as transversal as secrets. Therefore, we decided to separate it according to the following logic:

Developer environment

This section corresponds to the daily activities of the developer and how they manage to get access to and share the secrets they need to test their programs and scripts. As awareness about the problem of secrets sprawl rises, some developers encrypt secrets before sharing them, and potentially shield their working environment from leaks with pre-commit hooks. But that's not all: developers are also on the front line when it comes to secrets remediation. Progressively involving developers in the remediation process is a significant step toward a mature secrets management program.
This is also where we chose to talk about the evolution of secrets' rotation policies and process, eventually leading to full automation.

Source code management (source code & Infrastructure-as-Code)

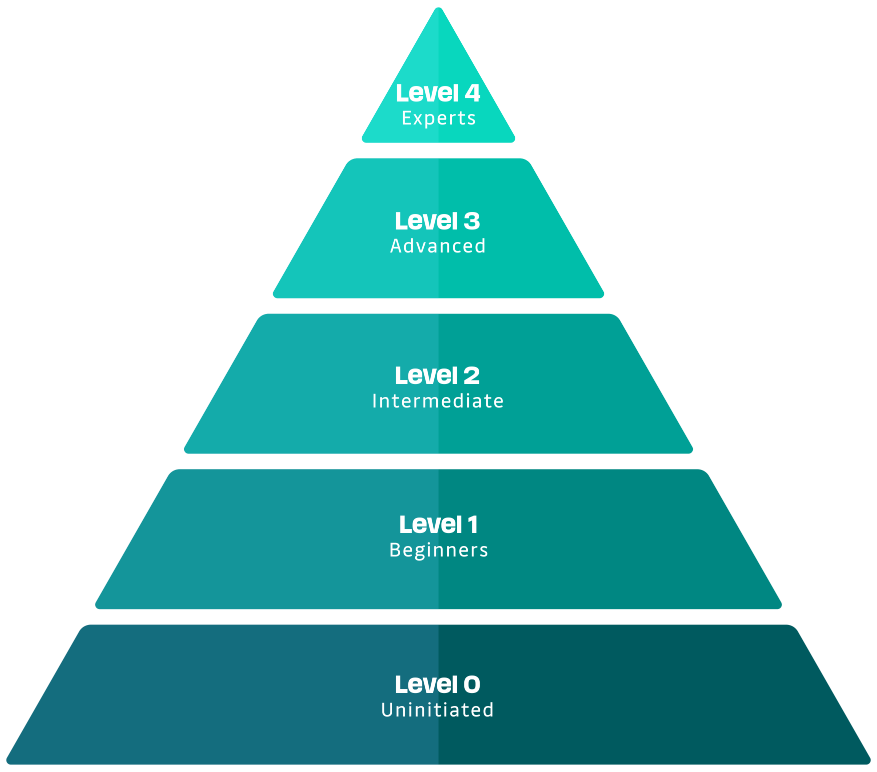
This section is about how source code and templates (Terraform, Dockerfile, etc.), at a global level, can be shielded from secrets sprawl. We consider that secrets found in IaC templates are probably giving access to cloud resources like storage, IAM systems, etc.. and should be removed first. Central repositories' administrators are in charge of setting up the right controls to continuously scan for secrets before they can be considered compromised.
This is also where we chose to talk about the evolution of the remediation process.

CI/CD pipelines & artifacts

This section is about the build process and the resulting artifacts. It is not uncommon to find secrets leaked in Docker images or even binaries. These should be removed in the first place, and the build process itself should eventually include a scanning step to make sure that no secrets can find their way into the artifacts or the build logs themselves. Also, the credentials used in the build process should be rotated and fine-tuned to a very restricted scope to prevent potential lateral movements.

Runtime environment

Finally, at runtime applications also need secrets. The classic examples would be a database connection string for a web app or third-party API keys. Provisioning these secrets at runtime requires the same thoughtful design decisions about secrets' lifetime, scopes, rotation, and, maybe more importantly, how to deal with a leak without causing downtime. We observed in the past that a leak could force engineers to temporally shut down part of the production, with direct consequences for the business. Preventing such an outcome definitely has its place in a secrets management mindmap.



Level 0

No processes or tools for managing secrets — secrets sprawl in the SDLC. No detection (and remediation) in place.

Level 1

Secrets are unencrypted at rest but grouped in configuration files shared across multiple teams. Scanning for secrets is triggered manually at times, but developers are rarely involved in remediation.

Level 2

Secrets are checked encrypted into repositories with decryption keys stored in a secure vault. Secrets scanning and rotation are performed periodically.

Level 3

Secrets are scoped, stored in a vault and shared using a secrets manager. Automated detection on shared repositories and final artifacts is continuous.

Level 4

Secrets are scoped, stored in a vault and shared with a secrets manager. Detection is preventive and integrated into development workflows (local workstations, CI pipelines, etc.). Developers remediate their incidents.

LEVEL 0 — UNINITIATED

High exposure risk

	Secrets management	Secrets detection
Developer environments	Secrets are **shared in clear** text through private channels and **stored unencrypted in local config files.**	No detection in place.
Source Control *(Source code & Infra-as-Code)*	Secrets can be found anywhere in files. They are **checked unencrypted into private repositories.**	No detection in place.
CI/CD pipelines & software artifacts	Secrets are **embedded in final artifacts such as container images.** VCS and 3rd party (e.g. code quality tools) access tokens are **harcoded in build scripts.**	No detection in place.
Runtime environments	Secrets are **embedded in deployment scripts.** Sensitive variables are **printed in server logs.**	Low or no observability on production secrets. No rotation planning or strategy.

LEVEL 1 — BEGINNERS

High exposure risk

	Secrets management	Secrets detection
Developer environments	Secrets are **unencrypted in config files** but can be encrypted before sharing with other developers.	No detection in place.
Source Control *(Source code & Infra-as-Code)*	Secrets are **grouped in config files** and accessed using environment variables. IaC secrets are **stored externally** — by the cloud services provider (e.g. AWS, GCP).	Source code and IaC templates are **manually and periodically scanned for secrets.** High-severity incidents are **remediated with limited help from developers.**
CI/CD pipelines & software artifacts	Final artifacts **do not contain any secrets.** Pipeline secrets are **stored in the build system.** Sensitive variables are **redacted from build logs.**	Build outputs (e.g., Docker images) are **scanned for secrets manually** before a release.
Runtime environments	Secrets are **grouped in a common config file.** Sensitive variables are **redacted from logs.** Secrets are **not scoped by environment.**	Production secrets are monitored through **a single pane of glass**, but there is no rotation strategy.

LEVEL 2 — INTERMEDIATE

Moderate exposure risk

	Secrets management	Secrets detection
Developer environments	Secrets are **stored in a vault and shared through a secrets manager.** Developer environment secrets are correctly scoped.	Scanning is **triggered manually at times by developers on their local workstations.**
Source Control *(Source code & Infra-as-Code)*	Secrets are **encrypted and checked into repositories** (master key is stored externally).	Critical repositories are **continuously scanned at the pull/merge requests stage.** Developers contribute to remediation but it is still not a well established and documented process.
CI/CD pipelines & software artifacts	Secrets are **stored in the build process.** Secrets are **scoped; permissions abide by the principle of least privilege.**	Build outputs (e.g. Docker images) are **continuously scanned for secrets.**
Runtime environments	Secrets (or master decryption key) are **stored in a vault and dynamically loaded with a secrets manager** with minimal access controls.	Production secrets are monitored, and there is a **remediation process in case of exposure or compromise.**

LEVEL 3 — ADVANCED

Low exposure risk

	Secrets management	Secrets detection
Developer environments	Secrets are **stored in a vault and shared exclusively through a secrets manager.** Secrets **rotation policy is well defined.**	Scanning **before pushing code** (pre-commit/ pre-push) is adopted by security champions. Developers are **systematically involved in the remediation process.**
Source Control *(Source code & Infra-as-Code)*	Secrets are **no longer embedded in the current** source code revision. Secrets **rotation policy is well defined.**	All repositories are **continuously scanned** for hardcoded credentials. Collaboration on incident remediation **is mandatory for all developers. Historical incidents** are being triaged.
CI/CD pipelines & software artifacts	Pipeline secrets are **stored in vault and loaded using a secrets manager.** Secrets are **scoped; permissions follow the principle of least privilege.**	Informative scanning is **enforced for all branches** (feature, hotfix, etc.) **in CI pipelines.**
Runtime environments	Secrets are **stored in a vault and dynamically loaded from a secrets manager.** Secrets are scoped and **access is monitored/logged.**	Production secrets are **scheduled for regular rotation.**

LEVEL 4 — EXPERTS

Limited exposure risk

	Secrets management	Secrets detection
Developer environments	Secrets are **stored in a vault with access controls and logging. Dynamic secrets with limited scope** are used for development when possible.	Scanning **before pushing code** (pre-commit/pre-push) **is adopted by all developers.** Developers are **systematically involved in the remediation process.**
Source Control *(Source code & Infra-as-Code)*	No presence of **valid hardcoded secrets in past or current revisions of source code.**	All **repositories are continuously monitored and blocking scans (pre-receive) are setup. Remediation workflows are automated** and fixing is handled by developers.
CI/CD pipelines & software artifacts	Pipeline secrets are **short-lived, scoped, and stored in an external vault.** If possible, build secrets are **replaced by OpenID Connect (OIDC) tokens** for auth.	Blocking scanning is **enforced for all branches** (features, hotfix, etc.) **in CI pipelines.**
Runtime environments	Secrets are **stored in a vault and loaded dynamically from a secrets manager. Restrictive access controls and logging** are enforced.	Production secrets rotation is **scheduled & automated.**

Bonus: Kubernetes

As more and more organizations are making the shift to cloud-native technologies, Kubernetes has become the de facto choice to orchestrate container-based applications. That's why we decided to give it its own place as a bonus in the Matrix, allowing us to be more specific about this unique technology.

Out of the box, Kubernetes supports 'Secrets' objects to store sensitive information like passwords, tokens, SSH keys, etc. securely. This construct eliminates the need for hard-coding sensitive data in the application code but needs to be handled with extreme caution.

	Level 0	Level 1	Level 2	Level 3	Level 4
kubernetes	Secrets are **hardcoded** in manifest.	Secrets are stored in a **Kubernetes Secrets Object.**	Secrets are loaded from **external vault.**	Secrets are loaded **directly into the pods.** Pods share the same credentials.	Secrets are ephemeral and **each pod has its own set of secrets.**

06

Other considerations

Like any security topic, secrets management does not exist in isolation. To conduct a thorough security posture analysis, it is important to consider related domains. The ones we have listed here have been deliberately excluded from the matrix since we consider that they go beyond the strict framework of secrets management.

However, this does not mean that they are optional: on the contrary, they require an equally important consideration.
In other words, if you want to put effort into improving secrets management policies, you cannot overlook the following:

Access controls

Access controls are used to answer the question: who has access to what and when? It is obvious that nothing can be secret if it can be read or updated by anyone. Engineers should not have access to all secrets in the secrets management system, so the least privilege principle should serve as a guide to progress in maturity. Our matrix does talk about the scope of the secrets at different levels: the secret management system needs to provide the ability to configure fine granular access controls on each object and component to accomplish the least privilege-required principle. Yet well-implemented access controls are a much bigger cybersecurity topic (including Identity and Access Management) that we do not have the vocation to cover in all its subtleties.

Cryptographic & encryption quality

The security of a secret depends, among other things, on its basic qualities: if the secret can be any character string chosen by a human, chances are that it will be easily cracked by brute force or a dictionary-like attack. On the other side of the spectrum, the most secure secrets are generated by high entropy hardware random number generators included in specialized HSMs.
They would probably be too costly to implement & operate for most use cases. Furthermore, having high entropy secrets is great, but useless if they’re stored in cleartext in a database. Choosing high entropy secrets, the right hashing & encryption algorithms for your secrets at rest & in transit is another important part of any secrets management policy.

Logging and auditing

Logging is an integral part of maintaining an organization's security posture, and this is also true for monitoring secrets. You should be able to know who requested a secret and for what system and role, when it was used and by which source, when it expired etc. Authentication and authorization errors are also a valuable source of security information. Having auditing tools and processes is definitely something to expect from a mature secrets management system.

07

Summary

Secrets management has a considerable impact on the security posture of organizations. With the advent of DevOps, the amount of sensitive information in use in software factories has exploded, creating a gap between theory and practice: in theory, all the "crown jewels" are closely guarded within a vault and scrupulously respect the principle of least privilege. In practice, teams continue to generate large quantities of secrets as they scale services and infrastructures, bypassing outdated controls. Secrets are easily exposed, sometimes without anyone noticing. This is a difficult problem to solve, even with all the flexibility automation brings us. That's why some organizations do not invest sufficient time & effort into it despite the percentage of security breaches originating from exposed secrets.

Reducing this attack surface requires the right controls to be placed along the DevOps cycle, and to encourage collaboration between developers, security engineers, and operations. Not taking into account the human factor in the management of secrets would be a serious mistake.

No matter the technology, leaks will happen. The response lies at the intersection of people, tools, and processes. Having a plan to be notified as early as possible when a leak happens and to face incidents with peace of mind is a must.

We hope that our maturity model will be useful to allow you to take stock of the actual state of secrets management in your organization, and more importantly, what are the steps to improve it.

08

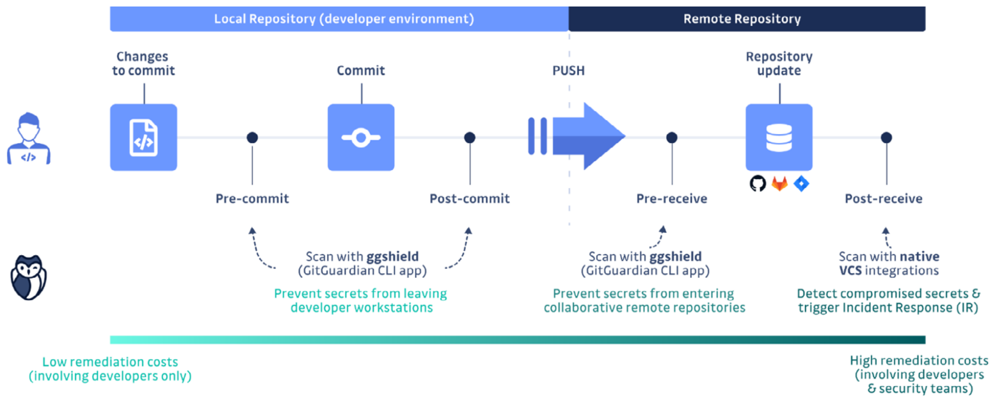
About GitGuardian

GitGuardian, founded in 2017 by Jérémy Thomas and Eric Fourrier, has rapidly emerged as the leader in automated secrets detection and is now focused on providing a comprehensive code security platform. The company has raised a $56M total investment to date from Eurazeo, Sapphire, Balderton, and notable tech entrepreneurs like Scott Chacon, co-founder of GitHub, and Solomon Hykes, co-founder of Docker.

GitGuardian helps software-driven organizations strengthen their overall security posture and comply with application security frameworks and standards by reducing the risks of secrets exposure across the SDLC. With GitGuardian's policy engine, security teams can monitor and enforce rules across all their VCS, DevOps tools, and infrastructure-as-code configurations.

With more than 210k installs, GitGuardian is by far the n°1 security application on the GitHub Marketplace. Its enterprise-grade features enable AppSec and Development teams to deliver secret-free code in a truly collaborative manner. By pulling developers closer to the remediation process, organizations can achieve shorter fix times.

Its detection engine is trained against more than a billion public GitHub commits every year, and it covers 350+ types of secrets such as API keys, database connection strings, private keys, certificates, and more.


Local Repository (developer environment)
Remote Repository
Changes to commit
Commit
PUSH
Repository update
Pre-commit
Post-commit
Pre-receive
Post-receive
Scan with ggshield (GitGuardian CLI app)
Prevent secrets from leaving developer workstations
Scan with ggshield (GitGuardian CLI app)
Prevent secrets from entering collaborative remote repositories
Scan with native VCS integrations
Detect compromised secrets & trigger Incident Response (IR)
Low remediation costs (involving developers only)
High remediation costs (involving developers & security teams)

GitGuardian monitors every step of the pipeline

Learn more about GitGuardian:

[GitGuardian - Website]

[GitGuardian - Internal Monitoring - GitHub Marketplace]

[State of Secrets Sprawl]

GitGuardian