PuigFlows

# Gestión de Secretos en Producción: de .env a Workload Identity, Rotación sin Downtime y Runbook de Fugas

Guía práctica para dejar de perder claves por operativa y no por criptografía: qué cuenta como secreto y qué no, por qué .env deja de funcionar en producción, las tres capas de detección que impiden que un secreto entre en el repositorio (pre-commit con gitleaks, escaneo del historial completo en CI y push protection) y SOPS con age cuando la configuración debe vivir en Git. Cubre el cambio de mentalidad que más riesgo elimina —preferir identidad a secreto con IRSA, Workload Identity y federación OIDC en CI—, incluida la política de confianza de IAM y el comodín en sub que convierte una arquitectura sin claves estáticas en una puerta abierta; credenciales dinámicas de PostgreSQL con Vault y sus dos trampas operativas (renovación en el pool y roles huérfanos); la diferencia real entre entregar el secreto por variable de entorno o por fichero montado (/proc, herencia a subprocesos y por qué env nunca se actualiza al rotar); configuración con pydantic-settings y SecretStr sobre secrets_dir; un envoltorio RotatingSecret que relee el fichero cuando el kubelet lo actualiza; Kubernetes con cifrado en reposo de etcd, RBAC por namespace y sincronización con External Secrets Operator (ClusterSecretStore, ExternalSecret y montaje como volumen); rotación sin downtime con el patrón de dos credenciales y la métrica por key_id que te dice cuándo es seguro revocar; redacción de secretos en los logs con structlog; un runbook de fuga que empieza por revocar y no por rotar; ocho errores recurrentes y checklist de producción.

**Josue Garcia**
26 de agosto de 2026

Lectura aprox. 38 min
Español

# Contenido

Índice de secciones

Casi ningún equipo pierde una clave de producción por un ataque sofisticado. La pierde porque alguien hizo `git add .` con un `.env` dentro, porque una excepción no capturada mandó la cadena de conexión entera a Sentry, o porque la clave del proveedor de pagos lleva cuatro años sin cambiar y ya vive en el portátil de tres personas que se fueron de la empresa. Los incidentes de secretos son casi siempre incidentes de operativa, no de criptografía.

Esta guía recorre el camino completo: desde el `.env` que todos empezamos usando hasta una arquitectura donde la mayoría de tus servicios no tienen ningún secreto de larga vida que perder. No es un catálogo de herramientas, es un orden de prioridades: primero eliminar el secreto, luego acortarle la vida, y sólo al final protegerlo bien.

La guía está pensada para leerse entera una vez y volver a ella por secciones. La primera mitad establece el orden de prioridades — qué es un secreto, cómo evitar que entre en el repositorio, cómo sustituirlo por identidad, cómo llega al proceso y cómo se rota sin downtime—. La segunda entra en el detalle de los sitios donde los secretos se pierden de verdad: el modelo de amenaza y el radio de explosión, los pipelines de CI/CD y la federación OIDC (incluido el cambio del claim inmutable de julio de 2026), las imágenes de contenedor, el estado de Terraform, las credenciales dinámicas de Vault con sus dos trampas operativas, el cifrado sobre de KMS, el caso del cliente móvil y web, y cómo medir si el programa está mejorando o solo generando documentos.

## Qué cuenta como secreto (y qué no)

Antes de montar nada, conviene delimitar. Un secreto es cualquier valor que, en manos de otro, permite actuar en tu nombre o leer datos que no le corresponden:

- Contraseñas de base de datos y cadenas de conexión completas.
- Claves de API de terceros (pagos, correo, proveedores de LLM).
- Claves privadas: TLS, firma de JWT, SSH, firma de artefactos.
- Secretos de firma de webhooks y claves de cifrado de datos en reposo.
- Tokens de sesión y credenciales de servicio a servicio.

Y esto *no* lo es: nombres de host internos, nombres de bucket, IDs de proyecto, claves públicas, flags de configuración. Meterlo todo en el gestor de secretos parece prudente y es contraproducente: convierte cada cambio trivial de configuración en un trámite, y cuando el proceso pesa demasiado la gente lo rodea. La regla práctica es preguntarse qué pasa si ese valor aparece en una captura de pantalla en un canal de Slack. Si la respuesta es "nada", es configuración.

## Por qué .env deja de funcionar

El fichero `.env` resuelve un problema real en local y crea cuatro en producción:

- **Está a una distracción de acabar en Git.** El `.gitignore` protege hasta que alguien copia `.env.example` a `.env.staging`, nombre que el patrón no cubre.
- **No tiene dueño ni historia.** Nadie sabe quién generó esa clave, cuándo, ni qué permisos tiene. Y por tanto nadie se atreve a rotarla.
- **Se propaga por copia.** El onboarding de cada persona nueva consiste, literalmente, en que alguien le pase los secretos de producción por un canal privado.
- **No se puede revocar de forma dirigida.** Si se filtra, no puedes invalidar una copia: invalidas la clave para todo el mundo, incluida producción.

El objetivo no es prohibir `.env`. En desarrollo local, con credenciales de juguete que no valen nada, es perfecto. El objetivo es que el fichero nunca contenga un valor que importe.

## Regla 1: el secreto no entra en el repositorio

Esta defensa tiene que ser automática, porque depender de la disciplina humana falla con probabilidad 1 a lo largo de suficientes commits. Se monta en tres capas.

### Capa 1: bloqueo antes del commit

Un hook de `pre-commit` con gitleaks escanea sólo lo que estás a punto de confirmar, así que es rápido y no molesta:

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/gitleaks/gitleaks
    # Fija una etiqueta concreta y actualízala de forma controlada:
    # un "latest" flotante en un hook es otra dependencia sin revisar.
    rev: v8.x.y
    hooks:
      - id: gitleaks
```

### Capa 2: escaneo del historial completo en CI

El hook local sólo ve el commit actual y cualquiera puede saltárselo con `--no-verify`. En CI hay que clonar el historial entero, porque un secreto borrado en un commit posterior sigue siendo recuperable en el anterior:

```
- uses: actions/checkout@v4
  with:
    fetch-depth: 0   # sin esto sólo se escanea el último commit

- uses: gitleaks/gitleaks-action@v2
```

### Capa 3: protección en la plataforma

Activa la protección de push de tu forja (en GitHub, *secret scanning* con *push protection*). Es la única capa que atrapa el push que hace alguien desde una máquina sin tus hooks, y además avisa al proveedor emisor: muchos servicios revocan automáticamente una clave que aparece en un repositorio público.

### Y si necesito la configuración en el repositorio: SOPS

Hay equipos que quieren la configuración versionada junto al código, incluida la parte sensible. SOPS cifra sólo los valores y deja las claves en claro, así que el diff sigue siendo legible y revisable:

```
# Genera una clave age; la privada NUNCA entra en el repositorio.
age-keygen -o age.key

# Cifra sólo los valores bajo data/stringData, no el YAML entero.
sops --encrypt \
```

```
    --age age1ql3z7hjy54pw3hyww5ayyfg7zqgvc7w3j2elw8zmrj2kg5sfn9aqmcac8p \
    --encrypted-regex '^(data|stringData)$' \
    secrets.yaml > secrets.enc.yaml

git add secrets.enc.yaml   # esto sí puede vivir en el repositorio
```

Es una solución honesta para equipos pequeños y para GitOps, con una advertencia importante: SOPS traslada el problema, no lo elimina. Ahora tienes que proteger la clave de descifrado, y el ciclo de rotación de un secreto pasa por un pull request y un despliegue. Si tu organización ya tiene un gestor de secretos, úsalo; SOPS brilla cuando no lo tienes.

## Regla 2: prefiere identidad a secreto

El mejor secreto es el que no existe. Cuando un proceso necesita hablar con un servicio de la misma nube, no le des una clave: dale una **identidad**. La plataforma firma un token corto que el proveedor valida, y no hay ningún valor de larga vida almacenado en ningún sitio.

- **AWS:** IRSA o EKS Pod Identity para pods; roles de instancia para EC2.
- **GCP:** Workload Identity Federation.
- **Azure:** Workload Identity / Managed Identity.
- **CI/CD:** federación OIDC desde GitHub Actions, GitLab CI o Buildkite.

El caso de CI es el de mayor retorno inmediato, porque las variables de CI son el sitio donde más claves estáticas de administrador se acumulan. Con OIDC desaparecen:

```
name: deploy
on:
  push:
    branches: [main]

permissions:
  contents: read
  id-token: write   # imprescindible: habilita el token OIDC del job

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Credenciales temporales de AWS (sin claves estáticas)
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/gha-deploy-checkout
          aws-region: eu-west-1

      - run: aws sts get-caller-identity
```

La pieza crítica no está en el workflow, sino en la política de confianza del rol. Aquí es donde se comete el error que anula todo el trabajo:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {
      "Federated": "arn:aws:iam::123456789012:oidc-provider/token.actions.githubusercontent.com"
    },
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
      "StringEquals": {
        "token.actions.githubusercontent.com:aud": "sts.amazonaws.com",
        "token.actions.githubusercontent.com:sub": "repo:mi-org/checkout:ref:refs/heads/main"
      }
    }
  }]
}
```

Si omites la condición sobre `sub`, o la escribes como `StringLike` con un comodín demasiado ancho (`repo:mi-org/*:*`, o peor, `repo:*`), estás permitiendo que *cualquier* workflow de GitHub —el de cualquier persona del mundo— asuma tu rol de despliegue. Es un fallo real y recurrente en auditorías. Ata siempre el `sub` al repositorio y a la referencia concreta, y usa un rol distinto por repositorio y por entorno.

## Regla 3: si hace falta un secreto, que sea corto y dinámico

Cuando el destino no soporta identidad federada —una base de datos PostgreSQL, por ejemplo— la siguiente mejor opción es generar la credencial al vuelo con una vida útil de horas. Vault lo hace con su motor de bases de datos: crea un rol de PostgreSQL nuevo por cada petición y lo destruye al expirar.

```
vault secrets enable database

vault write database/config/checkout \
  plugin_name=postgresql-database-plugin \
  allowed_roles="checkout-app" \
  connection_url="postgresql://{{username}}:{{password}}@pg.internal:5432/checkout?sslmode=require"
\
  username="vault-rotador" \
  password="..."

# Cada credencial emitida vive 1 hora y como mucho se renueva hasta 24.
vault write database/roles/checkout-app \
  db_name=checkout \
  creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{password}}' VALID UNTIL
'{{expiration}}'; GRANT app_readwrite TO \"{{name}}\";" \
  default_ttl="1h" \
  max_ttl="24h"
```

El cambio de mentalidad es grande: ya no existe "la contraseña de la base de datos". Existe un permiso que Vault concede a una identidad, y una credencial desechable. Una filtración deja de ser una emergencia y pasa a ser una anotación en el registro de auditoría, porque el valor caduca solo.

Dos detalles operativos que se aprenden por las malas. Primero: la credencial dinámica hay que *renovarla*, y tu

pool de conexiones tiene que saber reconectar cuando la vieja expira; si abres cien conexiones al arrancar y nunca las reciclas, en una hora tienes cien conexiones con un usuario que ya no existe. Segundo: cada credencial crea un rol real en PostgreSQL, así que un TTL demasiado corto con mucho tráfico llena el catálogo de roles huérfanos. Un TTL de una hora con `max_ttl` de 24 es un punto de partida sensato.

## Cómo llega el secreto al proceso: variables de entorno frente a ficheros

Es la decisión que más gente pasa por alto y la que más disgustos da. Las variables de entorno son cómodas y tienen tres problemas concretos:

- **Se heredan.** Cualquier subproceso que lances —un hook, un script de mantenimiento, una librería que invoca `curl`— recibe todo el entorno.
- **Se leen desde fuera.** Están en `/proc/<pid>/environ`, aparecen en volcados de memoria y muchos manejadores de errores las serializan enteras en el informe de la excepción.
- **No se actualizan nunca.** Cuando rotas el secreto, el proceso sigue con el valor con el que arrancó hasta que lo reinicias. Rotar implica desplegar.

Montar los secretos como ficheros resuelve los tres. En Kubernetes, un volumen de tipo `secret` se actualiza solo cuando cambia el Secret subyacente, sin reiniciar el pod. Este es el patrón de configuración que lo aprovecha:

```
# config.py
from functools import lru_cache

from pydantic import SecretStr, model_validator
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    model_config = SettingsConfigDict(
        env_file=None,                       # en producción NO se lee ningún .env
        secrets_dir="/var/run/secrets/app",  # un fichero por secreto
        extra="forbid",                      # una variable de más es un error, no un silencio
    )

    # SecretStr nunca se imprime: su repr es "**********".
    database_url: SecretStr
    stripe_api_key: SecretStr
    environment: str = "production"

    @model_validator(mode="after")
    def rechaza_claves_de_prueba(self) -> "Settings":
        if self.environment == "production":
            if self.stripe_api_key.get_secret_value().startswith("sk_test_"):
                raise ValueError("clave de prueba de Stripe en un entorno de producción")
        return self


@lru_cache
def get_settings() -> Settings:
    # Falla al arrancar si falta un secreto, en lugar de a las 3 de la mañana
    # en la primera petición que toque ese código.
```

```
    return Settings()
```

Dos decisiones de este fragmento merecen atención. `SecretStr` hace que el valor no aparezca en un `repr()`, en un `print()` del objeto de configuración ni en la mayoría de los informes de excepción: hay que pedirlo explícitamente con `get_secret_value()`, y esa llamada es fácil de auditar con un grep. Y `extra="forbid"` convierte un `DATABSE_URL` mal escrito en un fallo de arranque ruidoso en lugar de en un servicio que se levanta con el valor por defecto y falla raro.

Para que la rotación funcione sin reiniciar, hay que releer el fichero. Este envoltorio lo hace con un coste despreciable:

```
# secret_file.py
import threading
import time
from pathlib import Path


class RotatingSecret:
    """Secreto montado como fichero que se recarga cuando el kubelet lo actualiza.

    Kubernetes proyecta los volúmenes de secretos mediante enlaces simbólicos a un
    directorio ..data que reemplaza de forma atómica. Al hacer stat() sobre la ruta
    seguimos el enlace, así que el mtime cambia en cuanto rota el Secret.
    """

    def __init__(self, path: str, ttl_s: float = 5.0) -> None:
        self._path = Path(path)
        self._ttl = ttl_s
        self._lock = threading.Lock()
        self._value: str | None = None
        self._mtime: float = -1.0
        self._checked_at: float = 0.0

    def get(self) -> str:
        now = time.monotonic()
        with self._lock:
            fresco = self._value is not None and (now - self._checked_at) < self._ttl
            if fresco:
                return self._value

            mtime = self._path.stat().st_mtime
            if mtime != self._mtime or self._value is None:
                self._value = self._path.read_text().strip()
                self._mtime = mtime
            self._checked_at = now
            return self._value


stripe_key = RotatingSecret("/var/run/secrets/app/stripe_api_key")

# En el punto de uso se pide el valor, no se captura al importar el módulo.
# Ese detalle es lo que hace que la rotación llegue al código.
def cobrar(importe_cents: int) -> None:
```

```
    cliente = StripeClient(api_key=stripe_key.get())
    ...
```

El comentario final es el que de verdad importa: si haces `API_KEY = stripe_key.get()` a nivel de módulo, has vuelto exactamente al comportamiento de una variable de entorno. La rotación en caliente sólo funciona si el valor se lee en el momento de usarlo.

## Kubernetes: External Secrets Operator

Los Secrets de Kubernetes no están cifrados: están codificados en base64, que es un formato de transporte, no una protección. Antes de nada, dos ajustes de plataforma que no son opcionales:

- **Cifrado en reposo** en etcd con una `EncryptionConfiguration` respaldada por un KMS. Sin esto, un backup de etcd es un fichero con todos tus secretos en claro.
- **RBAC estricto.** Quien pueda hacer `get secrets` en un namespace, o simplemente crear un pod en él, puede leer todos sus secretos. El namespace es la frontera de seguridad real; trátalo como tal.

Con eso resuelto, el patrón dominante es dejar la fuente de verdad en un gestor externo y sincronizarla con External Secrets Operator. Primero se declara cómo hablar con el gestor:

```yaml
# En clusters antiguos la versión del API es external-secrets.io/v1beta1.
apiVersion: external-secrets.io/v1
kind: ClusterSecretStore
metadata:
  name: aws-secrets
spec:
  provider:
    aws:
      service: SecretsManager
      region: eu-west-1
      auth:
        jwt:
          # El ServiceAccount está anotado con el rol de IAM (IRSA):
          # el propio operador tampoco guarda ninguna clave estática.
          serviceAccountRef:
            name: external-secrets
            namespace: external-secrets
```

Y después, por cada aplicación, qué secretos quiere y con qué nombres los espera:

```yaml
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: checkout
  namespace: checkout
spec:
  refreshInterval: 1h          # cada cuánto se comprueba el origen
  secretStoreRef:
    name: aws-secrets
    kind: ClusterSecretStore
  target:
    name: checkout-secrets     # el Secret de Kubernetes que se crea y mantiene
```

```
    creationPolicy: Owner
    deletionPolicy: Delete     # si borras el ExternalSecret, se borra el Secret
  data:
    - secretKey: database_url  # nombre del fichero al montarlo
      remoteRef:
        key: prod/checkout/database
        property: url
    - secretKey: stripe_api_key
      remoteRef:
        key: prod/checkout/stripe
        property: secret_key
```

Fíjate en que los nombres de `secretKey` coinciden con los campos de la clase `Settings` de antes. No es casualidad: al montar el Secret como volumen, Kubernetes crea un fichero por clave, y `secrets_dir` de pydantic-settings los lee por nombre. Toda la cadena encaja sin código de pegamento:

```
      containers:
        - name: app
          volumeMounts:
            - name: secrets
              mountPath: /var/run/secrets/app
              readOnly: true
      volumes:
        - name: secrets
          secret:
            secretName: checkout-secrets
            defaultMode: 0400   # sólo lectura para el usuario del contenedor
```

El resultado es que rotar un secreto en AWS Secrets Manager se propaga solo: ESO actualiza el Secret, el kubelet actualiza el fichero montado y `RotatingSecret` lo relee en la siguiente petición. Cero despliegues, cero reinicios.

Una alternativa a considerar para el material más sensible es el **Secrets Store CSI Driver**, que monta directamente desde el gestor externo sin materializar un Secret de Kubernetes. Es más estricto (el valor nunca pasa por etcd) a cambio de perder la integración con todo lo que espera un Secret nativo.

## Rotación sin downtime: el patrón de dos credenciales

La razón por la que casi nadie rota sus claves es que rotar "de golpe" provoca un corte: revocas la vieja, y durante los segundos o minutos que tarda el despliegue en propagarse, las instancias que aún no se han enterado fallan. La solución es que el sistema acepte dos credenciales válidas a la vez durante una ventana de solape.

Del lado de quien *verifica* una clave —una API tuya, un webhook entrante— la implementación es directa, con comparación en tiempo constante para no filtrar información por el tiempo de respuesta:

```
import hmac

from prometheus_client import Counter
```

```
# La métrica es la parte importante: sin ella no sabes cuándo es seguro revocar.
uso_de_clave = Counter("api_key_use_total", "Uso por identificador de clave", ["key_id"])


def verificar_clave(presentada: str, claves_activas: dict[str, str]) -> str | None:
    """Devuelve el key_id que coincide, o None. Acepta la nueva y la anterior."""
    for key_id, valor in claves_activas.items():
        if hmac.compare_digest(presentada, valor):
            uso_de_clave.labels(key_id=key_id).inc()
            return key_id
    return None
```

El procedimiento completo tiene cuatro fases, y la tercera es la que la gente se salta:

1. **Emitir** la credencial nueva sin tocar la vieja. Ambas son válidas.
2. **Propagar**: los consumidores empiezan a usar la nueva. Con secretos montados como ficheros esto ocurre solo; si no, es un despliegue.
3. **Esperar y verificar**: observa la métrica `api_key_use_total` filtrada por el `key_id` antiguo hasta que lleve un tiempo razonable en cero. Ese "tiempo razonable" debe cubrir tus procesos más lentos: trabajos nocturnos, un cron mensual, ese servicio que sólo se despierta al cerrar el trimestre.
4. **Revocar** la vieja. Y comprobar que efectivamente ha dejado de funcionar, porque una revocación que falla en silencio es peor que no rotar: te da una falsa sensación de seguridad.

Sobre la frecuencia, una política que funciona sin volverse ceremonia: las identidades de carga de trabajo y los tokens de menos de 24 horas no necesitan política porque rotan solos; contraseñas de base de datos, cada 90 días si no son dinámicas; tokens entre servicios, cada 30; claves de terceros de larga vida, cada 180 días y aprovechando para revisar qué permisos tienen de verdad. Las claves de firma de artefactos y las CA raíz van aparte, con rotación anual planificada y solape explícito.

## La fuga por los logs

Los repositorios reciben toda la atención, pero una parte enorme de las filtraciones reales sale por la observabilidad: un `logger.info("conectando a %s", database_url)`, un traceback que serializa el objeto de configuración, una traza de OpenTelemetry con la cabecera `Authorization` como atributo del span. Y esos datos acaban en un sistema que suele tener permisos más laxos que producción.

La defensa principal es de diseño —`SecretStr` hace que el valor no llegue al log— pero conviene una red de seguridad en el procesador de logs:

```
import re

import structlog

PATRONES = re.compile(
    r"(sk_live_[A-Za-z0-9]+"
    r"|Bearer\s+[A-Za-z0-9._\-]+"
    r"|postgres(?:ql)?://[^\s@]+:[^\s@]+@)"
)

CLAVES_SENSIBLES = {"password", "token", "authorization", "api_key", "secret", "cookie"}
```

```
def redactar(_logger, _method, event_dict):
    for clave, valor in list(event_dict.items()):
        if clave.lower() in CLAVES_SENSIBLES:
            event_dict[clave] = "[REDACTADO]"
        elif isinstance(valor, str):
            event_dict[clave] = PATRONES.sub("[REDACTADO]", valor)
    return event_dict


structlog.configure(
    processors=[
        redactar,                                # antes de renderizar, siempre
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.JSONRenderer(),
    ]
)
```

Trátalo como lo que es: un airbag. Un filtro por expresiones regulares sólo atrapa los formatos que ya conoces, y el día que integres un proveedor nuevo con un prefijo distinto, no lo verá. La redacción compra tiempo; el diseño es lo que evita el incidente.

## Runbook: se ha filtrado un secreto

Cuando ocurre, el orden de las acciones importa más que la velocidad. Este es el que hay que seguir:

1. **Revocar antes que rotar.** Es el error número uno bajo presión: la gente genera una clave nueva, despliega, y se olvida de invalidar la vieja, que sigue funcionando perfectamente en manos de quien la tenga. Si el corte es asumible, revoca primero. Si no, emite la nueva, propágala y revoca en cuanto la métrica de uso llegue a cero.
2. **Asumir compromiso total.** Da igual que el repositorio fuera privado, que el commit se borrara en cinco minutos o que reescribieras el historial con `git filter-repo`. Los forks, los clones locales, la caché de la forja y los logs de CI conservan copias. Reescribir el historial es higiene posterior, no contención.
3. **Auditar desde la primera exposición.** No desde que te enteraste. Busca en CloudTrail o en los logs de auditoría del proveedor todo uso de esa credencial desde la fecha del commit original, prestando atención a IPs, regiones y horarios inhabituales.
4. **Revisar el radio de alcance.** ¿Qué podía hacer esa clave? Si la respuesta es "todo", tienes un segundo problema —permisos excesivos— y este es el momento de arreglarlo, no de restaurar el mismo nivel de acceso.
5. **Cerrar la puerta.** ¿Por qué no lo detuvo el hook? ¿Por qué no lo vio CI? Cada incidente debería terminar con una capa de detección nueva o arreglada, no sólo con una clave nueva.

## El modelo de amenaza: qué ocurre de verdad cuando se filtra una clave

Hay una razón por la que las guías de secretos suelen fracasar dentro de un equipo: se escriben como si el riesgo fuese abstracto. No lo es. Cuando una clave llega a un repositorio público, la recogen bots que vigilan el firehose de eventos de GitHub, no personas. El intervalo entre el push y el primer intento de uso se mide en segundos o minutos, no en días. Y el bot no necesita entender tu producto: prueba la credencial contra un catálogo de APIs conocidas, mira qué responde y vende o explota lo que funcione.

Eso cambia la pregunta que hay que hacerse. No es «¿cómo evito que se filtre un secreto?» —eso también, pero es la mitad fácil— sino «¿cuánto daño puede hacer esta credencial concreta en los primeros diez minutos?». A esa respuesta se le llama **radio de explosión**, y es lo que separa un incidente aburrido de uno que

termina en un informe a clientes.

Un ejercicio que cuesta una tarde y cambia decisiones durante años: coge el inventario de secretos de un servicio y, para cada uno, escribe en una línea qué haría un atacante con él suponiendo que ya lo tiene. No qué *debería* poder hacer según la documentación: qué puede hacer según los permisos reales que tiene hoy.

- **Clave de S3 con `s3:GetObject` sobre un prefijo concreto de un bucket:** lee esos ficheros. Molesto, acotado, auditable.
- **La misma clave con `s3:*` sobre `*`:** lee todo, borra todo, y puede subir contenido que tus usuarios descargarán como si fuese tuyo.
- **Token de un webhook de Slack:** publica mensajes en un canal. Suena inocuo hasta que alguien lo usa para hacer phishing interno desde una fuente que tu equipo considera de confianza.
- **Personal access token de GitHub con `repo`:** lee todo el código privado, y además puede modificar workflows, es decir, escalar hacia el resto de credenciales de CI.
- **Credencial de base de datos de la aplicación:** depende enteramente de si ese rol tiene `SELECT` sobre las tablas que necesita o es el dueño del esquema.

La lista casi siempre incomoda, porque revela que la mayoría de las credenciales están sobredimensionadas por inercia: alguien necesitaba desbloquearse un martes, pidió permisos amplios «de momento», y ese «de momento» lleva año y medio en producción. Reducir permisos no es un proyecto de seguridad separado; es la medida que hace que todas las demás importen menos.

Hay tres propiedades que quieres para cualquier credencial, en este orden de valor:

1. **Alcance**: qué recursos toca y con qué verbos. Es lo que limita el daño.
2. **Duración**: cuánto tiempo sigue siendo válida. Es lo que limita la ventana.
3. **Atribución**: si al usarla los logs dicen *quién* la usó. Es lo que hace posible responder.

Una credencial estática, compartida por cinco servicios y con permisos de administrador falla en las tres. Un token OIDC de quince minutos emitido a un workflow concreto de un repositorio concreto acierta en las tres, y no hay ningún secreto que guardar. Esa es toda la tesis de esta guía, expresada como propiedades en lugar de como herramientas.

## CI/CD: el sitio donde más secretos se pierden

Si tuviera que apostar dónde está la fuga que aún no has encontrado, apostaría por CI. Es el único sistema de tu organización que tiene, por diseño, acceso a producción, ejecuta código que cambia a diario y produce logs que se leen en público. Y además evoluciona: lo que era seguro en tu pipeline hace dos años puede no serlo hoy.

### El camino sin secretos: OIDC hacia el proveedor cloud

GitHub Actions puede pedir a GitHub un token JWT de corta vida que describe *quién es* el workflow: en qué repositorio corre, en qué rama, en qué entorno. AWS, GCP y Azure saben validar ese token contra el emisor de GitHub y devolver credenciales temporales. Resultado: cero claves de larga vida almacenadas en la configuración del repositorio.

Primero se registra el proveedor de identidad una sola vez por cuenta. Con Terraform:

```
resource "aws_iam_openid_connect_provider" "github" {
  url             = "https://token.actions.githubusercontent.com"
```

```
  client_id_list = ["sts.amazonaws.com"]
}
```

Y después el rol, con la parte que realmente importa: la condición sobre el claim `sub`.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": { "Federated": "arn:aws:iam::111122223333:oidc-provider/
token.actions.githubusercontent.com" },
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
      "StringEquals": {
        "token.actions.githubusercontent.com:aud": "sts.amazonaws.com",
        "token.actions.githubusercontent.com:sub": "repo:mi-org/mi-repo:environment:production"
      }
    }
  }]
}
```

Tres detalles que deciden si esto es seguridad real o teatro:

- **Nunca dejes la política solo con la condición `aud`.** Ese valor es idéntico para todos los repositorios de GitHub del mundo. Una política que solo comprueba `aud` permite que cualquier workflow de cualquier organización asuma tu rol.
- **Nunca uses `repo:mi-org/*:*`.** Eso significa cualquier rama de cualquier repositorio de la organización, incluida la rama de prueba que abrió ayer un colaborador externo.
- **Evita los operadores `ForAllValues:` en sentencias `Allow`.** Devuelven verdadero cuando el claim no está presente o está mal escrito, que es exactamente el caso en el que quieres que la evaluación falle. Usa `StringEquals`, o `StringLike` si necesitas un comodín acotado.

## El cambio del claim inmutable (julio de 2026)

Este es el detalle operativo más fácil de que te pille por sorpresa este año. Los repositorios creados en github.com a partir del 15 de julio de 2026 —y los anteriores que se hayan acogido— emiten un `sub` *inmutable*: además del nombre de la organización y del repositorio, añade el identificador numérico permanente de cada uno, separado por `@`. El motivo es sensato: impide que alguien que registre un nombre de organización reciclado acuñe tokens que encajen con una política de confianza antigua.

```
# Forma clásica (solo nombres)
repo:mi-org/mi-repo:environment:production

# Forma inmutable (nombres + IDs permanentes)
repo:mi-org@12345678/mi-repo@98765432:environment:production
```

El síntoma cuando esto te afecta es inconfundible y desconcertante: el workflow que llevaba meses funcionando falla de golpe con `Not authorized to perform sts:AssumeRoleWithWebIdentity`, sin que nadie haya tocado ni el rol ni el pipeline. La causa es que el token ahora lleva el formato nuevo y tu condición `StringEquals` compara contra el formato viejo. La solución es actualizar la condición a la forma inmutable.

Si no sabes qué está emitiendo tu workflow, no adivines: imprímelo. Existen acciones de depuración que

decodifican el payload del JWT y lo muestran. Ejecútalas siempre en un repositorio privado —el token caduca rápido, pero los valores de los claims describen tu topología interna.

## El enmascarado de secretos no es un límite de seguridad

GitHub sustituye por asteriscos los valores registrados como secretos cuando aparecen literalmente en los logs. Es una red de protección contra descuidos, no un control. Se le escapa cualquier transformación:

```
# Todo esto sale en claro en el log
echo "$MI_SECRETO" | base64
echo "$MI_SECRETO" | rev
echo "$MI_SECRETO" | cut -c1-10
echo "$MI_SECRETO" | fold -w1   # una letra por línea
```

Trátalo como lo que es: si un paso de tu pipeline ejecuta código de terceros con un secreto en el entorno, ese secreto está comprometido en el momento en que ese código quiera comprometerlo. La defensa no es el enmascarado, es no ponerlo en el entorno.

## Los disparadores peligrosos y el código de terceros

Dos riesgos concretos merecen una regla escrita en el equipo:

- `pull_request_target` ejecuta el workflow con el contexto del repositorio base —con acceso a los secretos— pero puede hacer checkout del código de la pull request. Si combinas ese disparador con un checkout de la rama del contribuidor y ejecutas cualquier cosa de ese árbol (un script de build, un hook de instalación de dependencias), acabas de darle tus secretos a quien abrió la PR. Si necesitas ese disparador, no hagas checkout del código no confiable, o divide el trabajo en dos workflows.
- **Acciones de terceros sin fijar.** `uses: alguien/accion@v3` resuelve una etiqueta mutable: quien controle el repositorio puede mover esa etiqueta a un commit nuevo cuando quiera. Fija por SHA completo y deja el número de versión en un comentario para que se pueda leer.

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production        # activa reglas de protección y revisores
    permissions:
      id-token: write              # imprescindible para OIDC
      contents: read               # lo mínimo para el checkout
    steps:
      - uses: actions/checkout@11bd71901bbe5b1630ceea73d27597364c9af683  # v4.2.2
      - uses: aws-actions/configure-aws-credentials@e3dd6a429d7300a6a4c196c26e071d42e0343502  #
v4.0.2
        with:
          role-to-assume: arn:aws:iam::111122223333:role/deploy-produccion
          aws-region: eu-west-1
      - run: ./scripts/deploy.sh
```

El bloque `permissions` es importante y se olvida siempre. Sin él, el `GITHUB_TOKEN` del job hereda los permisos por defecto de la organización, que en repositorios antiguos suelen ser de escritura sobre todo. Declararlo explícitamente y al mínimo convierte un job comprometido en un problema mucho más pequeño.

Y una capa más, gratis y muy infravalorada: los **entornos** de GitHub. Un entorno `production` puede exigir

revisores humanos, restringir qué ramas pueden desplegar y guardar sus propios secretos. Con OIDC, además, el nombre del entorno viaja dentro del claim `sub`, así que la restricción se aplica dos veces: una en GitHub y otra en la política de confianza de IAM. Si una de las dos se relaja por error, la otra sigue sujetando.

## Secretos en imágenes de contenedor: la fuga que viaja contigo

Una imagen de contenedor es un formato de archivo con capas, y cada capa es inmutable y se distribuye entera. Esa propiedad, que es lo que hace que los contenedores sean rápidos y cacheables, es también la que convierte un secreto mal colocado en un problema permanente: una vez la capa existe, borrar el fichero en una instrucción posterior no lo elimina, solo lo oculta en el sistema de ficheros final. La capa anterior sigue ahí, y cualquiera que tenga acceso al registro puede extraerla.

```
# MAL: el token queda en el historial de la imagen para siempre
FROM node:22-slim
ARG NPM_TOKEN
RUN echo "//registry.npmjs.org/:_authToken=$NPM_TOKEN" > ~/.npmrc  && npm ci  && rm ~/.npmrc
# este rm no sirve de nada
```

El `rm` crea una capa nueva que marca el fichero como borrado. La capa donde se escribió sigue existiendo, y el valor de `ARG` queda además en los metadatos de la imagen. Comprobarlo cuesta un comando:

```
# Ver el historial completo, sin truncar
docker history --no-trunc mi-imagen:latest

# Y el volcado real de capas, por si algo se coló en el sistema de ficheros
docker save mi-imagen:latest -o /tmp/img.tar
mkdir -p /tmp/img && tar -xf /tmp/img.tar -C /tmp/img
grep -ria "authToken|BEGIN PRIVATE KEY|aws_secret" /tmp/img | head
```

Si ese `grep` devuelve algo, el secreto está comprometido para todo el que haya hecho `pull` de esa etiqueta. No hay forma de arreglarlo publicando una imagen nueva: hay que rotar la credencial.

### La forma correcta: montajes de secreto de BuildKit

BuildKit —el motor de construcción por defecto de Docker desde hace ya varias versiones— resuelve esto con `RUN --mount=type=secret`. El valor se expone como un fichero dentro del contenedor de construcción solo durante esa instrucción `RUN`, y no se escribe en ninguna capa ni en la caché de construcción.

```
# syntax=docker/dockerfile:1.7
FROM node:22-slim AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN --mount=type=secret,id=npm_token,required=true     NPM_TOKEN="$(cat /run/secrets/npm_token)"
npm ci --omit=dev

FROM node:22-slim
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
```

```
USER node
CMD ["node", "server.js"]
```

```
# Desde un fichero
docker build --secret id=npm_token,src=./npm_token.txt -t mi-imagen:latest .

# Desde una variable de entorno, sin tocar el disco
export NPM_TOKEN="$(aws secretsmanager get-secret-value   --secret-id build/npm --query SecretString
--output text)"
docker build --secret id=npm_token,env=NPM_TOKEN -t mi-imagen:latest .
```

El flag `required=true` merece un comentario, porque evita el peor fallo posible: sin él, si te olvidas de pasar el secreto, el montaje simplemente está vacío y el build continúa con un token vacío. Con `required=true` el build falla ruidosamente, que es lo que quieres.

Dos advertencias importantes. La primera: el montaje protege el *almacenamiento*, no el *uso*. Si el comando que ejecutas imprime el secreto, lo escribe en un fichero de log dentro de la imagen o lo incrusta en un artefacto compilado, el montaje no te salva de nada. La segunda: la construcción multietapa no es opcional en este esquema. Si el secreto lo usa la etapa `deps` y la imagen final solo copia `node_modules`, nada de la etapa intermedia acaba publicado.

En tiempo de ejecución la regla es todavía más simple: **en la imagen no va ninguna credencial**. La imagen es un artefacto que se promociona sin cambios de staging a producción; si contuviera secretos, tendrías que construir una imagen distinta por entorno y perderías la propiedad más valiosa del despliegue basado en artefactos. Los secretos entran en el arranque, desde el gestor, vía identidad de la carga de trabajo.

## Terraform y el estado: la fuga que casi nadie audita

Aquí hay un punto ciego clásico. El fichero de estado de Terraform guarda en claro todos los atributos de todos los recursos que gestiona, incluidos los sensibles. Marcar una variable como `sensitive = true` solo evita que el valor aparezca en la salida de la consola: **en el estado se escribe igual**. Si tu backend es un bucket sin cifrar, o si alguien copia el `terraform.tfstate` a su portátil para depurar, la contraseña de la base de datos de producción se ha ido de paseo.

```
# Lo que hay realmente dentro del estado
terraform show -json | jq '.values.root_module.resources[]
  | select(.values.password != null)
  | {addr: .address, tiene_password: true}'
```

### Valores efímeros y argumentos de solo escritura

Terraform 1.10 introdujo los *valores efímeros* y 1.11 los extendió a recursos gestionados con los *argumentos de solo escritura*. Un bloque `ephemeral` produce un valor que Terraform usa durante la ejecución pero que nunca persiste ni en el plan ni en el estado. Un argumento de solo escritura acepta ese valor en un recurso y tampoco lo guarda.

```
# El valor se obtiene en cada ejecución y no se persiste en ningún artefacto
ephemeral "aws_secretsmanager_secret_version" "db" {
```

```
  secret_id = "produccion/postgres/maestra"
}

resource "aws_db_instance" "principal" {
  identifier     = "app-produccion"
  engine         = "postgres"
  instance_class = "db.m6g.large"
  username       = "appadmin"

  # Argumento de solo escritura: entra, no se guarda en el estado
  password_wo         = ephemeral.aws_secretsmanager_secret_version.db.secret_string
  password_wo_version = 3   # súbelo para forzar que se reenvíe el valor
}
```

El campo `_wo_version` es el truco que hace funcionar todo el mecanismo. Como Terraform no guarda el valor, no puede compararlo para detectar cambios; el número de versión es la señal explícita que le dice «este valor ha cambiado, vuelve a enviarlo». Si rotas la contraseña en el gestor y no subes ese número, Terraform no hará nada.

Los argumentos de solo escritura requieren Terraform 1.11 o superior *y* soporte del proveedor. No todos los recursos los tienen todavía, así que la estrategia realista es en tres niveles:

1. Usa `ephemeral` y argumentos `_wo` siempre que el proveedor los soporte.
2. Donde no, no metas el secreto en Terraform: crea el recurso con una contraseña generada aleatoriamente y déjala gestionada por rotación fuera de Terraform, o crea solo el *contenedor* del secreto y que sea la aplicación quien lo lea.
3. Y, en cualquier caso, trata el backend de estado como si contuviera secretos: cifrado en reposo, acceso restringido a un rol de CI, versionado activado y logs de acceso.

```
terraform {
  backend "s3" {
    bucket       = "tf-estado-produccion"
    key          = "app/terraform.tfstate"
    region       = "eu-west-1"
    encrypt      = true
    kms_key_id   = "arn:aws:kms:eu-west-1:111122223333:key/abcd-1234"
    use_lockfile = true
  }
}
```

Y una regla de higiene que ahorra sustos: el estado no se descarga al portátil. Si necesitas inspeccionarlo, hazlo desde el mismo entorno que ejecuta los `apply`, y si aun así lo descargas, bórralo de verdad al terminar — incluido el fichero de respaldo `terraform.tfstate.backup` que Terraform deja sin avisar.

## Vault en detalle: credenciales dinámicas sin sorpresas

La idea de las credenciales dinámicas es elegante: la aplicación no guarda una contraseña de base de datos, sino que se autentica ante Vault con su identidad y pide una credencial nueva, creada en ese momento, con un tiempo de vida limitado. Cuando el arriendo expira, Vault ejecuta un `DROP ROLE` y esa credencial deja de existir. Un volcado de memoria o un log filtrado dejan de ser un problema permanente y pasan a ser un problema con fecha de caducidad.

```
# Configuración del motor de base de datos
vault write database/config/postgres-produccion   plugin_name="postgresql-database-plugin"
allowed_roles="app-lectura,app-escritura"   connection_url="postgresql://{{username}}:{{password}}
@db.interno:5432/app?sslmode=require"   username="vault_gestor"   password="$PASSWORD_INICIAL"

# Muy importante: rota la contraseña del gestor inmediatamente.
# A partir de aquí solo Vault la conoce; ningún humano puede usarla.
vault write -force database/rotate-root/postgres-produccion

# Un rol con permisos acotados y TTL corto
vault write database/roles/app-lectura   db_name="postgres-produccion"   creation_statements="CREATE
ROLE "{{name}}" WITH LOGIN PASSWORD '{{password}}' VALID UNTIL '{{expiration}}'; GRANT SELECT ON ALL
TABLES IN SCHEMA public TO "{{name}}";"   revocation_statements="DROP ROLE IF EXISTS "{{name}}";"
default_ttl="1h"   max_ttl="24h"
```

Los valores por defecto de Vault son una hora de TTL y veinticuatro de TTL máximo, y conviene entender bien la diferencia porque de ahí salen los dos fallos operativos más comunes.

## Trampa 1: el pool de conexiones y el TTL máximo

Una credencial se puede *renovar* tantas veces como quieras hasta llegar al TTL máximo. Al llegar ahí, ya no hay renovación posible: hay que pedir una credencial nueva. Si tu aplicación usa un pool de conexiones —y la usa— las conexiones abiertas con la credencial vieja siguen vivas en el pool mientras el motor de la base de datos las mantenga, pero cualquier conexión nueva con esas credenciales fallará, y en muchos motores el `DROP ROLE` corta también las sesiones existentes.

El síntoma es característico: la aplicación funciona perfectamente durante horas y luego, a las veinticuatro en punto desde el arranque, empieza a devolver errores de autenticación en una fracción de las peticiones —justo las que tocan conexiones recicladas. La solución no es subir el TTL máximo hasta que el problema desaparezca; es que la capa de acceso a datos sepa reaccionar:

```
import hvac, time, threading
from sqlalchemy import create_engine

class PoolConCredencialesVault:
    """Renueva el arriendo y, cuando ya no se puede, reconstruye el pool."""

    def __init__(self, cliente: hvac.Client, rol: str, dsn_base: str):
        self.cliente, self.rol, self.dsn_base = cliente, rol, dsn_base
        self._engine = None
        self._lease_id = None
        self._construir()
        threading.Thread(target=self._bucle_renovacion, daemon=True).start()

    def _construir(self):
        r = self.cliente.secrets.database.generate_credentials(name=self.rol)
        self._lease_id = r["lease_id"]
        usuario, clave = r["data"]["username"], r["data"]["password"]
        anterior = self._engine
        # pool_pre_ping descarta conexiones muertas antes de entregarlas
        self._engine = create_engine(
            self.dsn_base.format(user=usuario, password=clave),
```

```
            pool_pre_ping=True, pool_recycle=1800, pool_size=10,
        )
        if anterior is not None:
            # drenaje suave: las conexiones en uso terminan su trabajo
            anterior.dispose(close=False)

    def _bucle_renovacion(self):
        while True:
            time.sleep(300)
            try:
                self.cliente.sys.renew_lease(lease_id=self._lease_id, increment=3600)
            except hvac.exceptions.InvalidRequest:
                # TTL máximo alcanzado: toca credencial nueva
                self._construir()

    @property
    def engine(self):
        return self._engine
```

Lo relevante no es la biblioteca concreta, es el patrón: renovar mientras se pueda, detectar el fallo de renovación como una señal esperada y no como un error, y reconstruir el pool drenando el anterior en lugar de matarlo. Con eso, la rotación deja de ser un evento y pasa a ser rutina.

## Trampa 2: la raíz asignada a un rol estático

El otro fallo es más sutil y rompe cosas de golpe. Si asignas las credenciales del usuario gestor (el que Vault usa para conectarse) a un rol estático para que Vault se las rote, en la siguiente rotación cambiará la contraseña de ese usuario *sin* actualizar el bloque `config/`. A partir de ese momento, Vault ya no puede autenticarse contra la base de datos y todos los usuarios dinámicos y estáticos de esa configuración dejan de emitirse. Regla: el usuario gestor se rota con `rotate-root`, nunca a través de un rol estático.

Vale la pena también mencionar el motor *transit*, que resuelve un problema distinto y muy útil: cifrado como servicio. La aplicación envía datos y recibe el texto cifrado, sin que la clave salga nunca de Vault. Sirve para cifrar campos concretos — un número de cuenta, un documento de identidad — sin repartir material criptográfico por los servicios.

```
vault secrets enable transit
vault write -f transit/keys/pii

# Cifrar (la app nunca ve la clave)
vault write transit/encrypt/pii plaintext=$(base64 <<< "ES9121000418450200051332")
# -> ciphertext: vault:v1:8SDd3WHDOjf7mq69CyCqYjBXAiQQAVZRkFM13ok481zoCmHnSeDX9vyf7w==

# Descifrar
vault write transit/decrypt/pii ciphertext="vault:v1:8SDd3WHD..."
```

El prefijo `v1` del texto cifrado es la versión de la clave. Cuando rotas la clave (`vault write -f transit/keys/pii/rotate`), los datos antiguos se siguen descifrando con la versión con la que se cifraron, y los nuevos usan la versión actual. Esto convierte la rotación de claves de cifrado —históricamente un proyecto de migración doloroso— en una operación de un comando.

## Elegir gestor: comparativa práctica

No hay una respuesta universal, pero sí hay criterios que ordenan la decisión mejor que las tablas de características.

- **AWS Secrets Manager**: unos 0,40 USD por secreto y mes más un coste por llamadas a la API. A cambio trae rotación gestionada con funciones prefabricadas para RDS, DocumentDB y Redshift, replicación entre regiones y versionado con etiquetas de fase. Es la opción correcta cuando el secreto *rota*.
- **SSM Parameter Store (estándar)**: almacenamiento y llamadas sin coste hasta el límite de parámetros, con cifrado por KMS si usas el tipo `SecureString`. No rota por sí solo y el rendimiento por defecto es modesto —del orden de decenas de operaciones por segundo, ampliable activando el modo de mayor rendimiento—. Es la opción correcta para configuración sensible que no rota.
- **GCP Secret Manager y Azure Key Vault**: modelos equivalentes, con versionado y con integración nativa con la identidad de la carga de trabajo de cada plataforma. Key Vault además separa secretos, claves y certificados, lo que ayuda cuando la gestión de certificados es un problema real y no un detalle.
- **HashiCorp Vault**: la opción con más capacidad —credenciales dinámicas, transit, PKI, múltiples backends de autenticación— y también la que tiene más coste operativo, porque hay que operarlo con alta disponibilidad y una historia de *unsealing* que funcione a las tres de la mañana. Vale la pena cuando quieres credenciales dinámicas o cuando necesitas un plano de secretos único sobre varias nubes.

Una estrategia mixta suele ganar en la práctica: lo que rota, en Secrets Manager; el resto de configuración sensible, en Parameter Store; y Vault solo si las credenciales dinámicas justifican operarlo.

### Cachear no es opcional

El error de coste y de latencia más habitual es leer el secreto en cada petición. Un servicio con mil peticiones por segundo que consulta el gestor en cada una hace ochenta y seis millones de llamadas al día: es caro, es lento y acabará chocando con los límites de tasa. El secreto se lee al arrancar, se guarda en memoria y se refresca en segundo plano.

```
import time, threading, boto3

class SecretoCacheado:
    """Lee una vez, refresca en segundo plano, nunca bloquea la petición."""

    def __init__(self, secret_id: str, ttl: int = 300):
        self._cliente = boto3.client("secretsmanager")
        self._id, self._ttl = secret_id, ttl
        self._valor, self._caduca = None, 0
        self._lock = threading.Lock()

    def get(self) -> str:
        ahora = time.monotonic()
        if self._valor is not None and ahora < self._caduca:
            return self._valor
        with self._lock:
            if self._valor is not None and time.monotonic() < self._caduca:
                return self._valor
            try:
                r = self._cliente.get_secret_value(SecretId=self._id)
                self._valor = r["SecretString"]
                self._caduca = time.monotonic() + self._ttl
            except Exception:
                # Degradación elegante: si el gestor no responde y tenemos
```

```
                # un valor previo, seguimos sirviendo con él.
                if self._valor is None:
                    raise
                self._caduca = time.monotonic() + 30
            return self._valor
```

El bloque `except` es la parte que la gente omite y luego lamenta. Si el gestor de secretos tiene una incidencia y tu servicio no puede arrancar ni servir sin él, acabas de convertir una dependencia de configuración en una dependencia de disponibilidad de tu ruta crítica. Servir con el último valor conocido durante una ventana corta es casi siempre la decisión correcta.

En entornos serverless, AWS ofrece una extensión de Lambda que cachea secretos y parámetros entre invocaciones y reduce drásticamente las llamadas a la API. Es una línea de configuración y suele pagar su propia adopción en la primera factura.

## Cifrado sobre: qué hace KMS por dentro

Merece la pena entender el mecanismo, porque explica por qué los gestores de secretos están construidos como están y evita un par de decisiones costosas.

Un servicio de gestión de claves no cifra tus datos con la clave maestra. Cifrar cien megabytes enviándolos a un servicio remoto sería absurdo por latencia y por límites de tamaño. Lo que hace es **cifrado sobre**: genera una clave de datos aleatoria (DEK), te la devuelve dos veces —una en claro y otra cifrada con la clave maestra (KEK), que nunca sale del módulo criptográfico— y tú cifras localmente con la DEK en claro, la borras de memoria y guardas junto al dato la versión cifrada de la DEK.

```
import boto3, os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

kms = boto3.client("kms")
ID_CLAVE = "arn:aws:kms:eu-west-1:111122223333:key/abcd-1234"

def cifrar(datos: bytes, contexto: dict[str, str]) -> dict:
    # KMS genera la DEK; devuelve la versión en claro y la cifrada
    r = kms.generate_data_key(KeyId=ID_CLAVE, KeySpec="AES_256",
                              EncryptionContext=contexto)
    dek, dek_cifrada = r["Plaintext"], r["CiphertextBlob"]
    try:
        nonce = os.urandom(12)
        ct = AESGCM(dek).encrypt(nonce, datos, None)
    finally:
        del dek   # fuera de memoria en cuanto deja de hacer falta
    return {"dek": dek_cifrada, "nonce": nonce, "ct": ct, "ctx": contexto}

def descifrar(sobre: dict) -> bytes:
    # El contexto debe coincidir exactamente o KMS rechaza la operación
    dek = kms.decrypt(CiphertextBlob=sobre["dek"],
                      EncryptionContext=sobre["ctx"])["Plaintext"]
    try:
        return AESGCM(dek).decrypt(sobre["nonce"], sobre["ct"], None)
    finally:
```

```
        del dek
```

El `EncryptionContext` es la pieza que casi todo el mundo ignora y que más valor aporta. Son pares clave-valor que se vinculan criptográficamente a la operación: si cifras con `{"tenant": "acme"}`, el descifrado solo funciona pasando exactamente ese contexto. Eso impide que un dato cifrado de un cliente se pueda descifrar en el contexto de otro aunque compartan clave maestra, y además el contexto aparece en los registros de auditoría de KMS, con lo que las trazas dejan de decir «alguien descifró algo» y pasan a decir qué.

Este mismo mecanismo es el que usan por debajo Secrets Manager, los `SecureString` de Parameter Store, el cifrado de secretos en *etcd* de Kubernetes y el motor transit de Vault. Cuando entiendes el sobre, la pregunta «¿dónde está realmente la confianza?» tiene una respuesta clara: en la política de la clave maestra. Un gestor de secretos impecable con una política de KMS que permite `kms:Decrypt` a media cuenta no protege gran cosa.

## En el cliente no existen los secretos

Es la conversación que se repite en todos los equipos que tienen aplicación móvil o frontend: «¿dónde guardamos la clave de la API en la app?». La respuesta honesta es que no se guarda, porque no hay ningún sitio donde esté a salvo. Un APK se descomprime, un bundle de JavaScript se lee, una app de iOS se extrae del dispositivo. La ofuscación sube el coste de encontrarla de cinco minutos a media hora, y eso es todo lo que compra.

La solución estructural es no darle a la aplicación cliente ninguna credencial que sirva para algo. Lo que se hace en su lugar:

- **Un backend intermedio (BFF).** El cliente habla con tu servidor con un token de sesión que representa a *ese usuario*; tu servidor guarda la clave del proveedor externo y hace la llamada. El token del usuario, si se roba, permite hacer lo que ese usuario puede hacer —lo que ya es tu modelo de autorización—, no vaciar tu cuota de un proveedor de pago.
- **Claves públicas restringidas.** Algunas APIs (mapas, analítica) ofrecen claves diseñadas para vivir en el cliente, restringidas por dominio de referencia, bundle ID o cuota. No son secretos: son identificadores con límites. Configura las restricciones, porque sin ellas dejan de estar acotadas.
- **URLs firmadas de corta vida.** Para subidas y descargas directas contra almacenamiento de objetos, el servidor firma una URL con permiso para una operación, un objeto y unos minutos. El cliente no ve credencial alguna.

```
# El servidor firma; el navegador sube directamente a S3 sin credenciales
import boto3

s3 = boto3.client("s3")

def url_de_subida(usuario_id: str, nombre: str) -> dict:
    return s3.generate_presigned_post(
        Bucket="subidas-usuarios",
        Key=f"u/{usuario_id}/{nombre}",
        Fields={"Content-Type": "image/jpeg"},
        Conditions=[
            {"Content-Type": "image/jpeg"},
            ["content-length-range", 1, 5 * 1024 * 1024],  # máximo 5 MB
        ],
        ExpiresIn=300,   # cinco minutos
```

```
    )
```

Fíjate en las `Conditions`: sin ellas la URL firmada permitiría subir un fichero de cualquier tamaño y cualquier tipo. Una URL firmada mal acotada es una credencial mal acotada; los mismos principios de alcance y duración se aplican igual.

## Detección: saber que ha pasado, y no por el correo de un desconocido

La prevención falla alguna vez. Lo que distingue a un equipo maduro es cuánto tarda en enterarse. Hay tres señales que dan mucho por poco esfuerzo.

**Auditoría de acceso al gestor.** Cada lectura de un secreto queda registrada. Esa traza sirve para dos cosas: detectar accesos anómalos y —más útil en el día a día— encontrar secretos muertos. Un secreto que lleva noventa días sin que nadie lo lea es un secreto que puedes borrar, y cada secreto borrado es una superficie de ataque menos.

```
# Quién ha leído secretos en las últimas 24 horas
aws cloudtrail lookup-events   --lookup-attributes
AttributeKey=EventName,AttributeValue=GetSecretValue   --start-time "$(date -u -d '24 hours ago' +
%Y-%m-%dT%H:%M:%SZ)"   --query 'Events[].CloudTrailEvent' --output text   | jq -r
'[.userIdentity.arn, .requestParameters.secretId] | @tsv'   | sort | uniq -c | sort -rn
```

**Tokens señuelo.** Un *honeytoken* es una credencial que parece real, no sirve para nada y avisa cuando alguien la usa. Coloca uno en un fichero `.env.example`, otro en la wiki interna, otro en una imagen de contenedor antigua. No producen falsos positivos —nadie tiene motivo legítimo para usarlos— así que cualquier alerta es una alerta de verdad. Es de las pocas medidas de detección con relación coste-beneficio casi absurda.

**Métricas del programa, no solo del incidente.** Si quieres que esto mejore de forma sostenida, mide cuatro cosas y revísalas una vez al mes:

1. **Cobertura de identidad**: porcentaje de cargas de trabajo que usan identidad federada en lugar de claves estáticas. Es la métrica que más correlaciona con reducción de riesgo real.
2. **Antigüedad de las credenciales estáticas que quedan**: el percentil 95 de días desde la última rotación. Si crece mes a mes, la rotación no está automatizada por mucho que exista el procedimiento.
3. **Tiempo hasta la revocación**: en el último simulacro o incidente real, cuánto pasó entre la detección y la invalidación efectiva. Este número solo baja si alguien practica.
4. **Secretos huérfanos**: cuántos llevan más de noventa días sin lecturas. Objetivo: cero, por borrado, no por excepción.

## Que el camino seguro sea el camino fácil

Un último punto, y probablemente el que más determina si algo de esto sobrevive seis meses. Todos los controles de este artículo fracasan si el camino correcto es más lento que el atajo. Cuando arrancar el proyecto en local con el gestor de secretos cuesta cuatro pasos y una petición de acceso, y copiar el `.env` del compañero por mensaje directo cuesta uno, la gente copia el `.env`. No es indisciplina: es la respuesta racional a un incentivo mal puesto.

El objetivo es que el desarrollador escriba un comando y todo funcione:

```
#!/usr/bin/env bash
# scripts/dev — arranca en local con credenciales temporales
set -euo pipefail

if ! aws sts get-caller-identity >/dev/null 2>&1; then
  echo "Iniciando sesión SSO..." >&2
  aws sso login --profile desarrollo
fi

# Los secretos se inyectan en el proceso hijo y no tocan el disco
exec aws-vault exec desarrollo --   env $(aws secretsmanager get-secret-value           --secret-id
dev/app --query SecretString --output text         | jq -r 'to_entries | map("(.key)=(.value)") | .
[]')   npm run dev
```

Tres propiedades que hacen que esto funcione: no hay ningún fichero de secretos en el disco del portátil, las credenciales caducan solas al terminar la sesión SSO, y el desarrollador no ha tenido que aprender nada nuevo —sigue escribiendo un comando para arrancar—. Si además el entorno de desarrollo apunta a una base de datos de desarrollo con datos sintéticos, has eliminado de golpe la categoría entera de «alguien tenía credenciales de producción en el portátil».

Complétalo con lo obvio pero raramente escrito: un documento de una página que diga dónde vive cada tipo de secreto, quién puede concederlo y qué hacer si se filtra. Cuando alguien nuevo entra un lunes, ese documento es la diferencia entre que adopte tu modelo o que se invente el suyo.

## Ocho errores que se repiten

- **Rotar sin revocar.** La clave vieja sigue viva. Lo cubrimos arriba porque es el más frecuente y el más caro.
- **Un único secreto compartido por todos los servicios.** Cuando se filtra, tienes que rotarlo todo a la vez, así que nunca se rota. Una credencial por servicio y entorno, siempre.
- **El comodín en la política de confianza OIDC.** `repo:mi-org/*` convierte una arquitectura sin claves estáticas en una puerta abierta.
- **Secretos en argumentos de línea de comandos.** Cualquier proceso del sistema los ve con `ps aux`, y quedan en el historial del shell. Pásalos por fichero o por entrada estándar.
- **Secretos en argumentos de `docker build`.** Un `ARG` queda grabado en el historial de capas de la imagen y lo recupera cualquiera con `docker history`. Usa `RUN --mount=type=secret`.
- **Leer el secreto una sola vez, al importar el módulo.** Anula por completo cualquier mecanismo de rotación en caliente.
- **Reutilizar credenciales de producción en staging.** Staging siempre tiene controles más laxos y más gente con acceso; es la vía de entrada preferida a producción.
- **No tener inventario.** Si no puedes responder "qué secretos existen, quién los usa y cuándo se tocaron por última vez", no puedes rotar nada con confianza. El inventario es el prerrequisito de todo lo demás.

## Checklist de producción

- Hook de `pre-commit` con gitleaks, escaneo del historial completo en CI y protección de push activada en la forja.
- Ninguna clave estática de nube en variables de CI: federación OIDC con la condición `sub` atada a repositorio y rama.
- Cargas de trabajo autenticándose con identidad de plataforma (IRSA, Workload Identity, Managed Identity) siempre que el destino lo permita.
- Gestor de secretos externo como fuente de verdad; en Kubernetes, sincronización con ESO y cifrado en reposo de etcd con KMS.

- Secretos entregados como ficheros montados, no como variables de entorno, y leídos en el punto de uso.
- Un secreto por servicio y por entorno, con permisos mínimos y revisados.
- Rotación documentada con ventana de solape y una métrica de uso por identificador de clave que permita revocar con seguridad.
- `SecretStr` o equivalente en la configuración, más redacción en el procesador de logs como red de seguridad.
- Runbook de fuga escrito y ensayado al menos una vez, con la revocación como primer paso.
- Inventario de secretos con dueño, propósito y fecha de última rotación.

## Por dónde empezar

Si tu equipo está en el punto de partida, el orden que más riesgo elimina por unidad de esfuerzo es este: activa hoy mismo la detección (hooks, CI y push protection), porque es media hora de trabajo y evita la clase de incidente más común. Después migra CI a OIDC, que suele ser la mayor concentración de claves de administrador estáticas de toda la organización. Sólo entonces aborda el gestor de secretos y la sincronización, que es el trozo con más piezas móviles. Y deja para el final las credenciales dinámicas: son el destino ideal, pero exigen que el resto ya esté en su sitio.

Ninguno de estos pasos requiere reescribir la aplicación. Requiere decidir, una vez, que los secretos son infraestructura y no un detalle de configuración.

PuigFlows

# Secrets Management in Production: From .env to Workload Identity, Zero-Downtime Rotation and a Leak Runbook

A practical guide to why teams lose keys to operations rather than cryptography, and how to stop: what counts as a secret and what does not, why .env breaks down in production, the three detection layers that keep secrets out of the repository (pre-commit with gitleaks, full-history scanning in CI and push protection), and SOPS with age when config has to live in Git. Covers the shift that removes the most risk —preferring identity over secrets with IRSA, Workload Identity and OIDC federation in CI—, including the IAM trust policy and the sub wildcard that turns a key-free architecture into an open door; dynamic PostgreSQL credentials with Vault and its two operational traps (pool renewal and orphaned roles); the real difference between delivering a secret via environment variable or mounted file (/proc, subprocess inheritance, and why env never updates on rotation); settings with pydantic-settings and SecretStr over secrets_dir; a RotatingSecret wrapper that re-reads the file when the kubelet updates it; Kubernetes with etcd encryption at rest, per-namespace RBAC and syncing via External Secrets Operator (ClusterSecretStore, ExternalSecret and volume mounting); zero-downtime rotation with the two-credential pattern and the per-key_id metric that tells you when revoking is safe; secret redaction in logs with structlog; a leak runbook that starts with revoking rather than rotating; eight recurring mistakes and a production checklist.

**Josue Garcia**
26 de agosto de 2026

Approx. 38 min read
English

# Contents

Section index

Almost no team loses a production key to a sophisticated attack. They lose it because someone ran `git add .` with a `.env` in the directory, because an uncaught exception shipped the full connection string to Sentry, or because the payment provider key hasn't changed in four years and now lives on the laptops of three people who left the company. Secret incidents are almost always operational incidents, not cryptographic ones.

This guide walks the whole path: from the `.env` we all start with to an architecture where most of your services have no long-lived secret left to lose. It isn't a tool catalogue, it's an order of priorities: first eliminate the secret, then shorten its life, and only at the end protect it properly.

This guide is meant to be read once end to end and then revisited section by section. The first half sets the order of priorities — what counts as a secret, how to keep it out of the repository, how to replace it with identity, how it reaches the process, and how to rotate it without downtime. The second half goes deep on the places where secrets actually get lost: the threat model and blast radius, CI/ CD pipelines and OIDC federation (including the July 2026 immutable claim change), container images, Terraform state, Vault dynamic credentials and their two operational traps, KMS envelope encryption, the mobile and web client case, and how to measure whether the programme is improving or merely generating documents.

## What counts as a secret (and what doesn't)

Before building anything, draw the line. A secret is any value that, in someone else's hands, lets them act on your behalf or read data that isn't theirs:

- Database passwords and full connection strings.
- Third-party API keys (payments, email, LLM providers).
- Private keys: TLS, JWT signing, SSH, artifact signing.
- Webhook signing secrets and encryption keys for data at rest.
- Session tokens and service-to-service credentials.

And this is *not*: internal hostnames, bucket names, project IDs, public keys, configuration flags. Putting everything in the secrets manager feels prudent and backfires: it turns every trivial config change into paperwork, and when the process gets heavy, people route around it. The practical test is to ask what happens if that value shows up in a screenshot in a Slack channel. If the answer is "nothing", it's configuration.

## Why .env stops working

The `.env` file solves a real problem locally and creates four in production:

- **It's one distraction away from Git.** Your `.gitignore` protects you until someone copies `.env.example` to `.env.staging`, a name the pattern doesn't cover.
- **It has no owner and no history.** Nobody knows who generated that key, when, or what permissions it carries. So nobody dares rotate it.
- **It spreads by copying.** Onboarding a new engineer literally consists of someone sending them production secrets over a private channel.
- **You can't revoke it selectively.** If it leaks, you can't invalidate one copy: you invalidate the key for everyone, production included.

The goal isn't to ban `.env`. Locally, with throwaway credentials that are worth nothing, it's perfect. The goal is that the file never holds a value that matters.

## Rule 1: the secret never enters the repository

This defence has to be automatic, because relying on human discipline fails with probability 1 across enough commits. Build it in three layers.

### Layer 1: block before the commit

A `pre-commit` hook running gitleaks scans only what you're about to commit, so it's fast and stays out of the way:

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/gitleaks/gitleaks
    # Pin a concrete tag and bump it deliberately: a floating "latest"
    # in a hook is just one more unreviewed dependency.
    rev: v8.x.y
    hooks:
      - id: gitleaks
```

### Layer 2: full-history scanning in CI

The local hook only sees the current commit, and anyone can skip it with `--no-verify`. In CI you have to clone the entire history, because a secret deleted in a later commit is still recoverable from the earlier one:

```
- uses: actions/checkout@v4
  with:
    fetch-depth: 0   # without this, only the last commit is scanned

- uses: gitleaks/gitleaks-action@v2
```

### Layer 3: platform-level protection

Turn on your forge's push protection (on GitHub, *secret scanning* with *push protection*). It's the only layer that catches a push from a machine without your hooks, and it also notifies the issuing provider: many services automatically revoke a key that appears in a public repository.

### If you really need config in the repo: SOPS

Some teams want configuration versioned alongside the code, sensitive parts included. SOPS encrypts only the values and leaves the keys in plaintext, so the diff stays readable and reviewable:

```
# Generate an age key; the private half NEVER enters the repository.
age-keygen -o age.key

# Encrypt only the values under data/stringData, not the whole YAML.
sops --encrypt \
     --age age1ql3z7hjy54pw3hyww5ayyfg7zqgvc7w3j2elw8zmrj2kg5sfn9aqmcac8p \
     --encrypted-regex '^(data|stringData)$' \
     secrets.yaml > secrets.enc.yaml
```

```
git add secrets.enc.yaml   # this one can live in the repository
```

It's an honest answer for small teams and for GitOps, with one important caveat: SOPS moves the problem, it doesn't remove it. Now you have to protect the decryption key, and rotating a secret goes through a pull request and a deploy. If your organisation already has a secrets manager, use it; SOPS shines when you don't have one.

## Rule 2: prefer identity over secrets

The best secret is the one that doesn't exist. When a process needs to talk to a service in the same cloud, don't hand it a key: hand it an **identity**. The platform signs a short-lived token the provider validates, and no long-lived value is stored anywhere.

- **AWS:** IRSA or EKS Pod Identity for pods; instance roles for EC2.
- **GCP:** Workload Identity Federation.
- **Azure:** Workload Identity / Managed Identity.
- **CI/CD:** OIDC federation from GitHub Actions, GitLab CI or Buildkite.

CI is where the immediate payoff is largest, because CI variables are where static admin keys pile up. With OIDC they disappear:

```
name: deploy
on:
  push:
    branches: [main]

permissions:
  contents: read
  id-token: write   # required: enables the job's OIDC token

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Short-lived AWS credentials (no static keys)
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/gha-deploy-checkout
          aws-region: eu-west-1

      - run: aws sts get-caller-identity
```

The critical piece isn't in the workflow, it's in the role's trust policy. This is where the mistake that undoes all the work gets made:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
```

```
    "Principal": {
      "Federated": "arn:aws:iam::123456789012:oidc-provider/token.actions.githubusercontent.com"
    },
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
      "StringEquals": {
        "token.actions.githubusercontent.com:aud": "sts.amazonaws.com",
        "token.actions.githubusercontent.com:sub": "repo:my-org/checkout:ref:refs/heads/main"
      }
    }
  }]
}
```

If you omit the `sub` condition, or write it as a `StringLike` with too broad a wildcard (`repo:my-org/*:*`, or worse, `repo:*`), you are letting *any* GitHub workflow — anyone's, anywhere — assume your deploy role. It's a real and recurring audit finding. Always bind `sub` to a specific repository and ref, and use a separate role per repository and per environment.

## Rule 3: if you need a secret, make it short-lived and dynamic

When the target doesn't support federated identity — a PostgreSQL database, say — the next best option is generating the credential on demand with a lifetime measured in hours. Vault does this with its database engine: it creates a fresh PostgreSQL role per request and drops it on expiry.

```
vault secrets enable database

vault write database/config/checkout \
  plugin_name=postgresql-database-plugin \
  allowed_roles="checkout-app" \
  connection_url="postgresql://{{username}}:{{password}}@pg.internal:5432/checkout?sslmode=require"
\
  username="vault-rotator" \
  password="..."

# Each issued credential lives 1 hour and can be renewed for at most 24.
vault write database/roles/checkout-app \
  db_name=checkout \
  creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{password}}' VALID UNTIL
'{{expiration}}'; GRANT app_readwrite TO \"{{name}}\";" \
  default_ttl="1h" \
  max_ttl="24h"
```

The mental shift is significant: "the database password" no longer exists. What exists is a permission Vault grants to an identity, and a disposable credential. A leak stops being an emergency and becomes an audit log entry, because the value expires on its own.

Two operational details you learn the hard way. First: dynamic credentials must be *renewed* , and your connection pool has to reconnect when the old one expires; if you open a hundred connections at startup and never recycle them, an hour later you have a hundred connections owned by a user that no longer exists. Second: every credential creates a real role in PostgreSQL, so a very short TTL under heavy traffic fills the

catalogue with orphaned roles. A 1-hour TTL with a 24-hour `max_ttl` is a sensible starting point.

## How the secret reaches the process: environment variables vs files

This is the decision most people skip and the one that causes the most grief. Environment variables are convenient and have three concrete problems:

- **They're inherited.** Any subprocess you spawn — a hook, a maintenance script, a library shelling out to `curl` — receives the entire environment.
- **They're readable from outside.** They sit in `/proc/<pid>/environ`, they show up in core dumps, and plenty of error handlers serialise the whole environment into the exception report.
- **They never update.** When you rotate the secret, the process keeps the value it booted with until you restart it. Rotating means deploying.

Mounting secrets as files solves all three. In Kubernetes, a `secret` volume updates itself when the underlying Secret changes, without restarting the pod. Here's the configuration pattern that takes advantage of it:

```
# config.py
from functools import lru_cache

from pydantic import SecretStr, model_validator
from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    model_config = SettingsConfigDict(
        env_file=None,                        # in production NO .env is read
        secrets_dir="/var/run/secrets/app",   # one file per secret
        extra="forbid",                       # a stray variable is an error, not silence
    )

    # SecretStr never prints itself: its repr is "**********".
    database_url: SecretStr
    stripe_api_key: SecretStr
    environment: str = "production"

    @model_validator(mode="after")
    def reject_test_keys(self) -> "Settings":
        if self.environment == "production":
            if self.stripe_api_key.get_secret_value().startswith("sk_test_"):
                raise ValueError("Stripe test key in a production environment")
        return self


@lru_cache
def get_settings() -> Settings:
    # Fail at boot if a secret is missing, rather than at 3am on the first
    # request that happens to hit that code path.
    return Settings()
```

Two decisions here are worth attention. `SecretStr` keeps the value out of `repr()`, out of a `print()` of the settings object, and out of most exception reports: you have to ask for it explicitly with `get_secret_value()`,

and that call is easy to audit with a grep. And `extra="forbid"` turns a misspelled `DATABSE_URL` into a loud startup failure instead of a service that boots with a default and fails in confusing ways.

For rotation to work without a restart, you have to re-read the file. This wrapper does it at negligible cost:

```
# secret_file.py
import threading
import time
from pathlib import Path


class RotatingSecret:
    """A file-mounted secret that reloads when the kubelet updates it.

    Kubernetes projects secret volumes through symlinks into a ..data directory
    that it swaps atomically. stat() on the path follows the symlink, so the
    mtime changes as soon as the Secret rotates.
    """

    def __init__(self, path: str, ttl_s: float = 5.0) -> None:
        self._path = Path(path)
        self._ttl = ttl_s
        self._lock = threading.Lock()
        self._value: str | None = None
        self._mtime: float = -1.0
        self._checked_at: float = 0.0

    def get(self) -> str:
        now = time.monotonic()
        with self._lock:
            fresh = self._value is not None and (now - self._checked_at) < self._ttl
            if fresh:
                return self._value

            mtime = self._path.stat().st_mtime
            if mtime != self._mtime or self._value is None:
                self._value = self._path.read_text().strip()
                self._mtime = mtime
            self._checked_at = now
            return self._value


stripe_key = RotatingSecret("/var/run/secrets/app/stripe_api_key")

# Ask for the value at the point of use; don't capture it at import time.
# That detail is what makes rotation actually reach your code.
def charge(amount_cents: int) -> None:
    client = StripeClient(api_key=stripe_key.get())
    ...
```

That last comment is the one that matters: if you write `API_KEY = stripe_key.get()` at module level, you're back to exactly the behaviour of an environment variable. Hot rotation only works if the value is read at the moment it's used.

## Kubernetes: External Secrets Operator

Kubernetes Secrets are not encrypted: they're base64-encoded, which is a transport format, not a protection. Two platform settings are non-negotiable first:

- **Encryption at rest** in etcd via an `EncryptionConfiguration` backed by a KMS. Without it, an etcd backup is a file containing every one of your secrets in the clear.
- **Strict RBAC.** Anyone who can `get secrets` in a namespace — or simply create a pod in it — can read every secret in it. The namespace is the real security boundary; treat it as one.

With that settled, the dominant pattern is keeping the source of truth in an external manager and syncing it with External Secrets Operator. First you declare how to talk to the manager:

```
# On older clusters the API version is external-secrets.io/v1beta1.
apiVersion: external-secrets.io/v1
kind: ClusterSecretStore
metadata:
  name: aws-secrets
spec:
  provider:
    aws:
      service: SecretsManager
      region: eu-west-1
      auth:
        jwt:
          # The ServiceAccount is annotated with the IAM role (IRSA):
          # the operator itself holds no static key either.
          serviceAccountRef:
            name: external-secrets
            namespace: external-secrets
```

Then, per application, which secrets it wants and under what names it expects them:

```
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: checkout
  namespace: checkout
spec:
  refreshInterval: 1h          # how often the source is checked
  secretStoreRef:
    name: aws-secrets
    kind: ClusterSecretStore
  target:
    name: checkout-secrets     # the Kubernetes Secret that gets created and kept in sync
    creationPolicy: Owner
    deletionPolicy: Delete     # delete the ExternalSecret, the Secret goes too
  data:
    - secretKey: database_url  # becomes the filename when mounted
      remoteRef:
        key: prod/checkout/database
        property: url
    - secretKey: stripe_api_key
```

```
      remoteRef:
        key: prod/checkout/stripe
        property: secret_key
```

Notice that the `secretKey` names match the fields of the `Settings` class above. That's not a coincidence: mounting the Secret as a volume makes Kubernetes create one file per key, and pydantic-settings' `secrets_dir` reads them by name. The whole chain fits together with no glue code:

```
      containers:
        - name: app
          volumeMounts:
            - name: secrets
              mountPath: /var/run/secrets/app
              readOnly: true
      volumes:
        - name: secrets
          secret:
            secretName: checkout-secrets
            defaultMode: 0400   # read-only for the container user
```

The result is that rotating a secret in AWS Secrets Manager propagates by itself: ESO updates the Secret, the kubelet updates the mounted file, and `RotatingSecret` re-reads it on the next request. Zero deploys, zero restarts.

One alternative worth considering for your most sensitive material is the **Secrets Store CSI Driver**, which mounts straight from the external manager without materialising a Kubernetes Secret at all. It's stricter (the value never touches etcd) at the cost of losing integration with everything that expects a native Secret.

## Zero-downtime rotation: the two-credential pattern

The reason almost nobody rotates their keys is that rotating "all at once" causes an outage: you revoke the old one, and for the seconds or minutes it takes the deploy to propagate, the instances that haven't caught up start failing. The fix is for the system to accept two valid credentials at the same time during an overlap window.

On the side that *verifies* a key — one of your APIs, an inbound webhook — the implementation is straightforward, using constant-time comparison so you don't leak information through response timing:

```
import hmac

from prometheus_client import Counter

# The metric is the important part: without it you don't know when it's safe to revoke.
key_use = Counter("api_key_use_total", "Usage per key identifier", ["key_id"])


def verify_key(presented: str, active_keys: dict[str, str]) -> str | None:
    """Return the matching key_id, or None. Accepts both the new and previous key."""
    for key_id, value in active_keys.items():
        if hmac.compare_digest(presented, value):
            key_use.labels(key_id=key_id).inc()
```

```
            return key_id
    return None
```

The full procedure has four phases, and the third is the one people skip:

1. **Issue** the new credential without touching the old one. Both are valid.
2. **Propagate**: consumers start using the new one. With file-mounted secrets this happens on its own; otherwise it's a deploy.
3. **Wait and verify**: watch `api_key_use_total` filtered by the old `key_id` until it has sat at zero for a reasonable window. "Reasonable" has to cover your slowest processes: nightly jobs, a monthly cron, that service which only wakes up at quarter close.
4. **Revoke** the old one. Then confirm it actually stopped working, because a revocation that fails silently is worse than not rotating: it gives you false confidence.

On frequency, a policy that works without turning into ceremony: workload identities and tokens with a TTL under 24 hours need no policy because they rotate on every use; database passwords every 90 days if they aren't dynamic; service-to-service tokens every 30; long-lived third-party keys every 180 days, using the occasion to review what permissions they actually hold. Artifact signing keys and root CAs are a separate case, with planned annual rotation and an explicit overlap.

## The leak that goes out through your logs

Repositories get all the attention, but a large share of real leaks exit through observability: a `logger.info("connecting to %s", database_url)`, a traceback that serialises the settings object, an OpenTelemetry span carrying the `Authorization` header as an attribute. And that data lands in a system whose permissions are usually laxer than production's.

The primary defence is by design — `SecretStr` keeps the value from reaching the log at all — but a safety net in the log processor is worth having:

```
import re

import structlog

PATTERNS = re.compile(
    r"(sk_live_[A-Za-z0-9]+"
    r"|Bearer\s+[A-Za-z0-9._\-]+"
    r"|postgres(?:ql)?://[^\s@]+:[^\s@]+@)"
)

SENSITIVE_KEYS = {"password", "token", "authorization", "api_key", "secret", "cookie"}


def redact(_logger, _method, event_dict):
    for key, value in list(event_dict.items()):
        if key.lower() in SENSITIVE_KEYS:
            event_dict[key] = "[REDACTED]"
        elif isinstance(value, str):
            event_dict[key] = PATTERNS.sub("[REDACTED]", value)
    return event_dict
```

```
structlog.configure(
    processors=[
        redact,                                  # before rendering, always
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.JSONRenderer(),
    ]
)
```

Treat it as what it is: an airbag. A regex filter only catches formats you already know about, and the day you integrate a new provider with a different prefix, it won't see it. Redaction buys time; design is what prevents the incident.

## Runbook: a secret has leaked

When it happens, the order of operations matters more than speed. This is the order:

1. **Revoke before you rotate.** This is mistake number one under pressure: people generate a new key, deploy, and forget to invalidate the old one, which keeps working perfectly for whoever holds it. If the outage is acceptable, revoke first. If not, issue the new one, propagate it, and revoke as soon as the usage metric hits zero.
2. **Assume full compromise.** It doesn't matter that the repository was private, that the commit was deleted five minutes later, or that you rewrote history with `git filter-repo`. Forks, local clones, the forge's cache and CI logs all keep copies. Rewriting history is cleanup after the fact, not containment.
3. **Audit from first exposure.** Not from when you found out. Search CloudTrail or the provider's audit logs for every use of that credential since the original commit date, paying attention to unusual IPs, regions and hours.
4. **Review the blast radius.** What could that key actually do? If the answer is "everything", you have a second problem — excessive permissions — and this is the moment to fix it, not to restore the same level of access.
5. **Close the door.** Why didn't the hook stop it? Why didn't CI see it? Every incident should end with a new or repaired detection layer, not just a new key.

## The threat model: what actually happens when a key leaks

There is a reason secrets guidance tends to bounce off engineering teams: it gets written as if the risk were abstract. It isn't. When a key lands in a public repository, it gets picked up by bots watching the GitHub events firehose, not by people. The gap between the push and the first attempt to use it is measured in seconds or minutes, not days. And the bot does not need to understand your product: it tries the credential against a catalogue of known APIs, sees what answers, and sells or exploits whatever works.

That changes the question worth asking. It isn't "how do I stop a secret from leaking?" — that too, but that's the easy half — it's "how much damage can this particular credential do in the first ten minutes?" That answer is called the **blast radius**, and it's what separates a boring incident from one that ends in a customer notification.

Here's an exercise that costs an afternoon and changes decisions for years: take one service's secret inventory and, for each entry, write one line describing what an attacker would do with it assuming they already have it. Not what it's *supposed* to allow according to the docs — what it allows given the permissions it actually carries today.

- **An S3 key with `s3:GetObject` on one prefix of one bucket:** reads those files. Annoying, bounded, auditable.
- **The same key with `s3:*` on `*`:** reads everything, deletes everything, and can upload content your users will download as though it came from you.

- **A Slack webhook token:** posts messages to a channel. Sounds harmless until someone uses it for internal phishing from a source your team already trusts.
- **A GitHub personal access token with `repo`:** reads all your private code, and can also modify workflows — which means escalating into the rest of your CI credentials.
- **An application database credential:** depends entirely on whether that role has `SELECT` on the tables it needs or owns the schema.

The list is almost always uncomfortable, because it reveals that most credentials are oversized by inertia: somebody needed to unblock themselves on a Tuesday, asked for broad permissions "for now", and that "for now" has been in production for eighteen months. Cutting permissions isn't a separate security project; it's the measure that makes every other measure matter less.

There are three properties you want from any credential, in this order of value:

1. **Scope**: which resources it touches and with which verbs. This is what bounds the damage.
2. **Lifetime**: how long it stays valid. This is what bounds the window.
3. **Attribution**: whether the logs say *who* used it. This is what makes response possible.

A static credential shared by five services with admin permissions fails all three. A fifteen-minute OIDC token issued to one specific workflow in one specific repository wins all three, and there is no secret to store at all. That's the whole thesis of this guide, expressed as properties rather than as tools.

## CI/CD: where most secrets are actually lost

If I had to bet on where the leak you haven't found yet is hiding, I'd bet on CI. It's the one system in your organisation that by design has production access, runs code that changes daily, and produces logs people read in public. It also evolves: what was safe in your pipeline two years ago may not be safe today.

### The secretless path: OIDC into your cloud provider

GitHub Actions can ask GitHub for a short-lived JWT that describes *who the workflow is*: which repository it runs in, which branch, which environment. AWS, GCP and Azure all know how to validate that token against GitHub's issuer and hand back temporary credentials. The result: zero long-lived keys stored in repository settings.

You register the identity provider once per account. With Terraform:

```
resource "aws_iam_openid_connect_provider" "github" {
  url            = "https://token.actions.githubusercontent.com"
  client_id_list = ["sts.amazonaws.com"]
}
```

Then the role, with the part that actually matters: the condition on the `sub` claim.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": { "Federated": "arn:aws:iam::111122223333:oidc-provider/
token.actions.githubusercontent.com" },
    "Action": "sts:AssumeRoleWithWebIdentity",
```

```
    "Condition": {
      "StringEquals": {
        "token.actions.githubusercontent.com:aud": "sts.amazonaws.com",
        "token.actions.githubusercontent.com:sub": "repo:my-org/my-repo:environment:production"
      }
    }
  }]
}
```

Three details decide whether this is real security or theatre:

- **Never leave the policy with only the `aud` condition.** That value is identical for every GitHub repository on earth. A policy that checks only `aud` lets any workflow from any organisation assume your role.
- **Never scope to `repo:my-org/*:*`.** That means any branch of any repository in the org, including the test branch an outside collaborator opened yesterday.
- **Avoid `ForAllValues:` operators in `Allow` statements.** They evaluate to true when the claim is absent or misspelled — which is precisely the case where you want evaluation to fail. Use `StringEquals`, or `StringLike` if you need a tightly bounded wildcard.

## The immutable claim change (July 2026)

This is the operational detail most likely to catch you off guard this year. Repositories created on github.com on or after 15 July 2026 — and older ones that have opted in — emit an *immutable* `sub`: alongside the organisation and repository names, it appends each one's permanent numeric ID, separated by `@`. The reasoning is sound: it stops someone who registers a recycled organisation name from minting tokens that match a stale trust policy.

```
# Legacy form (names only)
repo:my-org/my-repo:environment:production

# Immutable form (names + permanent IDs)
repo:my-org@12345678/my-repo@98765432:environment:production
```

The symptom when this hits you is unmistakable and baffling: a workflow that has worked for months suddenly fails with `Not authorized to perform sts:AssumeRoleWithWebIdentity`, with nobody having touched the role or the pipeline. The cause is that the token now carries the new format while your `StringEquals` condition compares against the old one. The fix is to update the condition to the immutable form.

If you aren't sure what your workflow is emitting, don't guess — print it. There are debugging actions that decode the JWT payload and display it. Always run them in a private repository: the token expires quickly, but the claim values describe your internal topology.

## Secret masking is not a security boundary

GitHub replaces registered secret values with asterisks when they appear verbatim in logs. That's a safety net against slips, not a control. Any transformation slips straight past it:

```
# All of this shows up in the log in cleartext
echo "$MY_SECRET" | base64
echo "$MY_SECRET" | rev
```

```
echo "$MY_SECRET" | cut -c1-10
echo "$MY_SECRET" | fold -w1   # one character per line
```

Treat it as what it is: if a step in your pipeline runs third-party code with a secret in the environment, that secret is compromised the moment that code decides to compromise it. The defence isn't masking — it's not putting it in the environment.

### Dangerous triggers and third-party code

Two specific risks deserve a written team rule:

- `pull_request_target` runs the workflow in the base repository's context — with access to secrets — but can check out the pull request's code. Combine that trigger with a checkout of the contributor's branch and then execute anything from that tree (a build script, a dependency install hook) and you have just handed your secrets to whoever opened the PR. If you need that trigger, don't check out untrusted code, or split the work across two workflows.
- **Unpinned third-party actions.** `uses: someone/action@v3` resolves a mutable tag: whoever controls that repository can move the tag to a new commit whenever they like. Pin to a full commit SHA and leave the version number in a comment so it stays readable.

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production        # activates protection rules and reviewers
    permissions:
      id-token: write              # required for OIDC
      contents: read               # the minimum needed for checkout
    steps:
      - uses: actions/checkout@11bd71901bbe5b1630ceea73d27597364c9af683  # v4.2.2
      - uses: aws-actions/configure-aws-credentials@e3dd6a429d7300a6a4c196c26e071d42e0343502  #
v4.0.2
        with:
          role-to-assume: arn:aws:iam::111122223333:role/deploy-production
          aws-region: eu-west-1
      - run: ./scripts/deploy.sh
```

The `permissions` block matters and is forgotten every time. Without it, the job's `GITHUB_TOKEN` inherits the organisation's default permissions, which in older repositories usually means write on everything. Declaring it explicitly and minimally turns a compromised job into a much smaller problem.

One more layer, free and badly underrated: GitHub **environments**. A `production` environment can require human reviewers, restrict which branches may deploy, and hold its own secrets. With OIDC, the environment name also travels inside the `sub` claim, so the restriction is enforced twice — once in GitHub and once in the IAM trust policy. If one of the two gets loosened by mistake, the other still holds.

## Secrets in container images: the leak that travels with you

A container image is a layered archive format, and every layer is immutable and shipped whole. That property — the thing that makes containers fast and cacheable — is also what turns a misplaced secret into a permanent problem: once the layer exists, deleting the file in a later instruction doesn't remove it, it only hides it in the final filesystem. The earlier layer is still there, and anyone with access to the registry can extract it.

```
# BAD: the token stays in the image history forever
FROM node:22-slim
ARG NPM_TOKEN
RUN echo "//registry.npmjs.org/:_authToken=$NPM_TOKEN" > ~/.npmrc  && npm ci  && rm ~/.npmrc
# this rm accomplishes nothing
```

The `rm` creates a new layer marking the file as deleted. The layer where it was written still exists, and the `ARG` value also lands in the image metadata. Checking costs one command:

```
# Full history, untruncated
docker history --no-trunc my-image:latest

# And the actual layer dump, in case something slipped into the filesystem
docker save my-image:latest -o /tmp/img.tar
mkdir -p /tmp/img && tar -xf /tmp/img.tar -C /tmp/img
grep -ria "authToken|BEGIN PRIVATE KEY|aws_secret" /tmp/img | head
```

If that `grep` returns anything, the secret is compromised for everyone who has pulled that tag. Publishing a new image doesn't fix it — you have to rotate the credential.

## The right way: BuildKit secret mounts

BuildKit — Docker's default build engine for several versions now — solves this with `RUN --mount=type=secret`. The value is exposed as a file inside the build container for the duration of that single `RUN` instruction, and it is never written to any layer or to the build cache.

```
# syntax=docker/dockerfile:1.7
FROM node:22-slim AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN --mount=type=secret,id=npm_token,required=true     NPM_TOKEN="$(cat /run/secrets/npm_token)"
npm ci --omit=dev

FROM node:22-slim
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
USER node
CMD ["node", "server.js"]
```

```
# From a file
docker build --secret id=npm_token,src=./npm_token.txt -t my-image:latest .

# From an environment variable, never touching disk
export NPM_TOKEN="$(aws secretsmanager get-secret-value   --secret-id build/npm --query SecretString
--output text)"
docker build --secret id=npm_token,env=NPM_TOKEN -t my-image:latest .
```

The `required=true` flag deserves a note, because it prevents the worst possible failure mode: without it, if you forget to pass the secret the mount is simply empty and the build carries on with a blank token. With

`required=true` the build fails loudly, which is what you want.

Two important caveats. First: the mount protects *storage*, not *use*. If the command you run prints the secret, writes it to a log file inside the image, or bakes it into a compiled artifact, the mount saves you from nothing. Second: multi-stage builds are not optional here. If the secret is used by the `deps` stage and the final image only copies `node_modules`, nothing from the intermediate stage ever gets published.

At runtime the rule is even simpler: **no credentials go into the image**. The image is an artifact promoted unchanged from staging to production; if it contained secrets you'd have to build a different image per environment and you'd lose the single most valuable property of artifact-based deployment. Secrets arrive at startup, from the manager, via workload identity.

## Terraform and state: the leak almost nobody audits

Here's a classic blind spot. Terraform's state file stores, in cleartext, every attribute of every resource it manages — including the sensitive ones. Marking a variable `sensitive = true` only keeps the value out of console output: **it still gets written to state**. If your backend is an unencrypted bucket, or if somebody copies `terraform.tfstate` to their laptop to debug something, the production database password has gone for a walk.

```
# What's actually inside your state
terraform show -json | jq '.values.root_module.resources[]
  | select(.values.password != null)
  | {addr: .address, has_password: true}'
```

### Ephemeral values and write-only arguments

Terraform 1.10 introduced *ephemeral values* and 1.11 extended them to managed resources through *write-only arguments*. An `ephemeral` block produces a value Terraform uses during the run but never persists to the plan or the state. A write-only argument accepts that value on a resource and likewise doesn't store it.

```
# Fetched on every run, never persisted to any artifact
ephemeral "aws_secretsmanager_secret_version" "db" {
  secret_id = "production/postgres/master"
}

resource "aws_db_instance" "main" {
  identifier     = "app-production"
  engine         = "postgres"
  instance_class = "db.m6g.large"
  username       = "appadmin"

  # Write-only argument: goes in, never lands in state
  password_wo         = ephemeral.aws_secretsmanager_secret_version.db.secret_string
  password_wo_version = 3   # bump this to force the value to be re-sent
}
```

The `_wo_version` field is the trick that makes the whole mechanism work. Because Terraform doesn't store the value, it can't diff it to detect changes; the version number is the explicit signal that says "this value changed,

send it again". If you rotate the password in the manager and don't bump that number, Terraform will do nothing.

Write-only arguments require Terraform 1.11 or newer *and* provider support. Not every resource has them yet, so the realistic strategy has three tiers:

1. Use `ephemeral` and `_wo` arguments wherever the provider supports them.
2. Where it doesn't, keep the secret out of Terraform entirely: create the resource with a randomly generated password managed by rotation outside Terraform, or create only the secret *container* and let the application read it.
3. And in every case, treat the state backend as if it contained secrets: encrypted at rest, access restricted to a CI role, versioning on, access logs enabled.

```
terraform {
  backend "s3" {
    bucket       = "tf-state-production"
    key          = "app/terraform.tfstate"
    region       = "eu-west-1"
    encrypt      = true
    kms_key_id   = "arn:aws:kms:eu-west-1:111122223333:key/abcd-1234"
    use_lockfile = true
  }
}
```

And one hygiene rule that saves a lot of grief: state does not get downloaded to laptops. If you need to inspect it, do it from the same environment that runs your applies — and if you download it anyway, delete it properly afterwards, including the `terraform.tfstate.backup` file Terraform leaves behind without telling you.

## Vault in depth: dynamic credentials without surprises

The idea behind dynamic credentials is elegant: the application stores no database password. Instead it authenticates to Vault with its identity and asks for a fresh credential, created on the spot, with a bounded lifetime. When the lease expires, Vault issues a `DROP ROLE` and that credential ceases to exist. A memory dump or a leaked log stops being a permanent problem and becomes a problem with an expiry date.

```
# Database engine configuration
vault write database/config/postgres-production   plugin_name="postgresql-database-plugin"
allowed_roles="app-read,app-write"   connection_url="postgresql://{{username}}:{{password}}
@db.internal:5432/app?sslmode=require"   username="vault_manager"   password="$INITIAL_PASSWORD"

# Critical: rotate the manager password immediately.
# From here on only Vault knows it; no human can use it.
vault write -force database/rotate-root/postgres-production

# A role with narrow permissions and a short TTL
vault write database/roles/app-read   db_name="postgres-production"   creation_statements="CREATE
ROLE "{{name}}" WITH LOGIN PASSWORD '{{password}}' VALID UNTIL '{{expiration}}'; GRANT SELECT ON ALL
TABLES IN SCHEMA public TO "{{name}}";"   revocation_statements="DROP ROLE IF EXISTS "{{name}}";"
default_ttl="1h"   max_ttl="24h"
```

Vault's defaults are a one-hour TTL and a twenty-four-hour max TTL, and it's worth understanding the

difference properly, because the two most common operational failures come straight out of it.

## Pitfall 1: connection pools and the max TTL

A credential can be *renewed* as many times as you like up to the max TTL. Once you hit it, renewal is no longer possible: you have to request a fresh credential. If your application uses a connection pool — and it does — connections already open with the old credential stay alive in the pool for as long as the database engine keeps them, but any new connection with those credentials will fail, and in many engines the `DROP ROLE` also terminates existing sessions.

The symptom is distinctive: the application runs perfectly for hours and then, exactly twenty-four hours after startup, starts returning authentication errors on a fraction of requests — precisely the ones that land on recycled connections. The fix isn't to raise the max TTL until the problem goes away; it's to make the data access layer handle it:

```
import hvac, time, threading
from sqlalchemy import create_engine

class VaultCredentialedPool:
    """Renews the lease and, when renewal is no longer possible, rebuilds the pool."""

    def __init__(self, client: hvac.Client, role: str, dsn_template: str):
        self.client, self.role, self.dsn_template = client, role, dsn_template
        self._engine = None
        self._lease_id = None
        self._build()
        threading.Thread(target=self._renewal_loop, daemon=True).start()

    def _build(self):
        r = self.client.secrets.database.generate_credentials(name=self.role)
        self._lease_id = r["lease_id"]
        user, password = r["data"]["username"], r["data"]["password"]
        previous = self._engine
        # pool_pre_ping discards dead connections before handing them out
        self._engine = create_engine(
            self.dsn_template.format(user=user, password=password),
            pool_pre_ping=True, pool_recycle=1800, pool_size=10,
        )
        if previous is not None:
            # graceful drain: in-flight connections finish their work
            previous.dispose(close=False)

    def _renewal_loop(self):
        while True:
            time.sleep(300)
            try:
                self.client.sys.renew_lease(lease_id=self._lease_id, increment=3600)
            except hvac.exceptions.InvalidRequest:
                # max TTL reached: time for a fresh credential
                self._build()

    @property
```

```
    def engine(self):
        return self._engine
```

What matters isn't the specific library, it's the pattern: renew while you can, treat renewal failure as an expected signal rather than an error, and rebuild the pool by draining the old one instead of killing it. With that in place, rotation stops being an event and becomes routine.

### Pitfall 2: the root user assigned to a static role

The other failure is subtler and breaks things all at once. If you assign the manager user's credentials — the ones Vault uses to connect — to a static role so Vault will rotate them for you, the next rotation changes that user's password *without* updating the `config/` block. From that moment Vault can no longer authenticate to the database, and every dynamic and static user under that configuration stops being issued. The rule: rotate the manager user with `rotate-root`, never through a static role.

The *transit* engine is also worth mentioning, because it solves a different and very useful problem: encryption as a service. The application sends data and receives ciphertext back, and the key never leaves Vault. It's the right tool for encrypting individual fields — a bank account number, a national ID — without scattering key material across your services.

```
vault secrets enable transit
vault write -f transit/keys/pii

# Encrypt (the app never sees the key)
vault write transit/encrypt/pii plaintext=$(base64 <<< "ES9121000418450200051332")
# -> ciphertext: vault:v1:8SDd3WHDOjf7mq69CyCqYjBXAiQQAVZRkFM13ok481zoCmHnSeDX9vyf7w==

# Decrypt
vault write transit/decrypt/pii ciphertext="vault:v1:8SDd3WHD..."
```

The `v1` prefix on the ciphertext is the key version. When you rotate the key (`vault write -f transit/keys/pii/rotate`), old data still decrypts with the version it was encrypted under, and new data uses the current one. That turns encryption key rotation — historically a painful migration project — into a one-command operation.

## Choosing a manager: a practical comparison

There's no universal answer, but there are criteria that order the decision better than feature tables do.

- **AWS Secrets Manager**: roughly USD 0.40 per secret per month plus a per-API-call charge. In return you get managed rotation with prebuilt functions for RDS, DocumentDB and Redshift, cross-region replication, and versioning with staging labels. It's the right choice when the secret *rotates*.
- **SSM Parameter Store (standard)**: no charge for storage or API calls up to the parameter limit, with KMS encryption if you use the `SecureString` type. It doesn't rotate on its own, and default throughput is modest — on the order of tens of operations per second, raisable by enabling higher-throughput mode. It's the right choice for sensitive configuration that doesn't rotate.
- **GCP Secret Manager and Azure Key Vault**: equivalent models, with versioning and native integration with each platform's workload identity. Key Vault additionally separates secrets, keys and certificates, which helps when certificate management is a real problem rather than a footnote.
- **HashiCorp Vault**: the most capable option — dynamic credentials, transit, PKI, multiple auth backends — and also the one with the highest operational cost, because you have to run it highly available with an unsealing

> story that works at three in the morning. Worth it when you want dynamic credentials, or when you need one secrets plane across several clouds.

A mixed strategy usually wins in practice: whatever rotates goes in Secrets Manager, the rest of your sensitive configuration goes in Parameter Store, and Vault only if dynamic credentials justify operating it.

## Caching isn't optional

The most common cost and latency mistake is reading the secret on every request. A service handling a thousand requests per second that calls the manager each time makes eighty-six million calls a day: it's expensive, it's slow, and it will eventually hit rate limits. Read the secret at startup, hold it in memory, refresh it in the background.

```
import time, threading, boto3

class CachedSecret:
    """Read once, refresh in the background, never block a request."""

    def __init__(self, secret_id: str, ttl: int = 300):
        self._client = boto3.client("secretsmanager")
        self._id, self._ttl = secret_id, ttl
        self._value, self._expires = None, 0
        self._lock = threading.Lock()

    def get(self) -> str:
        now = time.monotonic()
        if self._value is not None and now < self._expires:
            return self._value
        with self._lock:
            if self._value is not None and time.monotonic() < self._expires:
                return self._value
            try:
                r = self._client.get_secret_value(SecretId=self._id)
                self._value = r["SecretString"]
                self._expires = time.monotonic() + self._ttl
            except Exception:
                # Graceful degradation: if the manager is down and we have
                # a previous value, keep serving with it.
                if self._value is None:
                    raise
                self._expires = time.monotonic() + 30
            return self._value
```

That `except` block is the part people leave out and later regret. If the secrets manager has an incident and your service can neither start nor serve without it, you've just turned a configuration dependency into an availability dependency on your critical path. Serving the last known value for a short window is almost always the right call.

In serverless environments, AWS ships a Lambda extension that caches secrets and parameters between invocations and cuts API calls dramatically. It's one line of configuration and usually pays for its own adoption on the first bill.

## Envelope encryption: what KMS is doing under the hood

This mechanism is worth understanding, because it explains why secrets managers are built the way they are and it prevents a couple of expensive decisions.

A key management service does not encrypt your data with the master key. Encrypting a hundred megabytes by shipping it to a remote service would be absurd on latency and on size limits. What it does instead is **envelope encryption**: it generates a random data key (DEK), returns it to you twice — once in plaintext and once encrypted under the master key (KEK), which never leaves the cryptographic module — and you encrypt locally with the plaintext DEK, wipe it from memory, and store the encrypted DEK alongside the data.

```
import boto3, os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

kms = boto3.client("kms")
KEY_ID = "arn:aws:kms:eu-west-1:111122223333:key/abcd-1234"

def encrypt(data: bytes, context: dict[str, str]) -> dict:
    # KMS generates the DEK; returns the plaintext and encrypted versions
    r = kms.generate_data_key(KeyId=KEY_ID, KeySpec="AES_256",
                              EncryptionContext=context)
    dek, encrypted_dek = r["Plaintext"], r["CiphertextBlob"]
    try:
        nonce = os.urandom(12)
        ct = AESGCM(dek).encrypt(nonce, data, None)
    finally:
        del dek   # out of memory the moment it stops being needed
    return {"dek": encrypted_dek, "nonce": nonce, "ct": ct, "ctx": context}

def decrypt(envelope: dict) -> bytes:
    # The context must match exactly or KMS refuses the operation
    dek = kms.decrypt(CiphertextBlob=envelope["dek"],
                      EncryptionContext=envelope["ctx"])["Plaintext"]
    try:
        return AESGCM(dek).decrypt(envelope["nonce"], envelope["ct"], None)
    finally:
        del dek
```

The `EncryptionContext` is the piece almost everyone ignores and the one that adds the most value. It's a set of key-value pairs bound cryptographically to the operation: if you encrypt with `{"tenant": "acme"}`, decryption only succeeds when you pass exactly that context. That prevents one customer's encrypted data from being decrypted in another customer's context even though they share a master key — and the context also shows up in KMS audit records, so your trail stops saying "somebody decrypted something" and starts saying what.

This same mechanism is what sits underneath Secrets Manager, Parameter Store `SecureString` values, Kubernetes secret encryption in *etcd*, and Vault's transit engine. Once you understand the envelope, the question "where does the trust actually live?" has a clear answer: in the master key's policy. An immaculate secrets manager sitting on a KMS policy that grants `kms:Decrypt` to half the account isn't protecting much.

## There is no such thing as a secret on the client

This is the conversation that recurs in every team with a mobile app or a frontend: "where do we store the API key in the app?" The honest answer is that you don't, because there is nowhere it's safe. An APK unzips, a JavaScript bundle reads, an iOS app extracts off the device. Obfuscation raises the cost of finding it from five minutes to half an hour, and that's all it buys.

The structural fix is to give the client application no credential that does anything useful. What you do instead:

- **A backend-for-frontend.** The client talks to your server with a session token representing *that user*; your server holds the third-party key and makes the call. A stolen user token lets an attacker do what that user can do — which is already your authorisation model — not drain your quota with a paid provider.
- **Restricted public keys.** Some APIs (maps, analytics) offer keys designed to live on the client, restricted by referrer domain, bundle ID or quota. These are not secrets: they're identifiers with limits. Configure the restrictions, because without them they stop being bounded.
- **Short-lived signed URLs.** For direct uploads and downloads against object storage, the server signs a URL granting one operation on one object for a few minutes. The client never sees a credential at all.

```
# The server signs; the browser uploads straight to S3 with no credentials
import boto3

s3 = boto3.client("s3")

def upload_url(user_id: str, filename: str) -> dict:
    return s3.generate_presigned_post(
        Bucket="user-uploads",
        Key=f"u/{user_id}/{filename}",
        Fields={"Content-Type": "image/jpeg"},
        Conditions=[
            {"Content-Type": "image/jpeg"},
            ["content-length-range", 1, 5 * 1024 * 1024],  # 5 MB max
        ],
        ExpiresIn=300,   # five minutes
    )
```

Note the `Conditions`: without them the signed URL would allow uploading a file of any size and any type. A loosely scoped signed URL is a loosely scoped credential; the same principles of scope and lifetime apply exactly the same way.

## Detection: finding out yourself, not from a stranger's email

Prevention fails sometimes. What distinguishes a mature team is how long it takes to find out. Three signals give a lot for very little effort.

**Manager access auditing.** Every secret read is logged. That trail is good for two things: spotting anomalous access, and — more useful day to day — finding dead secrets. A secret nobody has read in ninety days is a secret you can delete, and every deleted secret is one less piece of attack surface.

```
# Who has read secrets in the last 24 hours
aws cloudtrail lookup-events   --lookup-attributes
AttributeKey=EventName,AttributeValue=GetSecretValue   --start-time "$(date -u -d '24 hours ago' +
%Y-%m-%dT%H:%M:%SZ)"   --query 'Events[].CloudTrailEvent' --output text   | jq -r
```

```
'[.userIdentity.arn, .requestParameters.secretId] | @tsv'   | sort | uniq -c | sort -rn
```

**Decoy tokens.** A *honeytoken* is a credential that looks real, does nothing, and alerts when someone uses it. Put one in a `.env.example` file, another in the internal wiki, another in an old container image. They produce no false positives — nobody has a legitimate reason to use them — so any alert is a real alert. It's one of the few detection measures with an almost absurd cost-to-benefit ratio.

**Programme metrics, not just incident metrics.** If you want this to improve steadily, measure four things and review them monthly:

1. **Identity coverage**: the percentage of workloads using federated identity instead of static keys. This is the metric that correlates most strongly with real risk reduction.
2. **Age of the static credentials that remain**: the 95th percentile of days since last rotation. If it grows month over month, rotation isn't automated no matter how well documented the procedure is.
3. **Time to revocation**: in your last drill or real incident, how long passed between detection and effective invalidation. This number only drops if somebody practises.
4. **Orphaned secrets**: how many have gone more than ninety days without a read. Target: zero, by deletion, not by exception.

## Make the secure path the easy path

One last point, and probably the one that most determines whether any of this survives six months. Every control in this article fails if the correct path is slower than the shortcut. When getting the project running locally with the secrets manager takes four steps and an access request, while copying a teammate's `.env` over DM takes one, people copy the `.env`. That isn't indiscipline: it's the rational response to a badly placed incentive.

The goal is that a developer types one command and everything works:

```
#!/usr/bin/env bash
# scripts/dev — start locally with temporary credentials
set -euo pipefail

if ! aws sts get-caller-identity >/dev/null 2>&1; then
  echo "Starting SSO session..." >&2
  aws sso login --profile development
fi

# Secrets are injected into the child process and never touch disk
exec aws-vault exec development --   env $(aws secretsmanager get-secret-value           --secret-id
dev/app --query SecretString --output text         | jq -r 'to_entries | map("(.key)=(.value)") | .
[]')   npm run dev
```

Three properties make this work: there is no secrets file on the laptop's disk, the credentials expire on their own when the SSO session ends, and the developer hasn't had to learn anything new — they still type one command to start. If the development environment also points at a development database with synthetic data, you've eliminated the entire category of "somebody had production credentials on a laptop" in one move.

Round it off with the obvious but rarely written thing: a one-page document saying where each type of secret lives, who can grant it, and what to do if it leaks. When someone new starts on a Monday, that document is the

difference between them adopting your model and inventing their own.

## Eight recurring mistakes

- **Rotating without revoking.** The old key stays alive. It's covered above because it's the most frequent and the most expensive.
- **One shared secret across every service.** When it leaks you have to rotate everything at once, so it never gets rotated. One credential per service and environment, always.
- **The wildcard in the OIDC trust policy.** `repo:my-org/*` turns a key-free architecture into an open door.
- **Secrets in command-line arguments.** Any process on the box sees them with `ps aux`, and they persist in shell history. Pass them via file or stdin.
- **Secrets in `docker build` arguments.** An `ARG` is baked into the image's layer history and anyone can recover it with `docker history`. Use `RUN --mount=type=secret`.
- **Reading the secret once, at import time.** It defeats every hot-rotation mechanism you built.
- **Reusing production credentials in staging.** Staging always has looser controls and more people with access; it's the preferred route into production.
- **Having no inventory.** If you can't answer "which secrets exist, who uses them, and when were they last touched", you can't rotate anything with confidence. The inventory is the prerequisite for everything else.

## Production checklist

- `pre-commit` hook with gitleaks, full-history scanning in CI, and push protection enabled on the forge.
- No static cloud keys in CI variables: OIDC federation with the `sub` condition bound to repository and branch.
- Workloads authenticating with platform identity (IRSA, Workload Identity, Managed Identity) wherever the target allows it.
- External secrets manager as the source of truth; in Kubernetes, sync via ESO and etcd encryption at rest with KMS.
- Secrets delivered as mounted files, not environment variables, and read at the point of use.
- One secret per service and environment, with least privilege, reviewed.
- Documented rotation with an overlap window and a per-key-id usage metric that makes revocation safe.
- `SecretStr` or equivalent in your settings, plus redaction in the log processor as a safety net.
- A written leak runbook, rehearsed at least once, with revocation as step one.
- A secrets inventory with owner, purpose and last rotation date.

## Where to start

If your team is at the beginning, the order that removes the most risk per unit of effort is this: turn on detection today (hooks, CI and push protection), because it's half an hour of work and it prevents the most common class of incident. Then migrate CI to OIDC, which is usually the single largest concentration of static admin keys in the whole organisation. Only then take on the secrets manager and the syncing, which is the piece with the most moving parts. Leave dynamic credentials for last: they're the ideal destination, but they require everything else to already be in place.

None of these steps requires rewriting your application. They require deciding, once, that secrets are infrastructure and not a configuration detail.