ConfigMap i Secret

Poziom: średni **Dotyczy: Kubernetes 1.30+** **Czas: ~20 min**

ConfigMap i Secret oddzielają konfigurację i dane wrażliwe od obrazu kontenera — ta strona pokazuje, jak wstrzykujemy je do poda, czym Secret realnie jest, a czym nie jest, i jak zarządzamy sekretami spoza samego Kubernetesa.

Skróty użyte na tej stronie (11)

Skrót	Co znaczy
AKS	Azure Kubernetes Service: Kubernetes w chmurze Microsoftu, w którym płaszczyznę sterowania utrzymuje dostawca
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
AWS	Amazon Web Services: chmura Amazonu
DER	binarny, nieczytelny dla człowieka format zapisu certyfikatu
EKS	Elastic Kubernetes Service: Kubernetes w chmurze Amazon Web Services, w którym płaszczyznę sterowania utrzymuje dostawca
etcd	etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft
GitOps	Git Operations: sposób prowadzenia klastra, w którym pożądany stan jest zapisany w repozytorium Git, a agent w klastrze sam doprowadza rzeczywistość do tego stanu
KMS	Key Management Service: usługa chmurowa przechowująca i zarządzająca kluczami szyfrującymi
RBAC	Role-Based Access Control: uprawnienia nadawane rolom, a role ludziom
TLS	Transport Layer Security: szyfrowanie połączeń, np. w HTTPS
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Fiszka

ZMIENIA SYSTEM

Utworzenie ConfigMap z pliku

```
kubectl create configmap NazwaUslugi-config --from-file=SciezkaDoPliku -n
NazwaPrzestrzeni
```

Tworzymy ConfigMap, którego klucz to nazwa pliku, a wartość to jego zawartość — szybki sposób bez pisania YAML-a, przydatny przy jednorazowym imporcie istniejącego pliku konfiguracyjnego.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

PLIK KONFIGURACYJNY

Utworzenie ConfigMap deklaratywnie

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: NazwaUslugi-config
  namespace: NazwaPrzestrzeni
data:
  APP_ENV: produkcja
  LOG_LEVEL: info
  app.properties: |
    timeout=30
    retries=3
```

Trzymamy zarówno pojedyncze pary klucz-wartość, jak i cały plik konfiguracyjny jako jedną wartość wieloliniową — oba warianty mogą współistnieć w jednym obiekcie.

PLIK KONFIGURACYJNY

Wstrzyknięcie ConfigMap jako zmiennych środowiskowych: envFrom

```
apiVersion: v1
kind: Pod
metadata:
  name: NazwaPoda
  namespace: NazwaPrzestrzeni
spec:
  containers:
```

```
    - name: NazwaKontenera
      image: NazwaObrazu
      envFrom:
        - configMapRef:
            name: NazwaUslugi-config
```

Wstrzykujemy wszystkie pary klucz-wartość z ConfigMap jako zmienne środowiskowe kontenera naraz, bez wymieniania ich pojedynczo.

□□ PLIK KONFIGURACYJNY

Wstrzyknięcie ConfigMap jako wolumenu z plikami

```
apiVersion: v1
kind: Pod
metadata:
  name: NazwaPoda
  namespace: NazwaPrzestrzeni
spec:
  containers:
    - name: NazwaKontenera
      image: NazwaObrazu
      volumeMounts:
        - name: config
          mountPath: /etc/NazwaUslugi
  volumes:
    - name: config
      configMap:
        name: NazwaUslugi-config
```

Każdy klucz z ConfigMap staje się osobnym plikiem w katalogu `/etc/NazwaUslugi` wewnątrz kontenera — różnica w zachowaniu przy aktualizacji względem zmiennych środowiskowych jest opisana niżej.

□□ ZMIENIA SYSTEM

Utworzenie Secret z linii poleceń

```
kubectl create secret generic NazwaSekretu --from-literal=haslo=NieWpisujHaslaWDokumentacji -n NazwaPrzestrzeni
```

Tworzymy Secret bezpośrednio z linii poleceń, żeby wartość nie trafiła do żadnego pliku YAML w repozytorium — sprawa rozwinięta w sekcji o tym, czego nigdy nie trzymamy w repozytorium.

YAML = YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Utworzenie Secret z certyfikatem TLS

```
kubectl create secret tls NazwaSekretu --cert=SciezkaDoPliku --key=SciezkaDoPliku -n NazwaPrzestrzeni
```

Tworzymy sekret typu `kubernetes.io/tls`, dokładnie w formacie, jakiego oczekuje pole `tls.secretName` obiektu Ingress opisanego na stronie Ingress i Gateway API: wejście z zewnątrz.

TLS = Transport Layer Security: szyfrowanie połączeń, np. w HTTPS · **API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podejrzenie zawartości Secret: kubectl get secret

```
kubectl get secret NazwaSekretu -n NazwaPrzestrzeni -o jsonpath='{.data.haslo}' | base64 -d
```

Odkodowujemy wartość z Secret z powrotem do postaci czytelnej — ta sama komenda pokazuje, dlaczego samo base64 nigdy nie było mechanizmem ochrony: dekodowanie nie wymaga żadnego klucza ani hasła.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Utworzenie sekretu do pobierania obrazów z prywatnego rejestru

```
kubectl create secret docker-registry NazwaSekretu --docker-server=NazwaHosta --docker-username=nazwa_usera --docker-password=NieWpisujHaslaWDokumentacji -n NazwaPrzestrzeni
```

Tworzymy sekret typu `kubernetes.io/dockerconfigjson`, który podpinamy pod pole `imagePullSecrets` poda, żeby kubelet mógł uwierzytelnić się w prywatnym rejestrze obrazów.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

PLIK KONFIGURACYJNY

Podpięcie imagePullSecrets do poda

```
apiVersion: v1
kind: Pod
metadata:
  name: NazwaPoda
  namespace: NazwaPrzestrzeni
spec:
  imagePullSecrets:
    - name: NazwaSekretu
  containers:
    - name: NazwaKontenera
      image: NazwaObrazu
```

Bez tego pola kubelet próbuje pobrać obraz z prywatnego rejestru anonimowo i kończy to błędem `ImagePullBackOff`, opisanym na stronie Diagnostyka: objaw, przyczyna, pierwsze kroki.

Z życia wzięte

Z ŻYCIA WZIĘTE

O 11:30 Tomek zmienia wartość `LOG_LEVEL` w ConfigMapie aplikacji katalogu produktów, żeby na chwilę włączyć bardziej szczegółowe logi, i czeka na efekt, patrząc na `kubectl logs -f`. Po pięciu minutach nic się nie zmienia, więc sprawdza manifest i widzi, że aplikacja czyta `LOG_LEVEL` jako zwykłą zmienną środowiskową przez `envFrom`, wstrzykiwaną raz przy starcie procesu, a nie z pliku w wolumenie, który aktualizowałby się sam. Dopiero po ręcznym

> usunięciu i odtworzeniu poda nowy proces startuje z nową wartością i szczegółowe logi faktycznie się pojawiają.

Czym jest ConfigMap i po co oddzielamy konfigurację od obrazu

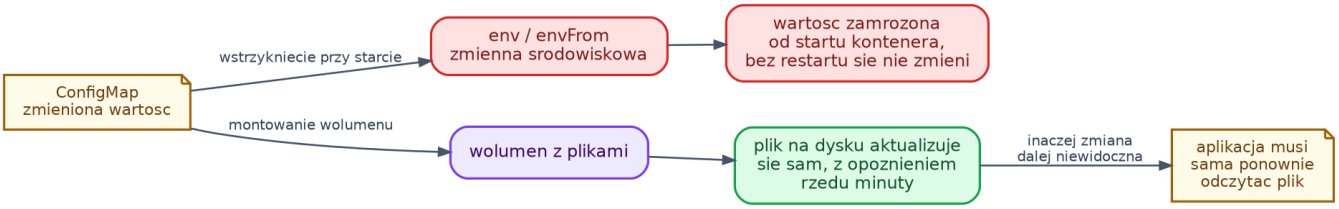
> **Pojęcie: ConfigMap.** Obiekt Kubernetesa przechowujący dowolne dane konfiguracyjne w postaci jawnego tekstu — par klucz-wartość albo całych plików — niezależnie od obrazu kontenera, który z nich korzysta.

Oddzielenie konfiguracji od obrazu ma jeden konkretny cel: ten sam obraz, zbudowany raz, trafia bez zmian na środowisko testowe i produkcyjne, różniąc się wyłącznie ConfigMapą podpiętą przy uruchomieniu. Bez tego rozdzielenia każda zmiana adresu bazy danych albo poziomu logowania wymagałaby przebudowania obrazu od nowa — a to jest dokładnie to, czego obraz kontenera jako niezmienny artefakt ma unikać.

Sposoby wstrzyknięcia i różnica przy aktualizacji

Trzy fiszki wyżej pokazują trzy drogi, którymi zawartość ConfigMap trafia do kontenera, i każda z nich zachowuje się inaczej przy zmianie danych po fakcie:

Sposób	Zachowanie przy aktualizacji ConfigMap
pojedyncza zmienna środowiskowa (`env` + `configMapKeyRef`)	wartość zamrożona na moment startu kontenera, nie odświeża się nigdy bez restartu poda
`envFrom`	tak samo jak wyżej — cały zestaw zmiennych zamrożony na starcie
wolumen z plikami	plik na dysku wewnątrz kontenera aktualizuje się automatycznie, z typowym opóźnieniem rzędu minuty, bez restartu poda



ConfigMap wstrzyknięty jako zmienna środowiskowa zamraża wartość na starcie kontenera, a wstrzyknięty jako wolumen aktualizuje się sam na dysku, choć aplikacja musi go i tak ponownie odczytać (kliknięcie otwiera pełny rozmiar)

To rozróżnienie ma bezpośrednie znaczenie praktyczne: aplikacja, która czyta konfigurację raz przy starcie procesu (typowe dla zmiennych środowiskowych) nie zobaczy zmiany nawet w wolumenie, dopóki proces się nie zrestartuje — mechanizm aktualizacji pliku na dysku nie pomaga, jeśli nikt tego pliku nie czyta ponownie. Aplikacje zaprojektowane do obserwowania zmian pliku konfiguracyjnego w locie (np. przez `inotify`) korzystają z wolumenu właśnie po to, żeby dostać nową konfigurację bez restartu.

Uważaj: to opóźnienie rzędu minuty przy aktualizacji przez wolumen wynika z okresu synchronizacji kubeletu, nie jest natychmiastowe — plan zakładający propagację zmiany „od razu” po `kubectl apply` na ConfigMapę bywa źródłem fałszywego alarmu, gdy ktoś sprawdza efekt sekundę po zmianie.

Czym jest Secret i czym nie jest

> **Pojęcie: Secret.** Obiekt Kubernetesa o tej samej strukturze co ConfigMap, przeznaczony do przechowywania danych wrażliwych — haseł, tokenów, kluczy prywatnych, certyfikatów. Różnica względem ConfigMap na poziomie samego API sprowadza się właściwie do jednego: wartości w polu `data` są zakodowane base64.

To jest najważniejsze zdanie tej strony: base64 to nie szyfrowanie. Base64 to odwracalne kodowanie binarnych danych do postaci tekstowej, bez żadnego klucza ani hasła po drugiej stronie — każdy, kto widzi zawartość obiektu Secret (przez `kubectl get secret -o yaml`, przez dostęp do etcd, przez backup etcd), odczytuje wartość jedną komendą `base64 -d`, dokładnie tak jak w fiszce wyżej. Kodowanie base64 istnieje wyłącznie po to, żeby dane binarne (np. certyfikat w formacie DER) dało się bezpiecznie zapisać w polu tekstowym JSON-a czy YAML-a, nie po to, żeby cokolwiek ukryć.

Szyfrowanie sekretów w etcd

> **Pojęcie: szyfrowanie danych w spoczynku (encryption at rest).** Mechanizm, w którym kube-apiserver szyfruje wartość obiektu Secret, zanim zapisze ją do etcd, i odszyfrowuje ją z powrotem tylko dla uprawnionego żądania przez API.

Bez tego mechanizmu Secret leży w bazie etcd w postaci zakodowanej wyłącznie base64 — czyli praktycznie jawnym tekstem dla każdego, kto dostanie dostęp do plików etcd albo do jego kopii zapasowej, opisanej na stronie Utrzymanie klastra, z pominięciem całego RBAC-u Kubernetesa. Szyfrowanie sekretów w etcd nie jest w Kubernetesie domyślnie włączone — wymaga jawnej konfiguracji `EncryptionConfiguration` na kube-apiserverze, wskazującej dostawcę szyfrowania (najprostszy to `aescbc`, zalecany produkcyjnie to zewnętrzny dostawca KMS przez `KMSv2`, w którym klucz szyfrujący nigdy nie trafia na dysk API servera). Zarządzane usługi Kubernetesa w chmurze (EKS, GKE, AKS) często oferują tę opcję jako włączaną funkcję, ale również nie zawsze domyślnie aktywną — sprawdzenie tego ustawienia jest jedną z pierwszych rzeczy, które warto zweryfikować w każdym nowym klastrze, niezależnie od tego, kto go postawił.

Zewnętrzne zarządzanie sekretami

Trzymanie sekretów bezpośrednio jako obiekty Secret w Kubernetesie ma dwa praktyczne ograniczenia: sam plik YAML z sekretem nie nadaje się do repozytorium Git (nawet zakodowany base64, z przyczyn wyżej), a rotacja hasła wymaga ręcznej aktualizacji obiektu w każdym klastrze z osobna. Dwa popularne narzędzia rozwiązują to inaczej, każde inny problem.

> **Pojęcie: External Secrets Operator.** Kontroler, który synchronizuje sekrety z zewnętrznego magazynu (HashiCorp Vault, AWS Secrets Manager, Azure Key Vault i inne) do obiektów Secret wewnątrz klastra, na podstawie obiektu `ExternalSecret` wskazującego, którego wpisu z zewnętrznego magazynu użyć. Rozwiązuje problem miejsca prawdy: hasło żyje i jest rotowane w jednym centralnym magazynie sekretów, a Kubernetes dostaje tylko jego aktualną kopię, odświeżaną automatycznie w ustalonym interwale.

```
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: NazwaSekretu
  namespace: NazwaPrzestrzeni
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: vault-backend
    kind: ClusterSecretStore
```

```
  target:
    name: NazwaSekretu
  data:
    - secretKey: haslo
      remoteRef:
        key: sklep/produkcja/baza
        property: haslo
```

> **Pojęcie: Sealed Secrets.** Mechanizm od Bitnami, który rozwiązuje inny problem — możliwość trzymania sekretu w repozytorium Git obok reszty manifestów. Sekret jawny szyfrujemy narzędziem `kubeseal` kluczem publicznym kontrolera działającego w klastrze, zapisujemy zaszyfrowany wynik jako obiekt `SealedSecret` (bezpieczny do trzymania w repozytorium, bo odszyfrować go potrafi tylko kontroler z odpowiadającym mu kluczem prywatnym), a kontroler w klastrze sam odszyfrowuje go z powrotem do zwykłego obiektu Secret.

Różnica między tymi dwoma podejściami sprowadza się do tego, gdzie leży źródło prawdy: przy External Secrets Operator jest nim zewnętrzny magazyn sekretów poza Gitem, a Sealed Secrets pozwala trzymać zaszyfrowaną wersję sekretu w tym samym repozytorium Git, w którym trzymamy resztę konfiguracji GitOps.

Czego nigdy nie trzymamy w repozytorium

- jawnego obiektu `Secret` w YAML-u, nawet zakodowanego base64 — dekodowanie nie wymaga żadnego sekretu
- hasła ani tokenu wpisanego wprost w `ConfigMap`, choćby tymczasowo „do testów”
- pliku `kubeconfig` z wbudowanym tokenem dostępu do klastra produkcyjnego
- klucza prywatnego certyfikatu TLS w żadnej postaci, nawet w gałęzi roboczej

Częste błędy

Traktowanie base64 jako ochrony. Sekret wklejony do publicznego repozytorium „bo i tak jest zakodowany” jest w praktyce jawnym hasłem — dekodowanie to jedna komenda bez żadnego klucza.

Brak włączonego szyfrowania w etcd na klastrze on-premise. Domyślna instalacja `kubeadm` nie włącza szyfrowania sekretów w etcd automatycznie — warto to sprawdzić explicité, zamiast zakładać, że Kubernetes robi to sam z siebie.

ConfigMap zamiast Secret dla wartości wrażliwej. ConfigMap nie ma żadnego mechanizmu kontroli dostępu innego niż zwykły RBAC na poziomie obiektu i nie jest traktowany przez narzędzia audytujące jako miejsce, w którym szuka się poświadczeń — hasło wpisane tam z przyzwyczajenia umyka przeglądom bezpieczeństwa skupionym na obiektach `Secret`.

□□ Ćwiczenia

> Ćwiczenia zakładają dostęp do klastra testowego z uprawnieniami do tworzenia obiektów w wybranej przestrzeni nazw.

Ćwiczenie 1 — dekodowanie Secret

```
kubectl create secret generic test-secret --from-literal=klucz=wartosc-testowa -n
NazwaPrzestrzeni
kubectl get secret test-secret -n NazwaPrzestrzeni -o jsonpath='{.data.klucz}'
kubectl get secret test-secret -n NazwaPrzestrzeni -o jsonpath='{.data.klucz}' | base64 -d
```

Tworzymy testowy Secret i odkodowujemy jego wartość bez żadnych dodatkowych uprawnień poza zwykłym odczytem obiektu.

Odpowiedź

Sama komenda `base64 -d` w pełni odtwarza oryginalną wartość — potwierdza to, że jedyną ochroną Secret jest RBAC kontrolujący dostęp do samego obiektu, a nie treść pola `data`.

Pytania kontrolne

1. Dlaczego aplikacja czytająca konfigurację ze zmiennej środowiskowej nie zauważy zmiany w ConfigMap bez restartu, mimo że plik zamontowany jako wolumen zaktualizowałby się sam?

Odpowiedź

Zmienne środowiskowe są wstrzykiwane raz, w momencie startu procesu w kontenerze, i system operacyjny nie ma mechanizmu ich odświeżenia w działającym procesie — plik na dysku aktualizuje kubelet niezależnie od procesu aplikacji, ale to i tak aplikacja musi go ponownie odczytać, żeby zobaczyć zmianę.

2. Jaki problem rozwiązuje Sealed Secrets, którego nie rozwiązuje samo szyfrowanie sekretów w etcd?

Odpowiedź

Szyfrowanie w etcd chroni dane już zapisane w klastrze przed odczytem z samego etcd, ale nie pozwala bezpiecznie trzymać sekretu w repozytorium Git przed jego wdrożeniem. Sealed Secrets rozwiązuje właśnie ten drugi problem — zaszyfrowana wersja w Gicie jest bezużyteczna bez klucza prywatnego kontrolera w konkretnym klastrze.

Powiązane: Ingress i Gateway API: wejście z zewnątrz · RBAC i bezpieczeństwo klastra · Diagnostyka: objaw, przyczyna, pierwsze kroki · Utrzymanie klastra

Źródła: dokumentacja kubernetes.io, rozdziały „Secrets", „ConfigMaps" i „Encrypting Confidential Data at Rest" · dokumentacja external-secrets.io i Bitnami Sealed Secrets (github.com/bitnami-labs/sealed-secrets)

Revision #33
Created 2026-09-25 10:33:06 UTC by Tomasz Konieczny
Updated 2026-10-26 14:17:08 UTC by Tomasz Konieczny