Bezpieczeństwo

RBAC, konta usług i kontekst bezpieczeństwa.

- RBAC i bezpieczeństwo klastra

RBAC i bezpieczeństwo klastra

Poziom: zaawansowany **Dotyczy: Kubernetes 1.30+** **Czas: ~30 min**

RBAC decyduje, kto i co wolno mu zrobić w klastrze, a kontekst bezpieczeństwa poda decyduje, co wolno procesowi wewnątrz kontenera — ta strona pokazuje oba mechanizmy razem z minimalnym zestawem praktyk, które warto włączyć od pierwszego dnia.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
PSP	Państwowa Straż Pożarna: służba, która gasi pożary, uczestniczy w odbiorach obiektów i kontroluje zabezpieczenia przeciwpożarowe
RBAC	Role-Based Access Control: uprawnienia nadawane rolom, a role ludziom
SRE	Site Reliability Engineering: sposób prowadzenia utrzymania wymyślony w Google, w którym inżynierowie oprogramowania pilnują niezawodności, z budżetem błędów i postmortemem bez szukania winnych

Fiszka

PLIK KONFIGURACYJNY

Utworzenie Role ograniczonej do jednej przestrzeni nazw

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: czytelnik-podow
  namespace: NazwaPrzestrzeni
rules:
```

```
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "watch"]
```

Definiujemy zestaw uprawnień do jednego typu zasobu, ograniczony do jednej przestrzeni nazw wskazanej w `metadata.namespace`.

□□ PLIK KONFIGURACYJNY

Powiązanie Role z podmiotem: RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: czytelnik-podow-binding
  namespace: NazwaPrzestrzeni
subjects:
  - kind: User
    name: nazwa_usera
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: czytelnik-podow
  apiGroup: rbac.authorization.k8s.io
```

Nadajemy uprawnienia z Role konkretnemu podmiotowi — sama Role bez powiązania nikomu niczego nie daje.

□□ PLIK KONFIGURACYJNY

Utworzenie ClusterRole widocznej we wszystkich przestrzeniach nazw

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: czytelnik-podow-globalny
rules:
```

```
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "watch"]
```

Definiujemy ten sam zestaw uprawnień co Role, ale bez przypisania do jednej przestrzeni nazw — sam obiekt jest zasięgiem globalnym niezależnie od tego, jak go później powiążemy.

□□ PLIK KONFIGURACYJNY

Powiązanie ClusterRole z podmiotem globalnie: ClusterRoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: czytelnik-podow-globalny-binding
subjects:
  - kind: Group
    name: zespol-sre
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: czytelnik-podow-globalny
  apiGroup: rbac.authorization.k8s.io
```

Nadajemy uprawnienia ze ClusterRole całej grupie użytkowników, we wszystkich przestrzeniach nazw klastra naraz.

□□ PLIK KONFIGURACYJNY

Utworzenie ServiceAccount

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: NazwaUslugi-sa
  namespace: NazwaPrzestrzeni
automountServiceAccountToken: false
```

Tworzymy tożsamość dla samego poda (nie dla człowieka), z jawnie wyłączonym automatycznym montowaniem tokenu — sprawa opisana niżej.

□□ TYLKO ODCZYT

Sprawdzenie własnych uprawnień: kubectl auth can-i

```
kubectl auth can-i delete pods --namespace NazwaPrzestrzeni
```

Sprawdzamy, czy bieżący użytkownik (albo konto usługi, z którego korzysta `kubectl`) ma prawo wykonać daną operację — bez wykonywania jej naprawdę.

Przykład:

```
no
```

□□ **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

□□ TYLKO ODCZYT

Sprawdzenie uprawnień innego podmiotu

```
kubectl auth can-i list secrets --namespace NazwaPrzestrzeni --
as=system:serviceaccount:NazwaPrzestrzeni:NazwaUslugi-sa
```

Sprawdzamy uprawnienia konkretnego konta usługi, tak jakbyśmy nim byli — przydatne przy diagnozowaniu, dlaczego aplikacja w podzie dostaje `Forbidden` z API Kubernetesa.

Przykład:

```
no
```

□□ **API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

□□ PLIK KONFIGURACYJNY

Ustawienie kontekstu bezpieczeństwa poda i kontenera

```
apiVersion: v1
kind: Pod
metadata:
  name: NazwaPoda
  namespace: NazwaPrzestrzeni
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 10000
    fsGroup: 10000
  containers:
    - name: NazwaKontenera
      image: NazwaObrazu
      securityContext:
        readOnlyRootFilesystem: true
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
```

Wymuszamy uruchomienie kontenera jako użytkownik nieuprzywilejowany, z systemem plików tylko do odczytu i bez żadnych dodatkowych uprawnień jądra — komplet ustawień odpowiadający poziomowi `restricted` Pod Security Admission, opisanemu niżej.

□□ ZMIENIA SYSTEM

Ustawienie poziomu Pod Security Admission dla przestrzeni nazw

```
kubectl label namespace NazwaPrzestrzeni pod-security.kubernetes.io/enforce=restricted
```

Wymuszamy, żeby żaden nowy pod w tej przestrzeni nazw nie mógł powstać, jeśli łamie zasady poziomu `restricted` — etykieta na samej przestrzeni nazw, bez żadnego dodatkowego kontrolera do zainstalowania.

□□ **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 16:00 nowy członek zespołu SRE zgłasza, że po nadaniu mu uprawnień do przestrzeni `produkcja` nadal nie może wylistować sekretów w przestrzeni `cache`. Tomek sprawdza obiekty RBAC i widzi, że ktoś powiązał `ClusterRole` z odpowiednimi uprawnieniami przez zwykły `RoleBinding` zamiast przez `ClusterRoleBinding`, więc uprawnienia zadziałały tylko w tej jednej przestrzeni nazw, w której RoleBinding został utworzony. Po dodaniu drugiego `RoleBinding` w przestrzeni `cache`, wskazującego tę samą `ClusterRole`, dostęp zaczyna działać dokładnie tam, gdzie powinien, bez nadawania niczego globalnie.

Podmiot i uprawnienie

Pojęcie: podmiot (subject). Ktoś albo coś, komu RBAC może przyznać uprawnienia: użytkownik (`User`), grupa użytkowników (`Group`) albo konto usługi (`ServiceAccount`) — jedyny typ podmiotu, który sam Kubernetes tworzy i zarządza jako własny obiekt API, w przeciwieństwie do użytkowników i grup, które Kubernetes rozpoznaje wyłącznie po tym, co przyjdzie w certyfikacie albo tokenie uwierzytelniającym, bez własnej bazy kont.

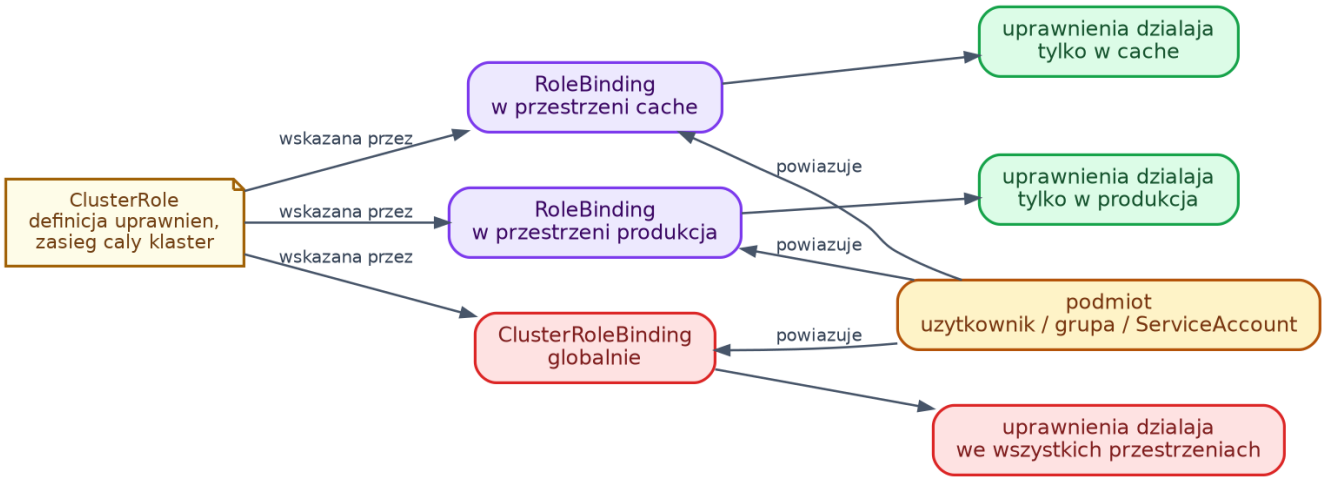
Pojęcie: uprawnienie w RBAC. Zestaw czasownika (`verb`: `get`, `list`, `watch`, `create`, `update`, `patch`, `delete`), typu zasobu (`resource`: `pods`, `secrets`, `deployments`) i grupy API (`apiGroup`), opisujący, co dokładnie wolno zrobić. RBAC działa na zasadzie białej listy — nic nie jest dozwolone, dopóki jakaś Role albo ClusterRole tego jawnie nie wymieni, a domyślnie nowy podmiot nie ma żadnych uprawnień poza tym, co dają mu wbudowane role klastra.

Role kontra ClusterRole, RoleBinding kontra ClusterRoleBinding

Obiekt	Zasięg	Z czym się wiąże
`Role`	jedna, konkretna przestrzeń nazw	tylko `RoleBinding` w tej samej przestrzeni nazw

Obiekt	Zasięg	Z czym się wiąże
`ClusterRole`	cały klaster (definicja bez przywiązania do przestrzeni nazw)	`ClusterRoleBinding` (globalnie) **albo** `RoleBinding` w jednej przestrzeni nazw (zawężenie zasięgu)
`RoleBinding`	jedna przestrzeń nazw	powiązanie podmiotu z `Role` lub z `ClusterRole` (w drugim przypadku uprawnienia z ClusterRole działają tylko w tej jednej przestrzeni)
`ClusterRoleBinding`	cały klaster	powiązanie podmiotu wyłącznie z `ClusterRole`

> **Pojęcie: ClusterRole użyta przez RoleBinding.** Wzorzec przydatny, gdy ten sam zestaw uprawnień (np. „czytaj pody, sekrety, configmapy") ma być dostępny w wielu przestrzeniach nazw, ale nie wszędzie naraz — definiujemy uprawnienia raz jako `ClusterRole`, a potem w każdej przestrzeni nazw, gdzie mają obowiązywać, tworzymy osobny `RoleBinding` wskazujący na tę samą `ClusterRole`. Unikamy w ten sposób duplikowania identycznej definicji `Role` w dziesięciu przestrzeniach nazw z osobna.



Ta sama ClusterRole wskazana przez RoleBinding działa tylko w przestrzeni tego RoleBindingu, a wskazana przez ClusterRoleBinding działa globalnie (kliknięcie otwiera pełny rozmiar)

ServiceAccount i token w podzie

> **Pojęcie: ServiceAccount.** Tożsamość, z której korzysta proces działający wewnątrz poda, gdy sam musi odpytać API Kubernetesa — np. kontroler napisany samodzielnie, który

sprawdza stan innych obiektów. Każdy pod, który nie dostał jawnie wskazanego ServiceAccount, korzysta z konta `default` w swojej przestrzeni nazw.

Domyślnie Kubernetes montuje token tego konta wewnątrz kontenera pod ścieżką `/var/run/secrets/kubernetes.io/serviceaccount/token`, niezależnie od tego, czy aplikacja w ogóle kiedykolwiek odpytuje API. To ma znaczenie dla bezpieczeństwa: kontener przejęty przez atakującego (np. przez podatność w samej aplikacji) automatycznie dostaje ten token na tacy, nawet jeśli aplikacja nigdy go nie potrzebowała.

Pojęcie: automountServiceAccountToken. Pole ustawiane na poziomie ServiceAccount albo poszczególnego poda, które wyłącza to automatyczne montowanie. Dla zdecydowanej większości aplikacji, które nie rozmawiają z API Kubernetesa (typowa aplikacja webowa, baza danych), ustawienie `automountServiceAccountToken: false` usuwa z kontenera token, który i tak nigdy nie zostałby użyty — jeden z tańszych sposobów na ograniczenie tego, co zyskuje atakujący po przejęciu jednego kontenera.

kubectl auth can-i jako narzędzie weryfikacji

`kubectl auth can-i` odpytuje API Kubernetesa dokładnie tym samym mechanizmem, który decyduje o autoryzacji prawdziwych żądań, ale bez wykonania samej operacji — odpowiedź `yes` albo `no` mówi wprost, czy dane uprawnienie istnieje, zamiast zgadywania na podstawie przeglądania obiektów `Role` i `RoleBinding` ręcznie. Flaga `--as` pozwala sprawdzić to z perspektywy dowolnego innego podmiotu, co czyni to narzędzie pierwszym krokiem przy diagnozowaniu błędów `Forbidden` zgłaszanych przez aplikację działającą pod innym kontem niż nasze własne.

Kontekst bezpieczeństwa poda i kontenera

Pojęcie: kontekst bezpieczeństwa (securityContext). Zestaw ustawień ograniczających to, co wolno procesowi wewnątrz kontenera na poziomie systemu operacyjnego, niezależny od RBAC (który kontroluje wyłącznie dostęp do API Kubernetesa, nie zachowanie procesu w kontenerze). Cztery ustawienia z Fiszki pokrywają zdecydowaną większość realnych

przypadków:

Ustawienie	Co wymusza
`runAsNonRoot`	kontener odmawia startu, jeśli obraz próbuje uruchomić proces jako `root` (UID 0)
`readOnlyRootFilesystem`	system plików kontenera montowany tylko do odczytu — proces może pisać wyłącznie do jawnie zamontowanych wolumenów
`allowPrivilegeEscalation` ustawione na `false`	proces wewnątrz kontenera nie może zdobyć więcej uprawnień niż miał na starcie, nawet przez binarki z bitem `setuid`
`capabilities.drop: [ALL]`	kontener traci wszystkie uprawnienia jądra Linuksa poza absolutnym minimum, zamiast dziedziczyć domyślny, szeroki zestaw uprawnień środowiska kontenerowego

Konsekwencja przejęcia kontenera skonfigurowanego bez żadnego z tych ustawień a kontenera z pełnym kompletem jest różnicą między procesem, który po wyjściu z aplikacji może próbować dalszej eskalacji w obrębie węzła, a procesem uwięzionym w ciasnym, jednorazowym środowisku bez żadnych dodatkowych narzędzi ataku wbudowanych w samo jądro.

Pod Security Admission i trzy poziomy

Pojęcie: Pod Security Admission (PSA). Wbudowany mechanizm kontroli admisji, obecny w Kubernetesie jako stabilny od wersji 1.25, sprawdzający przy tworzeniu każdego poda, czy jego konfiguracja spełnia jeden z trzech ustandaryzowanych poziomów bezpieczeństwa. To on zastąpił usunięty w tej samej wersji 1.25 mechanizm PodSecurityPolicy (PSP) — PSP był bardziej elastyczny, ale też znacznie trudniejszy w konfiguracji i diagnozowaniu (błędnie skonfigurowana polityka potrafiła bezgłośnie zablokować tworzenie wszystkich podów w klastrze), co było głównym powodem jego wycofania na rzecz prostszego, ustandaryzowanego modelu.

Poziom	Co dopuszcza
`privileged`	brak ograniczeń — poziom istniejący głównie dla komponentów systemowych i podów sieciowych działających na poziomie węzła
`baseline`	blokuje najbardziej znane drogi eskalacji uprawnień (tryb uprzywilejowany, dostęp do przestrzeni nazw hosta), pozostawiając swobodę w reszcie konfiguracji

Poziom	Co dopuszcza
`restricted`	wymusza aktualne dobre praktyki hardeningu poda: `runAsNonRoot`, brak eskalacji uprawnień, ograniczone `capabilities`, wymagany profil `seccomp`

Poziom ustawiamy etykietą na przestrzeni nazw, w trzech niezależnych trybach naraz: `enforce` (odrzuca pody niezgodne), `audit` (przepuszcza, ale zapisuje w logu audytu) i `warn` (przepuszcza, ale ostrzega w odpowiedzi `kubectl`) — co pozwala włączyć `warn` i `audit` na poziomie `restricted` na środowisku, które jeszcze mu nie odpowiada, zaobserwować skalę problemu, i dopiero potem przełączyć `enforce`.

```
kubectl label namespace NazwaPrzestrzeni pod-security.kubernetes.io/audit=restricted pod-security.kubernetes.io/warn=restricted
```

Minimalny zestaw praktyk od pierwszego dnia

- ustawienie `pod-security.kubernetes.io/enforce=baseline` (albo `restricted`, tam gdzie aplikacje na to pozwalają) na każdej przestrzeni nazw od momentu jej utworzenia, zamiast dokładania tego później
- `automountServiceAccountToken: false` jako wartość domyślna dla każdego ServiceAccount, z jawnym włączeniem tylko tam, gdzie aplikacja naprawdę odpytuje API
- `readOnlyRootFilesystem: true` i `capabilities.drop: [ALL]` jako punkt startowy dla każdego nowego kontenera, z jawnymi wyjątkami dopisywanymi dopiero wtedy, gdy aplikacja faktycznie tego wymaga
- osobne, wąskie `Role` zamiast pojedynczej szerokiej `ClusterRoleBinding` do wbudowanej roli `cluster-admin` dla kont usług aplikacji
- `kubectl auth can-i` jako pierwszy krok, zanim nadamy dodatkowe uprawnienia „na wszelki wypadek"

Częste błędy

Nadanie cluster-admin kontu usługi aplikacji, bo szybciej działa. Jedno przejęte konto usługi z rolą `cluster-admin` daje atakującemu pełną kontrolę nad całym klastrem, nie tylko nad jedną aplikacją — dokładnie odwrotność zasady najmniejszych uprawnień opisanej na stronie

Konta, role i uprawnienia: GRANT, REVOKE, zasada najmniejszych uprawnień.

RoleBinding wskazujący ClusterRole z przekonaniem, że to nada uprawnienia globalnie. RoleBinding zawsze ogranicza zasięg do własnej przestrzeni nazw, nawet jeśli wskazuje na ClusterRole — do globalnego nadania uprawnień potrzebny jest ClusterRoleBinding.

Ustawienie poziomu restricted na przestrzeni nazw bez wcześniejszego etapu audit/warn. Pody, które wcześniej działały bez ograniczeń bezpieczeństwa, przestają się tworzyć z dnia na dzień — bezpieczniejsza kolejność to najpierw `warn` i `audit`, obserwacja skali problemu, dopiero potem `enforce`.

□□ Ćwiczenia

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

Ćwiczenie 1 — sprawdzenie efektywnych uprawnień konta usługi

```
kubectl create serviceaccount test-sa -n NazwaPrzestrzeni
kubectl auth can-i list pods --as=system:serviceaccount:NazwaPrzestrzeni:test-sa -n
NazwaPrzestrzeni
kubectl create role test-role --verb=get,list --resource=pods -n NazwaPrzestrzeni
kubectl create rolebinding test-binding --role=test-role --
serviceaccount=NazwaPrzestrzeni:test-sa -n NazwaPrzestrzeni
kubectl auth can-i list pods --as=system:serviceaccount:NazwaPrzestrzeni:test-sa -n
NazwaPrzestrzeni
```

Tworzymy konto usługi bez żadnych uprawnień, sprawdzamy `can-i`, nadajemy uprawnienie przez Role i RoleBinding, sprawdzamy ponownie.

Odpowiedź

Pierwsze sprawdzenie zwraca `no` — świeże konto usługi nie ma żadnych uprawnień poza tym, co dają role wbudowane. Po powiązaniu z Role przez RoleBinding drugie sprawdzenie zwraca `yes`.

Pytania kontrolne

1. Dlaczego token ServiceAccount zamontowany w kontenerze jest problemem bezpieczeństwa nawet wtedy, gdy aplikacja nigdy go nie używa?

Odpowiedź

Bo w razie przejęcia kontenera przez atakującego (np. przez podatność samej aplikacji) token jest gotowy do użycia od razu, dając dostęp do API Kubernetesa z uprawnieniami konta usługi — niezależnie od tego, czy legalna aplikacja kiedykolwiek z niego korzystała.

2. Czym PSA jako mechanizm różni się fundamentalnie od usuniętego PodSecurityPolicy?

Odpowiedź

PSA ocenia zgodność poda z jednym z trzech ustandaryzowanych, wbudowanych poziomów (privileged, baseline, restricted) ustawianych etykietą na przestrzeni nazw — PodSecurityPolicy pozwalał na dowolnie złożone, własne polityki powiązane przez osobny mechanizm autoryzacji, co dawało większą elastyczność, ale też znacznie większe pole do błędu konfiguracyjnego i trudniejszą diagnozę, dlaczego dany pod nie może powstać.

Powiązane: Konta, role i uprawnienia: GRANT, REVOKE, zasada najmniejszych uprawnień · ConfigMap i Secret · Diagnostyka: objaw, przyczyna, pierwsze kroki

Źródła: dokumentacja kubernetes.io, rozdziały „Using RBAC Authorization", „Configure Service Accounts", „Pod Security Standards" i „Pod Security Admission" · kubernetes.io/blog, „Kubernetes 1.25: Pod Security Admission Controller in Stable" (sierpień 2022)