ORES
Научно-практический электронный журнал
Оригинальные исследования (ОРИС), *ISSN 2222-0402*
www.ores.su
original-research.ru

ВНЕДРЕНИЕ ДИНАМИЧЕСКОГО УПРАВЛЕНИЯ СЕКРЕТАМИ В КОНТЕЙНЕРИЗИРОВАННЫХ СРЕДАХ С ПОМОЩЬЮ VAULT AGENT

Парфенов Ярослав Евгеньевич,
студент, ФГАОУ ВО «Российский государственный университет нефти и газа (национальный исследовательский университет) имени И.М. Губкина»
Россия, г. Москва
Parfen0304@mail.ru

Полетухина София Сергеевна,
студент, ФГАОУ ВО «Российский государственный университет нефти и газа (национальный исследовательский университет) имени И.М. Губкина»
Россия, г. Москва
Sofia.Poletukhina@mail.ru

Аннотация

В данной статье рассмотрен практический подход к внедрению динамического управления секретами в контейнеризированной среде на базе ОС Альт и Kubernetes. Рассмотрена актуальность задачи, а также описана архитектура решения, его жизненный цикл от выдачи до ротации. Описан примере стенда, в котором секрет получается и обновляется без прямого взаимодействия с Vault API. Описаны преимущества и недостатки данный среды, а также предложены рекомендации для улучшения продуктивности системы.

Ключевые слова: ОС Альт, Vault Agent, Kubernetes, секреты, контейнеризированная среда, ротация секретов

IMPLEMENTING DYNAMIC SECRETS MANAGEMENT IN CONTAINERIZED ENVIRONMENTS USING VAULT AGENT

Yaroslav E. Parfenov,
student, National University of Oil and Gas «Gubkin University»
Moscow, Russia
Parfen0304@mail.ru

Sofia S. Poletukhina,
student, National University of Oil and Gas «Gubkin University»
Moscow, Russia
Sofia.Poletukhina@mail.ru

ABSTRACT

This article discusses a practical approach to implementing dynamic secrets management in a containerized environment based on Alt OS and Kubernetes. The relevance of the problem is discussed, and the solution architecture and its lifecycle from issuance to rotation are described. An example setup is presented in which a secret is retrieved and updated without direct interaction with the Vault API. The advantages and disadvantages of this environment are described, and recommendations for improving system performance are offered.

Keywords: Alt OS, Vault Agent, Kubernetes, secrets, containerized environment, secret rotation

В современном мире приложения все чаще используют контейнерные технологии, что приводит к необходимости повышения требований безопасности для передачи данных (паролей, API-ключей, токенов, сертификатов и токенов). Как правило, статические секреты приводят к утечкам, высокому времени жизни ключей и отсутствию должного контроля за их обращением, что, в свою очередь, приводит не только к финансовым, но и репутационным проблемам. Учитывая, что многие организации переходят на микросервисы, а также соблюдают меры различных регуляторов и внутренних политик информационной безопасности к минимизации площади атаки, особую актуальность приобретает внедрения динамического управления в контейнеризированных средах [1]. Внедрение динамической выдачи секретов в контейнеризированных средах демонстрирует возможность построение полнофункционального решения на базе отечественной инфраструктуры, обеспечивающего полностью автоматическую ротацию, аудит и централизованное хранение секретов.

Целью работы является разработка и экспериментальная проверка стенда динамического управления секретами в контейнеризированной среде с использованием Vault Agent, обеспечивающего автоматическую выдачу, ротацию, и аудит секретов без прямого обращения к Vault API.

Задачами работы являются:

1. Подготовка контейнеризированной среды на базе ОС Альт
2. Развертывание Vault в режиме dev внутри кластера Kubernetes
3. Настройка аутентификации подов
4. Создание хранилища секретов и определение политик доступа
5. Настройка Vault Agent для автоматической выдачи токена
6. Развертывание приложения-заглушки и обеспечение получения им секретов в режиме реального времени
7. Анализ полученных записей

Рассмотрим механизм автоматической выдачи и ротации секретов с помощью Vault Agent. Этот подход особенно удобен, так как его можно легко развернуть на учебных стендах для тренировки, а уже потом переносить в полноценную инфраструктуру [2].

На данный момент существует огромное количество традиционных подходов, в том числе и статические секреты, которые очень быстро устаревают, а самое главное - очень плохо контролируются. Однако существует динамический подход для решения поставленной задачи. Он предполагает наличие временных учетных данных, минимальное доверие к среде выполнения, упрощенную разработку, а также высокую степень наблюдаемости и аудита [4]. Особенно интерес к данному методу усиливается в контексте ОС Альт, так как он демонстрирует отечественную базу в роли полноценной платформы для безопасной работы.

Для начала внедрения динамического управления секретами в контейнеризованной среде необходимо подготовить базовую среду на ОС Альт. Для этого необходимо установить утилиты для скачивания бинарников и вызова API, docker, kubectl, kind (Kubernetes in Docker) [5].

Далее будут приведены конкретные способы установки необходимых элементов системы.

Для установки Docker в ОС Альт используем стандартные пакеты из официального репозитория. Следующие команды позволяют установить Docker, а также запустить службу и добавить ее в автозагрузку. Также проверим версию установленного пакета:

Листинг 1 - Установка Docker в ОС Альт

```
apt-get update
apt-get install -y docker docker-engine docker-cli containerd
systemctl enable --now docker
```

Листинг 2 - Проверка версии Docker и информации о конфигурации

```
docker --version
docker info | egrep 'Server Version|Storage Driver|Cgroup'
```

Листинг 3 - Установка kubectl и проверка установленного пакета:

```
#curl -L -o /usr/local/bin/kubectl \
"https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl"
#chmod +x /usr/local/bin/kubectl
#kubectl version --client
```

Листинг 4 - Установка и проверка kind(Kubernetes in Docker)

```
#curl -L -o /usr/local/bin/kind "https://kind.sigs.k8s.io/dl/v0.23.0/kind-linux-amd64"
#chmod +x /usr/local/bin/kind
#kind --version
```

Листинг 5 - Установка и проверка helm

```
#curl -fsSL https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
#helm version
```

Далее необходимо создать локальный кластер. Наиболее удобным вариантом является создание именно локального кластера, так как это не требует аренды облачных хранилищ или больших временных затрат на развертывание. Для упрощения изоляции и более удобного управления ролями необходимо создать локальный кластер Kubernetes, версия узла которого v1.31.0. Благодаря данному решению мы быстро можем быстро проверять доставку секретов. При создании кластера kind(Kubernetes in Docker) мы сразу настраиваем систему так, что каждый Pod получает свой IP, тем самым мы сразу распределяем объекты по их логическим местам: namespace Vault для сервера Vault и namespace app для самого приложения, что также упрощает работу со средой, помогая изолировать политики и права друг от друга [3].

Листинг 6 - Создание кластера Kubernetes с помощью kind

```
kind create cluster --name vaultlab --image kindest/node:v1.31.0
kubectl config use-context kind-vaultlab
kubectl get nodes -o wide
```

Листинг 7 - Создание namespace vault и namespace app

```
kubectl create namespace vault || true
kubectl create namespace app  || true
kubectl get ns
```

Затем необходимо развернуть HashiCorp Vault в dev-режиме внутри Kubernetes. Это довольно удобно и быстро для данного способа. Для настройки выдается хранилище в памяти, фиксированный root-токен и готовый API. Также необходимо заранее опубликовать сервис, чтобы приложение из кластера напрямую к нему обращалось. Также необходимо настроить порт-форвардинг для удобного доступа с хоста и проверки через браузер.

Листинг 8 - YAML Deployment и Service для Vault

```
cat <<'YAML' | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vault
  namespace: vault
spec:
  replicas: 1
  selector:
    matchLabels: { app: vault }
  template:
    metadata:
      labels: { app: vault }
    spec:
      serviceAccountName: default
      volumes:
        - name: vault-logs
          emptyDir: {}
      containers:
        - name: vault
          image: hashicorp/vault:1.16.3
          args: ["server","-dev","-dev-listen-address=0.0.0.0:8200"]
          env:
            - name: VAULT_DEV_ROOT_TOKEN_ID
              value: "root"
          ports:
            - name: http
              containerPort: 8200
          readinessProbe:
            httpGet: { path: /v1/sys/health, port: 8200 }
            initialDelaySeconds: 2
            periodSeconds: 2
          volumeMounts:
            - name: vault-logs
              mountPath: /vault/logs
---
apiVersion: v1
kind: Service
metadata:
  name: vault
  namespace: vault
spec:
```

```
  selector: { app: vault }
  ports:
    - name: http
      port: 8200
      targetPort: 8200
YAML
```

Листинг 9 - Команды проверки подов Vault

```
kubectl -n vault rollout status deploy/vault
kubectl -n vault get pods -o wide
```

Листинг 10 - Запуск port-forward

```
[ -f /tmp/vault_pf.pid ] && kill $(cat /tmp/vault_pf.pid) 2>/dev/null || true
kubectl -n vault port-forward svc/vault 8200:8200 >/tmp/vault_pf.log 2>&1 & echo $! >/tmp/Vault_pf.pid
```

Листинг 11 - Проверка статуса Vault API

```
export vault_ADDR="http://127.0.0.1:8200"
export vault_TOKEN="root"
curl -sS -o /dev/null -w "%{http_code}\n" "$vault_ADDR/v1/sys/health"
```

Далее переходим к настройке аутентификации подов в Vault. Для этого необходимо включить auth/kubernetes и дать право Vault проверять токен через Kubernetes TokenReview API. Чтобы это настроить, необходимо создать ServiceAccount Vault-auth в ns Vault и ClusterRoleBinding к роли system:auth-delegator.Затем все собранные и необходимые параметры передаются в Vault. Для реализации данного этапа необходимо сделать следующее:

Листинг 12 - Настройка Kubernetes-аутентификации в Vault

```
kubectl -n vault create serviceaccount vault-auth || true
kubectl create clusterrolebinding vault-auth-delegator \
  --clusterrole=system:auth-delegator \
  --serviceaccount=vault:vault-auth 2>/dev/null || true
SA_JWT="$(kubectl -n vault create token Vault-auth)"
KUBE_CA="$(kubectl -n kube-system get cm kube-root-ca.crt -o jsonpath='{.data.ca\.crt}')"
KUBE_HOST="https://kubernetes.default.svc:443"
curl -sS -H "X-Vault-Token: $Vault_TOKEN" \
  -X POST -d '{"type":"kubernetes"}' \
  "$Vault_ADDR/v1/sys/auth/kubernetes"
jq -n --arg token "$SA_JWT" --arg host "$KUBE_HOST" --arg ca "$KUBE_CA" \
  '{token_reviewer_jwt:$token, kubernetes_host:$host, kubernetes_ca_cert:$ca}' \
  > /tmp/k8s-auth.json
curl -sS -H "X-Vault-Token: $VAULT_TOKEN" -H "Content-Type: application/json" \
  -X POST --data-binary @/tmp/k8s-auth.json \
  "$Vault_ADDR/v1/auth/kubernetes/config"
```

Настраиваем само хранилище секретов(KV v2). Для этого размещаем там первичный секрет и описываем политику доступа, которая включает в себя только чтение записей, чтобы приложение самостоятельно не могло менять или удалять секреты. Это является одним из принципов минимально необходимого доступа.

Листинг 13 - Создание хранилища KV v2 и первичного секрета

```
curl -sS -H "X-Vault-Token: $VAULT_TOKEN" \
  -X POST -d '{"type":"kv-v2"}' \
  "$Vault_ADDR/v1/sys/mounts/secret"
curl -sS -H "X-Vault-Token: $VAULT_TOKEN" -H "Content-Type: application/json" \
```

```
  -X POST -d '{"data":{"username":"demo","password":"s3cr3t"}}' \
  "$VAULT_ADDR/v1/secret/data/app/config"
```

Листинг 14 - Настройка app-read в формате HCL:

```
cat > /tmp/app-read.hcl <<'HCL'
path "secret/data/app/*" {
  capabilities = ["read"]
}
HCL
```

Листинг 15 - Загрузка политики app-read в Vault

```
jq -Rs '{policy: .}' /tmp/app-read.hcl > /tmp/app-read.json
curl -sS -H "X-Vault-Token: $VAULT_TOKEN" -H "Content-Type: application/json" \
  --request PUT --data-binary @/tmp/app-read.json \
  "$VAULT_ADDR/v1/sys/policies/acl/app-read"
```

Для того чтобы политика работала корректно мы используем auth/kubernetes, в которой создаем роль app, где описываем то, что если в Vault появляется сервис из namespace app, то ему необходимо выдать токен с вышеуказанной политикой и TTL на 24 часа. Благодаря данным настройкам мы не позволяем любому другому сервису самостоятельно получить доступ к Vault.

Листинг 16 - Создание роли

```
curl -sS -H "X-Vault-Token: $VAULT_TOKEN" -H "Content-Type: application/json" \
  -X POST \
  -d '{"bound_service_account_names":"app-sa","bound_service_account_namespaces":"app","policies":"app-read","ttl":"24h"}' \
  "$VAULT_ADDR/v1/auth/kubernetes/role/app"
```

В результате мы получаем, что приложение просто читает необходимый ему файл, но при этом никак не взаимодействует с Vault напрямую. Как только секрет меняется, агент сообщает об этом приложению, которое перечитывает необходимое ему значение без перезапуска и лишних обращений к самому агенту.

Далее переходим к настройке самого пространства app, в котором создаются базовые роли и привязки, помогающие нам следить и участвовать в отладке пода. Заметим, что конкретно данный сервис никак не связан с выдачей секрета.

Листинг 17 - ServiceAccount и права доступа приложения в пространстве имен

```
cat <<'YAML' | kubectl apply -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-sa
  namespace: app
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: app-read
  namespace: app
rules:
  - apiGroups: [""]
    resources: ["pods","configmaps"]
    verbs: ["get","list","watch"]
---
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-read-binding
  namespace: app
subjects:
  - kind: ServiceAccount
    name: app-sa
    namespace: app
roleRef:
  kind: Role
  name: app-read
  apiGroup: rbac.authorization.k8s.io
YAML
```

Как уже говорилось ранее, для того, чтобы клиент успешно брал краткоживущий токен, необходимо прописать правила в общий том /vault/secrets.

Листинг 18 - ConfigMap с конфигом Vault agent

```
cat <<'YAML' | kubectl apply -f -
apiVersion: v1
kind: ConfigMap
metadata:
  name: vault-agent-config
  namespace: app
data:
  agent.hcl: |
    exit_after_auth = false
    pid_file = "/tmp/vault-agent.pid"

    auto_auth {
      method "kubernetes" {
        mount_path = "auth/kubernetes"
        config = { role = "app" }
      }
      sink "file" {
        config = { path = "/vault/secrets/token" }
      }
    }

    template {
      source      = "/vault/config/template.tpl"
      destination = "/vault/secrets/app-config.json"
      wait { min = "2s", max = "5s" }
    }
    template {
      source      = "/vault/config/template.env.tpl"
      destination = "/vault/secrets/app.env"
      wait { min = "2s", max = "5s" }
    }
    vault {
```

```
      address = "http://vault.vault.svc:8200"
    }
  template.tpl: |
    {{- with secret "secret/data/app/config" -}}
{"username":"{{ .Data.data.username }}","password":"{{ .Data.data.password }}"}
    {{- end -}}
  template.env.tpl: |
    {{- with secret "secret/data/app/config" -}}
    USERNAME={{ .Data.data.username }}
    PASSWORD={{ .Data.data.password }}
{{- end -}}
YAML
```

Для успешной работы приложения используются два контейнера. Первый - приложение-заглушка, которое каждые 10 секунд читает файл /vault/secrets/app.env и выводит в логи текущее значение логина и пароля. Благодаря этому, при изменении данных мы сразу можем увидеть это в логах. Второй контейнер - Vault Agent, который аутентифицируется в Vault и генерирует секрет в файл.

Листинг 19 - Повторное размещение ConfigMap

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app
  template:
    metadata:
      labels:
        app: app
    spec:
      serviceAccountName: app-sa
      volumes:
        - name: vault-secrets
          emptyDir: {}
        - name: vault-agent-config
          configMap:
            name: vault-agent-config
      containers:
        - name: app
          image: alpine:3.18
          command: ["/bin/sh", "-c"]
          args:
            - |
              echo "[app] start"
              while true; do
                echo "[app] current secret:"
```

```
          cat /vault/secrets/app.env 2>/dev/null || echo "no secret yet"
          sleep 10
        done
    volumeMounts:
     - name: vault-secrets
       mountPath: /vault/secrets
   - name: vault-agent
     image: hashicorp/vault:1.16.3
     args: ["agent", "-config=/vault/config/agent.hcl"]
     volumeMounts:
      - name: vault-agent-config
        mountPath: /vault/config
        readOnly: true
      - name: vault-secrets
        mountPath: /vault/secrets
```

Листинг 20 - Проверка запуска Deployment приложения app

kubectl -n app rollout status deploy/app

В завершении была проведена проверка работы Vault -> Vault Agent -> приложение-заглушка. Для этого используется команда kubectl -n app exec "$POD" -c app -- sh -lc 'ls -l /vault/secrets; echo "---"; cat /vault/secrets/app.env || true', которая позволяет увидеть два файла: token и app.env, в котором на момент проверки мы и видим текущие значения полей USERNAME и PASSWORD, что подтвержает успешное получение секрета из Vault. Также ниже мы можем видеть, что приложение-заглушка действительно каждый 10 секунд выводит в логи содержимое файла с секретами не обращаясь к Vault напрямую.

```
kubectl -n app exec "$POD" -c app -- sh -lc 'ls -l /vault/secrets; echo "---"; cat /vault/secr
ets/app.env || true'

# и смотрим, как приложение каждые 10 секунд печатает логин/пароль
kubectl -n app logs -f "$POD" -c app
total 8
-rw-r--r--    1 100      1000           28 Nov 21 11:37 app.env
-rw-r-----    1 100      1000           95 Nov 21 11:37 token
---
USERNAME=test2
PASSWORD=111
[app] started, polling every 10s
[2025-11-21T11:37:26+00:00] waiting for /vault/secrets/app.env
[2025-11-21T11:37:36+00:00] username=test2 password=111
[2025-11-21T11:37:46+00:00] username=test2 password=111
[2025-11-21T11:37:56+00:00] username=test2 password=111
[2025-11-21T11:38:06+00:00] username=test2 password=111
[2025-11-21T11:38:16+00:00] username=test2 password=111
[2025-11-21T11:38:26+00:00] username=test2 password=111
[2025-11-21T11:38:36+00:00] username=test2 password=111
```

Рисунок 1 - Проверка работы системы

Далее мы проверяем реакцию системы на изменение секрета. Для этого внутри pod`a Vault проводим данные изменения путем выполнения команды export VAULT_ADDR="http://127.0.0.1:8200"; export VAULT_TOKEN="root" vault kv put secret/app/config username="test2" password="111", но с другими значениями полей username и password

```
[root@host-15 ~]# kubectl -n vault exec -it "$VPOD" -- sh -lc '
export VAULT_ADDR="http://127.0.0.1:8200"
export VAULT_TOKEN="root"
vault kv put secret/app/config username="test2" password="111"
'
===== Secret Path =====
secret/data/app/config

======= Metadata =======
Key                Value
---                -----
created_time       2025-11-21T11:39:30.754655375Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            5
[root@host-15 ~]#
```

Рисунок 2 - Изменение секрета

Затем необходимо посмотреть актуальные логи и убедиться, что секрет действительно обновился и мы можем видеть эти обновления в режиме реального времени

```
[2025-11-21T11:41:37+00:00] username=test2 password=111
[2025-11-21T11:41:47+00:00] username=test2 password=111
[2025-11-21T11:41:57+00:00] username=test2 password=111
[2025-11-21T11:42:07+00:00] username=test2 password=111
[2025-11-21T11:42:17+00:00] username=test3 password=5452
```

Рисунок 3 - Проверка обновления секрета

Мы видим, что после изменения данных в Vault, приложение не перезапускается. Необходимо лишь дождаться очередного опроса приложением-заглушкой, чтобы увидеть, что данные действительно изменились и Vault Agent автоматически получил новую версию секрета и перегенерировал файл /vault/secrets/app.env.

Отдельно проверяется работа аудита Vault. Для этого на сервере секретов был включен файловый аудит с помощью команды vault audit enable file file_path=/vault/audit.log. После этого все обращения к Vault, чтение секрета и аутентификация по Kubernetes начинают записываться в указанный файл в JSON формате.


```
{"auth":{"accessor":"hmac-sha256:a7630ea00ca7d95bd2bb884b6be38996a57528e2376a8c24db
4388d21c27ef23","client_token":"hmac-sha256:2d58b12e48a23c1a583c287ee579aff97c78f57
90fa967d0f2d5fc119fc6404f","display_name":"kubernetes-app-app-sa","entity_id":"4ba3
cc39-54a0-964e-fefc-8802de93c672","metadata":{"role":"app","service_account_name":"
app-sa","service_account_namespace":"app","service_account_secret_name":"","service
_account_uid":"bcbda0ab-fa33-409e-8a12-64ab8dcd5639"},"policies":["app-read","defau
lt"],"policy_results":{"allowed":true,"granting_policies":[{"name":"app-read","name
space_id":"root","type":"acl"}]},"token_policies":["app-read","default"],"token_iss
ue_time":"2025-10-30T21:28:27Z","token_ttl":86400,"token_type":"service"},"time":"2
025-10-30T21:37:19.523986192Z","type":"response","request":{"client_id":"4ba3cc39-5
4a0-964e-fefc-8802de93c672","client_token":"hmac-sha256:f6a017a1154830545aa9c7659d1
1eedb028f70071947ccd61da62f1e8f352fcb","client_token_accessor":"hmac-sha256:a7630ea
00ca7d95bd2bb884b6be38996a57528e2376a8c24db4388d21c27ef23","id":"d5eb04b6-c08c-7d09
-2600-12955b8c7142","mount_accessor":"kv_ce417d9b","mount_class":"secret","mount_po
int":"secret/","mount_type":"kv","mount_running_version":"v0.17.0+builtin","namespa
ce":{"id":"root"},"operation":"read","path":"secret/data/app/config","remote_addres
s":"10.244.0.24","remote_port":55400},"response":{"data":{"data":{"password":"hmac-
sha256:c3958de477a139916c5b28ca93dcd5c855d809adbd4bd1ca4c6a8c72afb11a4d","username"
:"hmac-sha256:5f2d580c52b46b6509abaaaf64bda0bd23cc523517f186fc290e4aad22314b8f"},"m
etadata":{"created_time":"hmac-sha256:21d51a0f5d52d87948495bb80790b4fc3f1f263c05002
357044deec2b1fbaea7","custom_metadata":null,"deletion_time":"hmac-sha256:a459f99dc5
f3b021b81710cc521b6807d29262c2637166a24f4e782c9c2e8cfb","destroyed":false,"version"
:12}},"mount_accessor":"kv_ce417d9b","mount_class":"secret","mount_point":"secret/"
,"mount_running_plugin_version":"v0.17.0+builtin","mount_type":"kv"}}
```

Рисунок 4 - Аудит

Тут мы видим, что запрос пришёл через backend kubernetes, с ролью app, а в метаданных отражены имя сервисного аккаунта app-sa и namespace app, что демонстрирует привязку доступа к нужному сервису в кластере. В блоке request указано operation":"read" и path":"secret/data/app/config", то есть мы видим, что чтение секрета зафиксировано по ожидаемому пути, вместе с адресом источника запроса. В блоке response присутствуют данные секрета, а именно введенные ранее значения username и password в зашифрованном виде, а также номер версии. Просмотр последних записей выполнялся командой вида kubectl -n vault exec "$VPOD" -- tail -n 5 /vault/audit.log. Наличие таких записей подтверждает, что каждая выдача секрета через Vault Agent контролируется и может быть проанализирована в случае инцидента, а сама схема динамического управления секретами не только работает функционально, но и удовлетворяет требованиям наблюдаемости и безопасности.

Вывод

Проведенная работа подтвердила то, что использование динамического управления секретами в контейнеризированных средах без прямого обращения к Vault API достаточно эффективно, так как снижает сложность разработкии уменьшает поверхность атаки.

Автоматическая ротация секретов показала значительное преимущество перед статическим хранением. Изменение секрета в Vault привело к автоматическому обновлению файла без перезапуска приложения и самого контейнера. Данный подход обеспечивает непрерывность работы сервисов и минимизирует риск устаревания ключей.

Файловый аудит позволил зафиксировать все операции по чтению и аутентификации секретов, что позволяет повысить уровень наблюдаемости за секретами.

Разработанный стенд показал, что отечественные решения, такие как ОС Альт и Kubernetes позволяют строить современные и безопасные платформы, удовлетворяющие требованиям динамической ротации, изоляции и аудиту секретов. Также стоит отметить, что данная система подлежит расширению и улучшению в будущем.

Список литературы:

1. Уймин А. Г. Разработка методики тестирования системы безопасности автоматизированных систем управления технологическими процессами на основе корпоративного стандарта // Автоматизация и информатизация ТЭК. 2024. № 5 (610). С. 59-65.
2. Конаков П. О., Смоленцева Т. Е. Разработка архитектуры системы управления информационными ресурсами в рамках модели хранилища данных DATA VAULT 2.0 // Стольпинский вестник. 2022. №9. С. 4852-4870.
3. Лазарева Н. Б., Ловцова Н. Н. Автоматизация получения доступа к базам данных для компонентов в среде Kubernetes // ИВД. 2021. №3 (75). С. 28-36.
4. Самородских И. Л. Системы управление секретами // E-Scio. 2020. №3 (42). С. 267-281.
5. Липатова С. Е., Федоров В. О., Белов Ю. С. Kubernetes как элемент человеко-компьютерного взаимодействия // E-Scio. 2023. №1 (76). С.49-55.

References:

1. Uimin A. G. Development of a methodology for testing the security system of automated process control systems based on a corporate standard // Automation and informatization of the fuel and energy complex. 2024. No. 5 (610). pp. 59-65.

2. Konakov P. O., Smolentseva T. E. Development of the architecture of an information resource management system within the framework of the DATA VAULT 2.0 data warehouse model // Stolypinsky Vestnik. 2022. No. 9. pp. 4852-4870.
3. Lazareva N. B., Lovtsova N. N. Automation of access to databases for components in the Kubernetes environment // IVD. 2021. No. 3 (75). P.28-36.
4. Samorodskikh I. L. Secrets Management Systems // E-Scio. 2020. No. 3 (42). Pp. 267-281.
5. Lipatova S. E., Fedorov V. O., Belov Yu. S. Kubernetes as an Element of Human-Computer Interaction // E-Scio. 2023. No. 1 (76). Pp. 49-55.