УДК 004.056.55

ФРЕЙМВОРК ЗАХИСТУ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ З ВИКОРИСТАННЯМ СТРАТЕГІЇ ZERO TRUST

М. О. Гранік[1,a], О. В. Козленко [1]

[1] Навчально-науковий Фізико-технічний інститут

Анотація

У роботі пропонується концептуальний фреймворк захисту мікросервісної архітектури на основі принципів Zero Trust (моделі «нульової довіри»). Запропонований підхід поєднує інструменти керування ідентифікацією та доступом для кожного сервісу, мережеву сегментацію і шифрування трафіку на рівні service mesh, а також механізми забезпечення цілісності програмного забезпечення. Така багаторівнева модель безпеки покликана усунути притаманні мікросервісам вразливості шляхом тотальної перевірки довірених сторін і артефактів. У тезах розглянуто складові фреймворку (SPIFFE/SPIRE, OPA, Istio, Sigstore, SLSA) та їхню взаємодію для досягнення принципів Zero Trust. Наведено очікувані переваги від впровадження запропонованого рішення.

Ключові слова: Zero Trust, мікросервісна архітектура, ідентифікація, політики доступу, ланцюжок постачання ПЗ.

Вступ

Перехід від монолітів до мікросервісів ускладнив кібербезпеку. Традиційний захист периметру вже не працює в хмарних середовищах. Через розподілену архітектуру межі мережі стали розмитими, і наявність користувача або ресурсу всередині мережі не гарантує безпеку. Раніше внутрішній користувач вважався довіреним, але зараз це може призвести до атаки.

Концепція Zero Trust («ніколи не довіряй, завжди перевіряй»), запропонована для захисту сучасних розподілених систем [1], передбачає захищене з'єднання для всіх ресурсів незалежно від мережі, динамічний контроль доступу на основі контекстних політик, мінімальні права доступу для кожної сесії та постійний моніторинг стану системи.

1. Теоретична основа

Крупні IT-компанії переходять від **монолітів** до **мікросервісних** архітектур, що забезпечують гнучкість, масштабованість і надійність. Однак цей підхід породжує нові загрози кібербезпеки через динамічність розміщення компонентів, мережеву взаємодію між сервісами та інші архітектурні особливості. Це вимагає комплексного захисту (defense-in-depth) на всіх рівнях: коду, контейнерів, мережі та сервісів.

1.1. Принцип Zero Trust і його застосування

Концепція **Zero Trust** була вперше запропонована як нова модель кібербезпеки, що відповідає потребам розподілених систем та хмарних інфраструктур. На відміну від традиційного принципу "довіряй, але перевіряй", модель Zero Trust сповідує протилежний підхід: **"не довіряй, доки не перевіриш"**. Жодному суб'єкту або компоненту системи не надається довіра за замовчуванням – кожна спроба доступу повинна бути перевірена, незалежно від того, здійснюється вона зсередини корпоративної мережі чи ззовні. Умовно кажучи, Zero Trust виходить з припущення, що компрометація вже сталась (або може статись у будь-який момент), тому необхідно будувати систему так, ніби атакувальник вже присутній усередині.

Основні принципи Zero Trust можна сформулювати так [1]:

- Усі ресурси (сервіси, дані, обчислювальні вузли) розглядаються як потенційно відкриті для доступу, тому потребують захисту.
- Шифрування та захист застосовуються до всіх зєднань незалежно від місцезнаходження учасників взаємодії. Весь мережний трафік (як зовнішній, так і внутрішній між сервісами) має бути конфіденційним і цілісним.
- Автентифікація та авторизація виконуються для кожного запиту або сеансу доступу окремо (**granular per-session access**). Ніякого "постійного довіреного підключення" без повторної перевірки бути не повинно.
- Рішення про надання доступу ухвалюються на основі динамічних політик, **що враховують контекст** –- атрибути ідентичності клієнта (користувача або сервісу), стан його середовища, тип запитуваного ресурсу, інші поведінкові та середовищні ознаки. Політики побудовані за принципом найменших привілеїв: за замовчуванням

[a]mykgran-ipt25@lll.kpi.ua

доступ закрито (**deny by default**), дозволяється лише явно визначені дії.
- Стан усіх пристроїв і компонентів постійно моніториться і оцінюється.
- Довіреність є **динамічною**: автентифікація і авторизація здійснюються кожного разу при зверненні до ресурсу.

2. Розробка методології захисту мікросервісної архітектури

На основі розглянутих принципів та інструментів, розроблено комплексну методологію впровадження Zero Trust для мікросервісної архітектури в середовищі Kubernetes. Ця методологія охоплює кілька рівнів захисту, які впроваджуються послідовно по ходу життєвого циклу сервісу. Коротко ці рівні можна окреслити як:

1. Захист коду та збірки.
2. Безпечне розгортання.
3. Захист під час виконання (run-time).
4. Моніторинг і реагування.

2.1. Безпечна розробка і збірка (Secure SDLC)

На першому етапі, під час розробки мікросервісу, застосовуються практики DevSecOps для забезпечення початкової якості та безпеки коду. Всі вихідні репозиторії проходять перевірку на наявність відомих уразливостей у залежностях (засобами SCA – Software Composition Analysis) та статичний аналіз коду (SAST) для виявлення потенційно небезпечних конструкцій. В конвеєр CI/CD інтегруються інструменти аналізу, результати яких розробники зобов'язані врахувати перед злиттям коду. Таким чином, ще до побудови бінарних артефактів зменшується кількість вбудованих вразливостей. Після проходження тестів і перевірок код збирається у контейнерний образ. На цьому етапі в гру вступає захист ланцюжка поставок: згенерований Docker/OCI-образ підписується розробниками або CI-системою за допомогою Sigstore/cosign. Підпис містить криптографічний хеш образу, завірений приватним ключем (який може зберігатися у безпечному сховищі, наприклад HashiCorp Vault, або використовувати механізми ключів KMS хмарного провайдера). Відкритий ключ чи сертифікат, яким здійснено підпис, публікується у реєстрі Sigstore (Transparency Log) для подальшої верифікації. Одночасно можуть зберігатися додаткові метадані – інформація про збірку (так звана attestation, згідно з рівнями SLSA), які пізніше дозволять переконатися, що образ зібрано офіційним конвеєром CI, а не вручну. Підсумком цього етапу є те, що ми маємо контейнерний образ, який пройшов перевірки і має доказ легітимності у вигляді криптографічного підпису. Далі, при передачі цього образу у наступні середовища (staging, production), на кожному кроці перевіряється його цілісність та джерело походження.

2.2. Безпечне розгортання і оркестрація

При розгортанні мікросервісів у Kubernetes-кластері, методологія передбачає увімкнення декількох рівнів контролю, що відповідають принципам Zero Trust ще до запуску контейнера:

- Верифікація підписів образів: У кластері розгорнуто політику **OPA Gatekeeper** (або альтернативний admission-controller), який перехоплює запити на створення Pod. Ця політика перевіряє через **Sigstore** (Transparency Log та сертифікати) наявність дійсного підпису для кожного контейнерного образу, зазначеного у Pod.
- Політики конфігурації та ізоляції: Використовуючи OPA Gatekeeper, Kubernetes Policy Controller або інші інструменти, впроваджуються політики на рівні ресурсів кластера.
- Інтеграція **SPIRE** для атестації: На всіх вузлах Kubernetes запущені SPIRE Agent, а в кластері – SPIRE Server.

2.3. Захист під час виконання: автентифікація та авторизація запитів

Коли запущені сервіси починають взаємодіяти між собою та з зовнішніми клієнтами, вступають у дію механізми run-time захисту, які реалізують принцип «постійної перевірки» на кожному мережевому запиті.

Взаємна TLS між усіма сервісами забезпечується через Service Mesh (Istio + Envoy) і SPIRE сертифікати. Це дозволяє автоматично шифрувати всі запити між мікросервісами [2].

Авторизація на основі політик реалізується через OPA (Open Policy Agent). Кожен Envoy-проксі налаштований з external auth фільтром, що звертається до OPA для перевірки запитів.

OPA працює локально поруч із сервісом, що мінімізує затримки та підвищує відмовостійкість, оскільки відсутня єдина точка відмови [3].

Крім того, Istio забезпечує додаткові мережеві політики контролю, наприклад, глобальну політику "deny all"для блокування всього трафіку без явного дозволу [4].

2.4. Моніторинг, аналіз та реагування

Завершальним елементом методології є постійний моніторинг і аналітика для підтримки актуальності Zero Trust моделі та своєчасної реакції на інциденти:

- Централізоване логування і трасування: логи доступу з Envoy-проксі агрегуються в системах, як Elasticsearch або Loki, для подальшого аналізу. Використовуються метрики через Prometheus і трасування через Jaeger/Zipkin для моніторингу взаємодії сервісів, налаштовуються дашборди Grafana та алерти.
- Контроль цілісності і оновлення політик: регулярне оновлення сертифікатів SPIRE, контроль політик OPA через систему версій, що забезпечує актуальність політик відповідно до загроз.

- Аудит і відповідність: повне логування дозволяє здійснювати ретроспективний аналіз інцидентів, спрощує розслідування та вдосконалення політик, а також відповідає вимогам безпеки та аудиту.
- Динамічне реагування: система може ізолювати підозрілі сервіси через мікросегментацію, автоматично додаючи політику ізоляції в разі аномальної активності, що мінімізує вплив на решту системи.

Загалом, моніторинг і реакція є невід'ємною частиною Zero Trust, оскільки, за словами однієї з тез NIST, "потрібно збирати якомога більше інформації про стан системи і використовувати її для покращення безпеки".

2.5. Приклад впровадження SSDLC в рамках фреймворку

Для демонстрації процедури впровадження запропонованого фреймворку, виконаємо впровадження одного з етапів – Secure Software Development Lifecycle, або SSDLC. Для того, щоб це реалізувати, використаємо наступні інструменти:

- Gitlab CI;
- Trivy;
- Cosign;

Для демонстрації, було написано простий додаток на `Golang`, що виступає в ролі веб-серверу, що приймає запити від фронтенду та виконує запити до бази даних.

```
package main

import (
    "database/sql"
    "fmt"
    "log"
    "net/http"
    "os"
    _ "github.com/lib/pq"
)

func main() {
    connStr := os.Getenv("DB_COMM_STRING")
    if connStr == "" {
        log.Fatal(
            "Missing DB_COMM_STRING environment variable",
        )
    }

    db, err := sql.Open(
        "postgres",
        connStr,
    )
    if err != nil {
        log.Fatal(
            "Failed to connect to DB:",
            err,
        )
    }

    http.HandleFunc(
        "/api/message",
        func(w http.ResponseWriter, r *http.Request) {
            var message string
            err := db.QueryRow(
                "SELECT content FROM messages LIMIT 1",
            ).Scan(&message)

            if err != nil {
                http.Error(
                    w,
                    "DB error",
                    http.StatusInternalServerError,
                )
                return
            }

            fmt.Fprintf(w, message)
        },
    )

    log.Println("Backend listening on :8081")
    log.Fatal(
        http.ListenAndServe(":8081", nil),
    )
}
```

Бачимо в цьому коді потенційне місця для вразливостей:

1. Параметри підключення до бази даних
2. Залежності, вказані в початку

Наш конвеєр SSDLC складатиметься із декількох шагів:

1. `test` – тут буде виконуватись первинна ініціалізація `Golang` проєкту та сканування інструментом Trivy.
2. `build` – збірка контейнерного образу.
3. `scan` – сканування контейнерного образу.
4. `sign` – підпис контейнерного образу.

Більш за все нас цікавить test, scan, sign шаги, оскільки саме в них впроваджено SSDLC елементи.

```
test:
  stage: test
  image: golang:1.22-alpine
  script:
    - go mod init backend || true
    - go mod tidy
    - apk add --no-cache curl git
    - >-
      curl -sfl https://git.io/trivy-install |
      sh -s -- -b /usr/local/bin
    - >-
      trivy fs
      --scanners secret
      --secret-config .trivy-secret.yaml
      --exit-code 1
      --severity HIGH,CRITICAL
      --no-progress .
    - >-
      trivy fs
      --security checks vuln,config,license
      --exit-code 1
      --severity HIGH,CRITICAL
      --no-progress .
```

На етапі `test`, виконується ініціалізація проєкту та перевірка на вразливості інструментом Trivy. Він перевіряє наявність у коді секретів, які надають доступи до тих чи інших ресурсів. Патерни пошуку вказані у файлі `.trivy-secret.yaml`. Також інструмент перевіряє вразливості коду по базах CVE, NVD, GitHub Advisory Database, Alpine/Ubuntu/Debian/CentOS security advisories і так далі [5].

```
scan:
  stage: scan
  extends: .default_job_template
  script:
    - >-
      trivy_image
      --severity HIGH,CRITICAL
      --exit-code 1
      --no-progress
      "$IMAGE_TAG"
```

На етапі scan, використовується той самий інструмент, але вже для сканування зібраного контейнерного образу додатку.

```
sign:
```

```yaml
stage: sign
extends: .default_job_template
dependencies:
  - build
variables:
  COSIGN_EXPERIMENTAL: "1"
script:
  - >-
    echo "$COSIGN_PRIVATE_KEY" |
    base64 -d > cosign.key
  - >-
    echo "$COSIGN_PASSWORD" >
    cosign_password.txt
  - >-
    export COSIGN_PASSWORD="$(cat
    cosign_password.txt)"
  - >-
    cat build.env ||
    echo "build.env not found"
  - source build.env
  - >-
    cosign sign
    --key cosign.key
    "$IMAGE_WITH_DIGEST"
```

На етапі `sign`, виконується підпис контейнерного образу, що дозволяє перевірити аутентичність цього образу вже на етапі розгортання та використання, що запобігає атакам типу supply-chain [6].

В результаті отримуємо конвеєр, який виконує базовий функціонал SSDLC, що забезпечує безпечну розробку програмного забезпечення. Результат самого виконання наведено на наступних рисунках.

На рис. 1 земульовано ситуацію, коли в код потрапила строка для підключення до бази даних PostgreSQL, що містить в собі логін та пароль користувача БД.

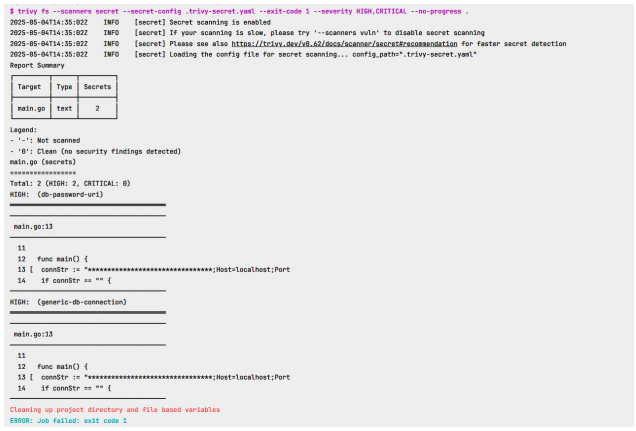

Рис. 1. Сповіщення Trivy про наявність секретних даних у коді

Бачимо, що це зупинило виконання конвеєру, тому наступне виконання збірки контейнерного образу не є можливим.

Виправиши це, виконання конвеєру піде далі, на рис. 2 та рис. 3 наведено результат успішного сканування контейнерного образу та його підпису.

Висновки

Під час виконання цієї роботи було розроблено фреймворк захисту мікросервісної архітектури із використанням стратегії Zero Trust, що об'єднує в собі існуючі інструменти для забезпечення безпеки

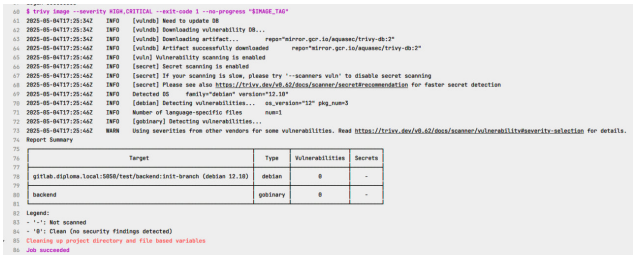

Рис. 2. Виконання сканування контейнерного образу

Рис. 3. Підпис контейнерного образу

контейнеризованих додатків та описує їхнє використання та взаємодію між ними для забезпечення максимальної безпеки програмного продукту на всьому життєвому циклі. За результатами впровадження одного з етапів даного фреймворку (SSDLC) вдалося налаштувати конвеєр сканування та підписування контейнерного образу додатку простої програми на `Golang`.

Перелік використаних джерел

1. Zero Trust Architecture / S. Rose, O. Borchert, S. Mitchell, S. Connelly. — 2020. — URL: https://doi.org/10.6028/NIST.SP.800-207.
2. *Foundation C. N. C.* SPIFFE: Secure Production Identity Framework For Everyone. — 2022. — URL: https://spiffe.io/docs/latest/.
3. *Project O. P. A.* Rego Language Reference. — 2023. — URL: https://www.openpolicyagent.org/docs/latest/policy-language/.
4. *Authors I.* Using Kubernetes Network Policy with Istio. — 2017. — URL: https://istio.io/v1.10/blog/2017/0.1-using-network-policy/.
5. *Security A.* Trivy Documentation. — 2025. — URL: https://aquasecurity.github.io/trivy/ ; Accessed: 2025-05-04.
6. *Project S.* Cosign Container Signing. — 2023. — URL: https://docs.sigstore.dev/cosign/overview/.