Практична робота №6

МЕТОДИ АВТЕНТИФІКАЦІЇ В NGINX

Мета заняття: набути практичних навичок налаштування різних методів автентифікації у вебсервері Nginx: базової автентифікації (Basic Auth), взаємної TLS-автентифікації (mTLS) за допомогою клієнтських сертифікатів, механізму auth_request із зовнішнім бекендом перевірки, автентифікації через OAuth2 Proxy із GitHub як провайдером та JWT-автентифікації через зовнішній сервіс валідації токенів.

Завдання на роботу

1. Підготувати середовище та налаштувати базову автентифікацію (Basic Auth) для захисту ресурсів Nginx.

Basic Auth (базова автентифікація) — найпростіший метод автентифікації, вбудований у Nginx через модуль ngx_http_auth_basic_module. Цей модуль входить до стандартної збірки Nginx і не потребує встановлення додаткових пакетів або перекомпіляції.

Принцип роботи Basic Auth полягає у наступному: при першому запиті до захищеного ресурсу сервер повертає відповідь із кодом 401 Unauthorized та заголовком WWW-Authenticate: Basic realm="...". Браузер відображає діалогове вікно для введення імені та пароля. Після введення браузер кодує рядок username:password у Base64 та надсилає його у заголовку Authorization: Basic <encoded_string> з кожним наступним запитом до цього ресурсу.

Важливо розуміти, що Base64 — це лише кодування, а не шифрування. Будь-хто, хто перехопить мережевий трафік, зможе декодувати облікові дані. Саме тому Basic Auth слід завжди використовувати виключно через HTTPS-з'єднання.

a. Підготувати робоче середовище. Переконатися, що Nginx встановлено та запущено:

```
systemctl status nginx
```

Переконатися, що Nginx працює (Active: active (running)). Перевірити версію:

```
nginx -v
```

b. Видалити стандартну конфігурацію `default`, щоб уникнути конфліктів із конфігураціями, що будуть створені у цій роботі:

```
sudo rm -f /etc/nginx/sites-enabled/default
sudo nginx -t && sudo systemctl reload nginx
```

c. Налаштувати прокидання портів у `Vagrantfile` для можливості перевірки через браузер на хост-машині. Зупинити VM та додати до `Vagrantfile` (рис. 1.c):

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/jammy64"

  # Прокидання HTTP та HTTPS портів на хост-машину
  config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.network "forwarded_port", guest: 443, host: 8443

  # Приватна мережа для IP-фільтрації
  config.vm.network "private_network", ip: "192.168.56.10"
end
```

Рис. 1.c – Vagrantfile з прокиданням портів

```
vagrant reload
```

Після перезапуску VM порти 80 та 443 всередині VM будуть доступні на хост-машині як `localhost:8080` та `localhost:8443` відповідно.

d. Додати DNS-записи для всіх демонстраційних доменів до файлу `hosts` на хост-машині. Це необхідно для перевірки через браузер.

Linux / macOS — відкрити файл з правами адміністратора:

```
sudo nano /etc/hosts
```

Windows — відкрити Блокнот (Notepad) від імені адміністратора та відкрити файл:

```
C:\Windows\System32\drivers\etc\hosts
```

Додати наступні рядки (рис. 1.d):

```
127.0.0.1   auth-demo.local
127.0.0.1   mtls-demo.local
127.0.0.1   authreq-demo.local
127.0.0.1   oauth-demo.local
127.0.0.1   jwt-demo.local
```

Рис. 1.d – DNS-записи для хост-машини

Ці самі записи також потрібно додати всередині VM (до `/etc/hosts` на гостьовій системі) для коректної роботи curl та внутрішніх запитів.

> **ВАЖЛИВА ПРИМІТКА:** На macOS після редагування `/etc/hosts` може знадобитися скинути DNS-кеш командою: `sudo dscacheutil -flushcache && sudo killall -HUP mDNSResponder`. На Windows — командою: `ipconfig /flushdns`.

e. Створити тестову директорію для захищеного контенту та файл індексної сторінки:

```
sudo mkdir -p /var/www/protected
echo "<h1>Protected Area</h1><p>You are authenticated!</p>" | sudo tee /var/www/protected/index.html
```

f. Встановити пакет `apache2-utils`, який містить утиліту `htpasswd` для генерації файлів паролів:

```
sudo apt update && sudo apt install -y apache2-utils
```

g. Створити файл `.htpasswd` із першим користувачем. Прапорець `-c` створює новий файл, `-B` використовує алгоритм bcrypt для хешування:

```
sudo htpasswd -cB /etc/nginx/.htpasswd user1
```

Ввести пароль двічі при запиті. Перевірити вміст створеного файлу:

```
cat /etc/nginx/.htpasswd
```

Nginx підтримує кілька алгоритмів хешування паролів у файлі `.htpasswd`:

— `bcrypt` (прапорець `-B`) — найбезпечніший, рекомендований для використання. Хеш починається з `$2y$`. Bcrypt автоматично додає сіль (salt) та підтримує налаштування кількості раундів хешування.

— `apr1` (MD5, прапорець `-m`) — стандарт Apache, хеш починається з `$apr1$`. Менш безпечний за bcrypt, але широко підтримується.

— `SHA-1` (прапорець `-s`) — застарілий, не рекомендується. Хеш починається з `{SHA}`.

— `crypt()` — системна функція Unix, обмежена 8 символами пароля. Використовується за замовчуванням без прапорців.

h. Додати другого користувача без прапорця `-c` (щоб не перезаписати файл):

```
sudo htpasswd -B /etc/nginx/.htpasswd user2
```

i. Альтернативний спосіб — згенерувати хеш пароля вручну за допомогою `openssl`:

```
echo "user3:$(openssl passwd -apr1 MySecretPass)" | sudo tee -a
/etc/nginx/.htpasswd
```

Формат файлу `.htpasswd` — один рядок на користувача: `ім'я:хеш_пароля`.

Права доступу до файлу мають бути обмежені — лише Nginx-процес повинен мати право читання:

```
sudo chmod 640 /etc/nginx/.htpasswd
sudo chown root:www-data /etc/nginx/.htpasswd
```

Перевірити, що файл містить записи для всіх трьох користувачів:

```
wc -l /etc/nginx/.htpasswd
```

У файлі має бути 3 рядки — по одному на кожного створеного користувача.

j. Створити конфігурацію Nginx для захисту ресурсу через Basic Auth (рис. 1.j):

Конфігурація визначає два блоки `location`: відкриту кореневу частину сайту та захищену директорію `/protected/`, для доступу до якої необхідно ввести ім'я та пароль.

```
server {
    listen 80;
    server_name auth-demo.local;

    # Відкрита частина сайту
    location / {
        root /var/www/html;
        index index.html;
    }

    # Захищена частина — вимагає Basic Auth
    location /protected/ {
        root /var/www;
        index index.html;

        auth_basic "Restricted Area";      # текст у діалозі авторизації
        auth_basic_user_file /etc/nginx/.htpasswd;  # шлях до файлу
паролів
    }
}
```

Рис. 1.j – Конфігурація Nginx із базовою автентифікацією

k. Зберегти конфігурацію у файл `/etc/nginx/sites-available/auth-demo`, створити символічне посилання та додати запис до `/etc/hosts`:

```
sudo ln -s /etc/nginx/sites-available/auth-demo /etc/nginx/sites-enabled/
echo "127.0.0.1 auth-demo.local" | sudo tee -a /etc/hosts
sudo nginx -t && sudo systemctl reload nginx
```

l. Перевірити роботу Basic Auth за допомогою curl — без облікових даних (очікується помилка 401):

```
curl -i http://auth-demo.local/protected/
```

m. Виконати запит із коректними обліковими даними:

```
curl -u user1:пароль http://auth-demo.local/protected/
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -i -u user1:пароль http://auth-demo.local/protected/
```

Зробити висновки за отриманими результатами.

n. Продемонструвати вразливість Base64 — перехоплення пароля. Виконати запит із прапорцем `-v` для перегляду заголовків:

```
curl -v -u user1:пароль http://auth-demo.local/protected/ 2>&1 | grep Authorization
```

Значення заголовка `Authorization: Basic dXNlcjE6...` — це лише Base64-кодування, а не шифрування. Декодувати його:

```
echo "dXNlcjE6cGFzc3dvcmQ=" | base64 -d
```

Це демонструє, чому Basic Auth без HTTPS є небезпечним — будь-хто, хто перехопить трафік, отримає облікові дані у відкритому вигляді.

Додатково перевірити, як Nginx логує невдалі спроби автентифікації:

```
sudo tail -5 /var/log/nginx/error.log
```

У error.log має з'явитися запис виду: `no user/password was provided for basic authentication` або `user "xxx" was not found` або `user "xxx": password mismatch`. Ці логи корисні для моніторингу та виявлення спроб підбору паролів (brute force).

o. Налаштувати комбінування Basic Auth з обмеженням за IP-адресою. Директива `satisfy` визначає логіку комбінування модулів контролю

доступу (`ngx_http_access_module` та `ngx_http_auth_basic_module`). Додати до конфігурації (рис. 1.о):

Режим `satisfy any` означає: доступ дозволено, якщо виконується ХОЧА Б ОДНА з умов — або запит з дозволеної IP-адреси, або надано правильний пароль. Це зручно для внутрішніх ресурсів: співробітники з корпоративної мережі входять без пароля, а зовнішні користувачі — з паролем.

Режим `satisfy all` означає: доступ дозволено лише якщо виконуються ВСІ умови одночасно — і правильна IP-адреса, і правильний пароль. Це забезпечує подвійний контроль доступу.

```
# Доступ дозволено з локальної мережі БЕЗ пароля
# або з будь-якої адреси З паролем
location /internal/ {
    root /var/www;
    index index.html;

    satisfy any;              # достатньо виконання ОДНОГО з правил

    allow 192.168.56.0/24;    # дозволити з приватної мережі
    deny all;                 # заборонити решту

    auth_basic "Internal Zone";
    auth_basic_user_file /etc/nginx/.htpasswd;
}
```

Рис. 1.о – Комбінування Basic Auth та IP-фільтрації через satisfy any

p. Створити тестову директорію та перевірити роботу:

```
sudo mkdir -p /var/www/internal
echo "<h1>Internal Zone</h1>" | sudo tee /var/www/internal/index.html
sudo nginx -t && sudo systemctl reload nginx
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -i http://auth-demo.local/internal/
```

Зробити висновки за отриманими результатами.

q. Змінити директиву на `satisfy all` та повторити перевірку — тепер потрібно одночасно і правильну IP-адресу, і пароль:

```
sudo nginx -t && sudo systemctl reload nginx
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -u user1:пароль http://auth-demo.local/internal/
```

Зробити висновки за отриманими результатами.

г. Перевірити роботу Basic Auth через браузер на хост-машині. Відкрити у браузері адресу `http://auth-demo.local:8080/protected/`.

Браузер має відобразити стандартне системне діалогове вікно автентифікації з полями для введення імені користувача та пароля. Текст у заголовку діалогу відповідає значенню директиви auth_basic ("Restricted Area").

Ввести облікові дані user1 та відповідний пароль. Після успішного входу має відобразитися сторінка з текстом "Protected Area" та "You are authenticated!".

Спробувати натиснути "Cancel" (Скасувати) у діалозі автентифікації — браузер відобразить стандартну сторінку помилки Nginx `401 Authorization Required`.

Зверніть увагу: після успішного входу браузер кешує облікові дані для поточної сесії та надсилає їх автоматично з кожним наступним запитом до цього домену. Для повторного тестування необхідно відкрити нове вікно в режимі інкогніто (Ctrl+Shift+N у Chrome, Cmd+Shift+N на macOS) або закрити та відкрити браузер.

2. Налаштувати взаємну TLS-автентифікацію (mTLS) за допомогою клієнтських сертифікатів.

Mutual TLS (mTLS) — метод автентифікації, при якому не лише клієнт перевіряє сертифікат сервера (як у звичайному HTTPS), а й сервер перевіряє сертифікат клієнта. Це забезпечує двосторонню автентифікацію на рівні транспортного протоколу.

Процес mTLS-з'єднання відбувається під час TLS Handshake. Після того, як сервер надсилає свій сертифікат клієнту, він також надсилає повідомлення CertificateRequest, вимагаючи від клієнта надати свій сертифікат. Клієнт

надсилає свій сертифікат, який сервер перевіряє за допомогою довіреного Certificate Authority (CA).

mTLS є найнадійнішим методом автентифікації серед розглянутих, оскільки він базується на криптографії з відкритим ключем. Типові сценарії використання: міжсервісна комунікація (service-to-service) у мікросервісній архітектурі, IoT-пристрої, корпоративні VPN, API для зовнішніх партнерів.

a. Створити директорію для зберігання сертифікатів та ключів:

```
sudo mkdir -p /etc/nginx/ssl/mtls
cd /etc/nginx/ssl/mtls
```

b. Створити кореневий центр сертифікації (CA) — згенерувати приватний ключ CA та самопідписаний сертифікат:

```
sudo openssl genrsa -out ca.key 4096
sudo openssl req -new -x509 -days 365 -key ca.key -out ca.crt \
    -subj "/C=UA/ST=Zhytomyr/O=Lab CA/CN=Lab Root CA"
```

Цей CA буде використано для підпису клієнтських сертифікатів. Nginx довірятиме лише тим клієнтам, чиї сертифікати підписано цим CA.

Пояснення параметрів команди `openssl req`:

— `-new -x509` — створити самопідписаний сертифікат (не CSR);

— `-days 365` — термін дії сертифіката (1 рік);

— `-key ca.key` — використати попередньо згенерований приватний ключ;

— `-subj` — Subject (відомості про власника): C=Country, ST=State, O=Organization, CN=Common Name.

Розмір ключа 4096 біт для CA забезпечує високу криптографічну стійкість. Для клієнтських та серверних сертифікатів достатньо 2048 біт.

c. Створити серверний сертифікат (якщо ще не налаштовано HTTPS):

```
sudo openssl genrsa -out server.key 2048
sudo openssl req -new -key server.key -out server.csr \
    -subj "/C=UA/ST=Zhytomyr/O=Lab/CN=mtls-demo.local"
sudo openssl x509 -req -days 365 -in server.csr -CA ca.crt -CAkey
ca.key \
    -CAcreateserial -out server.crt
```

d. Створити клієнтський сертифікат — згенерувати ключ, запит на підпис (CSR) та підписати його за допомогою CA:

```
sudo openssl genrsa -out client.key 2048
sudo openssl req -new -key client.key -out client.csr \
    -subj "/C=UA/ST=Zhytomyr/O=Lab/CN=Student Client"
sudo openssl x509 -req -days 365 -in client.csr -CA ca.crt -CAkey
ca.key \
    -CAcreateserial -out client.crt
```

Тепер є три пари ключ/сертифікат: CA, серверний, клієнтський.

e. Перевірити, що клієнтський сертифікат справді підписаний нашим CA:

```
openssl verify -CAfile ca.crt client.crt
```

Перевірити застосовану конфігурацію, виконавши команду:

```
openssl verify -CAfile ca.crt client.crt
```

Зробити висновки за отриманими результатами.

f. Налаштувати Nginx для mTLS — створити конфігурацію `/etc/nginx/sites-available/mtls-demo` (рис. 2.f):

Ключові директиви для mTLS:

— `ssl_client_certificate` — шлях до сертифіката CA, яким підписано клієнтські сертифікати. Nginx перевірятиме, чи клієнтський сертифікат підписаний саме цим CA;

— `ssl_verify_client on` — вимагати клієнтський сертифікат від кожного з'єднання. Без сертифіката Nginx відхилить з'єднання на етапі TLS Handshake;

— Змінні `$ssl_client_s_dn` (Subject Distinguished Name) та `$ssl_client_verify` (результат перевірки: SUCCESS або FAILED) доступні для використання в конфігурації — для логування, умовного доступу або передачі бекенду.

```
server {
    listen 443 ssl;
    server_name mtls-demo.local;

    # Серверний сертифікат і ключ
    ssl_certificate     /etc/nginx/ssl/mtls/server.crt;
    ssl_certificate_key /etc/nginx/ssl/mtls/server.key;

    # Клієнтська автентифікація
```

```
    ssl_client_certificate /etc/nginx/ssl/mtls/ca.crt;  # CA для
перевірки клієнтів
    ssl_verify_client on;       # вимагати клієнтський сертифікат

    # Протоколи та шифри
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_prefer_server_ciphers on;

    location / {
        root /var/www/html;
        index index.html;

        # Передати інформацію про клієнтський сертифікат бекенду
        add_header X-Client-DN $ssl_client_s_dn always;
        add_header X-Client-Verify $ssl_client_verify always;
    }
}
```

Рис. 2.f – Конфігурація Nginx для mTLS-автентифікації

g. Активувати конфігурацію та додати запис DNS:

```
sudo ln -s /etc/nginx/sites-available/mtls-demo /etc/nginx/sites-
enabled/
echo "127.0.0.1 mtls-demo.local" | sudo tee -a /etc/hosts
sudo nginx -t && sudo systemctl reload nginx
```

h. Перевірити підключення без клієнтського сертифіката — Nginx має відхилити з'єднання:

```
curl -k https://mtls-demo.local/
```

Очікується помилка 400 (No required SSL certificate was sent) або розрив з'єднання.

i. Виконати запит з клієнтським сертифікатом:

```
curl -k --cert /etc/nginx/ssl/mtls/client.crt \
     --key /etc/nginx/ssl/mtls/client.key \
     https://mtls-demo.local/
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -k -I --cert /etc/nginx/ssl/mtls/client.crt --key
/etc/nginx/ssl/mtls/client.key https://mtls-demo.local/
```

Зробити висновки за отриманими результатами.

j. Переглянути заголовки відповіді для отримання інформації з клієнтського сертифіката:

```
curl -k -I --cert /etc/nginx/ssl/mtls/client.crt \
     --key /etc/nginx/ssl/mtls/client.key \
     https://mtls-demo.local/ 2>&1 | grep -i x-client
```

Заголовок `X-Client-DN` містить Distinguished Name з клієнтського сертифіката (CN=Student Client), а `X-Client-Verify` — результат перевірки (SUCCESS).

k. Налаштувати умовний доступ на основі даних сертифіката. Додати до конфігурації блок `map` та перевірку (рис. 2.k):

```
# У блоці http {} головного nginx.conf або в окремому файлі
map $ssl_client_s_dn $allowed_client {
    default          0;
    ~CN=Student      1;   # дозволити лише сертифікати з CN, що містить
"Student"
}

# У блоці server {} конфігурації mtls-demo:
    location /admin/ {
        if ($allowed_client = 0) {
            return 403 "Access denied: certificate not authorized\n";
        }
        root /var/www;
        index index.html;
    }
```

Рис. 2.k – Умовний доступ на основі Distinguished Name клієнтського сертифіката

l. Змінити директиву `ssl_verify_client` на `optional` для окремих маршрутів, де сертифікат бажаний, але не обов'язковий (рис. 2.l):

```
server {
    # ...
    ssl_verify_client optional;  # запитувати сертифікат, але не вимагати

    location / {
        # Доступно всім
        root /var/www/html;
    }

    location /secure/ {
        # Доступно лише з валідним сертифікатом
        if ($ssl_client_verify != SUCCESS) {
            return 403;
        }
        root /var/www;
    }
}
```

Рис. 2.l – Конфігурація з необов'язковою клієнтською автентифікацією

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -k https://mtls-demo.local/ && curl -k https://mtls-
demo.local/secure/
```

Зробити висновки за отриманими результатами.

m. Створити список відкликаних сертифікатів (CRL) для можливості скасування доступу клієнта без зміни CA:

Certificate Revocation List (CRL) — це список сертифікатів, які було відкликано до закінчення їх терміну дії. Це необхідно, наприклад, коли приватний ключ клієнта було скомпрометовано або працівник звільнився.

```
cd /etc/nginx/ssl/mtls
sudo openssl ca -gencrl -keyfile ca.key -cert ca.crt -out crl.pem -
crldays 30 2>/dev/null || echo "CRL requires CA database setup"
```

Для повноцінної роботи CRL необхідна структура CA з базою даних (файл `index.txt`) та серійним номером (`serial`). У спрощеному варіанті можна використати команду `openssl crl` для створення порожнього CRL.

Для підключення CRL до Nginx додати директиву `ssl_crl /etc/nginx/ssl/mtls/crl.pem;` у блок `server {}`. Nginx перевірятиме кожен клієнтський сертифікат на наявність у списку відкликаних.

n. Розглянути типові помилки при налаштуванні mTLS:

— Помилка `400 Bad Request: No required SSL certificate was sent` — клієнт не надав сертифікат при `ssl_verify_client on`;

— Помилка `400 The SSL certificate error` — сертифікат клієнта не підписано довіреним CA (перевірити `ssl_client_certificate`);

— Помилка `SSL_ERROR_HANDSHAKE_FAILURE_ALERT` у браузері — формат сертифіката несумісний або сертифікат протермінований;

— Для імпорту клієнтського сертифіката у браузер потрібно конвертувати його у формат PKCS#12:

```
sudo openssl pkcs12 -export -out client.p12 -inkey client.key -in
client.crt -certfile ca.crt
```

При запиті Export Password ввести пароль для захисту файлу .p12 (цей пароль знадобиться при імпорті).

o. Скопіювати файл `client.p12` на хост-машину для імпорту у браузер. З хост-машини виконати:

```
vagrant ssh -- "sudo cat /etc/nginx/ssl/mtls/client.p12" > client.p12
```

Альтернативно, якщо налаштовано спільну директорію (synced folder), скопіювати файл у `/vagrant/` всередині VM.

р. Імпортувати клієнтський сертифікат у браузер. Процедура залежить від операційної системи та браузера:

Google Chrome / Chromium (yci ОС):

— Відкрити `chrome://settings/security` → Manage device certificates (або Manage certificates);

— На вкладці "Your certificates" або "Personal" натиснути "Import";

— Обрати файл client.p12, ввести пароль, встановлений при експорті;

— Підтвердити імпорт.

macOS (Safari / Chrome):

— Подвійний клік на файлі `client.p12` — він автоматично відкриється у Keychain Access;

— Ввести пароль файлу .p12, потім пароль системи для підтвердження;

— Сертифікат з'явиться у розділі "My Certificates" (login keychain).

Windows (Chrome / Edge):

— Подвійний клік на файлі `client.p12` — запуститься Certificate Import Wizard;

— Обрати "Current User", натиснути Next;

— Ввести пароль файлу .p12, увімкнути "Mark this key as exportable" за бажанням;

— Дозволити автоматичний вибір сховища сертифікатів → Finish.

Firefox (yci ОС):

— Відкрити `about:preferences#privacy` → Certificates → View Certificates;

— На вкладці "Your Certificates" натиснути "Import";

— Обрати файл client.p12, ввести пароль.

q. Перевірити mTLS через браузер. Відкрити у браузері `https://mtls-demo.local:8443/`.

   Браузер покаже попередження про самопідписаний серверний сертифікат — прийняти ризик та продовжити (Advanced → Proceed / Accept the Risk).

   Після цього браузер має запропонувати вибрати клієнтський сертифікат зі списку встановлених. Обрати "Student Client" та натиснути OK.

   При успішній автентифікації має відобразитися стандартна сторінка Nginx. Якщо клієнтський сертифікат не імпортовано або натиснуто "Cancel" — браузер покаже помилку з'єднання або сторінку 400 Bad Request.

3. Налаштувати механізм auth_request із зовнішнім бекендом автентифікації.

Директива auth_request — найпотужніший і найгнучкіший вбудований механізм автентифікації в open-source Nginx. Вона реалізована в модулі ngx_http_auth_request_module, який входить до стандартної збірки Nginx в Ubuntu/Debian.

Принцип роботи: при кожному запиті до захищеного location Nginx виконує внутрішній підзапит (subrequest) до зазначеного endpoint'у. Цей endpoint зазвичай є окремим сервісом автентифікації. Якщо сервіс повертає HTTP 200 — Nginx пропускає оригінальний запит далі. Якщо сервіс повертає 401 або 403 — Nginx блокує запит і повертає відповідний код помилки клієнту. Усі інші коди відповіді (наприклад, 500) вважаються помилкою і також блокують запит.

Ключова перевага auth_request — повна незалежність від конкретного протоколу автентифікації. Бекенд може реалізовувати будь-яку логіку: перевірку API-ключів, JWT-токенів, сесійних cookie, OAuth2-токенів, LDAP-запитів тощо. Nginx не знає і не потребує знань про деталі автентифікації — він лише аналізує HTTP-код відповіді.

a. Встановити Node.js для створення бекенда автентифікації (якщо ще не встановлено):

```
sudo apt install -y nodejs npm
```

b. Створити директорію проєкту та ініціалізувати його:

```
mkdir -p ~/auth-backend && cd ~/auth-backend
npm init -y
```

c. Створити файл `~/auth-backend/server.js` — бекенд, який перевіряє API-ключ у заголовку запиту (рис. 3.c):

```
const http = require('http');

// Список дійсних API-ключів
const VALID_KEYS = [
    'secret-api-key-12345',
    'another-valid-key-67890'
];

const server = http.createServer((req, res) => {
    // Отримати API-ключ із заголовка X-API-Key
    const apiKey = req.headers['x-api-key'];
    // Отримати оригінальний URI із заголовка, переданого Nginx
    const originalUri = req.headers['x-original-uri'] || 'unknown';

    console.log(`[AUTH] URI: ${originalUri}, Key: ${apiKey ? "present" :
"missing"}`);

    if (apiKey && VALID_KEYS.includes(apiKey)) {
        // Автентифікація успішна — повернути 200
        res.writeHead(200, {
            'X-Auth-User': 'api-client',    // передати ім'я користувача
назад
            'X-Auth-Status': 'authenticated'
        });
        res.end('OK');
    } else {
        // Автентифікація невдала — повернути 401
        res.writeHead(401, { 'Content-Type': 'text/plain' });
        res.end('Unauthorized');
    }
});

server.listen(3001, '127.0.0.1', () => {
    console.log('Auth backend listening on http://127.0.0.1:3001');
});
```

Рис. 3.c – Node.js-бекенд автентифікації за API-ключем

d. Запустити бекенд у фоновому режимі та переконатися, що він працює:

```
cd ~/auth-backend && node server.js &
curl -I http://127.0.0.1:3001/ -H "X-API-Key: secret-api-key-12345"
```

Очікується відповідь 200 OK. Без заголовка X-API-Key має повертатися 401 Unauthorized:

```
curl -I http://127.0.0.1:3001/
```

Пояснення роботи бекенда: сервер слухає на порту 3001 та перевіряє наявність заголовка `X-API-Key` у кожному запиті. Масив `VALID_KEYS` містить перелік допустимих ключів. При успішній перевірці сервер повертає 200 та додає заголовки `X-Auth-User` і `X-Auth-Status`, які Nginx зможе прочитати через `auth_request_set` та використати для подальшої обробки запиту — наприклад, передати бекенду ім'я автентифікованого користувача.

e. Створити конфігурацію Nginx `/etc/nginx/sites-available/authreq-demo`, яка використовує `auth_request` для перевірки кожного запиту (рис. 3.e):

```
server {
    listen 80;
    server_name authreq-demo.local;

    # Захищена частина — кожен запит проходить через auth_request
    location /api/ {
        auth_request /auth;                          # шлях до перевірки
        auth_request_set $auth_user $upstream_http_x_auth_user;
        auth_request_set $auth_status $upstream_http_x_auth_status;

        # Передати ім'я автентифікованого користувача далі
        proxy_set_header X-Authenticated-User $auth_user;

        root /var/www;
        index index.html;

        # Додати заголовки у відповідь для налагодження
        add_header X-Auth-User $auth_user always;
        add_header X-Auth-Status $auth_status always;
    }

    # Внутрішній location для підзапиту автентифікації
    location = /auth {
        internal;                                    # недоступний ззовні
        proxy_pass http://127.0.0.1:3001;      # адреса бекенда

        proxy_pass_request_body off;             # не передавати тіло
запиту
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-URI $request_uri;
        proxy_set_header X-API-Key $http_x_api_key;  # прокинути ключ
клієнта
    }
```

```
    # Відкрита частина
    location / {
        root /var/www/html;
        index index.html;
    }
}
```

Рис. 3.e – Конфігурація Nginx з auth_request до зовнішнього бекенда

f. Активувати конфігурацію:

```
sudo ln -s /etc/nginx/sites-available/authreq-demo /etc/nginx/sites-
enabled/
echo "127.0.0.1 authreq-demo.local" | sudo tee -a /etc/hosts
sudo mkdir -p /var/www/api
echo '{"status":"ok","data":"secret info"}' | sudo tee
/var/www/api/index.html
sudo nginx -t && sudo systemctl reload nginx
```

g. Перевірити запит без API-ключа (очікується 401):

```
curl -i http://authreq-demo.local/api/
```

h. Виконати запит із правильним API-ключем:

```
curl -i -H "X-API-Key: secret-api-key-12345" http://authreq-
demo.local/api/
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -i -H "X-API-Key: secret-api-key-12345" http://authreq-
demo.local/api/
```

Зробити висновки за отриманими результатами.

i. Перевірити запит із неправильним ключем:

```
curl -i -H "X-API-Key: wrong-key" http://authreq-demo.local/api/
```

Очікується код 401. Переглянути логи бекенда для аналізу роботи механізму автентифікації.

j. Пояснення механізму `auth_request_set`. Ця директива дозволяє "витягнути" значення із заголовків відповіді бекенда автентифікації та використати їх у подальшій обробці запиту. Синтаксис:

```
# $upstream_http_<header_name> — заголовок відповіді від auth-бекенда
# (дефіси замінюються на підкреслення, все у нижньому регістрі)

auth_request_set $auth_user $upstream_http_x_auth_user;
# Записує значення заголовка X-Auth-User із відповіді бекенда
# у змінну $auth_user для використання в основному запиті
```

Рис. 3.j – Механізм передачі даних через auth_request_set

> **ВАЖЛИВА ПРИМІТКА:** Директива auth_request виконує subrequest — Nginx не передає тіло оригінального запиту до бекенда автентифікації (якщо явно не налаштовано інакше). Це зроблено навмисно для продуктивності: бекенд автентифікації повинен приймати рішення лише на основі заголовків.

k. Розглянути роботу механізму `auth_request` на рівні HTTP-запитів. Для цього увімкнути логування у форматі debug для `authreq-demo`:

```
# У блоці http {} файлу /etc/nginx/nginx.conf додати:
log_format auth_debug '$remote_addr - $remote_user [$time_local] '
                      '"$request" $status '
                      '"auth_user=$http_x_api_key" '
                      '"upstream_status=$upstream_status"';

# У блоці server {} конфігурації authreq-demo:
access_log /var/log/nginx/authreq-debug.log auth_debug;
```

Рис. 3.k – Налаштування розширеного логування для аналізу auth_request

```
sudo nginx -t && sudo systemctl reload nginx
```

Виконати кілька запитів з ключем та без нього, потім переглянути лог:

```
sudo tail -10 /var/log/nginx/authreq-debug.log
```

У логах видно два запити на кожен клієнтський запит: (1) subrequest до `/auth` — це внутрішній запит auth_request; (2) основний запит до `/api/`. Якщо subrequest повернув 401, основний запит не виконується.

4. Налаштувати автентифікацію через OAuth2 Proxy із GitHub як провайдером.

OAuth2 — відкритий стандарт авторизації (RFC 6749), який дозволяє стороннім додаткам отримувати обмежений доступ до ресурсів користувача без передачі пароля. OAuth2 визначає декілька потоків (flows) авторизації. Найпоширеніший для веб-додатків — Authorization Code Flow.

Authorization Code Flow працює наступним чином: (1) додаток перенаправляє користувача на сторінку логіну провайдера (GitHub); (2) після успішного входу провайдер повертає тимчасовий authorization code на callback URL; (3) додаток обмінює цей code на access token, звертаючись напряму до API

провайдера; (4) access token використовується для отримання даних користувача.

OAuth2 Proxy (проєкт oauth2-proxy/oauth2-proxy) — це зворотний проксі, який інкапсулює весь цей потік. Він керує сесіями, cookie, і редиректами. Nginx використовує auth_request для делегування перевірки автентифікації OAuth2 Proxy. Така архітектура дозволяє захистити будь-який статичний сайт або бекенд автентифікацією через GitHub, Google, Keycloak та інші OAuth2/OIDC-провайдери без жодних змін у коді додатка.

a. Встановити Docker (якщо ще не встановлено) та додати користувача до групи docker:

```
sudo apt install -y docker.io
sudo usermod -aG docker vagrant
```

Перезайти в систему або виконати newgrp docker для застосування змін.

b. Зареєструвати OAuth App у GitHub. Перейти до Settings → Developer settings → OAuth Apps → New OAuth App. Заповнити поля:

```
Application name: Nginx Auth Lab
Homepage URL: http://oauth-demo.local
Authorization callback URL: http://oauth-demo.local/oauth2/callback
```

Рис. 4.b – Параметри реєстрації OAuth App у GitHub

Після створення додатка зберегти `Client ID` та згенерувати `Client Secret`. Ці значення знадобляться для конфігурації OAuth2 Proxy.

c. Згенерувати випадковий секрет для шифрування сесійних cookie:

```
openssl rand -hex 16
```

Зберегти отримане значення — воно буде використане як OAUTH2_PROXY_COOKIE_SECRET.

d. Запустити OAuth2 Proxy у Docker-контейнері (рис. 4.d):

```
docker run -d --name oauth2-proxy \
    --net=host \
    quay.io/oauth2-proxy/oauth2-proxy:latest \
    --provider=github \
    --client-id=<GITHUB_CLIENT_ID> \
    --client-secret=<GITHUB_CLIENT_SECRET> \
```

```
    --cookie-secret=<COOKIE_SECRET_HEX> \
    --cookie-secure=false \
    --upstream=static://200 \
    --http-address=127.0.0.1:4180 \
    --redirect-url=http://oauth-demo.local/oauth2/callback \
    --email-domain=* \
    --cookie-domain=oauth-demo.local \
    --whitelist-domain=oauth-demo.local \
    --scope="user:email"
```

Рис. 4.d – Запуск OAuth2 Proxy як Docker-контейнера

Замінити `<GITHUB_CLIENT_ID>`, `<GITHUB_CLIENT_SECRET>` та `<COOKIE_SECRET_HEX>` на реальні значення, отримані у попередніх кроках.

Пояснення ключових параметрів OAuth2 Proxy:

— `--provider=github` — використовувати GitHub як OAuth2-провайдера;

— `--upstream=static://200` — OAuth2 Proxy не проксіює запити далі, а лише виконує автентифікацію (Nginx сам обслуговує контент);

— `--http-address=127.0.0.1:4180` — слухати лише на localhost для безпеки (Nginx звертатиметься локально);

— `--cookie-secure=false` — дозволити cookie без HTTPS (лише для лабораторного середовища, у продакшені завжди true);

— `--email-domain=*` — дозволити вхід користувачам з будь-яким email (без обмеження за доменом);

— `--scope="user:email"` — запитати у GitHub дозвіл на читання email-адреси користувача.

e. Переконатися, що OAuth2 Proxy запустився коректно:

```
docker logs oauth2-proxy
curl -I http://127.0.0.1:4180/
```

f. Створити конфігурацію Nginx `/etc/nginx/sites-available/oauth-demo` (рис. 4.f):

Конфігурація містить три ключових блоки `location`:

— Кореневий `location /` — захищений через `auth_request`. При відсутності автентифікації Nginx повертає 401, який через `error_page 401` перенаправляє користувача на сторінку логіну OAuth2 Proxy;

— `location = /oauth2/auth` — внутрішній endpoint (`internal`), до якого Nginx робить subrequest. OAuth2 Proxy перевіряє наявність та валідність сесійної cookie;

— `location /oauth2/` — публічний endpoint для обробки OAuth2-потоку: сторінка логіну (`/oauth2/sign_in`), callback від GitHub (`/oauth2/callback`), вихід (`/oauth2/sign_out`).

```
server {
    listen 80;
    server_name oauth-demo.local;

    # Захищена частина — вимагає OAuth2-автентифікації
    location / {
        auth_request /oauth2/auth;
        error_page 401 = /oauth2/sign_in;

        # Передати дані автентифікованого користувача
        auth_request_set $user   $upstream_http_x_auth_request_user;
        auth_request_set $email  $upstream_http_x_auth_request_email;
        proxy_set_header X-User  $user;
        proxy_set_header X-Email $email;

        # Додати заголовки для налагодження
        add_header X-Authenticated-User $user always;
        add_header X-Authenticated-Email $email always;

        root /var/www/oauth-demo;
        index index.html;
    }

    # Внутрішній endpoint перевірки автентифікації
    location = /oauth2/auth {
        internal;
        proxy_pass http://127.0.0.1:4180;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Auth-Request-Redirect $request_uri;

        proxy_pass_request_body off;
        proxy_set_header Content-Length "";
    }

    # Обробка OAuth2 Proxy маршрутів (sign_in, callback тощо)
    location /oauth2/ {
        proxy_pass http://127.0.0.1:4180;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
```

```
        proxy_set_header X-Scheme $scheme;
        proxy_set_header X-Auth-Request-Redirect $request_uri;
    }
}
```

Рис. 4.f – Конфігурація Nginx з OAuth2 Proxy для автентифікації через GitHub

g. Створити контент для захищеного сайту, активувати конфігурацію:

```
sudo mkdir -p /var/www/oauth-demo
echo "<h1>OAuth2 Protected</h1><p>Welcome! You are authenticated via
GitHub.</p>" | sudo tee /var/www/oauth-demo/index.html
sudo ln -s /etc/nginx/sites-available/oauth-demo /etc/nginx/sites-
enabled/
echo "127.0.0.1 oauth-demo.local" | sudo tee -a /etc/hosts
sudo nginx -t && sudo systemctl reload nginx
```

h. Перевірити роботу OAuth2 через браузер на хост-машині. Прокидання портів та DNS-записи налаштовано у завданні 1 (підпункти c, d). Відкрити у браузері адресу `http://oauth-demo.local:8080/`.

Очікувана послідовність дій OAuth2 Authorization Code Flow:

1. Nginx отримує запит, виконує `auth_request` до OAuth2 Proxy → сесія відсутня → повертає 401;

2. Nginx через `error_page 401` перенаправляє на `/oauth2/sign_in`;

3. OAuth2 Proxy перенаправляє браузер на сторінку авторизації GitHub;

4. Користувач вводить логін/пароль GitHub та підтверджує доступ для додатка "Nginx Auth Lab";

5. GitHub повертає authorization code на `/oauth2/callback`;

6. OAuth2 Proxy обмінює code на access token, отримує дані користувача, створює сесійну cookie;

7. Браузер перенаправляється на оригінальну сторінку — відображається вміст "OAuth2 Protected".

Весь цей потік відбувається автоматично — користувач бачить лише сторінку GitHub та кінцевий результат.

i. Після успішного входу перевірити заголовки відповіді через DevTools браузера (F12 → Network → виділити запит → Headers). У Response Headers мають бути присутні:

`X-Authenticated-User` — ім'я користувача GitHub;

`X-Authenticated-Email` — email-адреса GitHub-акаунту.

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -I http://oauth-demo.local/
```

Зробити висновки за отриманими результатами.

У відповіді curl має бути статус 302 або 401 з перенаправленням на `/oauth2/sign_in` — без сесійної cookie автентифікація через curl неможлива. Це підтверджує, що OAuth2 Flow вимагає інтерактивного браузера.

j. Переглянути логи OAuth2 Proxy для розуміння повного циклу автентифікації:

```
docker logs oauth2-proxy --tail 20
```

У логах видно: отримання коду авторизації від GitHub, обмін коду на токен, створення сесійної cookie.

k. Перевірити, що після автентифікації OAuth2 Proxy створює сесійну cookie. Відкрити DevTools у браузері (F12), перейти на вкладку Application → Cookies та знайти cookie з префіксом _oauth2_proxy:

Cookie `_oauth2_proxy` містить зашифровані дані сесії, зокрема access token та інформацію про користувача. Cookie шифрується за допомогою `COOKIE_SECRET`, який було згенеровано на кроці с. Якщо змінити секрет, усі існуючі сесії стануть недійсними.

l. Перевірити роботу виходу із системи (logout). OAuth2 Proxy надає endpoint `/oauth2/sign_out`:

```
curl -i http://oauth-demo.local/oauth2/sign_out
```

Після виходу cookie видаляється, і при наступному зверненні до захищеного ресурсу користувач буде перенаправлений на сторінку логіну GitHub.

m. Розглянути типові помилки при налаштуванні OAuth2 Proxy:

— Помилка "redirect_uri_mismatch" — URL у конфігурації `--redirect-url` не збігається з Authorization callback URL у налаштуваннях GitHub OAuth App;

— Помилка "invalid cookie" — секрет cookie змінився або має неправильний формат (має бути рівно 16, 24 або 32 байти у hex);

— Помилка "upstream timeout" — OAuth2 Proxy не може зв'язатися з GitHub API (перевірити DNS та мережу);

— Помилка "no session found" — cookie не передається (перевірити `--cookie-domain` та `--cookie-secure`).

5. Налаштувати JWT-автентифікацію через auth_request із Node.js-сервісом валідації.

JSON Web Token (JWT) — відкритий стандарт (RFC 7519), який визначає компактний і самодостатній спосіб безпечної передачі інформації між сторонами у вигляді JSON-об'єкта з цифровим підписом.

JWT складається з трьох частин, розділених крапкою (header.payload.signature). Header містить метадані: алгоритм підпису (наприклад, HS256 — HMAC-SHA256) та тип токена (JWT). Payload містить claims — набір тверджень про сутність (зазвичай користувача): sub (subject — ідентифікатор), iat (issued at — час видачі), exp (expiration — час закінчення дії), а також довільні дані (role, permissions тощо). Signature — підпис, обчислений за формулою: HMAC-SHA256(base64url(header) + "." + base64url(payload), secret).

Ключова властивість JWT — самодостатність (self-contained). Сервер валідації може перевірити токен без звернення до бази даних чи зовнішнього сервісу — достатньо знати секретний ключ. Це робить JWT ідеальним для розподілених систем і мікросервісних архітектур, де кожен сервіс може незалежно перевіряти автентичність запиту.

У цьому завданні реалізується перевірка JWT через механізм auth_request: Nginx передає заголовок Authorization до Node.js-сервісу, який верифікує підпис токена, перевіряє термін дії та повертає дані з payload через заголовки відповіді.

a. Створити директорію проєкту та встановити залежності:

```
mkdir -p ~/jwt-auth && cd ~/jwt-auth
npm init -y
npm install jsonwebtoken
```

b. Створити файл `~/jwt-auth/jwt-server.js` — сервіс валідації JWT-токенів (рис. 5.b):

```
const http = require('http');
const jwt = require('jsonwebtoken');

// Секретний ключ для підпису та перевірки токенів
const SECRET = 'my-super-secret-key-for-lab';

const server = http.createServer((req, res) => {
    // Отримати заголовок Authorization із запиту
    const authHeader = req.headers['authorization'] || '';
    const originalUri = req.headers['x-original-uri'] || 'unknown';

    // Перевірити формат Bearer <token>
    if (!authHeader.startsWith('Bearer ')) {
        console.log(`[JWT] ${originalUri} — токен відсутній`);
        res.writeHead(401);
        return res.end('No token');
    }

    const token = authHeader.slice(7); // видалити "Bearer "

    try {
        // Верифікувати токен
        const decoded = jwt.verify(token, SECRET);
        console.log(`[JWT] ${originalUri} — OK, user: ${decoded.sub}`);

        // Передати дані з токена назад через заголовки
        res.writeHead(200, {
            'X-JWT-User': decoded.sub || 'unknown',
            'X-JWT-Role': decoded.role || 'user',
            'X-JWT-Exp':  String(decoded.exp || 0)
        });
        res.end('OK');
    } catch (err) {
        console.log(`[JWT] ${originalUri} — ПОМИЛКА: ${err.message}`);
        res.writeHead(401);
        res.end(err.message);
    }
});

server.listen(3002, '127.0.0.1', () => {
```

```
    console.log('JWT auth service listening on http://127.0.0.1:3002');
});
```

Рис. 5.b – Node.js-сервіс валідації JWT-токенів

c. Створити утиліту `~/jwt-auth/gen-token.js` для генерації тестових токенів (рис. 5.c):

```
const jwt = require('jsonwebtoken');

const SECRET = 'my-super-secret-key-for-lab';

// Аргументи: node gen-token.js <username> <role> <expiry>
const username = process.argv[2] || 'student';
const role = process.argv[3] || 'user';
const expiry = process.argv[4] || '1h';

const token = jwt.sign(
    {
        sub: username,
        role: role,
        iat: Math.floor(Date.now() / 1000)
    },
    SECRET,
    { expiresIn: expiry }
);

console.log('\nЗгенерований JWT-токен:');
console.log(token);
console.log('\nДекодований payload:');
console.log(JSON.stringify(jwt.decode(token), null, 2));
```

Рис. 5.c – Утиліта генерації тестових JWT-токенів

d. Запустити сервіс валідації та згенерувати тестові токени:

```
cd ~/jwt-auth && node jwt-server.js &
node gen-token.js student admin 1h
```

Зберегти згенерований токен — він знадобиться для тестування. Вивід містить як сам токен (рядок з трьох частин, розділених крапками), так і декодований payload у форматі JSON.

Структура виводу генератора:

— Перша частина токена (до першої крапки) — Base64url-кодований Header: `{"alg":"HS256","typ":"JWT"}`;

— Друга частина — Base64url-кодований Payload: `{"sub":"student","role":"admin","iat":...,"exp":...}`;

— Третя частина — Signature, обчислена за допомогою секретного ключа.

Перевірити, що токен складається саме з трьох частин, розділених крапкою. Кожну частину можна декодувати окремо:

```
echo "<ПЕРША_ЧАСТИНА_ТОКЕНА>" | base64 -d 2>/dev/null
```

e. Згенерувати токен з коротким терміном дії (5 секунд) для тестування перевірки expiry:

```
node gen-token.js testuser viewer 5s
```

Зачекати 10 секунд перед використанням цього токена — він має бути протермінованим.

f. Створити конфігурацію Nginx `/etc/nginx/sites-available/jwt-demo` (рис. 5.f):

Архітектура JWT-автентифікації через auth_request:

1. Клієнт надсилає запит із заголовком `Authorization: Bearer <token>`;

2. Nginx перехоплює запит і робить subrequest до `/jwt-verify`;

3. Location `/jwt-verify` проксіює запит до Node.js-сервісу на порту 3002;

4. Node.js-сервіс верифікує підпис токена, перевіряє термін дії та повертає 200 (OK) або 401 (Unauthorized);

5. При успішній перевірці сервіс додає до відповіді заголовки `X-JWT-User` та `X-JWT-Role`;

6. Nginx через `auth_request_set` копіює ці заголовки у змінні для подальшого використання.

```
server {
    listen 80;
    server_name jwt-demo.local;

    # Захищений API — вимагає валідний JWT
    location /api/ {
        auth_request /jwt-verify;

        # Отримати дані з токена через заголовки бекенда
        auth_request_set $jwt_user $upstream_http_x_jwt_user;
        auth_request_set $jwt_role $upstream_http_x_jwt_role;

        # Передати далі як заголовки
```

```
        add_header X-JWT-User $jwt_user always;
        add_header X-JWT-Role $jwt_role always;

        root /var/www/jwt-demo;
        index index.html;
    }

    # Внутрішній endpoint валідації JWT
    location = /jwt-verify {
        internal;
        proxy_pass http://127.0.0.1:3002;

        proxy_pass_request_body off;
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-URI $request_uri;
        # Прокинути заголовок Authorization від клієнта
        proxy_set_header Authorization $http_authorization;
    }

    # Відкрита частина
    location / {
        root /var/www/html;
        index index.html;
    }
}
```

Рис. 5.f – Конфігурація Nginx для JWT-автентифікації через auth_request

g. Активувати конфігурацію:

```
sudo mkdir -p /var/www/jwt-demo/api
echo '{"message":"JWT-protected data","items":[1,2,3]}' | sudo tee
/var/www/jwt-demo/api/index.html
sudo ln -s /etc/nginx/sites-available/jwt-demo /etc/nginx/sites-
enabled/
echo "127.0.0.1 jwt-demo.local" | sudo tee -a /etc/hosts
sudo nginx -t && sudo systemctl reload nginx
```

h. Перевірити запит без токена (очікується 401):

```
curl -i http://jwt-demo.local/api/
```

i. Виконати запит із валідним токеном (замінити <TOKEN> на згенерований раніше):

```
curl -i -H "Authorization: Bearer <TOKEN>" http://jwt-demo.local/api/
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -i -H "Authorization: Bearer <TOKEN>" http://jwt-demo.local/api/
```

Зробити висновки за отриманими результатами.

j. Перевірити запит із протермінованим токеном — використати токен, згенерований з expiry=5s після очікування:

```
curl -i -H "Authorization: Bearer <EXPIRED_TOKEN>" http://jwt-
demo.local/api/
```

Очікується 401 з повідомленням "jwt expired" у логах бекенда.

k. Перевірити запит із підробленим токеном — змінити декілька символів у валідному токені:

```
curl -i -H "Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.FAKE.SIGNATURE"
http://jwt-demo.local/api/
```

Очікується 401 з повідомленням "invalid signature" або "invalid token" у логах бекенда.

l. Переглянути логи сервісу валідації для аналізу всіх спроб автентифікації:

```
# Логи виводяться в термінал, де запущено jwt-server.js
# Приклад виводу:
# [JWT] /api/ — OK, user: student
# [JWT] /api/ — ПОМИЛКА: jwt expired
# [JWT] /api/ — ПОМИЛКА: invalid signature
```

m. Розглянути аспекти безпеки JWT-автентифікації:

JWT має кілька важливих особливостей з точки зору безпеки, які необхідно враховувати при впровадженні:

— **Відкликання токенів**: JWT є самодостатнім, тому сервер не може "відкликати" виданий токен до закінчення його терміну дії. Для вирішення цієї проблеми використовують чорні списки (blacklist) токенів у Redis або короткий час життя токенів (5–15 хвилин) з механізмом refresh token.

— **Алгоритм "none"**: деякі бібліотеки JWT підтримують алгоритм підпису "none", що дозволяє створювати токени без підпису. Завжди явно вказувати допустимі алгоритми при верифікації: `jwt.verify(token, secret, { algorithms: ["HS256"] })`.

— **Розмір токена**: JWT передається у кожному запиті в заголовку Authorization. Великі payload (з багатьма claims) збільшують розмір кожного HTTP-запиту. Зберігати в payload лише необхідний мінімум даних.

— **Зберігання на клієнті**: у веб-додатках JWT зазвичай зберігається в localStorage (вразливий до XSS) або httpOnly cookie (вразливий до CSRF). Кожен варіант має свої компроміси безпеки.

n. Налаштувати кастомні сторінки помилок для автентифікації. Додати до конфігурації `jwt-demo` обробку помилок (рис. 5.n):

```
    # У блоці location /api/ додати:
    error_page 401 = @jwt_error;

    # Окремий location для обробки помилок автентифікації
    location @jwt_error {
        default_type application/json;
        return 401 '{"error":"unauthorized","message":"Invalid or missing
JWT token"}\n';
    }
```

Рис. 5.n – Кастомна JSON-відповідь для помилок JWT-автентифікації

Це дозволяє API повертати структуровану JSON-відповідь замість стандартної HTML-сторінки Nginx, що зручніше для клієнтів API.

```
sudo nginx -t && sudo systemctl reload nginx
curl -i http://jwt-demo.local/api/
```

Тепер замість стандартної HTML-сторінки 401 клієнт отримує JSON-об'єкт із полями `error` та `message`, що полегшує обробку помилок у коді клієнтського додатка.

o. Порівняти продуктивність різних методів автентифікації. Виконати навантажувальний тест за допомогою утиліти ab (Apache Benchmark):

```
sudo apt install -y apache2-utils
```

Тест Basic Auth (100 запитів, 10 одночасних):

```
ab -n 100 -c 10 -A user1:пароль http://auth-demo.local/protected/
```

Тест JWT-автентифікації:

```
TOKEN=$(cd ~/jwt-auth && node gen-token.js bench admin 5m | grep -A1
"токен:" | tail -1)
ab -n 100 -c 10 -H "Authorization: Bearer $TOKEN" http://jwt-
demo.local/api/
```

Порівняти значення Requests per second та Time per request для обох методів. JWT через auth_request повинен бути повільнішим, оскільки кожен запит генерує додатковий subrequest до Node.js-сервісу. Basic Auth перевіряється безпосередньо Nginx без зовнішніх запитів. Ця

різниця у продуктивності є важливим фактором при виборі методу автентифікації для високонавантажених систем.

6. Виконати порівняльний аналіз методів автентифікації.

На основі виконаних завдань 1–5 провести порівняльний аналіз усіх розглянутих методів автентифікації. Для кожного методу визначити складність налаштування, рівень безпеки, типові сценарії застосування, переваги та недоліки. Цей аналіз є важливою частиною практичної роботи, оскільки в реальних проєктах вибір методу автентифікації суттєво впливає на архітектуру, безпеку та зручність системи.

a. Заповнити порівняльну таблицю методів автентифікації за наступною структурою. Для кожного з п'яти методів (Basic Auth, mTLS, auth_request + API-ключ, OAuth2 Proxy + GitHub, JWT через auth_request) зазначити:

— Складність налаштування (низька / середня / висока);

— Рівень безпеки (низький / середній / високий);

— Потреба у зовнішніх залежностях (які саме);

— Типовий сценарій застосування (один-два приклади);

— Головні переваги (два-три пункти);

— Головні недоліки (два-три пункти).

b. Для кожного методу навести конкретний приклад реального проєкту або системи, де цей метод був би найбільш доцільним. Обґрунтувати вибір з урахуванням вимог до безпеки, масштабу системи та зручності для користувачів.

Наприклад: mTLS — міжсервісна комунікація у Kubernetes-кластері, де кожен pod має власний сертифікат, виданий cert-manager. Обґрунтування: автоматичне управління сертифікатами, висока безпека без участі користувача, відсутність паролів у конфігурації.

c. Відповісти на запитання: який метод або комбінацію методів обрали б для захисту REST API мобільного додатка з мільйонами користувачів? Обґрунтувати рішення з урахуванням:

— масштабування (як метод працює під навантаженням);

— зручності для кінцевого користувача (мінімальні дії для автентифікації);

— безпеки (стійкість до перехоплення, повторного використання, підробки);

— витрат на інфраструктуру (додаткові сервіси, бази даних, сертифікати).

d. Відповісти на запитання: який метод обрали б для захисту внутрішнього корпоративного порталу з інтеграцією Active Directory? Порівняти варіанти: Basic Auth + LDAP, OAuth2 Proxy + OIDC (Keycloak/Entra ID), mTLS із корпоративним CA. Обґрунтувати вибір.

e. Проаналізувати можливість комбінування кількох методів автентифікації в одній системі. Навести приклад архітектури, де одночасно використовуються: mTLS для міжсервісного трафіку, JWT для API-клієнтів, OAuth2 для веб-інтерфейсу користувача. Пояснити, як Nginx може маршрутизувати запити до різних механізмів автентифікації залежно від типу клієнта.

Приклад такої архітектури: у Nginx конфігурації різні location блоки можуть використовувати різні механізми auth_request. Наприклад, /api/v1/ перевіряє JWT-токен, /dashboard/ використовує OAuth2 Proxy, а /internal/ вимагає клієнтський сертифікат через mTLS. Така конфігурація дозволяє забезпечити оптимальний рівень безпеки для кожного типу клієнта.

Розглянути схему маршрутизації автентифікації (рис. 6.e):

```
# Архітектура мультиметодної автентифікації в Nginx
#
# Клієнти:
#   Мобільний додаток → /api/  → auth_request → JWT validator (порт 3002)
#   Веб-браузер       → /app/  → auth_request → OAuth2 Proxy (порт 4180)
#   Мікросервіс       → /svc/  → ssl_verify_client on (mTLS)
#   Адмін-панель      → /admin/→ auth_basic + allow/deny (Basic + IP)
#
# Кожен механізм обирається залежно від:
#   - типу клієнта (програма / людина)
#   - рівня довіри (внутрішній / зовнішній)
#   - вимог до UX (логін-форма / API-ключ / сертифікат)
```

Рис. 6.e – Схема мультиметодної автентифікації

Контрольні запитання

1. Яку роль відіграє директива auth_basic_user_file та у якому форматі зберігаються паролі у файлі .htpasswd?
2. У чому полягає різниця між satisfy any та satisfy all при комбінуванні методів контролю доступу?
3. Яким чином працює взаємна TLS-автентифікація (mTLS) і чим вона відрізняється від стандартного HTTPS?
4. Які переваги має директива ssl_verify_client optional порівняно з ssl_verify_client on?
5. Як працює механізм auth_request в Nginx? Яку роль відіграє ключове слово internal?
6. Яким чином auth_request_set дозволяє передавати дані від бекенда автентифікації до основного запиту?
7. У чому полягає принцип роботи OAuth2 і яку роль у ньому відіграє Authorization Code Flow?
8. Які три частини містить JWT-токен і яке призначення кожної з них?
9. Чому Basic Auth без HTTPS є небезпечним і як зловмисник може перехопити облікові дані?
10. Яку комбінацію методів автентифікації доцільно використовувати для захисту публічного API з мільйонами користувачів? Обґрунтувати відповідь.