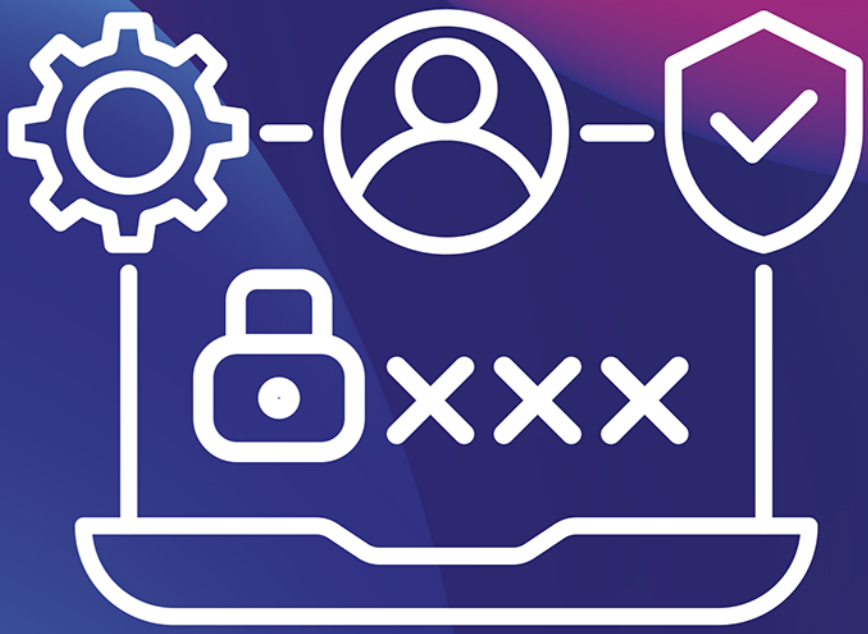

Jerzy Wawro

IDENTYFIKACJA I AUTORYZACJA

Poradnik administratora i inżyniera DevOps

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą AdobeStock.com.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/idaupa
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-289-3538-9

Printed in Poland.

Spis treści

ROZDZIAŁ 16.

Magazyny sekretów

Ilość sekretnych informacji (takich jak loginy, hasła, piny), jakie musimy zapamiętać, aby ich używać, ciągle rośnie. Reguły bezpieczeństwa sprawiają, że czas ważności sesji jest ograniczony i musimy te hasła wpisywać wielokrotnie. Chcemy też, by programy wykonywały za nas działania, które wymagają autoryzacji. W metodologii DevOps pojawiają się dodatkowe problemy z chronieniem sekretów: wpisane w kodzie hasła mogą pojawić się w repozytorium i zostać w ten sposób ujawnione osobom nieuprawnionym. Natomiast sekrety przekazane do kontenera poprzez zmienne systemowe (`env`) mogą zostać odczytane przez każdego, kto ma możliwość wykonania instrukcji `docker inspect`.

Niestety nie istnieje jeden standard ani dominujące (powszechnie przyjęte) rozwiązanie tych problemów. Dostępnych jest jednak szereg produktów, które warto poznać i stosować. Poniżej przedstawiono niektóre z nich.

16.1. ssh-agent

Program `ssh-agent` (*https://www.ssh.com/academy/ssh/agent*) zapamiętuje dane logowania do serwerów poprzez `ssh`. Program jest zawarty w pakiecie OpenSSH. Jeśli `ssh-agent` nie startuje automatycznie w naszym systemie, możemy dodać do pliku *.bashrc* polecenie uruchamiania agenta:

```
eval `ssh-agent -s`
```

w innej notacji:

```
eval $(ssh-agent -s)
```

Załóżmy, że mamy kilka serwerów, do których musimy się logować kilka razy dziennie. Wygenerowaliśmy sobie programem `ssh-keygen` parę kluczy z hasłem (*/home/demo/.ssh/test*).

Programem `ssh-copy-id` klucze zostały skopiowane na serwery, aby umożliwić logowanie (zob. rozdział poświęcony `ssh`):

```
ssh -i /home/demo/.ssh/test demo@server_ip
```

Za każdym razem musimy podawać hasło chroniące klucz — chyba że skorzystamy z `ssh-agent`. Wtedy hasło jest potrzebne tylko przy dodawaniu klucza:

```
ssh-add /home/demo/.ssh/test
```

Jak działa autoryzacja ssh z użyciem agenta? Zauważ, że w trakcie autoryzacji — jak to opisano w rozdziale dotyczącym SSH, następuje podpisanie kluczem prywatnym danych

identyfikacyjnych wysyłanych do serwera. Klient prosi o wykonanie podpisu agenta SSH i po otrzymaniu podpisanej wiadomości wysyła go na serwer.

Protokół agenta jest bardzo prosty. Wykonuje on tylko kilka podstawowych działań:

- Dodaje zwykłą parę kluczy (klucze publiczne i odszyfrowane klucze prywatne).
- Dodaje ograniczoną parę kluczy (klucze publiczne i odszyfrowane klucze prywatne).
- Dodaje klucz (zwykły lub ograniczony) z karty inteligentnej (tylko klucz publiczny).
- Usuwa klucz.
- Podaje listę przechowywanych kluczy.
- **Podpisuje wiadomość przechowywanym kluczem.**
- Blokuje lub odblokowuje całego agenta hasłem.

Ograniczony klucz (*https://smallstep.com/blog/ssh-agent-explained/*) to zwykle klucz o ograniczonym okresie istnienia lub taki, którego użycie wymaga wyraźnej akceptacji użytkownika.

Komenda `ssh-add` wykonuje wszystkie powyższe operacje **agenta** z wyjątkiem podpisywania. Gdy uruchomisz `ssh-add` bez żadnych parametrów, przeskanuje Twój katalog domowy w poszukiwaniu pewnych standardowych kluczy i doda je do Twojego agenta.

Domyślnie szuka:

~/.ssh/id_rsa

~/.ssh/id_ed25519

~/.ssh/id_dsa

~/.ssh/id_ecdsa

16.1.1. gpg-agent

Podobnie jak w przypadku `ssh-agent`, głównym celem jest brak konieczności wpisywania hasła za każdym razem, gdy używamy klucza. Program `gpg-agent` działa w tle (jako demon) i przechowuje w pamięci tajne klucze GPG. Gdy proces GPG potrzebuje klucza, kontaktuje się z uruchomionym programem `gpg-agent` przez gniazdo i wysyła odpowiednie żądanie (na przykład PKSIGN — podpisania skrótu). Wykaz wszystkich funkcji protokołu komunikacyjnego GPG znajdziesz tutaj: *https://www.gnupg.org/documentation/manuals/gnupg/Agent-Protocol.html*.

Jeśli proces agenta nie posiada klucza, spróbuje załadować zaszyfrowany klucz z Twojego zbioru kluczy i poprosi o hasło do jego rozszyfrowania.

Oprócz kluczy GPG `gpg-agent` może przechowywać klucze SSH i zastępować `ssh-agent`. Należy w tym celu uruchomić agenta z parametrem `--enable-ssh-support`.

Na przykład (w pliku *.bashrc*):

```
eval $(gpg-agent --daemon --enable-ssh-support --sh)
```

16.1.2. Bezpieczeństwo agentów

Ustawienia programu ssh-agent można znaleźć w zmiennych systemowych. Na przykład:

```
$ set | grep SSH
SSH_AGENT_PID=9178
SSH_AUTH_SOCK=/tmp/ssh-5IisbE2xeoQO/agent.9571
```

Widzimy, że program tworzy do komunikacji gniazdo systemu Unix w katalogu */tmp*.

To nie jest problem, jeśli sami korzystamy z maszyny, na której uruchomiono agenta. Jeśli jednak loguje się na niej inna osoba, może ona sprawdzić zawartość katalogu */tmp* i jeśli ma wystarczające uprawnienia, skorzystać z zostawionych w nim parametrów logowania:

```
$ SSH_AUTH_SOCK=/tmp/ssh-5IisbE2xeoQO/agent.9571 ssh-add -l
2048 SHA256:N8V9Zaxis/PXXXXXX /home/test/.ssh/klucz (RSA)

# Po ustawieniu zmiennej wystarczy proste wywołanie ssh
$ SSH_AUTH_SOCK=/tmp/ssh-5IisbE2xeoQO/agent.9571 ssh test@server
```

Wykorzystując dodatkowo program `socat`, można podsłuchać komunikację z `ssh-agent`.

To oczywiście nie jest możliwe, jeśli użytkownicy nie mają uprawnień `root`, a gniazdo ma uprawnienia tylko dla właściciela. Jednak powinniśmy chronić swoje hasła — nawet przed administratorami. Nie jest więc wskazane używanie `ssh-agent` na komputerach używanych przez wiele osób równocześnie. Także przekierowanie (*forwarding*) — które dla `ssh-agent` jest możliwe — niesie ze sobą niebezpieczeństwo.

Podobne problemy występują w przypadku `gpg-agenta`. W systemie Windows dodatkowo występuje problem zbyt długiego czasu przechowywania rozszyfrowanych kluczy: *https://security.szurek.pl/en/gpg-reaper-steal-gpg-private-keys*.

16.2. Plik z hasłami

Opisany wcześniej program GPG może służyć nie tylko do szyfrowania maili (do czego pierwotnie go wykorzystywano), ale na przykład do szyfrowania plików z hasłami.

W celu przetestowania tej możliwości wygenerujemy testowy klucz. Możemy wcześniej usunąć katalog (*~/.gnupg*) ze starymi hasłami, ale trzeba pamiętać, by równocześnie zabić demona agenta (przydaje się, jeśli zapomniałeś hasła):

```
gpgconf --kill gpg-agent
rm ~/.gnupg
gpg --gen-key
```

Testujemy (oczywiście zamiast jurek@example.com podaj użyty przy generowaniu e-mail):

```
gpg --list-keys
gpg --with-wkd-hash --fingerprint jurek@example.com
```

Skrypt, który tworzy testowy plik (*/tmp/test.txt*), szyfruje go kluczem gpg, a następnie go rozszyfrowuje:

```
HOMEDIR="/home/jurek/.gnupg/"
FILENAME="/tmp/test.txt"
ENCRYPTED="/tmp/test.asc"
RECIPIENT="jurek@tenar.pl"

echo "Testowy plik" > $FILENAME

rm $ENCRYPTED

gpg2  --armor -vv \
     --homedir $HOMEDIR \
     --recipient $RECIPIENT \
     --output $ENCRYPTED \
     --encrypt $FILENAME


ls -l $ENCRYPTED

echo "OK - teraz rozszyfrowanie"


gpg2         --armor  --no-version \
            --always-trust \
            --homedir $HOMEDIR \
            --decrypt $ENCRYPTED
```

Jeśli uruchomiony jest `gpg-agent`, program może nie pytać o hasło (zwraca się do agenta). Używamy `gpg2`, gdyż `gpg` nie ma opcji `--output`.

Użyte parametry:

- `--armor` — w wersji ASCII;
- `-vv` — zapobiega szukaniu przez WKD, można też użyć `--auto-key-locate →local`;
- `--always-trust` — pomiń walidację.

Program GPG możemy obudować skryptem w Pythonie, zarządzającym hasłami. Poniżej przykład takiego skryptu, za pomocą którego możemy zapamiętać hasła do kluczy prywatnych używanych w ssh.

```
import os
import subprocess
import json
import argparse
import pexpect
from pathlib import Path

RECIPIENT = "jurek@example.com"
GPG_BIN = 'gpg2'
PFILE = '.pass-store'

class CertsPasswordStore:
    def __init__(self, path=None):
        if path is None:
            self.store_path = Path.home() / PFILE
```

```
        else:
            self.store_path = Path(path).expanduser()
        self.passwords = []
        if self.store_path.exists():
            self.decrypt_passwords()

    def add_password(self, ident, password):
        # Sprawdzamy, czy identyfikator już istnieje, jeśli tak - aktualizujemy
        self.passwords = [p for p in self.passwords if p['cert'] != ident]
        self.passwords.append({'cert': ident, 'password': password})

    def encrypt_passwords(self):
        gpg = subprocess.Popen(
            [GPG_BIN, '-e', '--recipient', RECIPIENT, '--batch', '--yes',
               '-o', str(self.store_path)],
            stdin=subprocess.PIPE
        )
        gpg.stdin.write(json.dumps(self.passwords).encode())
        gpg.stdin.close()
        gpg.wait()
        if gpg.returncode != 0:
            raise Exception(f'Błąd szyfrowania pliku {self.store_path}')

    def decrypt_passwords(self):
        gpg = subprocess.Popen(
            [GPG_BIN, '--quiet', '--batch', '--decrypt'],
            stdin=open(self.store_path, 'rb'),
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE
        )
        stdout, stderr = gpg.communicate()
        if gpg.returncode == 0:
            self.passwords = json.loads(stdout)
        else:
            raise Exception(f'Błąd deszyfrowania: {stderr.decode()}')

def add_to_ssh_agent(cert_name, password):
    # Używa pexpect do automatycznego wpisania hasła do ssh-add.
    key_path = Path.home() / '.ssh' / cert_name
    if not key_path.exists():
        print(f"[-] Pomijam: Plik {key_path} nie istnieje.")
        return
    print(f"[*] Dodawanie klucza do agenta: {cert_name}")
    # Uruchomienie ssh-add dla konkretnego pliku
    child = pexpect.spawn(f'ssh-add {str(key_path)}')
    try:
        # Czekamy na prośbę o hasło (obsługa różnych wariantów językowych)
        index = child.expect(['Enter passphrase', 'Enter password',
                 r'[Hh]as.o', 'passphrase'], timeout=5)
        child.sendline(password)
        child.expect(pexpect.EOF)
        # Sprawdzenie wyniku po wysłaniu hasła
        output = child.before.decode() if child.before else ""
        if "Identity added" in output or child.exitstatus == 0:
            print(f"[+] Sukces: {cert_name} dodany.")
        else:
            print(f"[!] Błąd przy dodawaniu {cert_name}: {output.strip()}")
```

```
    except pexpect.TIMEOUT:
        print(f"[?] Timeout: {cert_name}. Może klucz nie ma hasła?")
    except pexpect.EOF:
        print(f"[+] Klucz {cert_name} został przetworzony (EOF).")

if __name__ == "__main__":
    usage = "Magazyn haseł SSH oparty na GPG"
    parser = argparse.ArgumentParser(description=usage)
    parser.add_argument('-i', '--init', help='Zainicjuj pusty magazyn', action="store_true")
    parser.add_argument('-a', '--add', help='Nazwa pliku klucza w ~/.ssh/')
    parser.add_argument('-p', '--password', help='Hasło do tego klucza')
    args = parser.parse_args()

    store = CertsPasswordStore()
    if args.init:
        store.encrypt_passwords()
        print(f"[+] Zainicjowano plik: {store.store_path}")
    elif args.add and args.password:
        store.add_password(args.add, args.password)
        store.encrypt_passwords()
        print(f"[+] Dodano hasło dla {args.add} i zaszyfrowano magazyn.")
    else:
        # Główny tryb: odszyfruj i dodaj wszystko do agenta
        if not store.passwords:
            print("[-] Magazyn jest pusty lub nie istnieje.")
        else:
            for p in store.passwords:
                add_to_ssh_agent(p['cert'], p['password'])
```

Skrypt uruchomiony z parametrami `-a -p` zapamiętuje hasła powiązane z nazwami kluczy w pliku *~/.pass-store*. Z parametrem `-i` inicjuje pusty magazyn, a bez parametrów uruchamia agenta `ssh` i przekazuje mu hasła.

Skrypt wymaga uruchomienia PGP z kluczami do podpisu i szyfrowania.

Przykład:

```
sudo apt install gnupg gpg-agent gnupg2
# Generowanie głównego klucza (signing)
gpg --quick-gen-key "jurek <jurek@example.com>" ed25519 sign 2y
# Pobranie odcisku palca nowo utworzonego klucza
fpr=$(gpg --list-secret-keys --with-colons "jurek <jurek@example.com>" | awk -F: '/^fpr/
{print $10; exit}')
# Dodanie subklucza (encryption) przy użyciu odcisku
gpg --quick-add-key "$fpr" cv25519 encr 2y
echo "sprawdź, czy masz klucz do podpisu i szyfrowania: [SC] i [E]"
gpg --list-keys jurek
#---------------
# Zainicjowanie pliku z hasłami:
./gpg_ssh.py -i
# dodanie hasła do certyfikatu ~/.ssh/certyfikat1
./gpg_ssh.py -a certyfikat1 -p hasło1
# Pobranie  z pliku haseł i dodanie do ssh_agent:
./gpg_ssh.py
```

Bardziej rozbudowany program tego typu (password manager) można znaleźć na stronie *https://github.com/marcwebbie/passpie*.

16.2.1. KeePass

Ciekawym programem do zapamiętywania haseł jest KeePass. Otwartą wersję dla wielu systemów znajdziesz na stronach *https://github.com/keepassxreboot/keepassxc*, *https://keepassxc.org/*.

Program można zintegrować z przeglądarką internetową i wykorzystać do zapamiętywania różnych sekretów używanych na stronach WWW.

W Pythonie dostępna jest biblioteka, która umożliwia odszukiwanie w pliku haseł. Testowy skrypt:

```
from pykeepass import PyKeePass

kp = PyKeePass('db.kdbx', password='haslo')
for entry in kp.find_entries(title='https://www.az.pl/login/'): #,
first=True):
  print(entry.title)
  print(entry.username)
  print(entry.password)
```

Można to wykorzystać, modyfikując wcześniej podany skrypt (do zapamiętywania haseł kluczy gpg).

16.2.2. Alternatywne rozwiązania

Istnieje całkiem sporo rozwiązań klasy KeePass. Na przykład programy rozwijane jako open source.

- *https://square.github.io/keywhiz/* — pamięta hasła w bazie mysql, licencja Apache, Java.
- *https://github.com/pinterest/knox* — baza sql (domyślnie — w pamięci), licencja Apache, język Go.
- *https://lyft.github.io/confidant/* — pamięta hasła w DynamoDB, Python, licencja Apache.
- *https://github.com/marcwebbie/passpie* — plik YAML szyfrowany przez PGP, Python, MIT.

Rozwój tego typu produktów zmierza do stworzenia architektury klient-serwer. Serwer obsługuje magazyn sekretów oraz API, poprzez które klient zapisuje i odczytuje sekrety. Ma to szczególne znaczenie w budowaniu bezpiecznych kontenerów. Serwer w jednym z kontenerów może służyć jako źródło sekretów dla pozostałych.

Na przykład Pinteresy Knox udostępnia implementacje klienta i serwera — https://github.com/pinterest/knox/tree/master/cmd. **UWAGA! Nie są to programy gotowe do zastosowania w wersji produkcyjnej.** Posługując się tymi przykładami, musimy sami zaimplementować odpowiednie rozwiązania (wymagana konfiguracja bazy danych i certyfikatów).

Powyższe przykłady mogą nam posłużyć do testów. Niestety nawet instrukcja instalacji na stronie GitHuba nie jest w pełni poprawna (stan na 30 września 2021). Poniżej krótki opis przeprowadzenia testu:

1. Zainstaluj Go w wersji co najmniej 1.15 (*https://golang.org/doc/install*).
2. Zainstaluj i uruchom serwer (Linux, Go 1.17).

```
# Wykonaj jako root / sudo
export PATH=$PATH:/usr/local/go/bin
export GOPATH=/usr/local/go
export GO17VENDOREXPERIMENT=1
go get -d github.com/pinterest/knox
go install github.com/pinterest/knox/cmd/dev_server@latest
```

3. Zainstaluj klienta.

```
# Wykonaj jako root / sudo
export PATH=$PATH:/usr/local/go/bin
export GOPATH=/usr/local/go
export GO17VENDOREXPERIMENT=1
go install github.com/pinterest/knox/cmd/dev_client@latest
```

4. Uruchom serwer.

```
export GOPATH=/usr/local/go
$GOPATH/bin/dev_server
```

5. Wygeneruj token z uprawnieniami read:org na stronie *https://github.com/settings/tokens* i uruchom test klienta (w innej instancji konsoli).

```
export GOPATH=/usr/local/go
export KNOX_USER_AUTH=<wygenerowany token>
echo -n "My first knox secret" | $GOPATH/bin/dev_client create
→ test_service:first_secret
```

Dalsze testy:

```
export GOPATH=/usr/local/go
export KNOX_USER_AUTH=<wygenerowany token>
#You can retrieve the secret using:
$GOPATH/bin/dev_client get test_service:first_secret
# as JSON
$GOPATH/bin/dev_client get -j test_service:first_secret
#You can see all key IDs using:
$GOPATH/bin/dev_client keys
```

16.2.3. Próba standaryzacji — SPIFFE

Opisane wcześniej mechanizmy ochrony sekretów nie narzucają żadnych standardów dotyczących struktury chronionych informacji. W systemie zbudowanym z wielu kontenerów korzystających ze wspólnych danych chronionych konieczne jest dokonanie pewnych dodatkowych ustaleń. Standaryzacja nazw (identyfikatorów) dla chronionych informacji oraz API pozwalającego na ich odczyt umożliwiłaby współdziałanie kontenerów tworzonych przez niezależne zespoły.

Próbą takiej standaryzacji jest SPIFFE (*https://www.cncf.io/blog/2020/06/22/toc-approves-spiffe-and-spire-to-incubation/*).

SPIFFE definiuje:

- identyfikatory dla chronionych informacji,
- dokument identyfikacyjny (*SPIFFE Verifiable Identity Document*, SVID),
- API (SPIFFE Workload API, *https://github.com/spiffe/spiffe/blob/main/standards/SPIFFE_Workload_API.md*).

Identyfikatory mają strukturę zgodną z URI: `spiffe://trust-domain-name/path`.

SVID może być zbudowany w oparciu o certyfikaty (X-509, *https://github.com/spiffe/spiffe/blob/main/standards/X509-SVID.md*) albo tokeny JWT (*https://github.com/spiffe/spiffe/blob/main/standards/JWT-SVID.md*).

SPIFFE API zostało wyspecyfikowane w języku Go (implementacja: *https://github.com/spiffe/spire*). Implementacja SPIFFE została wykorzystana w projekcie *https://github.com/pinterest/knox*. Dzięki temu Knox może być pomocne w ochronie danych w otwartym środowisku chmurowym (*https://medium.com/pinterest-engineering/secret-management-in-multi-tenant-environments-debc9236a744*).

16.3. Sekrety w Docker Swarm

Docker w trybie Swarm (*https://docs.docker.com/engine/swarm/*, stan na sierpień 2025) umożliwia tworzenie magazynu sekretów (*secrets*). Sekrety są przechowywane w postaci zaszyfrowanej. Gdy usługa (*service*) uzyskuje dostęp do sekretu, jest on odszyfrowywany w pamięci i udostępniany poprzez system plików montowany w ścieżce */run/secrets/<secret_name>* (dla systemu Linux) lub C: (dla systemu Windows). Można także zdefiniować inną ścieżkę.

W systemie Linux odszyfrowane sekrety są przechowywane w pamięci (tmpfs), co zwiększa bezpieczeństwo. W systemie Windows sekrety zapisywane są na dysku, co obniża bezpieczeństwo w porównaniu z Linuksem. Szczegółowe różnice implementacyjne między systemami Linux i Windows opisano w dokumentacji (*https://docs.docker.com/engine/swarm/secrets/*, stan na sierpień 2025).

Uwaga: Swarm jest zintegrowany z Dockerem, ale dostęp do sekretów możliwy jest tylko dla usług uruchamianych za pomocą `docker service` lub `docker stack`, a nie dla kontenerów uruchamianych poleceniem `docker run`. Sekrety można definiować i udostępniać tylko w zainicjowanym trybie Swarm.

16.3.1. Wymagania

- Docker 20+ zainstalowany.
- Python 3.12+ dla przykładu.
- System operacyjny: zalecany Linux dla wyższego bezpieczeństwa.

16.3.2. Inicjalizacja klastra Swarm

Aby rozpocząć, zainicjuj klaster Swarm na serwerze (*node*):

```
docker swarm init
```

Jeśli chcesz dodać worker node do klastra, użyj polecenia `docker swarm join --token <token> <manager-ip>:2377` (token uzyskasz z `docker swarm join-token worker`).

16.3.3. Tworzenie sekretów i zarządzanie nimi

Tworzenie sekretu z `stdin` (kreska - oznacza odczyt z `stdin`):

```
printf "my_super_secret_password" | docker secret create my_secret -
```

Alternatywnie sekret można utworzyć z pliku:

```
docker secret create my_secret /path/to/secret.txt
```

Wyświetlenie listy sekretów:

```
docker secret ls
```

Szczegółowe informacje o sekrecie:

```
docker secret inspect my_secret
```

Usuwanie sekretu (ostrożnie, operacja nieodwracalna):

```
docker secret rm my_secret
```

16.3.4. Przykładowy skrypt Pythona

Poniższy skrypt Pythona odczytuje sekret i wyświetla go w przeglądarce za pomocą prostego serwera HTTP:

```
#!/usr/bin/env python3
from http.server import SimpleHTTPRequestHandler
import socketserver
import logging

def get_secret():
    secret_fpath = '/run/secrets/my_secret'
    try:
        with open(secret_fpath, 'r') as f:
            return f.read().strip()
    except FileNotFoundError:
        return 'not exists!!'

class MyHandler(SimpleHTTPRequestHandler):
    def do_GET(self):
        logging.info("GET request,\nPath: %s\nHeaders:\n%s\n", str(self.path),
        ↪str(self.headers))
        self.send_response(200)
        self.send_header('Content-Type', 'text/html')
        self.end_headers()
        message = 'Secret: ' + get_secret()
        self.wfile.write(bytes(message, 'utf8'))
```

```
PORT = 8080

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s')
with socketserver.TCPServer(("", PORT), MyHandler) as httpd:
    httpd.serve_forever()
```

Uruchom skrypt lokalnie i przetestuj w przeglądarce pod adresem *http://localhost:8080*. Wyświetli się komunikat: „Secret: not exists!!!" (bo sekret jest dostępny tylko w usłudze Swarm).

16.3.5. Dockerfile

Plik `Dockerfile` do zbudowania obrazu:

```
FROM python:3.12-slim
WORKDIR /app
COPY srv.py /app
RUN chmod +x /app/srv.py
CMD ["python", "/app/srv.py"]
```

Budowanie obrazu:

```
docker build -t py3:latest .
```

16.3.6. Uruchomienie usługi Swarm

Plik *swarm-secret-stack.yml* definiuje usługę z dostępem do sekretu:

```
version: "3.9"
services:
  demo1:
    image: py3:latest
    secrets:
      - my_secret
    ports:
      - target: 8080
        published: 85
        mode: host
    deploy:
      replicas: 1
secrets:
  my_secret:
    external: true
```

Uruchomienie stacka:

```
docker stack deploy -c swarm-secret-stack.yml demo1
```

Sprawdź usługę w przeglądarce — *http://localhost:85*. Wyświetli się sekret: „Secret: my_super_secret_password".

Podgląd logów:

```
docker service logs demo1_demo1 -f
```

Nazwa serwisu to `<stack_name>_<service_name>`, np. `demo1_demo1` (sprawdź `docker service ls`).

16.3.7. Najlepsze praktyki bezpieczeństwa

- **Nigdy** nie zapisuj wprost sekretów w kodzie lub pliku Dockerfile — używaj Swarm secrets.
- W produkcji używaj zewnętrznych narzędzi, np. HashiCorp Vault (zob. dokumentację na stronie *https://docs.docker.com/engine/swarm/secrets/#use-external-secrets*).
- Ogranicz dostęp — ustaw `deploy: mode: manager` w YAML, aby usługa działała tylko na menedżerach.
- Rotacja sekretów — twórz nowe wersje i aktualizuj usługi:
  ```
  docker service update --secret-rm old_secret --secret-add new_secret demo1_demo1
  ```
- Monitoruj zmiany poprzez `docker events --filter 'type=secret'`.

16.3.8. Częste problemy

- **Błąd „no such secret”**. Sprawdź, czy sekret istnieje (`docker secret ls`).
- **Porty nie działają**. Upewnij się, że firewall pozwala na ruch (np. `ufw allow 85`).
- **IPv4 vs IPv6**. Jeśli porty nie działają, spróbuj:
  ```
  sudo sysctl -w net.ipv6.conf.all.forwarding=1
  ```
  i zrestartuj sieć (w Debianie/Ubuntu edytuj */etc/default/docker-network*).
- **W Windowsie** sekrety są zapisywane na dysku — unikaj w produkcji.

16.3.9. Porównanie sekretów Swarm dla Linuksa i Windowsa

ASPEKT	LINUX	WINDOWS
Przechowywanie sekretów	W pamięci (tmpfs)	Na dysku
Bezpieczeństwo	Wyższe	Niższe (ryzyko wycieku)
Dostęp	*/run/secrets/*	*C:\ProgramData \Docker\secrets*

16.3.10. Źródła

- Oficjalna dokumentacja Docker Swarm, *https://docs.docker.com/engine/swarm/*, stan na sierpień 2025.
- Dokumentacja sekretów, *https://docs.docker.com/engine/swarm/secrets/*, stan na sierpień 2025.
- Rozwiązywanie problemów z portami, (*https://stackoverflow.com/questions/29957143/make-docker-use-ipv4-for-port-binding*.

16.4. Sekrety Kubernetesa

Podobnie jak dla Dockera Swarm, sekrety mogą być pamiętane na serwerze Kubernetes.

Wzorując się na przykładzie *https://kubernetes.io/docs/tasks/configmap-secret/managing-secret-using-kubectl/*, możemy stworzyć testowy sekret na serwerze (tu: DigitalOcean):

```
kubectl create secret generic admin-pass \
 --from-file=./username.txt \
 --from-file=./password.txt
```

Pliki *username.txt* i *password.txt* zawierają chronione dane.

Uzyskujemy sekret w strukturze JSON:

```
{
 "kind": "Secret",
 "apiVersion": "v1",
 "metadata": {
  "name": "admin-pass",
  "namespace": "default",
  "selfLink": "/api/v1/namespaces/default/secrets/admin-pass",
  "uid": "e3bb1c5e-43e6-4bb0-9246-60d2ba89c269",
  "resourceVersion": "62837062",
  "creationTimestamp": "2021-09-23T12:07:39Z"
 },
 "data": {
  "password.txt": "xxxxxxxxxxxxxxxxxxxxxx=",
  "username.txt": "YWRtaW4="
 },
 "type": "Opaque"
}
```

W pliku *username.txt* był wpisany identyfikator „admin", co możemy sprawdzić poleceniem:

```
echo "YWRtaW4="| base64 --decode
```

Możemy też przygotować JSON z definicją sekretu i wysłać go na serwer poleceniem:

```
kubectl apply -f secret2.json
```

Manifest sekretu w formacie json (plik *secret2.json*) będzie wyglądał bardzo podobnie jak powyżej:

```
{
"kind": "Secret",
"apiVersion": "v1",
"metadata": {
  "name": "admin2-pass",
  "namespace": "default",
  "selfLink": "/api/v1/namespaces/default/secrets/admin2-pass"
 },
"data": {
  "password.txt": "xxxxxxxxxxxxxx",
  "username.txt": "YWRtaW4yCg=="
 },
  "type": "Opaque"
}
```

Sekrety mogą być widoczne poprzez system plików (jak w Dockerze Swarm) albo poprzez zmienne środowiskowe. Przetestujmy odczyt (podobnie jak poprzednio) programem Python zaszytym w kontenerze. Zakładamy, że sekrety będą zapisywane do katalogu */secrets*.

Program odczytu (*srv1.py*) udostępnia sekret na porcie 8088:

```
#!/usr/bin/env python3

import os
import http.server as SimpleHTTPServer
import socketserver as SocketServer
import logging

def get_secret():
  secret_fpath = '/secrets/password.txt'
  if os.path.exists(secret_fpath):
    s = open(secret_fpath).read().rstrip('\n')
    return s
  else:
    return 'not exists!!'

class MyHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):

  def do_GET(self):
    logging.info("GET request,\nPath: %s\nHeaders:\n%s\n", str(self.path),
str(self.headers))
    self.send_response(200)
    self.send_header('Content-Type', 'text/html')
    self.end_headers()
    message = 'Sekret: ' + get_secret()
    self.wfile.write(bytes(message, 'utf8'))
    return

PORT=8088

logging.basicConfig(level=logging.INFO)
with SocketServer.TCPServer(("", PORT), MyHandler) as httpd:
    httpd.serve_forever()
```

Definicja Dockera (Dockerfile):

```
FROM python:3.8-slim

WORKDIR /app

COPY srv1.py /app
EXPOSE 8088
CMD [ "python", "/app/srv1.py" ]
```

Przygotowujemy obraz:

```
docker build -t py3a .
```

Tu sprawy się nieco komplikują, bo trudno jest zbudować *POD* bez odwołania do rejestru (*Registry*) obrazów. Jeśli nie chcemy korzystać z *https://hub.docker.com/*, musimy mieć rejestr prywatny. Można też sprawdzić, czy czasem nasz provider internetu nie udostępnia takiej usługi.

Hosting DigitalOcean, na którym były przeprowadzane testy, taką usługę udostępnia (niewielki rejestr jest za darmo).

Możemy więc wykonać (*https://docs.docker.com/engine/reference/commandline/push/*):

```
docker image tag py3a:latest registry.digitalocean.com/mojrejestr/py3a:latest
docker push registry.digitalocean.com/mojrejestr/py3a:latest
```

Do testów można spróbować pobrać obraz:

```
docker pull registry.digitalocean.com/mojrejestr/py3a
```

Możemy przystąpić do tworzenia poda. Definicja w pliku *demo1s.yaml*:

```
apiVersion: v1
kind: Pod
metadata:
  name: py3a
spec:
  containers:
  - name: py3a
    image: registry.digitalocean.com/mojrejestr/py3a:latest
    ports:
    - containerPort: 8088
      name: http
      protocol: TCP
    volumeMounts:
      - name: demo-secret-volume
        mountPath: "/secrets/"
        readOnly: true
  volumes:
    - name: demo-secret-volume
      secret:
        secretName: admin2-pass
```

Zwróć uwagę na punkt montowania sekretów (`mountPath`) oraz definicję portu kontenera.

Uruchamiamy poda:

```
kubectl apply -f demo1s.yaml
```

Sprawdzamy:

```
kubectl get pods -o wide
```

Ważny jest parametr `-o wide`. Pozwala odczytać lokalny adres IP, pod którym możemy wykonać testowe połączenie. Wynik powyższej instrukcji:

```
NAME   READY   STATUS    RESTARTS   AGE   IP             NODE      ...
py3a   1/1     Running   0          40m   10.244.0.152   ...
```

Na komputerze węzła (lokalnie) możemy przetestować otwarcie strony — na przykład programem `curl` lub `lynx`:

```
lynx 10.244.0.152:8088
```

Powinien pojawić się odczytany sekret.

Oczywiście w wersji produkcyjnej sekret nie może być w ten sposób ujawniany! Także adres IP uzyskany jak powyżej w typowym przypadku nie będzie wykorzystywany. Zamiast tego musimy stworzyć serwis (*service*) i ewentualnie Ingress.

Powyższy przykład służy jedynie do przetestowania możliwości dostępu do sekretów Kubernetesa.

Kompleksowy przewodnik po rozwiązaniach uwierzytelniających

Ten podręcznik jest przeznaczony przede wszystkim dla administratorów i deweloperów systemów informatycznych. Dotyka niezwykle istotnego zagadnienia, przed którym stoją inżynierowie DevOps: jak sprawić, by poprzez autoryzację chronić dostęp do danych i serwisów, gdy programy komunikują się ze sobą — często bez udziału człowieka?

Autor wyjaśnia teoretyczne podstawy i prezentuje praktyczne rozwiązania z wykorzystaniem technologii kontenerów Docker. Ta technologia bowiem doskonale nadaje się do testowania i prezentowania licznych przykładów zawartych w podręczniku.

Książka opisuje nowoczesne rozwiązania, kluczowe dla rozwoju systemów informatycznych w Polsce. Dla osób posługujących się językiem Python jest to pozycja niezbędna.

- **Poznaj fundamentalne różnice między identyfikacją, autoryzacją a uwierzytelnianiem i sposoby ich implementacji**
- **Dowiedz się, jaką rolę odgrywają szyfry i algorytmy ważne w procesie uwierzytelnienia**
- **Opanuj wiedzę dotyczącą technologii i metod związanych z ochroną serwerów, a także funkcjonowania ich najważniejszych elementów, takich jak poczta (SMTP), VPN, LDAP, Kerberos, PAM**
- **Zapoznaj się z nowoczesnymi metodami uwierzytelniania, takimi jak weryfikacja dwuetapowa, standardy SAML i OAuth 2**
- **Zobacz, jak te rozwiązania zostały zastosowane w usługach wprowadzanych przez polskie prawo, w tym e-Doręczenia, KWIE, KSeF**

Jerzy Wawro jest inżynierem z bogatym doświadczeniem we wdrażaniu i w utrzymaniu dużych systemów informatycznych. Zajmuje się optymalizacją i cyfryzacją procesów w przedsiębiorstwach i na uczelniach, rozwojem e-learningu i nauczania zdalnego, a także sztuczną inteligencją. Brał udział w wielu projektach realizowanych w nowoczesnych technologiach, w tym w projekcie kryptograficznej waluty niefiducjarnej e-Talar. Jest autorem licznych publikacji i projektów otwartych — między innymi popularnego podręcznika *Uczymy się programować w Pythonie*. Założyciel i prezes fundacji Galicea, która promuje zastosowania technologii dla rozwoju społecznego (*galicea.pl*).

Helion

helion.pl

HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej! ▶

ISBN 978-83-289-3538-9

Cena: 99,00 zł