

## Polecenie które pomaga, gdy już jest bieżąca sesja zapisana:

`podman rm numer`

-co gdy server nie działa bo bieżąca sesja już jest?

`podman ps -a`

szukamy naszego:

972ed565671c docker.io/mazurkatarzyna/hashing-md5-ex1:latest python app.py 14 minutes ago Up  
14 minutes ago 0.0.0.0:10001->10001/tcp hashingmd5ex1

następnie └─ \$podman stop 972e

i teraz możemy usunąć sesję podman rm numer

i jeszcze raz odpalić!

## Zajęcia 1

### Podłączenie do serwera, wyciąganie danych z niego, dodawanie ich

Poniższe zadania służą do zapoznania się z działaniem 2 narzędzi, których będziemy często używać. Są to [cURL](#) oraz [jq](#). Będziemy korzystać z serwera, który przechowuje dane w poniżej postaci (w formacie [JSON](#)):

```
users = {  
    "alice": {  
        "name": "Alice Smith",  
        "email": "alice@example.com",  
        "age": 25,  
        "city": "Warsaw",  
        "hobbies": ["reading", "chess", "cycling"]  
    },  
    "bob": {  
        "name": "Bob Johnson",  
        "email": "bob@example.com",  
        "age": 17,  
        "city": "Krakow",  
        "hobbies": ["football", "games"]  
    }  
}  
  
items = [  
    {"id": 1, "name": "itemX", "price": 15.99},  
    {"id": 2, "name": "superItem", "price": 99.90},  
    {"id": 3, "name": "item123", "price": 5.49},  
    {"id": 4, "name": "itemY", "price": 29.00}  
]
```

Po uruchomieniu serwer działa pod adresem <http://127.0.0.1:5000>. Udostępniane przez serwer endpointy możesz sprawdzić pod adresem <http://127.0.0.1:5000/docs>. Możesz je tam również przetestować.

Uruchom serwer za pomocą polecenia:

```
docker run -p 5000:5000 mazurkatarzyna/curl-jq-server:latest
```

W pierwszym terminalu odpalimy server:

```
podman run -p 5000:5000 docker.io/mazurkatarzyna/curl-jq-server:latest
```

W drugim będziemy wypisywać polecenia

Wchodzacy na server, mozemy sobie kopowac polecenia z niego

The screenshot shows a web browser window with the URL <http://127.0.0.1:5000/docs/>. The page title is "JQ + CURL Practice Server 1.0 OAS 3.0". Below the title, it says "Server for training curl and jq commands with JSON APIs." A "Servers" dropdown menu is open, showing "http://localhost:5000". The main content area displays a list of API endpoints under the "default" section:

- GET /user/{username} Get user by username
- GET /items List all items
- POST /items Add a new item
- PUT /items/{item\_id} Update an item
- DELETE /items/{item\_id} Delete an item
- GET /status Server status
- POST /echo Echo received JSON

Below the list, there is a note: "man run -p 5000 docker.io/mazurkatarzyna/curl-jq-server:latest".  
A detailed view of the "/user/{username}" endpoint is shown in a modal dialog:

- Parameters**:

Name	Description
username <span style="color:red;">*</span> required	string (path) alice
- Responses**:
  - Curl**:

```
curl -X 'GET' \
'http://localhost:5000/user/alice' \
-H 'accept: */*'
```
  - Request URL**:

```
http://localhost:5000/user/alice
```
  - Server response**:

Code	Details
200	<b>Response body</b> <pre>{   "age": 25,   "city": "Warsaw",   "email": "alice@example.com",   "hobbies": [     "reading",     "chess",     "cycling"   ] }</pre>
- Curl**:

```
curl -X 'GET' \
'http://localhost:5000/user/alice' \
-H 'accept: */*'
```

```
curl -X 'GET' \
'http://localhost:5000/user/alice' \
-H 'accept: */*'
```

U mnie nie dziala na localhostie, wiec robimy przez ip:

```
curl -X GET http://192.168.1.14:5000/user/alice -H 'accept: */*' | jq
```

W jaki sposób sprawdzić ip?

*hostname -i*

Zapytanie – wyjaśnienie:

- ◆ **-H** oznacza **Header** (nagłówek HTTP).

Używasz go, by dodać własny nagłówek do żądania wysyłanego przez curl.

- ◆ **accept: \*/\*** to właśnie **treść tego nagłówka**.

To mówi serwerowi:

„Akceptuję dowolny typ odpowiedzi” (czyli każdy Content-Type).

### 0.1 Wyświetl wszystkie informacje użytkownika Alice.

```
curl -X GET http://192.168.1.14:5000/user/alice -H 'accept /' | jq
```

### 0.2 Wyświetl imię i nazwisko użytkownika Alice.

```
curl -X GET http://192.168.1.14:5000/user/alice -H 'accept /' | jq -r ".name"
```

Opcja `-r` w komendzie `jq` oznacza „**raw output**”, czyli **surowe wyjście**.  
👉 Bez `-r`, `jq` zwraca dane w **formacie JSON**, np. z cudzysłowniami:  
bash  
"alice"  
👉 Z `-r`, `jq` wypisze tylko **czysty tekst**, bez cudzysłowów i formatowania:  
bash  
alice

### 0.3 Wyświetl adres e-mail użytkownika Bob.

```
curl -X GET http://192.168.1.14:5000/user/bob -H 'accept /' | jq -r ".email"
```

### 0.4 Wyświetl wiek użytkownika Alice.

```
curl -X GET http://192.168.1.14:5000/user/alice -H 'accept /' | jq -r ".age"
```

### 0.5 Wyświetl nazwę miasta, w którym mieszka użytkownik Bob.

```
curl -X GET http://192.168.1.14:5000/user/bob -H 'accept /' | jq -r ".city"
```

### 0.6 Wyświetl hobby użytkownika Alice.

```
curl -X GET http://192.168.1.14:5000/user/alice -H 'accept /' | jq -r ".hobbies"
```

### 0.7 Sprawdź, czy jednym z hobby użytkownika Bob są gry.

```
curl -X 'GET' 'http://localhost:5000/user/bob' -H 'accept: /' | jq -n '(.hobbies | index("games")) != null'
```

**0.8** Wyświetl pierwsze hobby użytkownika Alice.

```
curl -X GET http://192.168.1.14:5000/user/alice -H 'accept: /' | jq -r ".hobbies[0]"
```

**0.9** Sprawdź, ile hobby ma użytkownik Bob, a ile użytkownik Alice.

```
curl -X GET http://192.168.1.14:5000/user/alice -H 'accept: /' | jq -r ".hobbies | length"
```

**0.10** Wyświetl nazwę użytkownika i miasto jako jeden ciąg znaków, np. "Alice Smith (Warsaw)". Zmodyfikuj polecenie, aby tekst był wyświetlany jako "Alice Smith from Warsaw".

```
curl -X GET http://192.168.1.14:5000/user/alice -H 'accept: /' | jq -r "(.name) ((.city))"
```

```
http://192.168.1.14:5000/user/alice -H 'accept: /' | jq -r "(.name) ((.city))"
```

```
curl -X GET http://192.168.1.14:5000/user/alice -H 'accept: /' | jq -r "'(.name) from ((.city))'"
```

**0.11** Wyświetl wszystkie przedmioty. Wyświetl nazwy wszystkich przedmiotów.

```
curl -X GET http://192.168.1.14:5000/items | jq -r
```

```
[  
  {  
    "id": 1,  
    "name": "itemX",  
    "price": 15.99  
  },  
  {  
    "id": 2,  
    "name": "superItem",  
    "price": 99.9  
  },  
  {  
    "id": 3,  
    "name": "item123",  
    "price": 5.49  
  },  
  {  
    "id": 4,  
    "name": "itemY",  
    "price": 29.0  
  }]
```

```
curl -X GET http://192.168.1.14:5000/items -H 'accept: /' | jq -r '.[].name'
```

```
100 157 1  
itemX  
superItem  
item123  
itemY
```

**0.12** Wyświetl przedmioty droższe niż 20 pln.

```
curl -X GET http://192.168.1.14:5000/items -H 'accept: /' | jq -r '.[].price > 20'
```

```
100 157  
false  
true  
false  
true  
true
```

```
curl -X GET http://192.168.1.14:5000/items -H 'accept: /' | jq -r '[] | select(.price > 20)'
```

```
{  
  "id": 2,  
  "name": "superItem",  
  "price": 99.9  
}  
{  
  "id": 4,  
  "name": "itemY",  
  "price": 29.0  
}
```

### 0.13 Wyświetl przedmioty tańsze niż 30 pln.

```
curl -X GET http://192.168.1.14:5000/items -H 'accept /' | jq -r '.[] | select(.price <30)'
```

```
natalka@pc-53:~$ curl -X GET http://192.168.1.14:5000/items -H 'accept /*' | jq -r '.[] | select(.price <30)'  
% Total    % Received % Xferd  Average Speed   Time     Time      Current  
          Dload Upload Total Spent   Left Speed  
100 157 100 157 0 0 56781 0 --:--:-- --:--:-- --:--:-- 78500  
{  
  "id": 1,  
  "name": "itemX",  
  "price": 15.99  
}  
{  
  "id": 3,  
  "name": "item123",  
  "price": 5.49  
}  
{  
  "id": 4,  
  "name": "itemY",  
  "price": 29.0  
}
```

### 0.14 Posortuj przedmioty według ceny rosnąco, a następnie malejąco.

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq -r 'sort_by(.price)'
```

```
[  
  {  
    "id": 3,  
    "name": "item123",  
    "price": 5.49  
  },  
  {  
    "id": 1,  
    "name": "itemX",  
    "price": 15.99  
  },  
  {  
    "id": 4,  
    "name": "itemY",  
    "price": 29.0  
  },  
  {  
    "id": 2,  
    "name": "superItem",  
    "price": 99.9  
  }  
]
```

Na odwrot:

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq -r 'sort_by(.price) | reverse'
```

### 0.15 Pobierz sumę cen wszystkich przedmiotów.

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq 'map(.price) | add'
```

**0.16** Wyświetl przedmioty, których nazwa zawiera "item".

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept */* | jq -r '.[] | select(.name | test("item"))'
```

Z ingorowaniem wielkości liter V

```
[User@DellOptiPlex-0110 ~]$ curl -X 'GET' 'http://127.0.0.1:5000/items' -H 'accept:/' | jq '.[] | select(.name | test("item", "1"))'
% Total % Received % Xferd Average Speed Time Time Current
          Dload Upload Total Spent Left Speed
100 157 100 157    0     0 64849    0 --:--:-- --:--:-- 78500
[{"id": 1,
 "name": "itemX",
 "price": 15.99
},
 {"id": 2,
 "name": "superItem",
 "price": 99.9
},
 {"id": 3,
 "name": "item123",
 "price": 5.49
},
 {"id": 4,
 "name": "itemY",
 "price": 29}
```

**0.17** Wyświetl przedmioty w przedziale cenowym 10-30 pln.

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq 'map(select(.price >= 10.00 and .price <= 30.00))'
```

**0.18** Wyświetl użytkowników, którzy mają więcej niż 2 hobby.

**0.19** Wyślij do serwera request typu POST (endpoint `/echo`) i wyświetl odpowiedź.

```
curl -X 'POST'
```

```
http://192.168.1.14:5000/echo -H 'accept: /' 'Content-Type: application/json' -d '{}'
```

```
[x]-[user@parrot]:[-]
$ curl -X 'POST' 'http://localhost:5000/echo' -H 'accept:/' -H 'Content-Type: application/json' -d '{}'
{"Received":{}}
```

**0.20** Wyślij do serwera request typu **POST**, który doda nowy przedmiot (użyj endpointa `/items`). Wyświetl wszystkie przedmioty po dodaniu.

```
curl -X 'POST'
```

```
'http://192.168.1.14:5000/items' -H 'accept: /' -H 'Content-Type: application/json' -d '{"id": 5, "name": "itemNext", "price": 30.00 }'
```

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq
```

**0.21** Za pomocą requestu typu PUT, aktualizuj cenę przedmiotu o numerze 1. Wyświetl wszystkie przedmioty.

curl -X 'PUT'

```
'http://192.168.1.14:5000/items/1' -H 'accept: /' -H 'Content-Type: application/json' -d '{ "price":1.34}'
```

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq
```

**0.22** Za pomocą requestu typu DELETE, usuń przedmiot o numerze 3. Wyświetl wszystkie przedmioty.

<http://192.168.1.14:5000/items/3>

```
curl -X 'DELETE'
```

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq 'max_by(.price | tonumber)'
```

```
100 195 100 195
{
  "id": 2,
  "name": "superItem",
  "price": 99.9
}
```

**0.24** Wyświetl wszystkie przedmioty z ceną podwyższoną o 10%.

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq 'map(.price | tonumber * 1.1)'
```

**0.25** Pobierz nazwy wszystkich użytkowników i ich wiek.

**0.26** Wyświetl hobby Alice jako string połączony przecinkami.

```
curl -X 'GET' http://192.168.1.14:5000/user/alice -H 'accept /' | jq '.hobbies | join(",")'
```

```
100 116 100 116 0 0 39780
"reading, chess, cycling"
curl: (7) Failed to connect to 192.168.1.14 port 5000 after 0 ms
```

**0.27** Pobierz pierwszy i ostatni przedmiot z listy.

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq '.[].[0],.[[-1]]'
```

```
100 195 100 195 0
[
  {
    "id": 1,
    "name": "itemX",
    "price": 1.34
  },
  {
    "id": 5,
    "name": "itemNowy",
    "price": 15
  }
]
```

**0.28** Sprawdź, czy są przedmioty droższe niż 100 pln.

```
curl -X 'GET' http://192.168.1.14:5000/items -H 'accept /' | jq '.[] | .price > 100'
```

```
100 195
false
false
false
false
false
```

Lub Kinga ma tak:

```
[user@parrot]~
└─ $ curl -s 'http://localhost:5000/items' | jq '.[] | select(.price | tonumber) > 100' |
```

**0.29** Utwórz nowy przedmiot za pomocą POST /items. Wyświetl odpowiedź serwera i następnie listę wszystkich przedmiotów.

To ju bylo

chyba xD

**0.30** Zaktualizuj przedmiot `id=2`. Wyświetl odpowiedź serwera i następnie listę wszystkich przedmiotów. To tez raczej było ale tylko price aktualizowalismy, da sie tak samo wszystkie argumenty wypisac i aktualizowac

## Zajecia 2 – Funkcje haszujace, haszowanie

### GitHub

Na potrzeby zajęć swoje własne repozytorium na Githubie. Rezpozytorium będziesz wykorzystywać, aby wrzucać do niego rozwiązań zadań z przedmiotu. Możesz nazwać repozytorium dowolnie, ale najlepiej wykorzystaj nazwę:

`imie-nazwisko-bsk-umcs.`

### Wirtualne środowisko pracy

Wirtualne środowisko pracy w Python, znane również jako `virtualenv` lub `venv`, jest sposobem za zapewnienie działania naszego programu niezależnie od maszyny na której jest uruchamiany ORAZ sposobem na uruchomienie innych programów na NASZEJ maszynie, tak aby instalowane z nią zależności nie zakłóciły pracy innych programów. Wirtualne środowisko pracy, jest swojego rodzaju odizolowanym katalogiem, zawierającym instalację języka programowania Python oraz zainstalowane biblioteki na potrzeby programu, który będziemy w nim uruchamiać.

### Instalacja

```
sudo apt update  
sudo apt install python3-pip  
python3 -m pip install --upgrade pip  
sudo apt install python3-venv  
python3 -m pip install --user virtualenv
```

### Tworzenie środowiska

```
python3 -m venv venv  
source ./venv/bin/activate
```

### Instalowanie pakietów

```
pip list  
pip install black  
black test.py
```

### Dezaktywacja (koniec pracy) wirtualnego środowiska

```
deactivate
```

### Dobre praktyki

W zadaniach najczęściej będziemy wykorzystywać język programowania Python. Aby zachować dobre praktyki programistyczne, będziemy używać `black'a` oraz `pylama`. Jeśli to możliwe, możesz wykorzystać inny język programowania (jest to zależne od zadania). Pamiętaj jednak również o zachowaniu dobrych

## Narzędzia

### hash-identifier

**hash-identifier** to narzędzie służące do identyfikacji różnych typów hashy kryptograficznych. Hash funkcjonuje jako skrót wiadomości (ang. *message digest*), który jest stosowany w kryptografi w celu sprawdzenia integralności danych lub w zabezpieczeniach hasel. Aby użyć **hash-identifier**, wystarczy uruchomić narzędzie z poziomu terminala, a następnie wprowadzić hash, który chcemy zidentyfikować. Program automatycznie przeanalizuje format i poda możliwe algorytmy, z jakimi może być związany podany hash. Narzędzie jest szczególnie przydatne podczas analizy danych pozyskanych z audytów bezpieczeństwa, forensyki cyfrowej, czy przy łamaniu hasel. **hash-identifier** to proste, ale użyteczne narzędzie, które przyspiesza proces identyfikacji algorytmów hashujących. Dzięki jego intuicyjnej obsłudze, jest to wartościowe narzędzie w arsenale specjalistów zajmujących się bezpieczeństwem informacji.

### hashid

**hashid** HashID to niewielkie, ale bardzo przydatne narzędzie wiersza poleceń służące do automatycznej identyfikacji typu (algorytmu) dowolnego ciągu znaków zakodowanego w formie hash (skrótu). HashID analizuje wejściowy ciąg (np. 5f4dcc3b5aa765d61d8327deb882cf99) i zgaduje, jakim algorytmem został on wytworzony

**praktyk** (MD5, SHA-1, SHA-256, NTLM, MySQL, bcrypt, itp.).

### OpenSSL

To wieloplatformowa, otwarta implementacja protokołów SSL (wersji 2 i 3) i TLS (wersji 1) oraz algorytmów kryptograficznych ogólnego przeznaczenia. Dostępna jest dla systemów uniksopodobnych (m.in. Linux, BSD, Solaris), OpenVMS i Microsoft Windows. OpenSSL zawiera biblioteki implementujące wspomniane standardy oraz mechanizmy kryptograficzne, a także zestaw narzędzi konsolowych (przede wszystkim do tworzenia kluczy oraz certyfikatów, zarządzania urządzeniami certyfikacji, szyfrowania, dekryptażu i obliczania podpisów cyfrowych). OpenSSL pozwala na używanie wszystkich zastosowań kryptografii. Poniżej kilka kluczowych cech OpenSSL:

- **Wszechstronność:** OpenSSL oferuje bogaty zestaw narzędzi i bibliotek do obsługi różnych protokołów kryptograficznych, w tym SSL/TLS, kryptografię klucza publicznego, szyfrowanie danych i wiele innych.
- **Bezpieczeństwo sieciowe:** OpenSSL dostarcza narzędzia do zarządzania certyfikatami, generowania kluczy, podpisywania cyfrowego i szyfrowania danych, co umożliwia tworzenie bezpiecznych aplikacji internetowych i usług.
- **Wsparcie dla różnych platform:** OpenSSL jest dostępny na wielu platformach, w tym na systemach Unix, Linux, macOS, Windows oraz innych, co czyni go popularnym narzędziem wśród programistów i administratorów systemów.
- **Otwarty kod źródłowy:** OpenSSL jest projektem open-source, co oznacza, że jego kod jest dostępny publicznie i może być modyfikowany oraz rozwijany przez społeczność, co sprzyja ciągłemu ulepszaniu i dostosowywaniu narzędzi do różnych potrzeb i zastosowań.
- **Wsparcie dla wielu protokołów:** OpenSSL obsługuje wiele standardowych protokołów kryptograficznych, takich jak SSL/TLS, SSH, S/MIME, PKCS, co czyni go wszechstronnym narzędziem do implementacji bezpiecznych komunikacji w różnych aplikacjach i systemach.

## Linki

- <https://www.openssl.org/>
- <https://www.kali.org/tools/hash-identifier/>
- <https://pypi.org/project/hashID/>
- <https://pypi.org/project/black/>
- <https://pypi.org/project/pylama/>
- <https://www.toptal.com/developers/gitignore>
- <https://pypi.org/project/bcrypt/>
- <https://httpd.apache.org/docs/current/programs/htpasswd.html>
- [https://hashcat.net/wiki/doku.php?id=example\\_hashes](https://hashcat.net/wiki/doku.php?id=example_hashes)
- [https://en.wikipedia.org/wiki/Hash\\_function](https://en.wikipedia.org/wiki/Hash_function)
- <https://www.baeldung.com/cs/hashing>
- <https://cryptobook.nakov.com/cryptographic-hash-functions>
- <https://argon2.online/>
- <https://manpages.ubuntu.com/manpages/questing/man1/scrypt.1.html>
- <https://informatykzakladowy.pl/jak-serwer-sprawdza-haslo>
- [https://cheatsheetseries.owasp.org/cheatsheets/Password\\_Storage\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html)
- <https://www.cyber.mil.pl/funkcje-skrotu/>

## Teoria

Funkcje haszujące to deterministyczne algorytmy, które przekształcają dane wejściowe dowolnej długości na ciąg znaków o stałej długości – zwany skrótem (ang. *hash*). Ich podstawowe cechy to:

- **Jednostronność:** trudność odtworzenia danych wejściowych ze skrótu.
- **Stała długość skrótu:** niezależnie od długości danych wejściowych.
- **Unikalność:** minimalizacja kolizji – różne dane powinny dawać różne skróty.
- **Deterministyczność:** te same dane wejściowe dają ten sam wynik.
- **Szybkość:** szybkie przetwarzanie dużych danych.
- **Odporność na kolizje:** trudność znalezienia dwóch różnych danych z tym samym skrótem.

## Zastosowanie funkcji haszujących

- **Zabezpieczanie haseł:** przechowywanie skrótów haseł zamiast samych haseł.
- **Weryfikacja integralności danych:** np. sumy kontrolne plików.
- **Kryptografia:** podpisy cyfrowe, generowanie kluczy.
- **Systemy autoryzacji i uwierzytelniania:**
- **Ochrona prywatności i anonimizacja danych:**

## Klasyczne funkcje haszujące

- **MD5 (Message Digest 5):**
  - 128-bitowy skrót.
  - Szybka, ale nieodporanna na kolizje.
  - Złamana – niezalecana do zastosowań bezpieczeństwa.
- **SHA-1 (Secure Hash Algorithm 1):**
  - 160-bitowy skrót.
  - Używana w przeszłości m.in. w certyfikatach SSL.
  - Również uważana za złamana (od 2017 potwierdzone kolizje).
- **SHA-256:**
  - 256-bitowy skrót.
  - Obecnie uznawana za bezpieczną, ale szybka – nieoptymalna do przechowywania haseł.

## Nowoczesne funkcje haszujące

- **bcrypt:**
  - Opiera się na algorytmie Blowfish.
  - Używa mechanizmu kosztu (**cost**), który określa czas haszowania.
  - Automatycznie dodaje sól (salt), dzięki czemu ten sam tekst hasłowy daje różne hasza.
  - Dobrze chroni przed atakami słownikowymi i tablicami tęczowymi.
- **scrypt:**
  - Wprowadza wymagania pamięciowe – utrudnia użycie specjalistycznego sprzętu (GPU/ASIC).
  - Umożliwia konfigurację pamięci, CPU oraz długości wyjścia.
- **Argon2** (zwycięzca PHC 2015):
  - Trzy warianty: **Argon2d**, **Argon2i**, **Argon2id**.
  - Pozwala ustawić parametry: pamięć, czas, liczbę wątków, sól.
  - Bardzo odporna na ataki brute-force i tęczowe tablice.
  - Obecnie rekommendowana funkcja do przechowywania haseł.

## Porównanie cech: stare vs nowoczesne funkcje

Cechy funkcji haszujących	Stare funkcje (MD5, SHA-1, SHA-256)	Nowoczesne funkcje (bcrypt, scrypt, Argon2)
Jednostronność	Tak	Tak
Stała długość skrótu	Tak	Tak
Unikalność (mała kolizja)	Nie (MD5/SHA-1) / Częściowo (SHA-256)	Tak (odporne na kolizje)
Deterministyczność	Tak	Tak (ale z losową solą hasz daje inny wynik)
Szybkość	Bardzo szybkie (łatwiejsze do złamania)	Wolne (celowo, zwiększa koszt ataku)
Odporność na kolizje	Niska / średnia	Wysoka
Odporność na ataki GPU/rainbow tables	Nie	Tak
Konfigurowalność (koszt, pamięć, wątki)	Brak	Tak (duża elastyczność w doborze parametrów)

## Podsumowanie

Klasyczne funkcje haszujące takie jak MD5 czy SHA-1 nie zapewniają już dziś wystarczającego poziomu bezpieczeństwa – głównie z powodu podatności na kolizje oraz braku mechanizmów obrony przed atakami brute-force. Dlatego też w kontekście przechowywania haseł zaleca się stosowanie nowoczesnych funkcji haszujących takich jak bcrypt, scrypt czy Argon2, które dzięki zastosowaniu soli, kosztu czasowego i wymagań pamięciowych skutecznie utrudniają łamanie haseł nawet przy użyciu nowoczesnego sprzętu.

**losowa sola - pełna wartość doklejana do słowa**

**openssl version – wersja openssl**

**openssl list -cipher-algorithms - pokazuje wszystkie algorytmy**

**openssl list -digest-algorithms - haszujące funkcje (tu sprawdzamy jak się nazywają)**

**openssl dgst -md5**

**echo -n "hello" | openssl dgst -md5**

1.1 Pod adresem <http://127.0.0.1:10001> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:10001/hash> oraz <http://127.0.0.1:10001/submit>.

Twoim zadaniem jest obliczenie skrótu MD5 podanego przez serwer losowego słowa. Wynikowy hash powinien być w formacie szesnastkowym (hex). Aby otrzymać od serwera słowo do zahashowania oraz identyfikator sesji, wyślij zapytanie HTTP GET do endpointa <http://127.0.0.1:10001/hash>. Otrzymasz w odpowiedzi unikalny identyfikator sesji oraz losowe słowo do zahashowania.

Następnie powinieneś obliczyć skrót MD5 otrzymanego słowa, kodując wynik w formacie szesnastkowym (hex).

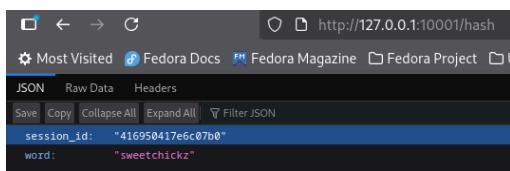
Po obliczeniu hash'a wyślij go do serwera metodą HTTP POST na endpoint <http://127.0.0.1:10001/submit>, przekazując w formacie JSON zarówno identyfikator sesji, jak i obliczony skrót MD5 (hex). Serwer zweryfikuje poprawność przesłanego hasha i zwróci informację o sukcesie lub błędzie.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 10001:10001 mazurkatarzyna/hashing-md5-ex1:latest
```

- (b) Do obliczenia skrótu MD5 użyj narzędzia OpenSSL.

- (c) Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka HTTP "Content-Type: application/json".



## 1. Uruchamiamy serwer

```
podman run -dp 10001:10001 --name hashingmd5ex1 docker.io/mazurkatarzyna/hashing-md5-ex1:latest
```

## 2. Haszujemy wybrane słowo

```
echo -n "sweetchickz" | openssl dgst -md5
```

## 3. Wysylamy zapytanie o to zahaszowane słowo

```
curl -X POST http://127.0.0.1:10001/submit -d
'{"session_id": "416950417e6c07b0", "hash_hex": "982a7cad9cfbe626c209933f85e4bde1"}' -H 'Content-Type: application/json'
```

```
natalka@pc-53:~$ echo -n "sweetchickz" | openssl dgst -md5
MD5(stdin)= 982a7cad9cfbe626c209933f85e4bde1
natalka@pc-53:~$ curl -X POST http://127.0.0.1:10001/submit -d '{"session_id": "416950417e6c07b0", "hash_hex": "982a7cad9cfbe626c209933f85e4bde1"}' -H 'Content-Type: application/json'
{"result": "Correct MD5 hash!"}
```

~Wysłanie JSON via curl (bezpośrednio i z pliku)~

### 1) wysyłanie z pliku

Utwórz plik: nano payload.json i wpisz (przykład):

```
{"session_id": "957725326449cc9e", "hash_hex": "5aa2f7fa75b0a9c3267d6f0feed3d2fb"}
```

```
curl -s -X POST http://127.0.0.1:10001/submit \
-H "Content-Type: application/json" \
--data @payload.json
```

### 2) Pobieranie odpowiedzi do pliku i wyjmowanie pola word

```
curl -s -X POST http://127.0.0.1:10001/hash -o response.json
word=$(jq -r '.word' response.json)
```

Co robią te polecenia:

- `curl -s -X POST http://127.0.0.1:10001/hash -o response.json` — wysyła POST na `/hash`, zapisuje odpowiedź do pliku `response.json`. `-s` ukrywa pasek postępu/curl messages.
- `jq -r '.word' response.json` — `jq` to narzędzie do parsowania JSON. `-r` zwraca surowy string (bez cudzysłowów). Przykład: jeśli `response.json` to `{"word": "abc"}`, to wynik to `abc`.
- Wracając do zmiennej: `word=$(jq -r '.word' response.json)` — teraz w `$word` masz słowo.

Uwaga: jeśli serwer zwraca błąd lub JSON bez pola `.word` — `jq` zwróci `null` lub pusty string; warto sprawdzić.

### 3)) Wysyłanie wyniku z użyciem zmiennych — uwaga na cytowania!

```
sol=$(printf '%s' "$word" | openssl dgst -md4 | awk '{print $2}')
```

Tu awk '{print \$2}' wyciąga drugie pole (hash). To działa z formatem MD4(stdin)= <hash>.

Możesz też użyć cut:

```
sol=$(printf '%s' "$word" | openssl dgst -md4 | cut -d' ' -f2)
```

Wysylamy zapytanie:

```
curl -s -X POST http://127.0.0.1:10001/submit
```

```
-H "Content-Type: application/json"
```

```
-d "{\"session_id\": \"$id\", \"hash_hex\": \"$sol\"}"
```

b) Lepsze i czytelniejsze: użyj jq do zbudowania poprawnego JSON (uniezależnia od problemów z escapowaniem):

```
payload=$(jq -n --arg id "$id" --arg sol "$sol" '{session_id: $id, hash_hex: $sol}'")
```

```
curl -s -X POST http://127.0.0.1:10001/submit -H "Content-Type: application/json" -d "$payload"
```

1.2 Pod adresem <http://127.0.0.1:10002> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:10002/hash> oraz <http://127.0.0.1:10002/submit>.

Twoim zadaniem jest obliczenie skrótu SHA-256 podanego przez serwer losowego słowa. Wynikowy hash powinien być w formacie szesnastkowym (hex). Aby otrzymać od serwera słowo do zahashowania oraz identyfikator sesji, wyślij zapytanie HTTP GET do endpointa <http://127.0.0.1:10002/hash>. Otrzymasz w odpowiedzi unikalny identyfikator sesji oraz losowe słowo do zahashowania.

Następnie powinieneś obliczyć skróć SHA-256 otrzymanego słowa, kodując wynik w formacie szesnastkowym (hex).

Po obliczeniu hash'a wyślij go do serwera metodą HTTP POST na endpoint <http://127.0.0.1:10002/submit>, przekazując w formacie JSON zarówno identyfikator sesji, jak i obliczony skróć MD5 (hex). Serwer zweryfikuje poprawność przesłanego hasha i zwróci informację o sukcesie lub błędzie.

(a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 10002:10002 mazurkatarzyna/hashing-sha256-ex1:latest
```

(b) Do obliczenia skrótu SHA-256 użyj narzędzia OpenSSL.

(c) Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka HTTP "Content-Type: application/json".

```
echo -n "glamour24" | openssl dgst -sha-256
```

```
curl -X POST http://127.0.0.1:10002/submit -d '{"session_id": "f5a227b49ae8b6e6", "hash_hex": "6af7c0aa2a0fe26d3466845e6cce06c4138f29cef76e12857098cb010c2f7f7e"}' -H "Content-Type: application/json"
```

1.3 Pod adresem <http://127.0.0.1:10003> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:10003/hash> oraz <http://127.0.0.1:10003/submit>.

Twoim zadaniem jest obliczenie skrótu SHA-512 podanego przez serwer losowego słowa. Wynikowy hash powinien być w formacie szesnastkowym (hex). Aby otrzymać od serwera słowo do zahashowania oraz identyfikator sesji, wyslij zapytanie HTTP GET do endpointa <http://127.0.0.1:10003/hash>. Otrzymasz w odpowiedzi unikalny identyfikator sesji oraz losowe słowo do zahashowania.

Następnie powinieneś obliczyć skrót SHA-512 otrzymanego słowa, kodując wynik w formacie szesnastkowym (hex).

Po obliczeniu hash'a wyslij go do serwera metodą HTTP POST na endpoint <http://127.0.0.1:10003/submit>, przekazując w formacie JSON zarówno identyfikator sesji, jak i obliczony skrót SHA-512 (hex). Serwer zweryfkuje poprawność przesłanego hasha i zwróci informację o sukcesie lub błędzie.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 10003:10003 mazurkatarzyna/hashing-sha-512-ex1:latest
```

- (b) Do obliczenia skrótu SHA-512 użyj narzędzia OpenSSL.

- (c) Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dołączeniu nagłówka HTTP "Content-Type: application/json".

## Uruchomienie serwera:

```
podman run -dp 10003:10003 --name hashingsha512ex1 docker.io/mazurkatarzyna/hashing-sha-512-ex1:latest
```

```
natalka@fedora:~$ echo -n "car250" | openssl dgst -sha-512
SHA2-512(stdin)= ba3f270faf6336e5399e21aee6624fc2ae60f137abb1be31c2d0adadd49c7f206e1ee28a86244a24867f3615a886f51bc31707b13814
914c29288e39157240e
natalka@fedora:~$ curl -X POST http://127.0.0.1:10003/submit -d '{"session_id": "c7f2d0571a988b47", "hash_hex": "ba3f270faf6336e5399e21aee6624fc2ae60f137abb1be31c2d0adadd49c7f206e1ee28a86244a24867f3615a886f51bc31707b13814914c29288e39157240e"}' -H "Content-Type: application/json"
>{"result": "Correct SHA-512 hash!"}
```

1.4 Pod adresem <http://127.0.0.1:10004> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:10004/hash> oraz <http://127.0.0.1:10004/submit>.

Twoim zadaniem jest obliczenie skrótu **Argon** podanego przez serwer losowego słowa. Wynikowy hash powinien być w formacie szesnastkowym (hex). Aby otrzymać od serwera słowo do zahashowania, parametry użyte do haszowania oraz identyfikator sesji, wyslij zapytanie HTTP GET do endpointa <http://127.0.0.1:10004/hash>. Otrzymasz w odpowiedzi unikalny identyfikator sesji, losowe słowo do zahashowania i niezbędne parametry.

Następnie powinieneś obliczyć skrót **Argon** otrzymanego słowa, kodując wynik w formacie szesnastkowym (hex).

Po obliczeniu hash'a wyslij go do serwera metodą HTTP POST na endpoint <http://127.0.0.1:10004/submit>, przekazując w formacie JSON zarówno identyfikator sesji, jak i obliczony skrót **Argon** (hex). Serwer zweryfkuje poprawność przesłanego hasha i zwróci informację o sukcesie lub błędzie.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 10004:10004 mazurkatarzyna/hashing-argon-ex1:latest
```

- (b) Do obliczenia skrótu Argon narzędzia OpenSSL dostępnego jako kontener Dockerowy. W tym celu uruchom kontener, i wejdź do jego środka. Haszowanie za pomocą algorytmu Argon dostępne jest pod nazwą kdf.

```
docker run -it mazurkatarzyna/openssl-332-ubuntu:latest
openssl kdf -help
```

- (c) Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dołączeniu nagłówka HTTP "Content-Type: application/json".

```
podman run -dp 10004:10004 --name hashingargonex1 docker.io/mazurkatarzyna/hashing-argon-ex1:latest
```

Wypluwa nam hash: curl -X GET <http://127.0.0.1:10004/hash>

W drugim terminalu wchodzimy do środka kontenera, aby w nim pracować:

```
podman exec -it hashingargonex1 /bin/bash
```

Sprawdzamy wersje openssl: openssl version

Tworzenie hasła z użyciem argon:

```
openssl kdf -keylen 24 \
```

```
-kdfopt pass:kokook \
```

```
-kdfopt salt:NaCl2024 \
```

```
-kdfopt iter:1 \
```

```
-kdfopt memcost:8192 \
```

ARGON2D

Parametr	Znaczenie
-keylen 24	długość klucza (hasha) w bajtach → 24 bajty = 48 znaków hex
-kdfopt pass:kokook	hasło wejściowe ( kokook )
-kdfopt salt:NaCl2024	sól (salt), ważna dla bezpieczeństwa
-kdfopt iter:1	liczba iteracji (tu: 1 – dla testów, w praktyce dajesz więcej, np. 10+)
-kdfopt memcost:8192	pamięć w KB używana przez algorytm
ARGON2D	typ Argon2 (D = data-dependent) — jedna z wersji (są jeszcze ARGON2I, ARGON2ID)

Zamianianie dużych liter na małe:

```
root@2248bddf510e:/app# openssl kdf -keylen 24 -kdfopt pass:kokook -kdfopt salt:NaCl2024 -kdfopt iter:1 -kdfopt memcost:8192 ARGON2D | tr -d ':' | tr '[:upper:]' '[:lower:]'  
e6200991560d390e4d916a14a3221ebe29306d2b25ea856a
```

### na kolokwium moze:

argon2i gdybysmy chcieli wykorzystać, to tak samo, takie same argumenty przyjmują, może się tylko nazwa zmienić

```
curl -X POST http://127.0.0.1:10004/submit -d '{"session_id":"9b6b6ad45e21e8b3",  
"hash_hex":"e6200991560d390e4d916a14a3221ebe29306d2b25ea856a"}' -H "Content-Type:  
application/json"
```

- To **wysyłanie wyniku (hasha)**, który wygenerowałaś wcześniej za pomocą openssl kdf ... ARGON2D, do serwera działającego na porcie 10004.
- Serwer (hashing-argon-ex1) ma endpoint /submit, który **sprawdza**, czy przesłany hash jest poprawny (czy pasuje do zadania, np. do konkretnej sesji session\_id).

### ⌚ Gdzie to uruchomić:

- Możesz to zrobić **na pierwszym terminalu (tym z hosta), nie wewnątrz kontenera**.
- Terminal 1 → uruchamiasz i kontrolujesz serwer (podman run ... + curl).

- Terminal 2 → wchodzisz do kontenera tylko po to, żeby **obliczyć hash**.

1.5 Pod adresem <http://127.0.0.1:10005> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:10005/hash> oraz <http://127.0.0.1:10005/submit>.

Twoim zadaniem jest obliczenie skrótu **bcrypt** podanego przez serwer losowego słowa. Wynikowy hash powinien być w formacie szesnastkowym (hex). Aby otrzymać od serwera słowo do zahashowania oraz identyfikator sesji, wyślij zapytanie HTTP GET do endpointa <http://127.0.0.1:10005/hash>. Otrzymasz w odpowiedzi unikalny identyfikator sesji i losowe słowo do zahashowania.

Następnie powinieneś obliczyć skrót **bcrypt** otrzymanego słowa, kodując wynik w formacie szesnastkowym (hex).

Po obliczeniu hash'a wyślij go do serwera metodą HTTP POST na endpoint <http://127.0.0.1:10005/submit>, przekazując w formacie JSON zarówno identyfikator sesji, jak i obliczony skrót **brypt** (hex). Serwer zweryfikuje poprawność przesłanego hasha i zwróci informację o sukcesie lub błędzie.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 10005:10005 mazurkatarzyna/hashing-bcrypt-ex1:latest
```

- (b) Do obliczenia skrótu **brypt** użyj narzędzia **htpasswd** dostępnego jako kontener Dockerowy.

```
docker run mazurkatarzyna/htpasswd:latest --help
```

- (c) Aby wysłać odpowiedź do serwera, użyj narzędzia **cURL**. Pamiętaj o dołączeniu nagłówka HTTP "Content-Type: application/json".

podman run -dp 10005:10005 --name hashingbcryptex1 docker.io/mazurkatarzyna/hashing-bcrypt-ex1:latest

To **drugi kontener** (inny serwer), który działa niezależnie od poprzedniego.

Zamiast Argon2, ten używa **brypt** — kolejnego popularnego algorytmu do haszowania haseł.

Używamy kontenerów z narzędziami do rozpoznawania hasła

podman run -it docker.io/mazurkatarzyna/hash-identifier:latest

Lub

podman run -it docker.io/mazurkatarzyna/hashid:latest

- Te obrazy uruchamiają w terminalu narzędzia:  
hash-identifier  
hashid
- Oba analizują ciąg hasha (np. 6adfb183a4a2c94a2f92dab5ade762a47889a5a1) i pokazują, jakie **algorytmy** mogły go wygenerować (np. SHA1, MD5, NTLM itp.).

#### • 4 Przykład: analiza konkretnego hasha

Masz hash:

Copy code

```
6adfb183a4a2c94a2f92dab5ade762a47889a5a1
```

Uruchamiasz:

Copy code

```
bash
```

```
hashid 6adfb183a4a2c94a2f92dab5ade762a47889a5a1
```

Narzędzie poda Ci coś w tym stylu:

Copy code

```
less
```

```
Possible Hashes:
```

```
[+] SHA-1
```

```
[+] Double SHA-1
```

```
[+] Tiger-160
```

```
...
```

czyli mówi, że to prawdopodobnie SHA-1.

## Sprawdzenie przez OpenSSL

Teraz robisz test, żeby to potwierdzić:

```
echo -n "helloworld" | openssl dgst -sha1
```

Wynik: SHA1(stdin)= 6adfb183a4a2c94a2f92dab5ade762a47889a5a1

I on się zgadza z tym z polecenia! Reasumując, użyto funkcji sha1 do zahaszowania wyrazu.

- 1.6 Napisz skrypt w języku Python, w którym wygenerujesz hash **MD5** dowolnego ciągu znaków podawanego jako argument wywołania skryptu. Sprawdź poprawność wygenerowanego hasza porównując go z wynikiem otrzymanym przy pomocy md5sum lub openssl. Wykorzystaj bibliotekę [hashlib](#). Możesz wykorzystać poniższy szkic kodu:

```
import hashlib

def md5_hash_string(input_string: str) -> str:
    """
    Returns the MD5 hash of a given string.
    """
    pass

if __name__ == "__main__":
    text = "hello world"
    print(f"MD5 hash of '{text}': {md5_hash_string(text)}")
```

```
import hashlib as hs

def md5_hash_string(input_string: str) -> str:
    hash_text=hs.md5(input_string.encode())
    return hash_text.hexdigest()

if __name__ == "__main__":
    text = "hello world"
    print(f"MD5 hash of '{text}': {md5_hash_string(text)}")

MD5 hash of 'hello world': 5eb63bbbe01eeed093cb22bb8f5acdc3
```

```
import hashlib as hs
```

```

def md5_hash_string(input_string: str) -> str:
    hash_text=hs.md5(input_string.encode())
    return hash_text.hexdigest()

if __name__ == "__main__":
    text = "hello world"
    print(f"MD5 hash of '{text}': {md5_hash_string(text)}")

```

- 1.7 Napisz skrypt w języku Python, w którym wygenerujesz hash **SHA-1** dowolnego ciągu znaków podawanego jako argument wywołania skryptu. Sprawdź poprawność wygenerowanego hasza porównując go z wynikiem otrzymanym przy pomocy `sha1sum` lub `openssl`. Wykorzystaj bibliotekę `hashlib`. Możesz wykorzystać poniższy szkielet kodu:

```

import hashlib

def sha1_hash_string(input_string: str) -> str:
    """
    Returns the SHA-1 hash of a given string.
    """
    pass

if __name__ == "__main__":
    text = "hello world"
    print(f"SHA-1 hash of '{text}': {sha1_hash_string(text)}")

```

```

import hashlib as hs

def sha1_hash_string(input_string: str) -> str:
    hash_text=hs.sha1(input_string.encode())
    return hash_text.hexdigest()

if __name__ == "__main__":
    text = "hello world"
    print(f"SHA-1 hash of '{text}': {sha1_hash_string(text)}")

SHA-1 hash of 'hello world' : 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed

```

```

import hashlib as hs

def sha1_hash_string(input_string: str) -> str:
    hash_text=hs.sha1(input_string.encode())
    return hash_text.hexdigest()

if __name__ == "__main__":
    text = "hello world"
    print(f"SHA-1 hash of '{text}': {sha1_hash_string(text)}")

```

- 1.8 Napisz skrypt w języku Python, w którym wygenerujesz hash **SHA-256** dowolnego ciągu znaków podawanego jako argument wywołania skryptu. Sprawdź poprawność wygenerowanego hasza porównując go z wynikiem otrzymanym przy pomocy `openssl`. Wykorzystaj bibliotekę `hashlib`. Możesz wykorzystać poniższy szkielet kodu:

```

import hashlib

def sha256_hash_string(input_string: str) -> str:
    """
    Returns the SHA-256 hash of a given string.
    """
    pass

if __name__ == "__main__":
    text = "hello world"
    print(f"SHA-256 hash of '{text}': {sha256_hash_string(text)}")

```

Tak samo jak powyżej tylko sha-256

**1.9** Napisz skrypt w języku Python, w którym wygenerujesz hash **bcrypt** dowolnego ciągu znaków podawanego jako argument wywołania skryptu. Sprawdź poprawność wygenerowanego hasza porównując go z wynikiem otrzymanym przy pomocy **htpasswd**. Do napisania programu potrzebna jest biblioteka **bcrypt** zainstalowana w kontenerze Dockerowym. Uruchom kontener, i wejdź do jego środka - biblioteka jest już zainstalowana w kontenerze. Aby napisać program, wykorzystaj **nano**.

(a) Uruchom kontener za pomocą poniższego polecenia:

```
docker run -it mazurkatarzyna/hashing-bcrypt-ex2:latest  
nano template.py
```

(b) Do weryfikacji skrótu **bcrypt** użyj narzędzia **htpasswd** dostępnego jako kontener Dockerowy.

```
docker run mazurkatarzyna/htpasswd:latest --help
```

Możesz wykorzystać poniższy szkielet kodu, który dostępny jest również wewnątrz kontenera jako plik **template.py**:

```
import bcrypt  
  
def bcrypt_hash_password(password: str) -> str:  
    """  
    Hash a password using bcrypt.  
    """  
    pass  
  
def bcrypt_verify_password(password: str, hashed: str) -> bool:  
    """  
    Verify a password against a bcrypt hash.  
    """  
    pass  
  
# Example usage  
if __name__ == "__main__":  
    pwd = "my_secure_password"
```

```
import bcrypt  
  
def bcrypt_hash_password(password: str) -> str:  
    #potrzebujemy bajtów, więc najpierw kodujemy tekst  
    hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt())  
    return hashed.decode()  
  
def bcrypt_verify_password(password: str, hashed: str) -> bool:  
    return bcrypt.checkpw(password.encode(), hashed.encode())  
  
# Example usage  
if __name__ == "__main__":  
    pwd = "my_secure_password"  
    hashed_pwd=bcrypt_hash_password(pwd)  
    print(f"Oryginalne hasło: {pwd}")  
    print(f"Wygenerowany hash bcrypt: \n {hashed_pwd}")  
  
    if bcrypt_verify_password(pwd, hashed_pwd):  
        print("Poprawne hasło!")  
    else:  
        print("Hasło nie pasuje!")
```

```
root@b58dcefc4a14:/app# nano template.py  
root@b58dcefc4a14:/app# python3 template.py  
Oryginalne hasło: my_secure_password  
Wygenerowany hash bcrypt:  
$2b$12$FXreYLjchElaHxToV2axw0w4qdQbVNGcP/dX/NRv0WhztS/mVKg2S  
Poprawne hasło!
```

## SHA-1, SHA-256, SHA-512, MD5

Są one tworzone głównie do:

- sprawdzania integralności plików (czy dane się nie zmieniły),
- generowania skrótów danych, podpisów cyfrowych, itp.

👉 Nie są zaprojektowane do bezpiecznego przechowywania haseł.

Dlaczego?

Bo są **bardzo szybkie** 🔌

I to, co jest plusem w zastosowaniach technicznych, jest **minusem w kontekście bezpieczeństwa**:

- Atakujący może w sekundę sprawdzić **miliony haset** (brute-force / rainbow tables).

## BCRYPT:

Jest stworzona, by utrudnić życie atakującemu 

Ma kilka **mechanizmów zabezpieczeń**, których SHA nie ma:

## ◊ Wolne obliczenia

- bcrypt jest **celowo powolny** – każda próba zajmuje ułamek sekundy.
  - dzięki temu ataki brute-force są **bardzo kosztowne**.

#### ◆ Losowa sól (salt)

- bcrypt **automatycznie generuje sól** (unikalny串ug dodany do hasła przed haszowaniem),
  - dzięki temu nawet jeśli dwie osoby mają to samo hasło, ich hashe będą **inne**.

- ◊ Możliwość ustawienia „kosztu” (liczby rund)

- możesz sam ustalić, jak bardzo ma być wolny (np. `bcrypt.gensalt(rounds=12)`).
  - im wyższy koszt, tym trudniej złamać hash, ale też wolniej działa logowanie.

**1.10** Mając dany początkowy ciąg znaków `helloworld`, który następnie został zahashowany, określ, jaka funkcja skrótu została wykorzystana do utworzenia hasha: `6adfb183a4a2c94a2f92dab5ade762a47889a5a1`.  
Możesz wykorzystać narzędzie `hash-identifier`:

```
docker run -it mazurkatarzyna/hash-identifier:latest
```

Czy hash-identifier odnalał nazwę hasha? Jeśli nie, zaproponuj inny sposób na rozwiązanie zadania.

```
HASH: 6adfb183a4a2c94a2f92dab5ade762a47889a5a1
```

```
Possible Hashes:  
[+] SHA-1  
[+] MySQL5 - SHA-1(SHA-1($pass))
```

```
Least Possible Hashes:
```

```
[+] Tiger-160  
[+] Haval-160  
[+] RipeMD-160  
[+] SHA-1(HMAC)  
[+] Tiger-160(HMAC)  
[+] RipeMD-160(HMAC)  
[+] Haval-160(HMAC)  
[+] SHA-1(MaNGOS)  
[+] SHA-1(MaNGOS2)  
[+] sha1($pass.$salt)  
[+] sha1($salt.$pass)  
[+] sha1($salt.md5($pass))  
[+] sha1($salt.md5($pass).$salt)  
[+] sha1($salt.sha1($pass))  
[+] sha1($salt.sha1($salt.sha1($pass)))  
[+] sha1($username.$pass)  
[+] sha1($username.$pass.$salt)  
[+] sha1(md5($pass))  
[+] sha1(md5($pass).$salt)  
[+] sha1(md5($pass))  
[+] sha1($pass)  
[+] sha1($pass).$salt  
[+] sha1($pass).substr($pass,0,3)  
[+] sha1($salt.$pass)  
[+] sha1($salt.sha1($pass))  
[+] sha1(strtolower($username).$pass)
```

```
-----  
Bye!
```

```
natalka@fedora:~$ echo -n "helloworld" | openssl dgst -sha1  
SHA1(stdin)= 6adfb183a4a2c94a2f92dab5ade762a47889a5a1  
natalka@fedora:~$
```

Hash jest poprawny, taki sam tuu w poleceniu

- 1.11 Mając dany początkowy ciąg znaków helloworld, który następnie został zahashowany, określ, jaka funkcja skrótu została wykorzystana do utworzenia hasza:  
\$2y\$10\$xbAv5a46CQYPay5UISCNeFwpVdx2qvhCBE0Z/YtfxoVXh0GrVKQa. Możesz wykorzystać narzędzie hash-identifier:

```
docker run -it mazurkatarzyna/hash-identifier:latest
```

Czy hash-identifier odnalazł nazwę hasha? Jeśli nie, zaproponuj inny sposób na rozwiązywanie zadania.

u mnie hashid trzeba było zainstalować:

Pip install hashid

Hashid 'hashnasz'

```
natalka@fedora:~$ hashid '$2y$10$xbAv5a46CQYPay5UISCNeFwpVdx2qvhCBE0Z/YtfxoVXh0GrVKQa'  
Analyzing '$2y$10$xbAv5a46CQYPay5UISCNeFwpVdx2qvhCBE0Z/YtfxoVXh0GrVKQa'  
[+] Blowfish(OpenBSD)  
[+] Woltlab Burning Board 4.x  
[+] bcrypt  
natalka@fedora:~$
```

```
import bcrypt  
hashed = b'$2y$10$xbAv5a46CQYPay5UISCNeFwpVdx2qvhCBE0Z/YtfxoVXh0GrVKQa'  
password=b"elloworld"  
  
ok=bcrypt.checkpw(password,hashed)  
  
if not ok and hashed.startswith(b"$2y$"):  
    ok=bcrypt.checkpw(password, b"$2y$" + hashed[4:])  
print("Match?",ok)
```

Kod:

```
import bcrypt
```

```
hashed = b"$2y$10$xbyAv5a46CQYPay5UISCNeFWpVdx2qvhCBEOZ/YtfxoVXhOGrVKQa"
password=b"elloworld"

ok=bcrypt.checkpw(password,hashed)
```

```
if not ok and hashed.startswith(b"$2y$"):
    ok=bcrypt.checkpw(password, b"$2y$" + hashed[4:])
print("Match?",ok)
```

```
natalka@fedora:~$ python3 hash.py
Match? True
```

- 1.12 Mając dany początkowy ciąg znaków `helloworld`, który następnie został zahashowany, określ, jaka funkcja skrótu została wykorzystana do utworzenia hasha: `6adfb183a4a2c94a2f92dab5ade762a47889a5a1`. Możesz wykorzystać narzędzie `hashid`:

```
docker run mazurkatarzyna/hashid:latest yourhash
```

Czy `hashid` odnalazł nazwę hasha? Jeśli nie, zaproponuj inny sposób na rozwiązanie zadania.

To zadanie chyba już było

- 1.13 Mając dany początkowy ciąg znaków `helloworld`, który następnie został zahashowany, określ, jaka funkcja skrótu została wykorzystana do utworzenia hasha: `$2y$10$xbyAv5a46CQYPay5UISCNeFWpVdx2qvhCBEOZ/YtfxoVXhOGrVKQa`. Możesz wykorzystać narzędzie `hashid`:

```
docker run mazurkatarzyna/hashid:latest yourhash
```

Czy `hashid` odnalazł nazwę hasha? Jeśli nie, zaproponuj inny sposób na rozwiązanie zadania.

To też chyba było

- 1.14 Mając dany początkowy ciąg znaków `R3iSrSNmgU9SFHxVekUD`, który następnie został zahashowany, określ, jaka funkcja skrótu została wykorzystana do utworzenia hasha: `48cab4b54bef42fddaa6353c68a20b369f40026e`. Aby rozwiązać zadanie, napisz skrypt w języku Python. Podpowiedź: algorytm jest dostępny w bibliotece `hashlib`.

```
import hashlib

input_text = "R3iSrSNmgU9SFHxVekUD"
target_hash = "48cab4b54bef42fddaa6353c68a20b369f40026e"

def find_matching_algorithms(text: str, target: str):
    matches = []
    # używamy algorithms_available - daje szerszy zestaw algorytmów dostępnych na systemie
    for algo in sorted(hashlib.algorithms_available):
        try:
            h = hashlib.new(algo, text.encode()).hexdigest()
        except (ValueError, TypeError):
            # nieobsługiwany algorytm w tej konfiguracji Pythona
            continue
        if h == target.lower():
            matches.append((algo, h))
    return matches

if __name__ == "__main__":
    res = find_matching_algorithms(input_text, target_hash)
    if res:
        print("Znaleziono dopasowanie:")
        for algo, h in res:
            print(f" - algorytm: {algo} -> {h}")
    else:
        print("Brak dopasowania w hashlib.algorithms_available.")
        print("Upewnij się, że używasz pełnej listy algorytmów (algorithms_available) lub że biblioteka "
              "implementuje dodatkowe algorytmy (np. RIPEMD-160) w Twojej wersji OpenSSL/Python.")
```

```
natalka@fedora:~$ python3 hash14.py
Znaleziono dopasowanie:
 - algorytm: ripemd160 -> 48cab4b54bef42fddaa6353c68a20b369f40026e
```

# ZAJECIA 3 – szyfrowanie symetryczne

NAJWAZNIEJSZE KOMENDY TYCH ZAJEC:

**Szyfrowanie:** openssl enc -aes-256-ecb -in zad1.txt -K \$(cat key) -out zad1.enc -base64

**Generowanie klucza:** openssl rand -hex 16 > key25

**Deszyfrowanie:** openssl enc -d -aes-256-ecb -in zad1.enc -out zad1.dec -K \$(cat key) -base64

**Odszyfrowanie z funkcja pbkdf2 bez soli:**

```
openssl enc -d -aes-256-ecb -pbkdf2 -iter 356 -nosalt -pass pass:"winter44" -in zad29.enc -out zad29.txt
```

**Szyfrowanie z wektorem inicjalizujacym:**

```
echo -n "<SLOWO>" > zad.txt
```

```
openssl enc -aes-128-cbc -K "$(cat key26.hex)" -iv "$(cat iv26.hex)" -in zad.txt -out enc26.b64 -base64
```

## Zadanie 2.1

2.1 Pod adresem <http://127.0.0.1:2001> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2001/encrypt> oraz <http://127.0.0.1:2001/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2001:2001 --name ex1 docker.io/mazurkatarzyna/symmetric-enc-ex1:latest  
podman run -p 2001:2001 --name ex1 docker.io/mazurkatarzyna/symmetric-enc-ex1:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:2001/encrypt> używając metody HTTP GET. Otrzymasz w odpowiedzi unikalny identyfikator sesji (`session_id`), losowe słowo do zaszyfrowania (`word`) oraz klucz szyfrujący podany jako 32-bajtowy ciąg szesnastkowy (`key_hex`).
- (c) Zaszyfruj słowo odebrane od serwera użyciu algorytmu AES-256 w trybie ECB wykorzystując odebrany od serwera klucz szyfrujący.
- (d) Po wykonaniu szyfrowania, zaszyfrowany ciąg znaków zakoduj w formacie `base64`.
- (e) Po uzyskaniu zaszyfrowanego i zakodowanego wyniku wyślij go do serwera poprzez endpoint <http://127.0.0.1:2001/submit> używając metody HTTP POST, w którym przekażesz zarówno identyfikator sesji (jako `session_id`), jak i zaszyfrowany ciąg w formacie `base64` (jako `encrypted_b64`). Serwer w odpowiedziwróci odpowiednią informację o sukcesie lub błędzie.

### UWAGI:

- Aby zaszyfrować wylosowane przez serwer słowo, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użij narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

**Openssl list –cipher-commands – tu mamy funkcje szyfrujące**

```
openssl ciphers -v
```

```
echo -n "hello" > zad1.txt
```

```
openssl rand -help
```

**Tworzenie klucza:** 256 bitow, rand zawsze w bajtach nam generuje ten klucz -> 256/8=32

**Zapisywanie klucza do pliku:**

```
openssl rand -hex 32 > key
```

```
└─ $ cat zad1.txt
```

```
Hello(.python_env) └─[admin@parrot]─[~]
```

```
└─ $ cat key
```

660a9933a4e63f70f28b35bc252f6bcc05c70a6ae94f3ae1c0756d965eb7bf6

Mamy teraz dane wejsciowe i klucz, poniewaz to ocb to mozemy wykonac szyfrowanie  
openssl enc –help

**Na pewno -e bo chcemy szyfrować, -d bo chcemy deszyfrować**

## Padding

HELLO -> HE | LL | OO

Nasz algorytm tylko po dwa znaki szyfruje, jak użyjemy opcji do paddingu, to nam uzupełni zerami, gdy zabraknie nam cyferek i będzie w stanie to zaszyfrować

## Szyfrowanie:

```
openssl enc -aes-256-ecb -in zad1.txt -K $(cat key) -out zad1.enc
```

Dodanie -base64 powoduje, że mamy bardziej czytelny tekst zaszyfrowany i możemy go wysłać do serwera (poprzez skopiowanie)

Z base64 jest zaszyfrowana i kodowana

Hello -> AES-256-ECB -> zad1.enc -base64-> zad.enx

Musimy teraz odkodowac, najpierw base64, nastepnie deszyfrowanie (AES-256-ECB) jak sie uda to dostajemy napis -> Hello

## AES-256-ECB – kod

## Teraz chcemy odszyfrować:

```
openssl enc -d -aes-256-ecb -in zad1.enc -out zad1.dec -K $(cat key) -base64
```

## Zadanie 2.2

**2.2** Pod adresem <http://127.0.0.1:2002> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2002/decrypt> oraz <http://127.0.0.1:2002/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2002:2002 --name ex2 docker.io/mazurkatarzyna/symmetric-enc-ex2:latest  
podman run -p 2002:2002 --name ex2 docker.io/mazurkatarzyna/symmetric-enc-ex2:latest
```

- (b) Wyslij request do endpointa <http://127.0.0.1:2002/decrypt> używając metody HTTP GET. Otrzymasz w odpowiedzi unikalny identyfikator sesji (`session_id`), zakodowane i zaszyfrowane słowo (`encrypted_b64`) oraz klucz deszyfrujący podany jako 32-bajtowy ciąg szesnastkowy (`key_hex`).
- (c) Odkoduj otrzymane słowo, stosując kodowanie `base64`. Następnie odszyfruj odkodowane słowo algorytmem AES-256 w trybie ECB z użyciem podanego klucza deszyfrującego.
- (d) Po uzyskaniu odszyfrowanego i odkodowanego wyniku wyslij go do serwera poprzez endpoint <http://127.0.0.1:2002/submit> używając metody HTTP POST, w którym przekażesz zarówno identyfikator sesji (jako `session_id`), jak i odkodowane i odszyfrowane słowo (jako `decrypted_word`). Serwer w odpowiedzi zwróci odpowiednią informację o sukcesie lub błędzie.

```
(.python_env) [admin@parrot] ~  
└─$ cat zad2.enc  
TnsG6z3G0jZnPq7UMsYkYw==  
(.python_env) [admin@parrot] ~  
└─$ cat key2  
7aa82fd9088b7c3b04f55eaaa931e5c65cab080f22106962ad3ae5a12c8fcce4c  
(.python_env) [admin@parrot] ~  
└─$ openssl enc -d -aes-256-ecb -in zad2.enc -out zad2.dec -K $(cat key2) -base64  
(.python_env) [admin@parrot] ~  
└─$ cat zad2.dec  
UMCS(.python_env) [admin@parrot] ~  
└─$
```

## Zadanie 2.3

**2.3** Pod adresem <http://127.0.0.1:2003> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2003/encrypt> oraz <http://127.0.0.1:2003/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2003:2003 --name ex3 docker.io/mazurkatarzyna/symmetric-enc-ex3:latest  
podman run -p 2003:2003 --name ex3 docker.io/mazurkatarzyna/symmetric-enc-ex3:latest
```

- (b) Wyslij request do endpointa <http://127.0.0.1:2003/encrypt> używając metody HTTP GET. Otrzymasz w odpowiedzi unikalny identyfikator sesji (`session_id`), losowe słowo do zaszyfrowania (`word`) oraz klucz szyfrujący podany jako 32-bajtowy ciąg szesnastkowy (`key_hex`).
- (c) Zaszyfruj otrzymane od serwera słowo, stosując algorytm CAMELLIA-128 w trybie ECB wykorzystując odebrany od serwera klucz szyfrujący.
- (d) Po wykonaniu szyfrowania, zaszyfrowany ciąg znaków zakoduj w formacie `base64`.
- (e) Po uzyskaniu zaszyfrowanego i zakodowanego wyniku wyslij go do serwera poprzez endpoint <http://127.0.0.1:2003/submit> używając metody HTTP POST, w którym przekażesz zarówno identyfikator sesji (jako `session_id`), jak i zaszyfrowany ciąg w formacie `base64` (jako `encrypted_b64`). Serwer w odpowiedzi zwróci odpowiednią informację o sukcesie lub błędzie.

### UWAGI:

- Aby zaszyfrować wylosowane przez serwer słowo, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użyj narzędzia curl. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

## Skad wiemy jakie hex? Hex bedzie tutaj 128/8=16

Na samym poczatku sobie randomowe słowo wpisalismy (na zajeciach nie dzialal docker)

Echo -n "costam" > zad3.txt

## Odpowiedz na zadanie:

```
openssl rand -hex 16 > key25  
openssl enc -camellia-128-ecb -in zad3.txt -K $(cat key2) -out zad3.enc
```

## Zadanie 2.4

2.4 Pod adresem <http://127.0.0.1:2004> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2004/decrypt> oraz <http://127.0.0.1:2004/submit>.

(a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2004:2004 --name ex4 docker.io/mazurkatarzyna/symmetric-enc-ex4:latest
podman run -p 2004:2004 --name ex4 docker.io/mazurkatarzyna/symmetric-enc-ex4:latest
```

(b) Wyślij request do endpointa <http://127.0.0.1:2004/decrypt> używając metody HTTP GET. Otrzymasz w odpowiedzi unikalny identyfikator sesji (`session_id`), zakodowane i zaszyfrowane słowo (`encrypted_b64`) oraz klucz deszyfrujący podany jako 32-bajtowy ciąg szesnastkowy (`key_hex`).

(c) Odkoduj otrzymane słowo, stosując kodowanie `base64`. Następnie odszyfruj odkodowane słowo algorymem CAMELLIA-128 w trybie ECB z użyciem podanego klucza deszyfrującego.

(d) Po uzyskaniu odszyfrowanego i odkodowanego wyniku wyślij go do serwera poprzez endpoint <http://127.0.0.1:2004/submit> używając metody HTTP POST, w którym przekażesz zarówno identyfikator sesji (jako `session_id`), jak i odkodowane i odszyfrowane słwo (jako `decrypted_word`). Serwer w odpowiedzi wróci odpowiednią informację o sukcesie lub błędzie.

#### UWAGI:

- Aby odszyfrować zaszyfrowane przez serwer słwo, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użyj narzędzia curl. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

The screenshot shows a Parrot OS desktop environment. In the top right corner, there's a system tray icon for a terminal window labeled "Parrot Terminal". The main window is a terminal window titled "Parrot Terminal" containing the following command-line session:

```
Programy Miejsca System
● ● ●
Edycja Widok Szukaj Terminal Pomoc
cc424917b32b862ee396841aed0e5b8 IntelliJ IDEA
(.python_env) └─[admin@parrot]─[~]
└─$ cat zad3.txt
slowo(.python_env) └─[admin@parrot]─[~]
└─$ openssl enc -camellia-128-ecb -in zad3.txt -K $(cat key) -out zad3.enc
hex string is too long, ignoring excess
(.python_env) └─[admin@parrot]─[~] Community Edition
└─$ openssl rand -hex 16 > key2
(.python_env) └─[admin@parrot]─[~]
└─$ openssl enc -camellia-128-ecb -in zad3.txt -K $(cat key) -out zad3.enc
hex string is too long, ignoring excess
(.python_env) └─[admin@parrot]─[~]
└─$ openssl enc -camellia-128-ecb -in zad3.txt -K $(cat key2) -out zad3.enc
(.python_env) └─[admin@parrot]─[~]
└─$ openssl enc -d -camellia-128-ecb -in zad3.out -K $(cat key2) -out zad3.txt
Can't open 'zad3.out' for reading, No such file or directory
400780007D7F0000: error:80000002:system library:BIO_new_file:No such file or directory:../crypto/bio/bss_file.c:67:calling fopen(zad3.out, "rb")
400780007D7F0000: error:10000000:BIO routines:BIO_new_file:no such file:../crypto/bio/bss_file.c:75:anie base64 powoduje, że mamy bardziej czytelny tekst zaszyfrowany i możemy go wysłać do serwera (poprzez skopiowanie)
(.python_env) └─[admin@parrot]─[~]
└─$ openssl enc -d -camellia-128-ecb -in zad3.enc -K $(cat key2) -out zad3.txt
(.python_env) └─[admin@parrot]─[~]
└─$ cat zad3.txt
slowo(.python_env) └─[admin@parrot]─[~]
└─$ cat zad3.enc
(.python_env) └─[admin@parrot]─[~]
└─$ openssl enc -camellia-128-ecb -in zad3.txt -K $(cat key) -out zad3.enc -base64
hex string is too long, ignoring excess
(.python_env) └─[admin@parrot]─[~]
└─$ openssl enc -camellia-128-ecb -in zad3.txt -K $(cat key2) -out zad3.enc -base64
(.python_env) └─[admin@parrot]─[~]
└─$ cat zad3.enc
/12JgwBV77sCfa6yIGbDw==
(.python_env) └─[admin@parrot]─[~]
└─$ openssl enc -d -camellia-128-ecb -in zad3.enc -K $(cat key2) -out zad3.dec -base64
(.python_env) └─[admin@parrot]─[~]
└─$ cat zad3.dec
slowo(.python_env) └─[admin@parrot]─[~]
└─$
```

The terminal window has a scroll bar at the bottom. Below the terminal, there's a status bar with icons for menu, terminal, security, and file operations. To the right of the terminal, there's a browser window showing a SharePoint page with some text and a warning message about base64 encoding.

└─\$ cat zad25.txt | md5sum

bef0fb6eb96c747beaf048c9a5a44a1b -

(.python\_env) └─[admin@parrot]─[~]

└─\$ cat zad25.dec | md5sum

bef0fb6eb96c747beaf048c9a5a44a1b -

**Szyfrowanie i deszyfrowanie nie wpływa na zawartość pliku; w pewnym momencie plik był nieczytelny ale finalnie jest do odczytania**

## Zadanie 2.5

**2.5** Pod adresem <http://127.0.0.1:2005> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2005/encrypt> oraz <http://127.0.0.1:2005/submit>.

(a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2005:2005 --name ex5 docker.io/mazurkatarzyna/symmetric-enc-ex5:latest
podman run -p 2005:2005 --name ex5 docker.io/mazurkatarzyna/symmetric-enc-ex5:latest
```

(b) Wyślij request do endpointa <http://127.0.0.1:2005/encrypt> używając metody HTTP GET. Otrzymasz w odpowiedzi unikalny identyfikator sesji (`session_id`) oraz losowe słowo do zaszyfrowania (`word`).

(c) Wygeneruj klucz szyfrujący (klucz musi mieć 192 bity).

(d) Zaszyfruj otrzymane od serwera słowo, stosując algorytm ARIA-192 w trybie ECB wykorzystując wygenerowany przez Ciebie (w punkcie **2.5c**) klucz szyfrujący.

(e) Po wykonaniu zaszyfrowania zaszyfrowany ciąg znaków zakoduj w formacie `base64`.

(f) Po uzyskaniu zaszyfrowanego i zakodowanego wyniku wyślij go do serwera poprzez endpoint <http://127.0.0.1:2005/submit> używając metody HTTP POST, w którym przekażesz: identyfikator sesji (`session_id`), zaszyfrowany ciąg w formacie `base64` (`encrypted_b64`), wygenerowany przez Ciebie klucz szyfrujący w formacie szesnastkowym (`key_hex`, 192 bity). Serwer w odpowiedzi zwróci odpowiednią informację o sukcesie lub błędzie.

#### **UWAGI:**

- Aby zaszyfrować wylosowane przez serwer słowo i wygenerować klucz szyfrujący wykorzystaj narzędzie `OpenSSL`.
- Aby wysłać odpowiedź do serwera, użyj narzędzia `cURL`. Pamiętaj o dodaniu nagłówka protokołu HTTP `Content-Type: application/json`.

```
natalka@fedora:~$ curl -X GET http://127.0.0.1:2005/encrypt
{"session_id":"34bb31dfaca36a0b","word":"blur13"}
natalka@fedora:~$ echo -n "blur13" > zad25.txt
natalka@fedora:~$ cat key25
d16c18f33584acbf19fa4efa3e1bb2b9cbf66311cf30fb3
natalka@fedora:~$ openssl enc -aria-192-ecb -in zad25.txt -K $(cat key25) -out zad25.enc -base64
natalka@fedora:~$ cat zad25.enc
TWF7mPbKLn0d25/F5izVA==
natalka@fedora:~$ curl -X POST http://127.0.0.1:2005/submit -d '{"session_id":"34bb31dfaca36a0b", "encrypted_b64":"TWF7mPbKLn0d25/F5izVA==", "key_hex":"d16c18f33584acbf19fa4efa3e1bb2b9cbf66311cf30fb3"}' -H "Content-Type:application/json"
{"result":"Correct encryption!"}
natalka@fedora:~$
```

### **1) Komenda na uzyskanie hasha:**

```
curl -X GET http://127.0.0.1:2005/encrypt
```

```
echo -n "blur13" > zad25.txt
```

### **2) Tworzenie klucza 192 => 192/8=24**

```
openssl rand -hex 24 > key25
```

### **3) Hashowanie slowa:**

```
openssl enc -aria-192-ecb -in zad25.txt -K $(cat key25) -out zad25.enc -base64
```

```
Cat zad25.enc
```

```
cat klucz25
```

### **4) Wysyłanie zapytania na server:**

```
curl -X POST http://127.0.0.1:2005/submit -d '{"session_id":"34bb31dfaca36a0b",
"encrypted_b64":"TWF7mPbKLn0d25/F5izVA==",
"key_hex":"d16c18f33584acbf19fa4efa3e1bb2b9cbf66311cf30fb3"}' -H "Content-Type:application/json"
```

## **Zadanie 2.6**

2.6 Pod adresem <http://127.0.0.1:2006> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2006/encrypt> oraz <http://127.0.0.1:2006/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2006:2006 --name ex6 docker.io/mazurkatarzyna/symmetric-enc-ex6:latest  
podman run -p 2006:2006 --name ex6 docker.io/mazurkatarzyna/symmetric-enc-ex6:latest
```

- (b) Wyślij żądanie HTTP GET do endpointa <http://127.0.0.1:2006/encrypt>. Otrzymasz w odpowiedzi unikalny identyfikator sesji (`session_id`) oraz losowe słowo do zaszyfrowania (`word`).
- (c) Wygeneruj klucz szyfrujący (klucz musi mieć 128 bitów).
- (d) Wygeneruj wektor inicjalizacyjny (IV) (klucz musi mieć 128 bitów).
- (e) Zaszyfruj pobrane od serwera słowo rzy użyciu algorytmu AES-128 w trybie CBC, z kluczem szyfrującym oraz wektorem inicjalizującym (IV) wygenerowanymi przez Ciebie (w punktach 2.6c i 2.6d).
- (f) Po wykonaniu szyfrowania zakoduj zaszyfrowany ciąg do formatu `base64`.
- (g) Po uzyskaniu zaszyfrowanego i zakodowanego wyniku wyślij go do serwera poprzez endpoint <http://127.0.0.1:2006/submit> używając metody HTTP POST, przekazując następujące dane: identyfikator sesji (`session_id`), zaszyfrowany ciąg w formacie `base64` (`encrypted_b64`), klucz szyfrujący w formacie szesnastkowym (`key_hex`, 128 bitów) oraz IV w formacie szesnastkowym (`iv_hex`, 128 bitów). Serwer w odpowiedzi zwróci odpowiednią informację o sukcesie lub błędzie.

#### UWAGI:

- Aby zaszyfrować wylosowane przez serwer słowo, wygenerować klucz szyfrujący oraz IV, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

## CBC – dodatkowa losowość tekstu którego dostajemy po zaszyfrowaniu

### Wektor inicjalizacyjny to losowa wartość

```
(.python_env) └─[admin@parrot]─[~]
```

```
└─ $openssl enc -d -aes-128-cbc -in zad26.enc -K $(cat key26) -iv  
00000000000000000000000000000000 -out zad26.dec -base64
```

```
(.python_env) └─[admin@parrot]─[~]
```

```
└─ $cat zad26.dec
```

zaszyfrować

Działaaa :)

Ale ona zrobiła ten wektor w taki sposób, że, utworzyła go tak jak klucz, czyli `openssl rand -hex 16 > iv26`

No i klucz tak samo: `openssl rand -hex16 > klucz26`

```
echo -n "<SLOWO>" > zad.txt
```

Następnie szyfrujemy:

```
openssl enc -aes-128-cbc -K "$(cat key26.hex)" -iv "$(cat iv26.hex)" -in zad.txt -out zad.out -base64
```

2.7 Pod adresem <http://127.0.0.1:2007> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2007/decrypt> oraz <http://127.0.0.1:2007/submit>.

(a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2007:2007 --name ex7 docker.io/mazurkatarzyna/symmetric-enc-ex7:latest  
podman run -p 2007:2007 --name ex7 docker.io/mazurkatarzyna/symmetric-enc-ex7:latest
```

(b) Wyslij żądanie HTTP GET do endpointa <http://127.0.0.1:2007/decrypt>. W odpowiedzi otrzymasz identyfikator sesji (`session_id`), zakodowany w formacie base64 ciąg zaszyfrowanego tekstu (`encrypted_b64`), użyté hasło (`password`) oraz IV w formacie szesnastkowym (`iv_hex`). Serwer zaszyfrował losowo wybrane słowo przy użyciu algorytmu AES-256-CBC, z kluczem wygenerowanym na podstawie hasła przy użyciu funkcji PBKDF2, oraz losowego wektora inicjalizującego (IV).

(c) Odkoduj otrzymane słowo, stosując kodowanie base64. Następnie odszyfruj odkodowane słowo algorytmem AES-256-CBC, z użyciem kluczem wygenerowanym na podstawie hasła przy użyciu funkcji PBKDF2, oraz losowego wektora inicjalizującego (IV).

(d) Pamiętaj, że do szyfrowania NIE użyto soli, więc w deszyfrowaniu również NIE JEST potrzebna.

(e) Wyslij do serwera odpowiedź poprzez endpoint <http://127.0.0.1:2007/submit>, używając metody HTTP POST, przekazując dane w formacie JSON: identyfikator sesji (`session_id`) oraz odszyfrowane słowo (`decrypted_word`). Serwer w odpowiedzi zwróci odpowiednią informację o sukcesie lub błędzie.

#### UWAGI:

- Aby odszyfrować zaszyfrowane przez serwer słowo, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

## 1. Uruchomienie kontenera:

```
podman run -p 2007:2007 --name ex7 docker.io/mazurkatarzyna/symmetric-enc-ex7:latest
```

## 2. Pobranie danych z servera

```
curl -X GET http://127.0.0.1:2007/decrypt
```

Wyjdzie:

```
{"encrypted_b64": "+NMLdMcfcTHXoyrWx39MO/ujcvl/Za1J+MoGVnB9usw=", "iv_hex": "cf3e23c9cf03cc686157dd4445dc7d8d", "password": "jump44", "session_id": "e02f1255229897a2"}
```

```
echo "+NMLdMcfcTHXoyrWx39MO/ujcvl/Za1J+MoGVnB9usw=" | base64 -d > zad27.enc
```

## 3.Odszyfrowanie:

```
openssl enc -d -aes-256-cbc -pbkdf2 -nosalt -pass pass:"jump44" -in zad27.enc -out zad27.out -iv cf3e23c9cf03cc686157dd4445dc7d8d
```

## 4. Sprawdzenie wyniku

```
natalka@fedora:~$ cat zad27.out
```

labline

Wysłanie zapytania do servera: (to juz z chatu randomowe słowo, robimy tak samo z tym naszym xD)

```
curl -X POST http://127.0.0.1:2009/submit -H "Content-Type: application/json" -d '{ "session_id": "a42f8de3b111", "decrypted_word": "szyfrowanie" }'
```

2.8 Pod adresem <http://127.0.0.1:2008> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2008/decrypt> oraz <http://127.0.0.1:2008/submit>.

(a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2008:2008 --name ex8 docker.io/mazurkatarzyna/symmetric-enc-ex8:latest  
podman run -p 2008:2008 --name ex8 docker.io/mazurkatarzyna/symmetric-enc-ex8:latest
```

- (b) Wyślij żądanie HTTP GET do endpointa <http://127.0.0.1:2008/decrypt>. W odpowiedzi otrzymasz identyfikator sesji (`session_id`), zakodowany w formacie base64 ciąg zaszyfrowane, losowe słowo (`encrypted_b64`), użyte hasło (`password`) oraz IV w formacie szesnastkowym (`iv_hex`). Serwer zaszyfrował losowe wybrane słowo przy użyciu algorytmu 3DES, z kluczem wygenerowanym na podstawie hasła przy użyciu funkcji PBKDF2, oraz losowego wektora inicjalizującego (IV).
- (c) Odkoduj otrzymane słowo, stosując kodowanie base64. Następnie, odszyfruj otrzymany ciąg, stosując algorytm 3DES, funkcję PBKDF2 do wyprowadzenia klucza z hasła oraz przekazany IV.
- (d) Pamiętaj, że do szyfrowania NIE użyto soli, więc w deszyfrowaniu również NIE JEST potrzebna.
- (e) Wyślij do serwera odpowiedź poprzez endpoint <http://127.0.0.1:2008/submit>, używając metody HTTP POST, przekazując dane w formacie JSON: identyfikator sesji (`session_id`) oraz odszyfrowane słowo (`decrypted_word`). Serwer w odpowiedzi zwróci odpowiednią informację o sukcesie lub błędzie.

---

Bezpieczeństwo Systemów Komputerowych - Szyfrowanie Symetryczne | Katarzyna Mazur

#### UWAGI:

- Aby odszyfrować zaszyfrowane przez serwer słowo, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

### 1. Uruchomienie kontenera

```
podman run -p 2008:2008 --name ex7 docker.io/mazurkatarzyna/symmetric-enc-ex8:latest
```

### 2. Zapisywanie zaszyfrowanego hasła (encrypted\_b64):

```
echo "9RV4Uqan1d3VMP7pOByZkzKaXhaGEm+U" | base64 -d > zad28.enc
```

### 3.Odszyfrowanie

```
openssl enc -d -des-ede3-cbc -pbkdf2 -nosalt -pass pass:"may272008" -in zad28.enc -out zad28.out -iv 90e2e0c8fa6b4ef5
```

### 4. Sprawdzenie wyniku:

```
Cat zad28.out
```

```
natalka@fedora:~$ curl -X GET http://127.0.0.1:2008/decrypt  
{"encrypted_b64":"9RV4Uqan1d3VMP7pOByZkzKaXhaGEm+U","iv_hex":"90e2e0c8fa6b4ef5","password":"may272008","session_id":"7f168ab8c4673f6c"}  
natalka@fedora:~$ echo "9RV4Uqan1d3VMP7pOByZkzKaXhaGEm+U" | base64 -d > zad28.enc  
natalka@fedora:~$ openssl enc -d -des-ede3-cbc -pbkdf2 -nosalt -pass pass:"may272008" -in zad28.enc -out zad28.out -iv 90e2e0c8fa6b4ef5  
natalka@fedora:~$ cat zad28.out  
yoYo13
```

Wysłanie zapytania do servera: (to juz z chatu randomowe słowo, robimy tak samo z tym naszym xD)

```
curl -X POST http://127.0.0.1:2009/submit -H "Content-Type: application/json" -d '{ "session_id": "a42f8de3b111", "decrypted_word": "szyfrowanie" }'
```

**2.9** Pod adresem <http://127.0.0.1:2009> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2009/decrypt> oraz <http://127.0.0.1:2009/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2009:2009 --name ex9 docker.io/mazurkatarzyna/symmetric-enc-ex9:latest  
podman run -p 2009:2009 --name ex9 docker.io/mazurkatarzyna/symmetric-enc-ex9:latest
```

(b) Wżądanie HTTP GET do endpointa <http://127.0.0.1:2009/decrypt>. W odpowiedzi otrzymasz identyfikator sesji (`session_id`), zakodowany w formacie base64 ciąg zaszyfrowane, losowe słowo (`encrypted_b64`) oraz użyte hasło (`password`). Serwer zaszyfrował losowo wybrane słowo przy użyciu algorytmu AES-256-ECB, z kluczem wygenerowanym na podstawie hasła przy użyciu funkcji PBKDF2 z liczbą iteracji równą 356.

- (c) Odkoduj otrzymane słowo, stosując kodowanie `base64`. Następnie, odszyfruj otrzymany ciąg, stosując algorytm AES-256-ECB, z kluczem wygenerowanym na podstawie hasła przy użyciu funkcji PBKDF2 z liczbą iteracji równą 356.
- (d) Pamiętaj, że do szyfrowania NIE użyto soli, więc w deszyfrowaniu również NIE JEST potrzebna.
- (e) Wyślij do serwera odpowiedź poprzez endpoint <http://127.0.0.1:2009/submit>, używając metody HTTP POST, przekazując dane w formacie JSON: identyfikator sesji (`session_id`) oraz odszyfrowane słowo (`decrypted_word`).

#### **UWAGI:**

- Aby odszyfrować zaszyfrowane przez serwer słowo, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

## To samo co wcześniejsz robiemy

**2.10** Pod adresem <http://127.0.0.1:2010> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2010/decrypt> oraz <http://127.0.0.1:2010/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2010:2010 --name ex10 docker.io/mazurkatarzyna/symmetric-enc-ex10:latest  
podman run -p 2010:2010 --name ex10 docker.io/mazurkatarzyna/symmetric-enc-ex10:latest
```

(b) Wyślij żądanie HTTP GET do endpointa <http://127.0.0.1:2010/decrypt>. W odpowiedzi otrzymasz identyfikator sesji (`session_id`), zakodowany w formacie base64 ciąg zaszyfrowanego tekstu (`encrypted_b64`), klucz użyty do szyfrowania (`key_hex`) oraz IV (jeśli wylosowany algorytm go wymaga, `iv_hex`).

- (c) Twoim zadaniem jest odszyfrowanie zaszyfrowanego słowa przesłanego przez serwer. Serwer szyfruje losowo wybrane słowo przy użyciu losowego algorytmu dostępnego w bibliotece OpenSSL.
- (d) Odgadnij, jaki algorytm został wykorzystany do zaszyfrowania słowa, i odszyfruj otrzymany ciąg. (Pamiętaj, że po zaszyfrowaniu ciąg był zakodowany przy użyciu kodowania base64, więc najpierw odkoduj, a później odszyfruj ciąg.)
- (e) Wyślij do serwera odpowiedź poprzez endpoint <http://127.0.0.1:2010/submit>, używając metody HTTP POST, przekazując dane w formacie JSON: identyfikator sesji (`session_id`), odszyfrowane słowo (`decrypted_word`) i znaleziony algorytm, którym słowo zostało zaszyfrowane (`algorithm`).

#### **UWAGI:**

- Aby odszyfrować zaszyfrowane przez serwer słowo, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

## 1. Uruchomienie kontenera

podman run -p 2010:2010 --name ex10 docker.io/mazurkatarzyna/symmetric-enc-ex10:latest

## 2. Wysłanie zapytania o dane

curl -X GET <http://127.0.0.1:2010/decrypt>

Mi taki wyszedł:

curl -X GET <http://127.0.0.1:2010/decrypt>

No i jak chodzi o odkodowanie, to polecają zrobić skrypt w python:

```

Alternatywnie w Pythonie (jeśli wolisz)

python

import requests
r = requests.get("http://127.0.0.1:2010/decrypt").json()
key_hex = r["key_hex"]
iv_hex = r.get("iv_hex", "")

key_len = len(key_hex)//2
iv_len = len(iv_hex)//2

print(f"key={key_len}B, iv={iv_len}B")

if iv_len == 0:
    print("ECB mode")
elif iv_len == 16:
    print("CBC AES")
elif iv_len == 8:
    print("CBC DES/3DES")

```

Zeby łatwiej sprawdzac wszystkie algorytmy xD

2.11 Pod adresem <http://127.0.0.1:2011> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2011/encrypt> oraz <http://127.0.0.1:2011/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2011:2011 --name ex11 docker.io/mazurkatarzyna/symmetric-enc-ex11:latest
podman run -p 2011:2011 --name ex11 docker.io/mazurkatarzyna/symmetric-enc-ex11:latest
```

- (b) Wyślij żądanie HTTP GET do endpointa <http://127.0.0.1:2011/encrypt>. W odpowiedzi otrzymasz identyfikator sesji (`session_id`), obrazek do zaszyfrowania (`image_data_base64`), klucz do szyfrowania (`key_hex`) oraz IV (`iv`).
- (c) Do zaszyfrowania obrazka użyj algorytmu ARIA-128-CTR (bez paddingu), wykorzystując odebrany od serwera klucz szyfrujący oraz IV.
- (d) Wyślij do serwera odpowiedź poprzez endpoint <http://127.0.0.1:2011/submit>, używając metodę HTTP POST, przekazując dane w formacie JSON: identyfikator sesji (`session_id`) i zaszyfrowany obrazek zakodowany w base64 (obrazek jest najpierw zaszyfrowany, a potem zakodowany) (jako `encrypted_data`). Serwer zwróci w odpowiedzi informację o tym, czy obrazek jest prawidłowo zaszyfrowany.

#### UWAGI:

- Aby zaszyfrować wylosowany przez serwer obrazek, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użyj narzędzia curl. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

`podman run -p 2011:2011 --name ex11 docker.io/mazurkatarzyna/symmetric-enc-ex11:latest`

`curl -X GET http://127.0.0.1:2011/encrypt`

`echo "iVBORw0KGgoAAAANSUhEUgAAAAUA..." | base64 -d > image.png`

### Zaszyfruj obraz ARIA-128-CTR

`openssl enc -aria-128-ctr -K aabbccddeeff00112233445566778899 -iv 0102030405060708090a0b0c0d0e0f10 -in image.png -out image.enc`

### Zakoduj zaszyfrowany plik w Base64

`base64 image.enc > image.enc.b64`

`curl -X POST http://127.0.0.1:2011/submit -H "Content-Type: application/json" -d '{ "session_id": "b85d01b9f3e2", "encrypted_data": """$(cat image.enc.b64)"""}'`

2.12 Pod adresem <http://127.0.0.1:2012> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:2012/decrypt> oraz <http://127.0.0.1:2012/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 2012:2012 --name ex12 docker.io/mazurkatarzyna/symmetric-enc-ex12:latest  
podman run -p 2012:2012 --name ex12 docker.io/mazurkatarzyna/symmetric-enc-ex12:latest
```

- (b) Wyslij żądanie HTTP GET do endpointa <http://127.0.0.1:2012/decrypt>. W odpowiedzi otrzymasz identyfikator sesji (`session_id`), obrazek do odszyfrowania (`image_data_base64`), klucz do odszyfrowania (`key_hex`) oraz IV (`iv`).

- (c) Do odszyfrowania obrazka użyj algorytmu ARIA-256-CFB (bez paddingu), wykorzystując odebrany od serwera klucz deszyfrujący oraz IV.

- (d) Wyslij do serwera odpowiedź poprzez endpoint <http://127.0.0.1:2012/submit>, używając metody HTTP POST, przekazując dane w formacie JSON: identyfikator sesji (`session_id`) i odszyfrowany obrazek zakodowany w `base64` (obrazek jest najpierw jest odszyfrowany, a potem zakodowany) (jako `decrypted_data`). Serwer zwróci w odpowiedzi informację o tym, czy obrazek jest prawidłowo odszyfrowany.

- (e) Zapisz odszyfrowany obrazek na dysku z rozszerzeniem `*.png`.

**UWAGI:**

- Aby odszyfrować wylosowany przez serwer obrazek, wykorzystaj narzędzie OpenSSL.
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP `Content-Type: application/json`.

`podman run -p 2012:2012 --name ex12 docker.io/mazurkatarzyna/symmetric-enc-ex12:latest`

`curl -X GET http://127.0.0.1:2012/decrypt`

`echo "U2FsdGVkX1+XgNAgAAABAgMEBQYHCAkKCw==" | base64 -d > image.enc`

```
openssl enc -d -aria-256-cfb -K  
00112233445566778899aabbcdddeff00112233445566778899aabbcdddeff -iv  
0102030405060708090a0b0c0d0e0f10 -in image.enc -out image.png
```

**Mozna sobie otworzyc ten plik zeby sprawdzic**

**Zakoduj odszyfrowany obrazek z powrotem do Base64**

`base64 image.png > image_decoded.b64`

```
curl -X POST http://127.0.0.1:2012/submit -H "Content-Type: application/json" -d '{ "session_id":  
"f2d5c4e8ab44", "decrypted_data": """$(cat image_decoded.b64)"""}'
```

```
{"status":"success"}
```

ZAJECIA 4 – klucze asymetryczne

A

Sk\_A

Pk\_a

B

Sk\_b

Pk\_b

Kolejny etap: Wymiana kluczy prywatnych i publicznych

Szyfrujemy kluczem publicznym osoby do ktorej chcemy wyslac, ale tez swoim kluczem prywatnym mozna (tylko swoim)

# Zajecia 3 - SZYFROWANIE ASYMETRYCZNE

## Zadanie 3.1

Generujemy parę kluczy (a i b), algorytm rsa, klucz publiczny prywatny, mamy je wyeksportowac do plikow.

Wysylamy do servera je, czy sie wszystko zgadza

**3.1** Pod adresem <http://127.0.0.1:3001> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:3001/pubkey> oraz <http://127.0.0.1:3001/privkey>.

(a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3001:3001 --name ex1 docker.io/mazurkatarzyna/asymmetric-enc-ex1:latest  
podman run -p 3001:3001 --name ex1 docker.io/mazurkatarzyna/asymmetric-enc-ex1:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3001:3001 --name ex1 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex1:latest  
podman run -p 3001:3001 --name ex1 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex1:latest
```

(b) Wygeneruj parę kluczy RSA (publiczny i prywatny). Wyeksportuj oba klucze do plików.

(c) Używając metody HTTP POST, wyślij parę kluczy RSA do serwera poprzez endpointy <http://127.0.0.1:3001/upload/public> (jako file) oraz <http://127.0.0.1:3001/upload/private> (również jako file). W odpowiedzi serwer zwróci odpowiednią informację (szczegóły klucza).

### UWAGI:

- Aby wygenerować parę kluczy RSA, wykorzystaj narzędzie OpenSSL. Użyj argumentu [genpkey](#) oraz [pkey](#).
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówków protokołu HTTP: Content-Type: application/json, Content-Type: multipart/form-data.

## 1. Generowanie pary kluczy

### PRYWATNY i PUBLICZNY:

```
openssl genpkey -algorithm RSA -out priv.pem -pkeyopt rsa_keygen_bits:1024
```

```

Programy Miejsca System
Parrot Terminal
Plik Edycja Widok Szukaj Terminal Pomoc
└─ $ openssl genpkey -algorithm RSA -out priv.pem -pkeyopt rsa_keygen_bits:1024
+++
(.python_env) [admin@parrot] ~
└─ $ openssl pkey -noout priv.pem
pkey: Use -help for summary.
(.python_env) [x] [admin@parrot] ~
└─ $ cat priv.pem
-----BEGIN PRIVATE KEY-----
MIICeATBADANBgkqhkiG9w0BAQEFAASCamIwgJeAgEAAoGBAnx2f2i/12MrdjyF
WFFyS7RwTyOYuadA0ismiQwegzS2emWF8y5t4v11Y/yziimMFfn8yWfofZLW6
kAtvLR124BMegK0ec212vyxEHluqVdu3m+N2RUt0R1J/06ulgJ/TG8R1juXL6
0gEyUFAQUda8+Hsga8BCvfoXkg3ZAgMBAECgYEAbRvOB6TerlrM/m8QuSeffRw
BhSwd59q56q0jkqabWht5fehgMZip4Q6WMafxFRA2Bj3DR17XqvmtAxjjeA359le3.1
vkvk6zzGkAoElc45IulpXVFI7aoevuSpvLkyR82Jnoz9efKKMRay9wmwYaihd
790s134bsJsd0YHqWl0cCQDyewkzlzT6L/SMKqehyDjbJQfd1BkD38tKoWb+tz
do1VKVj90u179Jv0jX0EnubQk11THQiyv77m8W+HAEAGqD1KFc310aTEt
J06KoIeaufc30HvRAsgxInzqiev8CarzusBbi+R2Rraw4AZAgcqMhdhVg9UERA
wLwvwJBALBPYfHnw0tzZ9dfu8P3bJnIY5tg4LtenDLQ8qNQntuIdy60+3Lb
2Xnb6hz/JV7v7404EKPi1/eUzHigvD8CQE493ogAtKZSAGVVmio+Yj/HxfkpZ6Q
yhJtbBxMNsfznqglkwBzRPE9abokX0RnKOBPLn+yaJxgRoOeyDcCQDyJ1U0
Tyt03Pf+Fm+nxUc51Cdcf7jE+/KfJB55Ezqza69Ycx0fv1f4Uyas+SBGZInNB
QAVMcIhYs/jMKUyP
-----END PRIVATE KEY-----
(.python_env) [admin@parrot] ~
└─ $ openssl pkey -text -in priv.pem
-----BEGIN PRIVATE KEY-----
MIICeATBADANBgkqhkiG9w0BAQEFAASCamIwgJeAgEAAoGBAnx2f2i/12MrdjyF
WFFyS7RwTyOYuadA0ismiQwegzS2emWF8y5t4v11Y/yziimMFfn8yWfofZLW6
kAtvLR124BMegK0ec212vyxEHluqVdu3m+N2RUt0R1J/06ulgJ/TG8R1juXL6
0gEyUFAQUda8+Hsga8BCvfoXkg3ZAgMBAECgYEAbRvOB6TerlrM/m8QuSeffRw
-----END PRIVATE KEY-----
1. Generowanie pary kluczy
2. Eksportowanie klucza prywatnego
3. Eksportowanie klucza publicznego

```

`openssl pkeyutil –help`

**Wyswietlanie przez cat** – zakodowane w base64,

**Wyswietlanie przez openssl pkey -text -in priv.pem** - forma tekstowa klucza

## PUBLICZNY (wyodrębniony z prywatnego)

`openssl pkey -in priv.pem -pubout -out pub.pem`

Wyswietlenie:

`└─ $cat pub.pem`

-----BEGIN PUBLIC KEY-----

MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDcWX9ov9djK3Y8hVhWEqgu0cE8

jmLmnQNlrJokMHoM0tnpln/F+subeL9dWP8s4q5zHxZ/Mln6H2S1upAE75UYtuAT

HhpDhHnNtdr8qyhB7qlXbt5wltkVLTkYifzurjICf0xvBkYo7ly+tIBMIIBUAFHW

vPh7IGvAQr39F5IN2QIDAQAB

-----END PUBLIC KEY-----

**Mozemy jeszcze tak wyswietlic (klucz prywatny i publiczny):**

`openssl pkey -in priv.pem -text –noout`

**Mozemy jeszcze tak wyswietlic(klucz publiczny)**

```
Openssl pkey -n priv.pem -text_pub -noout
```

## 2. Wysyłanie do serwera:

### Klucza publicznego:

```
└─ $curl -X 'POST' \
  'http://127.0.0.1:3001/upload/public' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@pub.pem'
```

```
{
  "bits": 1024,
  "bytes": 272,
  "error": null,
  "is_rsa": true,
  "type": "public"
}
```

### Klucza prywatnego:

```
curl -X 'POST' 'http://127.0.0.1:3001/upload/private' -H 'accept: application/json' -H 'Content-Type: multipart/form-data' -F 'file=@priv.pem'
```

## Zadanie 3.2

HTTP: Content-Type: application/json, Content-Type: multipart/form-data.

3.2 Pod adresem <http://127.0.0.1:3002> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:3002/upload/ec/public> oraz <http://127.0.0.1:3002/upload/ec/private>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3002:3002 --name ex2 docker.io/mazurkatarzyna/asymmetric-enc-ex2:latest
podman run -p 3002:3002 --name ex2 docker.io/mazurkatarzyna/asymmetric-enc-ex2:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3002:3002 --name ex2 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex2:latest
podman run -p 3002:3002 --name ex2 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex2:latest
```

- (b) Wygeneruj parę kluczy EC - krzywych eliptycznych (publiczny i prywatny). Wykorzystaj krzywą prime256v1. Wyeksportuj oba klucze do plików.

- (c) Używając metody HTTP POST, wyślij parę kluczy EC do serwera poprzez endpointy <http://127.0.0.1:3002/upload/ec/public> (jako file) oraz <http://127.0.0.1:3002/upload/ec/private> (również jako file). W odpowiedzi serwer zwróci odpowiednią informację (szczegóły klucza).

### UWAGI:

- Aby wygenerować parę kluczy EC, wykorzystaj narzędzie OpenSSL. Użyj argumentu [genpkey](#) oraz [ec](#).
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówków protokołu HTTP: Content-Type: application/json, Content-Type: multipart/form-data.

## Generowanie pary kluczy EC

To ja z chata: openssl genpkey -algorithm EC -pkeyopt ec\_paramgen\_curve:P-256 -out ecpriv.pem

To ona: openssl genpkey -algorithm EC -out ecpriv.pem -pkeyopt ec\_paramgen\_curve:prime256v1

## Wyodrębnienie klucza publicznego:

openssl ec -in ecpriv.pem -pubout -out ecpub.pem

Wysyłanie na serwer:

```
└─ $curl -X 'POST' 'http://127.0.0.1:3002/upload/ec/public' -H 'accept: /' -H 'Content-Type: multipart/form-data' -F 'file=@ecpub.pem'
```

Odpowiedz: { "bytes": 178, "curve": "NIST P-256", "error": null, "is\_ec": true, "type": "public" }

## Zadanie 3.3 - tak samo

3.3 Pod adresem <http://127.0.0.1:3003> działa prosty serwer HTTP udostępniający jeden endpoint: <http://127.0.0.1:3003/checkkeys>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3003:3003 --name ex3 docker.io/mazurkatarzyna/asymmetric-enc-ex3:latest  
podman run -p 3003:3003 --name ex3 docker.io/mazurkatarzyna/asymmetric-enc-ex3:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3003:3003 --name ex3 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex3:latest  
podman run -p 3003:3003 --name ex3 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex3:latest
```

- (b) Wygeneruj parę kluczy RSA (publiczny i prywatny) o długości 1024 bitów. Wyeksportuj oba klucze do plików.
- (c) Używając metody HTTP POST, wyslij parę kluczy RSA do serwera poprzez endpoint <http://127.0.0.1:3003/checkkeys> (jako pliki `private_key.pem` oraz `public_key.pem` w jednym requeście). W odpowiedzi serwer zwróci informację o poprawności i dopasowaniu kluczy (czy klucz publiczny odpowiada przesłanemu kluczowi prywatnemu).

### UWAGI:

- Aby wygenerować parę kluczy RSA, wykorzystaj narzędzie OpenSSL. Użyj argumentu `genpkey` oraz `pkey`.
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP `Content-Type: application/json`.

## Uruchomienie serwera:

podman run -p 3003:3003 --name ex3 docker.io/mazurkatarzyna/asymmetric-enc-ex3:latest

## Generowanie pary kluczy RSA:

openssl genpkey -algorithm RSA -out priv.pem -pkeyopt rsa\_keygen\_bits:1024

## Wyodrębnienie klucza publicznego

openssl pkey -in priv.pem -pubout -out pub.pem

Wysyłanie do serwera (mi nie działa.. Cos tam było chyba z tymi /n, nie znam sposobu..)

curl -X POST '<http://127.0.0.1:3003/checkkeys>'

-H 'Content-Type: application/json'

```
-d '{ "private_key_pem": "-----BEGIN RSA PRIVATE KEY----- resztaKlucza -----END RSA PRIVATE KEY----",  
"public_key_pem": "-----BEGIN PUBLIC KEY-----resztaKlucza-----END PUBLIC KEY----" }'
```

## Zadanie 3.4

3.4 Pod adresem <http://127.0.0.1:3004> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:3004/getprivkey> oraz <http://127.0.0.1:3004/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3004:3004 --name ex4 docker.io/mazurkatarzyna/asymmetric-enc-ex4:latest  
podman run -p 3004:3004 --name ex4 docker.io/mazurkatarzyna/asymmetric-enc-ex4:latest
```

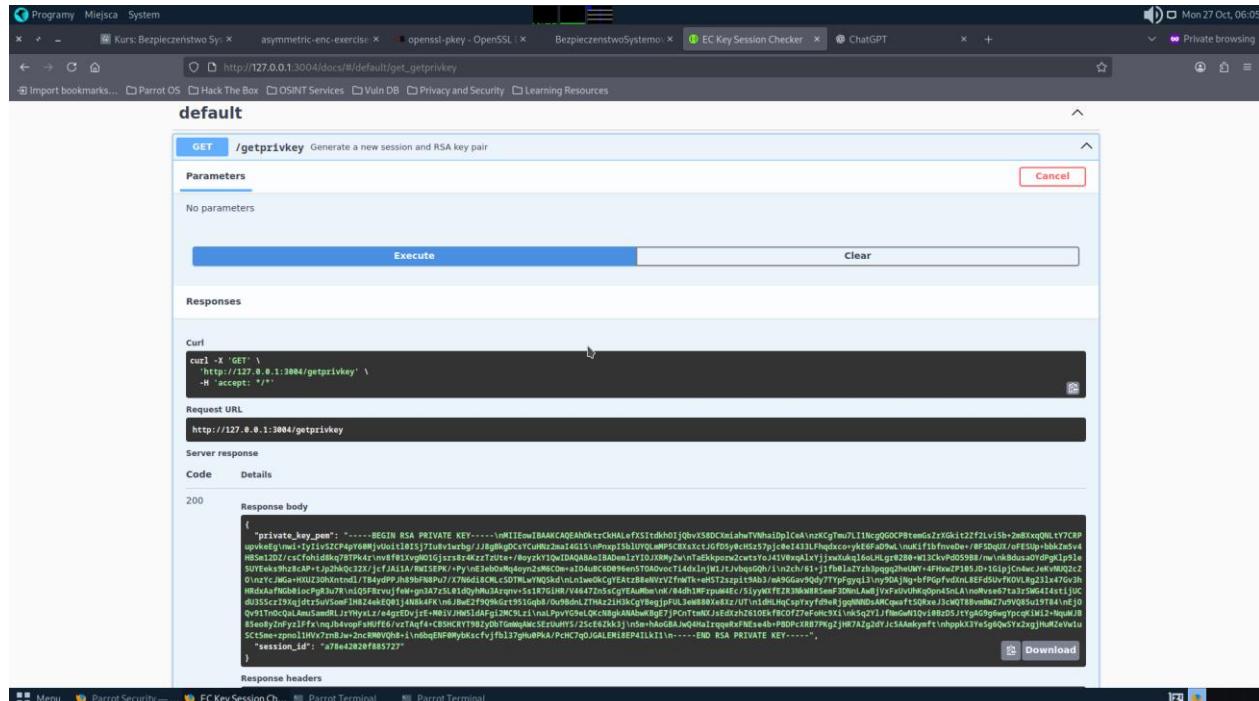
Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3004:3004 --name ex4 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex4:latest  
podman run -p 3004:3004 --name ex4 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex4:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:3004/getprivkey> używając metody HTTP GET. Otrzymasz w odpowiedzi identyfikator sesji (w nagłówku X-Session-ID), oraz klucz prywatny RSA (`private_key_pem`).
- (c) Na podstawie otrzymanego klucza prywatnego, wygeneruj odpowiadający mu klucz publiczny. Ilu bitowy jest klucz?
- (d) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:3004/submit> wygenerowany klucz publiczny (jako plik, `public_key_pem`) oraz identyfikator sesji (`session_id`).

### UWAGI:

- Aby wygenerować klucz publiczny, wykorzystaj narzędzie OpenSSL. Użyj argumentu `pkey`.
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.



```
curl -X 'GET' \  
  'http://127.0.0.1:3004/getprivkey' \  
 -H 'accept: */*'
```

```
Request URL  
http://127.0.0.1:3004/getprivkey
```

```
Server response  
Code Details  
200 Response body  
{  
  "private_key_pem": "-----BEGIN RSA PRIVATE KEY-----\n  \\nMIIEogIBAAKCAQEAqhiYxTvOzZl0cm6f3hcTqAXpFGZhwgNIL4X92B3yY9EVZN7r\\nnrlxd8+v8DlBrItOb+S\n  v3wn1Yg7RVGfsPjaTp430Lkpz/EvKPRK+Zq7MuK1vf2SJ\\nyr1qRtNLD+ioGbdBO6Xhecg7DEWmv8ITxYB+b\n  S5e5m+zn011Hv7zmJw+2nRmVQh6+1+nbqENF#Mybkcfvfb137glu#PKA/PcHC7qQJGALEM18P4tlkI:\\n-----END RSA PRIVATE KEY-----",  
  "session_id": "a7be4202ef88572"
```

Wstęp do zadania:

Wysyłanie do servera zapytania o klucz prywatny:

```
curl -X GET http://127.0.0.1:3004/getprivkey
```

Echo -n “-----BEGIN RSA PRIVATE KEY-----

```
\\nMIIEogIBAAKCAQEAqhiYxTvOzZl0cm6f3hcTqAXpFGZhwgNIL4X92B3yY9EVZN7r\\nnrlxd8+v8DlBrItOb+S\n  v3wn1Yg7RVGfsPjaTp430Lkpz/EvKPRK+Zq7MuK1vf2SJ\\nyr1qRtNLD+ioGbdBO6Xhecg7DEWmv8ITxYB+b
```

```
oGmrE0FofHzvme9eM1FmlmDp3ku\nYypHDIv+K5LOotSNEpnubdJnqGKltZ9ciUYh7TWP54DKW/6ntt9zK
bZNkLoq4Bi6\ngEOvc5Y5/LUXOyTO1LGT6lL7UfpthcD5UGcde8cgvKQMxqe4ATr+H2AjQeqT5Uqo\nClaw
3la+J3zpK1qD8nXFV/onxBcZEANUCENNbQIDAQABAolBACnn2WtuKqr4jXnL\ny1MZ+FvC6QN/nclAsP4U
Us+1wQvabRzm1sHIQOu1nTDfylBKGc7zVmjqSupXuUe\nnnVNOet4i2O+2pBCaU37saUA+/GzbPcby7Ae
o1tAQKkQHG7MKNOYPHhe0vudtGYHQ\nvLJBz+RFyNxWNXcKdvSH9mQWnnlCAWWPPGAK/aUkoRC27
Z9etivqLq+PTn3AZ+QK\nnOOpqgfKIMr0rgsF9gPs+MsVHu+pTY0gzh0UC2/5P0Ve/BGiml9FfeVhPx+s6Cd0
\nkL6E+OH2wEKFyN0DXPCZkd3gkZwuWNO2ERW6LiqaYPQsc52ObrYs9GT2pwmJXYa6\nnUoqfswECgYE
Aw4gHCcUgHx+wdvnBl+xCo3ZfmpEYobRLJ8s1evzUEpY2inXcVJyk\nGxtuefEqqfWfzwiZCwFLHX148h37q
Gq9af0l5ULMpix9nz6ovCoDXwV40YEWuPdp\nn4DQadkDIOcl1Zi4x/vh7SBCt89WVEkFooHA0XSjQVavHtb
l0bidS2M0CgYEA3rLj\n6/3xVHHhGxp6crR4xfUjjMFT6vmlljuvqThKy2iPtSrHClkmdsKm1qfJqjB0z1tY\nnkRp
qCdvpGNVfWQiGmMyz34n7Arx+rLWBh1jZ5eDQ0Hs2X0rvKhVq528/W7WngiL\nn+ril6/hwb/OjqjvB8B64
mSWR9VqskZo7KWhGxyECgYA0FVrSlmitAbLuri6MOkmx\n4wkqUfX+tNjEG9P+E7Sl0s9qaGStQSBRfCgc
YUodBalw63hgvxJ0I4UA8U2kmflk\nndmMOw5Fhvj4kGfu2S3aka/+3xqv3zyerqhVWZIEkXKMx6al1qZiUJRBf
CX+O+AT\nxG+eVQ3RlDVZSbli2WSvDQKBgEEiUUdUnqD0LcKfm/CwmCJN8Hfak5DGKurYSihU\nnLowPq
Phz6oM8T+OsSt+9c14zjfaX4O+PqjP3/dUlDt8bf0JxnKpk0OWb+/DHpXr\nQoep6NpjDcMOuRwUn/nxyLT
XFi5ikY5jKqjZoQkdWWEQ2UhBs/wW4PnB63cqhOM3\nfHwBAoGAD1uJZl0CE36ThOiU3S8QKrPPljzAtBG
mMTUWXTHo8cvelmVPfLhbysCq\nrHu1UiL0jWkCCmftXHoVnBumgsWq394hrU6t6i7nKnT4L9DbwGgT9
c8+GrFqEau\n6D7vfZjEORxgQIRamr8VYmuIW4DuxYlQ9X/Tm5tCqvZ251+9VSw=\n----END RSA
PRIVATE KEY----" > zad34.priv
```

Trzeba pousuwac /n

### KOMENDA do usuwania nowych linii:

```
cat zad34.priv | tr -d '\n' > zad34_no_newline.priv
```

```
openssl pkey -in zad34.priv -pubout -out zad34.pub
```

```
curl -X GET http://127.0.0.1:3004/getprivkey -o response.json
```

```
echo -n '{"private_key_pem": "'\"$(cat response.json)\""}' > responsecorr.json
```

```
cat responsecorr.json | tr -d '\n' > responsecorrno_newline.json
```

```
Cat responsecorrno_newline.json | jq .private_key_pem
```

Wyslanie klucza do servera:

```
Curl -X POST http://127.0.0.1:3004/submit -H "Content-Type: application/json" -d "$(jq -n -arg session_id "xxxx" --arg pub "$(cat pub.pem)" '{session_id: $session_id, public_key_pem: &pub}')"
```

Wysylamy do servera, dane w jsonie i -d bo wysylamy dane, uzywamy jq do tego, aby nie bawic sie w usuwanie nowych linii, tam dodajemy tylko argumenty, potem nastepuje wyslanie do servera session id, no i public key bez nowych linii

**Szyfrowanie i deszyfrowanie za pomoca kluczy ktore mamy (oddzielne zadanie jej)**

```
echo -n "Hello" > slowo.txt
```

```
openssl genpkey -algorithm RSA -out keys.pem -pkeyopt rsa_keygen_bits:1024
```

```
openssl pkey -in keys.pem -pubout -out pub.pem
```

### Szyfrowanie tego słowa kluczem publicznym:

```
openssl pkeyutl -encrypt -pubin -inkey pub.pem -in slowo.txt -pkeyopt rsa_padding_mode:oaep -out ciphertext.enc
```

### Tutaj zeby tekst byl bardziej czytelny:

```
cat ciphertext.enc | base64 > ciphertext.txt
```

### Szyfrujemy publicznym, odszyfrujemy prywatnym

### Odszyfrowanie:

```
└─ $openssl pkeyutl -decrypt -inkey keys.pem -in ciphertext.enc -pkeyopt rsa_padding_mode:oaep -out decrypted.txt
```

## Zadanie 3.5

**3.5** Pod adresem <http://127.0.0.1:3005> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:3005/getprivkey> oraz <http://127.0.0.1:3005/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3005:3005 --name ex5 docker.io/mazurkatarzyna/asymmetric-enc-ex5:latest  
podman run -p 3005:3005 --name ex5 docker.io/mazurkatarzyna/asymmetric-enc-ex5:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3005:3005 --name ex5 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex5:latest  
podman run -p 3005:3005 --name ex5 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex5:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:3005/getprivkey> używając metody HTTP GET. Otrzymasz w odpowiedzi identyfikator sesji (w nagłówku X-Session-ID), oraz klucz prywatny EC (`private_key_pem`).
- (c) Na podstawie otrzymanego klucza prywatnego, wygeneruj odpowiadający mu klucz publiczny. Ilu bitowy jest klucz?
- (d) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:3005/submit> wygenerowany klucz publiczny (jako plik, `public_key_pem`) oraz identyfikator sesji (`session_id`).

### UWAGI:

- Aby wygenerować klucz publiczny, wykorzystaj narzędzie OpenSSL. Użyj argumentu [ec](#).
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

## Zadanie 3.6

**3.6** Pod adresem <http://127.0.0.1:3006> działa prosty serwer HTTP udostępniający 2 endpointy: <http://127.0.0.1:3006/encrypt> oraz <http://127.0.0.1:3006/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3006:3006 --name ex6 docker.io/mazurkatarzyna/asymmetric-enc-ex6:latest  
podman run -p 3006:3006 --name ex6 docker.io/mazurkatarzyna/asymmetric-enc-ex6:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3006:3006 --name ex6 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex6:latest  
podman run -p 3006:3006 --name ex6 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex6:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:3006/encrypt> używając metody HTTP GET. Otrzymasz w odpowiedzi unikalny identyfikator sesji (w nagłówku X-Session-ID), losowe słowo do zaszyfrowania (w nagłówku X-Word) oraz klucz publiczny RSA w formacie PEM (jako `public_key.pem`).
- (c) Zaszyfruj otrzymane słowo, stosując algorytm RSA-2048 z użyciem klucza publicznego oraz wybranego trybu paddingu: OAEP.
- (d) Wyślij request do serwera poprzez endpoint <http://127.0.0.1:3006/submit> używając metody HTTP POST, identyfikator sesji (`session_id`), jak i zaszyfrowane słwo (jako plik, `encrypted_file`).
- (e) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność przesłanego zaszyfrowanego i zakodowanego słowa.

#### **UWAGI:**

- Aby zaszyfrować odebrane od serwera słwo, wykorzystaj narzędzie OpenSSL. Użyj argumentu [pkeyutl](#).
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

### Zadanie 3.7

**3.7** Pod adresem <http://127.0.0.1:3007> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:3007/decrypt> oraz <http://127.0.0.1:3007/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3007:3007 --name ex7 docker.io/mazurkatarzyna/asymmetric-enc-ex7:latest  
podman run -p 3007:3007 --name ex7 docker.io/mazurkatarzyna/asymmetric-enc-ex7:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3007:3007 --name ex7 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex7:latest  
podman run -p 3007:3007 --name ex7 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex7:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:3007/decrypt> używając metody HTTP GET. Otrzymasz w odpowiedzi unikalny identyfikator sesji (w nagłówku X-Session-ID), klucz prywatny oraz zakodowane i zaszyfrowane słwo, oba zapakowane w pliku `*.zip`. (Serwer wygenerował losowe słwo, zaszyfrował je algorytmem RSA (klucz 4096-bit) i zakodował w formacie base64.)
- (c) Odkoduj podane słwo, następnie odszyfruj je przy użyciu klucza prywatnego i algorytmu RSA-4096 z paddingiem OAEP.
- (d) Wyślij request do serwera poprzez endpoint <http://127.0.0.1:3007/submit> używając metody HTTP POST, identyfikator sesji (`session_id`) oraz odszyfrowane słwo (`decrypted_word`).
- (e) W odpowiedzi serwer zweryfikuje poprawność odszyfrowanego słowa i zwróci informację o sukcesie lub błędzie.

#### **UWAGI:**

- Aby odszyfrować odebrane od serwera słwo, wykorzystaj narzędzie OpenSSL. Użyj argumentu [pkeyutl](#).
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

### Zadanie 3.8 - to juz umiemy chyba

**3.8** Pod adresem <http://127.0.0.1:3008> działa prosty serwer HTTP udostępniający jeden endpoint: <http://127.0.0.1:3008/decrypt>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3008:3008 --name ex8 docker.io/mazurkatarzyna/asymmetric-enc-ex8:latest  
podman run -p 3008:3008 --name ex8 docker.io/mazurkatarzyna/asymmetric-enc-ex8:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3008:3008 --name ex8 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex8:latest  
podman run -p 3008:3008 --name ex8 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex8:latest
```

- (b) Wygeneruj parę kluczy RSA (jako `keys.pem`).
- (c) Wyeksporuj klucz publiczny z pary kluczy do pliku (jako `public_key.pem`).
- (d) Wybierz słowo i zapisz je do pliku (jako `plaintext.txt`).
- (e) Zaszyfruj wybrane przez siebie słowo (`plaintext.txt`) wygenerowanym kluczem publicznym (`public_key.pem`). Wykorzystaj padding OAEP.
- (f) Zakoduj zaszyfrowane słowo i zapisz je do pliku (`ciphertext.txt`).
- (g) Wyślij request do endpointa <http://127.0.0.1:3008/decrypt> używając metody HTTP POST i przeslij do serwera:
- wygenerowany klucz prywatny (jako plik `private_key.pem`),
  - wygenerowany klucz publiczny (jako plik `public_key.pem`),
  - wybrane przez siebie słowo (jako plik `plaintext`),
  - wybrane przez siebie słowo zaszyfrowane kluczem publicznym (`public_key.pem`) z paddingiem OAEP i zakodowane w base64 (jako plik `ciphertext.txt`).
- (h) Zadaniem serwera jest sprawdzenie, czy zaszyfrowane słowo zostało zaszyfrowane podanym kluczem, oraz, czy słowa zgadzają się po odszyfrowaniu.

#### **UWAGI:**

- Aby zaszyfrować losowe słowo, wykorzystaj narzędzie OpenSSL. Użyj argumentu [pkeyutl](#).
- Aby wysłać odpowiedź do serwera, użyj narzędzia cURL. Pamiętaj o dodaniu nagłówka protokołu HTTP Content-Type: application/json.

## Zadanie 3.9

**3.9** Pod adresem <http://127.0.0.1:3009> działa prosty serwer HTTP udostępniający jeden endpoint: <http://127.0.0.1:3009/sign>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3009:3009 --name ex9 docker.io/mazurkatarzyna/asymmetric-enc-ex9:latest  
podman run -p 3009:3009 --name ex9 docker.io/mazurkatarzyna/asymmetric-enc-ex9:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3009:3009 --name ex9 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex9:latest  
podman run -p 3009:3009 --name ex9 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex9:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:3009/sign> używając metody HTTP GET. Otrzymasz w odpowiedzi unikalny identyfikator sesji (`session_id`), klucz prywatny RSA (`private_key.pem`) oraz słowo (`word`).
- (c) Podpisz otrzymane słowo (`word`) kluczem prywatnym (`private_key.pem`), a następnie zakoduj wynik do formatu base64. Przy podpisywaniu należy zwrócić uwagę na parametry paddingu PSS:
- Używana funkcja skrótu: SHA-256
  - Salt length: długość soli powinna odpowiadać długości skrótu (32 bajty dla SHA-256), aby podpis był zgodny z weryfikacją po stronie serwera.
- (d) Używając metody HTTP POST, wyślij do serwera poprzez endpoint <http://127.0.0.1:3009/submit> podpisane słowo (jako `signature_b64`) oraz identyfikator sesji (`session_id`).
- (e) Serwer zweryfikuje podpis przy użyciu klucza publicznego i zwróci odpowiednią informację o sukcesie lub błędzie.

**Jak zrobimy odwrotnie: najpierw prywatnym szyfrujemy, a potem publicznym odszyfrujemy – nazywamy to podpisaniem, np. Mamy ze do podpisywania sa wykorzystane funkcje skrotu, salt itd.**

Na słownie zrobimy z poprzedniego zadania:

#### **Podpiswanie:**

```
openssl dgst -sha256 -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:32 -sign keys.pem -out slowo.sig slowo.txt
```

Udalo sie podpisac slowo, w rezultacie mamy cat slowo.sig

### Weryfikacja takiego podpisu:

```
openssl dgst -sha256 -verify pub.pem -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:32 -signature slowo.sig slowo.txt
```

### Zadanie 3.10

**3.10** Pod adresem <http://127.0.0.1:3010> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:3010/verify> oraz <http://127.0.0.1:3010/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 3010:3010 --name ex10 docker.io/mazurkatarzyna/asymmetric-enc-ex10:latest  
podman run -p 3010:3010 --name ex10 docker.io/mazurkatarzyna/asymmetric-enc-ex10:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 3010:3010 --name ex10 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex10:latest  
podman run -p 3010:3010 --name ex10 ghcr.io/mazurkatarzynaumcs/asymmetric-enc-ex10:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:3010/verify> używając metody HTTP GET. Otrzymasz w odpowiedzi klucz publiczny RSA (public\_key\_pem), podpis (signature\_b64) oraz słowo (word).
- (c) Zweryfikuj podpis przy użyciu algorytmu RSA-2048 oraz trybu paddingu PSS. Przy weryfikacji należy zwrócić uwagę na parametry paddingu PSS:
- Używana funkcja skrótu: SHA-256
  - Salt length: długość soli powinna odpowiadać długości skrótu (32 bajty dla SHA-256), aby weryfikacja lokalna była zgodna z serwerem.
- (d) Używając metody HTTP POST, wyślij do serwera poprzez endpoint <http://127.0.0.1:3009/submit> słowo (jako word), klucz publiczny (jako public\_key\_pem), podpis w formacie base64 (jako signature\_b64) oraz informację, czy weryfikacja lokalna zakończyła się sukcesem (true/false) (jako user\_verified).
- (e) Serwer sprawdzi, czy Twoja lokalna weryfikacja jest zgodna z faktycznym podpisem i zwróci odpowiednią informację.

## ZAJĘCIA 4 - łamanie haseł

**PRZYDATNE NA KOŁOKWIUM: Szukanie ciągów znaków (funkcji hashujących)**  
*Hashcat -help | grep "bcrypt"*

Slowo

hash slowa: hash-md5

Lista hasel:

password

```
pass  
admin  
hasla  
1234567
```

Bierzemy 1 słowo ze słownika, porównujemy z

### Zadanie 4.1

Odpalamy serwer,

<http://127.0.0.1:4001/hash> - generujemy swój hash

```
(.python_env) [admin@parrot] [~/Pobrane/słowniki/ex41]  
└─ $echo -n "13b9797da8926a1462f9317057af122b" > hash.txt  
  
(.python_env) [admin@parrot] [~/Pobrane/słowniki/ex41]  
└─ $hashcat --help
```

### Złamanie hasła:

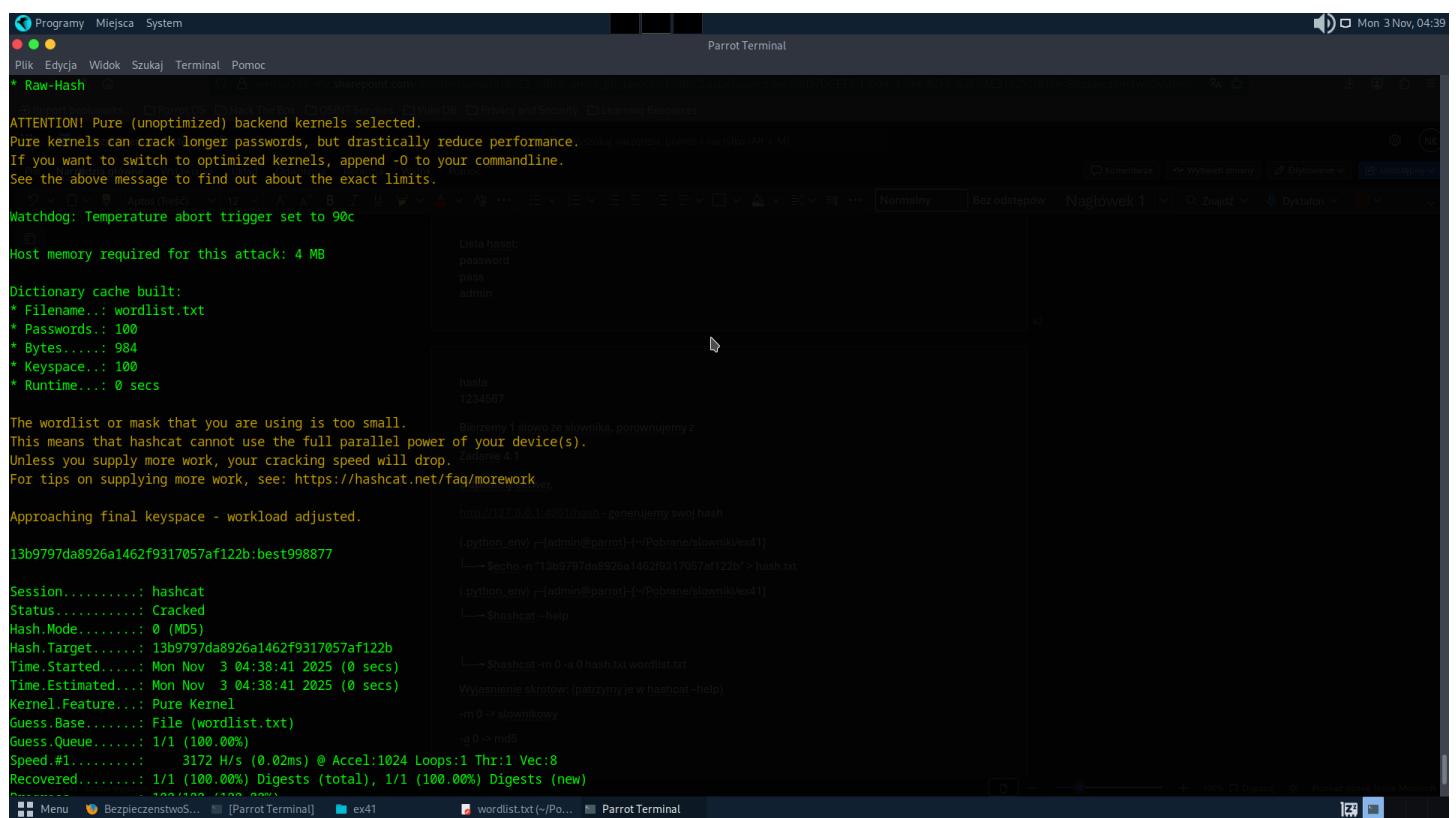
└─ \$**hashcat -m 0 -a 0 hash.txt wordlist.txt**

Wyjaśnienie skrótu: (patrzmy je w hashcat –help)

-m 0 -> md5 (typ hasha)

-a 0 -> słownikowy (typ łamania hasła)

Tu nas interesuje głównie status (że jest cracked)



The screenshot shows a terminal window titled 'Parrot Terminal' with the date 'Mon 3 Nov, 04:39'. The terminal output is as follows:

```
Programy Miejsca System  
Plik Edycja Widok Szukaj Terminal Pomoc  
* Raw-Hash  
ATTENTION! Pure (unoptimized) backend kernels selected.  
Pure kernels can crack longer passwords, but drastically reduce performance.  
If you want to switch to optimized kernels, append -O to your commandline.  
See the above message to find out about the exact limits.  
Watchdog: Temperature abort trigger set to 90c  
Host memory required for this attack: 4 MB  
Dictionary cache built:  
* Filename..: wordlist.txt  
* Passwords.: 100  
* Bytes.....: 984  
* Keystream..: 100  
* Runtime....: 0 secs  
The wordlist or mask that you are using is too small.  
This means that hashcat cannot use the full parallel power of your device(s).  
Unless you supply more work, your cracking speed will drop.  
For tips on supplying more work, see: https://hashcat.net/faq/morework  
Approaching final keystream - workload adjusted.  
13b9797da8926a1462f9317057af122b:best998877  
Session.....: hashcat  
Status.....: Cracked  
Hash.Mode.....: 0 (MD5)  
Hash.Target....: 13b9797da8926a1462f9317057af122b  
Time.Started....: Mon Nov 3 04:38:41 2025 (0 secs)  
Time.Estimated...: Mon Nov 3 04:38:41 2025 (0 secs)  
Kernel.Feature....: Pure Kernel  
Guess.Base.....: File (wordlist.txt)  
Guess.Queue.....: 1/1 (100.00%)  
Speed.#.....: 3172 H/s (0.02ms) @ Accel:1024 Loops:1 Thr:1 Vec:8  
Recovered.....: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
```

Teraz mozemy go sobie wyswietlic:

```
(.python_env) └─[admin@parrot]─[~/Pobrane/slowniki/ex41]
```

```
└─ $hashcat -m 0 hash.txt --show
```

```
13b9797da8926a1462f9317057af122b:best998877
```

```
(.python_env) └─[admin@parrot]─[~/Pobrane/slowniki/ex41]
```

```
└─ $cat hash.txt
```

```
13b9797da8926a1462f9317057af122b(.python_env) └─[admin@parrot]─[~/Pobrane/slowniki/ex41]
```

Wysylanie do serwera odpowiedzi:

```
curl -X POST http://127.0.0.1:4001/submit -H "Content-Type:application/json" -d  
'{"word":"best998877"}'
```

## Zadanie 4.2

```
podman run -p 4002:4002 --name ex2 docker.io/mazurkatarzyna/pass-cracking-ex2:latest
```

<http://127.0.0.1:4002/hash>

Zapisujemy nasz hash do pliku tekstowego:

```
echo -n "a1e677d57872e73fd8a7789f0abfcbe64cf20187" > hash2.txt
```

I lamiemy:

```
hashcat -m 100 -a 0 hash2.txt wordlist.txt
```

```
└─ $hashcat -m 100 hash.txt --show
```

```
a1e677d57872e73fd8a7789f0abfcbe64cf20187:evania
```

```
(.python_env) └─[admin@parrot]─[~/Pobrane/slowniki/ex42]
```

```
└─ $cat hash.txt
```

```
a1e677d57872e73fd8a7789f0abfcbe64cf20187(.python_env)
```

```
└─[admin@parrot]─[~/Pobrane/slowniki/ex42]
```

Wysylamy do serwera:

```
curl -X POST http://127.0.0.1:4002/submit -H "Content-Type:application/json" -d '{"word":"evania"}'
```

## Zadanie 4.3

**4.3** Pod adresem `http://127.0.0.1:4003` działa prosty serwer HTTP udostępniający dwa endpointy: `http://127.0.0.1:4003/hash` oraz `http://127.0.0.1:4003/submit`.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4003:4003 --name ex3 docker.io/mazurkatarzyna/pass-cracking-ex3:latest  
podman run -p 4003:4003 --name ex3 docker.io/mazurkatarzyna/pass-cracking-ex3:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4003:4003 --name ex3 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex3:latest  
podman run -p 4003:4003 --name ex3 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex3:latest
```

- (b) Wyślij request do endpointa `http://127.0.0.1:4003/hash` używając metody HTTP GET. Otrzymasz w odpowiedzi hash **SHA-256** do złamania (jako hash).
- (c) Wykorzystując narzędzia [hashcat](#) oraz [John the Ripper](#) i dostępne [słowniki](#), spróbuj złamać hash.
- (d) Za pomocą metody HTTP POST, wyślij pod endpoint `http://127.0.0.1:4003/submit` wartość złamanej hasza (jako word).
- (e) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

## 1. Uruchomienie serwera

### 2. Zapisanie hasha do pliku

```
curl -X GET http://127.0.0.1:4003/hash | jq -r .hash > hash.txt
```

### 3. Złamanie hashu

```
hashcat -m 17400 -a 0 hash.txt wordlist.txt
```

### 4. Wyswietlenie hasla

```
hashcat -m 1400 hash.txt --show
```

### 5. Wysłanie hasła do serwera

```
curl -X POST http://127.0.0.1:4003/submit -H "Content-Type:application/json" -d '{"word":"vizxv"}'
```

## Zadanie 4.4

**4.4** Pod adresem <http://127.0.0.1:4004> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4004/hash> oraz <http://127.0.0.1:4004/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4004:4004 --name ex4 docker.io/mazurkatarzyna/pass-cracking-ex4:latest  
podman run -p 4004:4004 --name ex4 docker.io/mazurkatarzyna/pass-cracking-ex4:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4004:4004 --name ex4 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex4:latest  
podman run -p 4004:4004 --name ex4 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex4:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:4004/hash> używając metody HTTP GET. Otrzymasz w odpowiedzi **hash SHA-512** do złamania (jako hash).
- (c) Wykorzystując narzędzia [hashcat](#) oraz [John the Ripper](#) i dostępne [słowniki](#), spróbuj złamać hash.
- (d) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4004/submit> wartość złamanej hasza (jako word).
- (e) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

## 1.Uruchomienie serwera

## 2.Zapisanie hasha do pliku

```
curl -X GET http://127.0.0.1:4004/hash | jq -r .hash > hash.txt
```

## 3.Złamanie hashu

```
hashcat -m 17400 -a 0 hash.txt wordlist.txt
```

## 4.Wyswietlenie hasla

```
hashcat -m 1400 hash.txt --show
```

## 5. Wysłanie hasła do serwera

```
curl -X POST http://127.0.0.1:4004/submit -H "Content-Type:application/json" -d '{"word":"vizxv"}'
```

## Zadanie 4.5

### Uruchomienie serwera:

```
podman run -p 4005:4005 --name ex5 docker.io/mazurkatarzyna/pass-cracking-ex5:latest
```

3200   bcrypt \$2*\$, Blowfish (Unix)	Operating System
---------------------------------------	------------------

### Zapisanie hasza do pliku hash.txt (z samej konsoli):

```
curl -X GET http://127.0.0.1:4005/hash | jq -r .hash > hash.txt
```

### Złamanie hasła:

```
hashcat -m 3200 -a 0 hash.txt wordlist.txt
```

```
(.python_env) └─[admin@parrot]─[~/Pobrane/slowniki/ex45]
```

```
└─$ls
```

```
wordlist.txt
```

```
(.python_env) └─[admin@parrot]─[~/Pobrane/slowniki/ex45]
```

```
└─$curl -X GET http://127.0.0.1:4005/hash | jq -r .hash > hash.txt
```

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
---------	------------	---------	---------------	------	------	------	---------

Dload	Upload	Total	Spent	Left	Speed
-------	--------	-------	-------	------	-------

```
100 133 100 133 0 0 734 0 --:--:--:--:--:--:--:-- 738
```

```
(.python_env) └─[admin@parrot]─[~/Pobrane/slowniki/ex45]
```

```
└─$cat hash.txt
```

```
$2b$12$Z0lNaeHEj3gOu.HRG52EfeFJvxXarvSkeNd5P0t6HzuGVPXVPcHMW
```

```
(.python_env) └─[admin@parrot]─[~/Pobrane/slowniki/ex45]
```

```
└─$hashcat -m 3200 -a 0 hash.txt wordlist.txt
```

### **Wyswietlenie złamaneego hasła:**

```
hashcat -m 3200 hash.txt --show
```

### **Wysłanie do serwera:**

```
$curl -X POST http://127.0.0.1:4005/submit -H "Content-Type:application/json" -d  
'{"word":"lazarashvili"}
```

## Zadanie 4.6

**4.6** Pod adresem <http://127.0.0.1:4006> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4006/hash> oraz <http://127.0.0.1:4006/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4006:4006 --name ex6 docker.io/mazurkatarzyna/pass-cracking-ex6:latest  
podman run -p 4006:4006 --name ex6 docker.io/mazurkatarzyna/pass-cracking-ex6:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4006:4006 --name ex6 ghcr.io/mazurkatarzynamcs/pass-cracking-ex6:latest  
podman run -p 4006:4006 --name ex6 ghcr.io/mazurkatarzynamcs/pass-cracking-ex6:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:4006/hash> używając metody HTTP GET. Otrzymasz w odpowiedzi hash scrypt do złamania (jako hash).
- (c) Wykorzystując narzędzia [hashcat](#) oraz [John the Ripper](#) i dostępne [słowniki](#), spróbuj złamać hash.
- (d) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4006/submit> wartość złamanej hasza (jako word).
- (e) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

Wszystko tak samo, zapisalam sobie hasla do oddzialnego pliku, a komenda do zlamania hasla taka:

Hashcat -m 8900 -a 0 hash.txt slownik.txt

```
curl: (3) URL Rejected: Port number was not a decimal number between 0 and 65535
natalka@fedora:~$ curl -X POST http://127.0.0.1:4006/submit -H "Content-Type:application/json" -d '{"word":"123456789"}'
{"success": true, "message": "Gratulacje! Poprawnie złamano hash!", "word": "123456789", "hash": "SCRYPT:1024:1:1:8+CnezzCekxqrFtpTR2SKA==:XsWB5uCoG653FVKYbXkNyXoPD24gsrbsc3kCRGYJzrg="}natalka@fedora:~$ curl -X POST http://127.0.0.1:4006/submit -H "Con
```

Zadanie 4.7

#### **Generowanie slownika:**

crunch 3 3 0123456789 -o digits.txt

Dlugosc 3 znakow, z samych cyfr sie sklada

(.python\_env) └─[admin@parrot]─[~]

```
└─ $echo -n "9e3cfcc48eccf81a0d57663e129aef3cb" > hash.txt
```

#### **Zlamanie hasla:**

```
└─ $hashcat -m 0 -a 0 hash.txt digits.txt
```

#### **Wyswietlenie hasla:**

Hashcat -m 0 hash.txt --show

#### **Wyslanie do serwera:**

```
└─ $curl -X POST http://127.0.0.1:4007/submit -H "Content-Type:application/json" -d '{"word":"798"}'
```

Zadanie 4.8

Hashcat -m 0 -a

crunch 4 4 ABCDEFGHIJKLMNOPQRSTUVWXYZ -o uppercase4.txt

No i w sumie dzialamy tak samo z tym slownikiem, server nie dziala wiec nie moge ):

Zadanie 4.9

**4.9** Pod adresem <http://127.0.0.1:4009> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4009/hash> oraz <http://127.0.0.1:4009/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4009:4009 --name ex9 docker.io/mazurkatarzyna/pass-cracking-ex9:latest  
podman run -p 4009:4009 --name ex9 docker.io/mazurkatarzyna/pass-cracking-ex9:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4009:4009 --name ex9 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex9:latest  
podman run -p 4009:4009 --name ex9 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex9:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:4009/hash> używając metody HTTP GET. Otrzymasz w odpowiedzi hash MD5 do złamania (jako hash).
- (c) Wiedząc, że **hasło to słowo pass z sufiksem składającym się z 4 cyfr od 0 do 9**, czyli np. **pass0000**, **pass0001**, ... , za pomocą narzędzia [crunch](#) wygeneruj słownik, którego użyjesz do złamania hasza.
- (d) Do złamania hasza użyj narzędzia [hashcat](#) lub [John the Ripper](#).
- (e) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4009/submit> wartość złamanej hasza (jako word).
- (f) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

**crunch 8 8 -t pass%%%%% -o pass\_digits.txt**

## Zadanie 4.10

**4.10** Pod adresem <http://127.0.0.1:4010> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4010/hash> oraz <http://127.0.0.1:4010/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4010:4010 --name ex10 docker.io/mazurkatarzyna/pass-cracking-ex10:latest  
podman run -p 4010:4010 --name ex10 docker.io/mazurkatarzyna/pass-cracking-ex10:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4010:4010 --name ex10 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex10:latest  
podman run -p 4010:4010 --name ex10 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex10:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:4010/hash> używając metody HTTP GET. Otrzymasz w odpowiedzi hash MD5 do złamania (jako hash).
- (c) Wiedząc, że **hasło to permutacje słów admin, password oraz 123**, za pomocą narzędzia [crunch](#) wygeneruj słownik, którego użyjesz do złamania hasza.
- (d) Do złamania hasza użyj narzędzia [hashcat](#) lub [John the Ripper](#).
- (e) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4010/submit> wartość złamanej hasza (jako word).
- (f) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

**crunch 3 16 -p admin password 123 -o perms.txt**

## Zadanie 4.11

4.11 Pod adresem <http://127.0.0.1:4011> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4011/hash> oraz <http://127.0.0.1:4011/submit>.

(a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4011:4011 --name ex11 docker.io/mazurkatarzyna/pass-cracking-ex11:latest  
podman run -p 4011:4011 --name ex11 docker.io/mazurkatarzyna/pass-cracking-ex11:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4011:4011 --name ex11 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex11:latest  
podman run -p 4011:4011 --name ex11 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex11:latest
```

(b) Wyślij request do endpointa <http://127.0.0.1:4011/hash> używając metody HTTP GET. Otrzymasz w odpowiedzi hash MD5 do złamania (jako hash).

(c) Wiedząc, że hasło składa się z 5 znaków, gdzie:

- pierwszy znak to wielka litera,
- drugi znak to mała litera,
- trzeci znak to cyfra od 0 do 9,
- czwarty i piąty znak to znaki specjalne (czyli np. !, @, #, \$ itp.),

za pomocą narzędzia [crunch](#) wygeneruj słownik, którego użyjesz do złamania hasza.

(d) Do złamania hasza użyj narzędzia [hashcat](#) lub [John the Ripper](#).

(e) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4011/submit> wartość złamanej hasza (jako word).

(f) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

crunch 5 5 -t ,@%^^ -o test.txt

- , — wielka litera (A–Z)
- @ — mała litera (a–z)
- % — cyfra (0–9)
- ^ — znak specjalny (symbol)

## Zadanie 4.12

podman run -p 4012:4012 --name ex12 docker.io/mazurkatarzyna/pass-cracking-ex12:latest

<http://127.0.0.1:4012/hash>

(.python\_env) └─[admin@parrot]─[~/zaj4]

└─ \$hashcat -a 3 -m 100 hash.txt '?l?l?l?l'

?l - mała litera

-a 3 (za pomocą masek łamanie hasła)

-m 100 (sha1)

Tutaj możemy zobaczyć jakie znaczki do jakich znaków:

<https://github.com/noobosaurus-r3x/Cheat-sheets/blob/main/Hashcat%20Cheat%20Sheet.md#mask-attack>

Takie coś tam jest:

- **Common Masks:**
- ?l: Lowercase letters (a-z)

- ?u: Uppercase letters (A-Z)
- ?d: Digits (0-9)
- ?s: Special characters (!@#\$%^&\*)
- ?a: All characters (?l?u?d?s)
- 

```
(.python_env) └─[admin@parrot]─[~/zaj4]
└── $hashcat -m 100 hash.txt --show
8eab2fb3af872058efda1ca0fd969b7f86597d8d:jzyw
(.python_env) └─[admin@parrot]─[~/zaj4]
└── $curl -X POST http://127.0.0.1:4012/submit -H "Content-Type:application/json" -d '{"word":"jzyw"}'
curl: (6) Could not resolve host: -X
curl: (6) Could not resolve host: POST
{"success": true, "message": "Gratulacje! Poprawnie złamano hash!", "word": "jzyw", "hash":
"8eab2fb3af872058efda1ca0fd969b7f86597d8d"}.(python_env) └─[admin@parrot]─[~/zaj4]
```

## Zadanie 4.13

Pierwszy ze znakow jest wielka litera, a reszta to znaki od 0 do 9

### Wlaczenie serwera:

```
podman run -p 4013:4013 --name ex13 docker.io/mazurkatarzyna/pass-cracking-ex13:latest
```

Pobranie hashu, zapisanie go do pliku

```
curl -X GET http://127.0.0.1:4013/hash | jq -r .hash > hash2.txt
```

### Zlamanie hasla:

```
└── $hashcat -m 100 -a 3 hash2.txt '?u?d?d?d'
```

### Pokazanie hasla:

```
└── $hashcat -m 100 hash2.txt --show
```

5820d242dd775842a463ba8aea60e7660a277a22:P858

### Wyslanie do serwera hasla:

```
(.python_env) └─[admin@parrot]─[~/zaj4]
```

```
└── $curl -X POST http://127.0.0.1:4013/submit -H "Content-Type:application/json" -d '{"word":"P858"}'
```

## Zadanie 4.14

```
curl -X GET http://127.0.0.1:4014/hash | jq -r .hash > hash3.txt
```

```
hashcat -a 3 -m 100 hash3.txt -1 abc -2 468 -3 '*%:' '?1?2?3'
```

-1 – pierwszy zbior, -2 – drugi zbior, -3 – trzeci zbior

Wysłanie do serwera:

```
curl -X POST http://127.0.0.1:4014/submit -H "Content-Type:application/json" -d '{"word":"c8%"}'
```

## Zadanie 4.15 - sami przetestowac

4.15 Pod adresem <http://127.0.0.1:4015> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4015/hash> oraz <http://127.0.0.1:4015/submit>.

(a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4015:4015 --name ex15 docker.io/mazurkatarzyna/pass-cracking-ex15:latest  
podman run -p 4015:4015 --name ex15 docker.io/mazurkatarzyna/pass-cracking-ex15:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4015:4015 --name ex15 ghcr.io/mazurkatarzynamcs/pass-cracking-ex15:latest  
podman run -p 4015:4015 --name ex15 ghcr.io/mazurkatarzynamcs/pass-cracking-ex15:latest
```

(b) Wyślij request do endpointa <http://127.0.0.1:4015/hash> używając metody HTTP GET. Otrzymasz w odpowiedzi hash SHA-1 do złamania (jako hash).

(c) Wiedząc, że hasło jest adresem MAC, zaczynającym się od 00:14:22 (OUI Dell), gdzie kolejne dwa bajty również są stałe, i równe FF:FF, złam hash.  
(Przykładowe hasła to: 00:14:22:ff:ff:01, 00:14:22:ff:ff:02 ...)

(d) Nie generuj własnych, ani nie używaj gotowych słowników. Wykorzystaj metodę bruteforce, definiując wzorzec hasła używając odpowiedniej maski. (Maski to wzorce definiujące strukturę hasła. Zamiast sprawdzać wszystkie możliwe kombinacje, możesz określić dokładny format hasła.)

(e) Do złamania hasła użyj narzędzia [hashcat](#) lub [John the Ripper](#).

(f) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4015/submit> wartość złamanej hasza (jako word).

(g) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

```
hashcat -a 3 -m 100 hash.txt "00:14:22:ff:ff:?h?h"
```

?h to jeden znak z zestawu hex (0–9, a–f)

[Haveibeenpwned.com](http://Haveibeenpwned.com) - sprawdzenie czy twoje hasło wyciekło gdzieś

## Zadanie 4.16

```
podman run -p 4016:4016 --name ex16 docker.io/mazurkatarzyna/pass-cracking-ex16:latest
```

```
hashcat -a 6 -m 1400 hash.txt
```

-a 6 -> hybrydowy tryb: maska+hasło

Slowo maska/ maska slowo (to decyduje czy 6 czy 7)

Najpierw maska a potem ze słownika to 6, jak na odwrot to 7 by było (sprawdzić bo chyba XD)

Lamanie hasła:

```
hashcat -a 6 -m 1400 hash.txt input.txt '?d?d'
```

```
hashcat -m 1400 hash.txt --show
```

```
curl -X POST http://127.0.0.1:4016/submit -H "Content-Type:application/json" -d '{"word":"123410"}'
```

## Zadanie 4.17

4.17 Pod adresem <http://127.0.0.1:4017> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4017/hash> oraz <http://127.0.0.1:4017/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4017:4017 --name ex17 docker.io/mazurkatarzyna/pass-cracking-ex17:latest  
podman run -p 4017:4017 --name ex17 docker.io/mazurkatarzyna/pass-cracking-ex17:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4017:4017 --name ex17 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex17:latest  
podman run -p 4017:4017 --name ex17 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex17:latest
```

- (b) Wyslij request do endpointa <http://127.0.0.1:4017/hash> używając metody HTTP GET. Otrzymasz w odpowiedzi hash **SHA-256** do złamania (jako hash).
- (c) Wiedząc, że hasło jest słowem ze słownika zawierającego najpopularniejsze hasła, do którego dodano na początku małą literę (każda od a do z), cyfrę (od 0 do 9), oraz znak specjalny ze zbioru znaków !, @, #, %, &, \*, złam hash. (Hasło to jedno z 10 pierwszych hasłów [ze słownika zawierającego najpopularniejsze hasła](#) z dodanymi 3 znakami na początku słowa, czyli np. f4@shadow czy k9!sunshine.)
- (d) W celu złamania hasza, wykorzystaj [słownik popularnych hasł](#), oraz zastosuj odpowiednią maskę, dodającą do każdego słowa ze słownika 2 cyfry na końcu słowa.

(e) Do złamania hasza użyj narzędzia [hashcat](#) lub [John the Ripper](#).

(f) Za pomocą metody HTTP POST, wyslij pod endpoint <http://127.0.0.1:4017/submit> wartość złamanej hasza (jako word).

(g) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

```
hashcat --stdout -1 '!@#$%&*' -a 7 '?l?d?1' slownik.txt > tmp_prefix_full.txt hashcat --stdout -a 6  
tmp_prefix_full.txt '?d?d' > candidates_full.txt hashcat -m 1400 -a 0 hash.txt candidates_full.txt
```

```
?l?d?s slownik.txt ?d?d
```

Tu nie wychodzi to zadanie 😞

## Zadanie 4.18

Zamienienie liter:

1. Tworzenie słownika (skopiuj wszystkie hasła i wklej do pliku)
2. Zamień litery w tym słowniku:

```
sed -e 's/a/@/g' -e 's/e/3/g' -e 's/i/1/g' names.txt | head -n 20
```

3. Złam hash:

```
hashcat -m 1400 -a 0 hash.txt names_leet.txt
```

```
hashcat -m 1400 hash.txt --show (pokazanie hashu)
```

4. Wyslij na serwer

```
curl -X POST http://127.0.0.1:4018/submit -H "Content-Type:application/json" -d '{"word":"sz@rlotk@"}'
```

## Zadanie 4.19

4.19 Pod adresem <http://127.0.0.1:4019> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4019/hash> oraz <http://127.0.0.1:4019/submit>.

(a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4019:4019 --name ex19 docker.io/mazurkatarzyna/pass-cracking-ex19:latest  
podman run -p 4019:4019 --name ex19 docker.io/mazurkatarzyna/pass-cracking-ex19:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4019:4019 --name ex19 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex19:latest  
podman run -p 4019:4019 --name ex19 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex19:latest
```

(b) Wyślij request do endpointa <http://127.0.0.1:4019/hash> używając metody HTTP GET. Otrzymasz w odpowiedzi hash **SHA-256** do złamania (jako hash).

(c) Wiedząc, że hasło to jedno z męskich imion z języka polskiego, przekształconych w następujący sposób:

- pierwsza litera imienia jest wielką literą,
- litery są zamienione w następujący sposób: a na 4, e na 3, o na 0,
- na końcu słowa jest dodany znak \_ oraz bieżący rok i znak !

złam hash. Przykładowe hasła to: J4c3k\_2025!, Andrz3j\_2025!, ... (Sprawdź też czym jest [Leet speak.](#))

(d) W celu złamania hasza, wykorzystaj [słownik polskich imion męskich](#), oraz stwórz plik z powyższymi regułami.

(e) Do złamania hasza użyj narzędzia [hashcat](#) lub [John the Ripper](#).

(f) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4019/submit> wartość złamanej hasza (jako word).

(g) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

Wczesniej należy zapisac sobie słownik w pliku oraz pobrac hash.txt

curl -X GET http://127.0.0.1:4018/hash

Echo -n "naszhash" > hash.txt

### 1. Zamienienie pierwszej litery na dużą:

awk '{print toupper(substr(\$0,1,1)) substr(\$0,2)}' names.txt > step1.txt

**3** substr(\$0, 1, 1)

👉 oznacza: „weź z całej linii \$0 **pierwszy znak**”.

czyli z "piotr" → "P"

**4** toupper(...)

To funkcja, która zamienia litery na **wielkie**.

czyli toupper("p") → "P"

**5** substr(\$0, 2)

👉 oznacza: „weź z \$0 **od drugiego znaku do końca**”.

Ponieważ nie ma trzeciego argumentu (długości), awk przyjmuje „do końca”.

czyli z "piotr" → "otr"

### 2. Zamienienie literek na inne + dodanie koncowki z rokiem:

```
sed -e 's/a/4/g' -e 's/e/3/g' -e 's/o/0/g' -e 's/$/_2025!/' step1.txt > final.txt
```

### 3. Złamanie hashu:

```
hashcat -m 1400 -a 0 hash.txt final.txt
```

### 4. Wysłanie na serwer:

```
curl -X POST http://127.0.0.1:4019/submit -H "Content-Type:application/json" -d '{"word":"J4n_2025!"}'
```

## Zadanie 4.20

**4.20** Pod adresem <http://127.0.0.1:4020> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4020/hash> oraz <http://127.0.0.1:4020/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4020:4020 --name ex20 docker.io/mazurkatarzyna/pass-cracking-ex20:latest  
podman run -p 4020:4020 --name ex20 docker.io/mazurkatarzyna/pass-cracking-ex20:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4020:4020 --name ex20 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex20:latest  
podman run -p 4020:4020 --name ex20 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex20:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:4020/hash> używając metody HTTP GET. Otrzymasz w odpowiedzi hash **SHA-256** do złamania (jako hash).
- (c) Wiedząc, że hasło to słowo ze [słownika CERT Polska](#), z dodanym znakiem ! na początku i znakiem # na końcu słowa, złam hash.
- (d) W celu złamania hasza, wykorzystaj słownik popularnych polskich haseł, zebrany podczas [wycieków danych](#), oraz stwórz plik z powyższymi regułami.
- (e) Do złamania hasza użyj narzędzia [hashcat](#) lub [John the Ripper](#).
- (f) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4020/submit> wartość złamanej hasza (jako word).
- (g) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie - zweryfikuje poprawność złamanej hasza.

### 1. Uruchomic serwer

### 2. Stworzyc słownik z tymi slowami (przez vim np. I kopiuj wklej)

### 3. Wygenerowanie słownika z ! na poczatku (mozna tez po prostu ‘!’ bez tego –1

```
hashcat --stdout -1 '!' -a 7 '?1' wordlist_pl.txt > word20.txt
```

### 4. Wygenerowanie słownika z ‘#’ na koncu

```
hashcat --stdout -a 6 word20.txt '#' > words20.txt
```

### 5. Złamanie hashu:

```
hashcat -m 1400 -a 0 hash20.txt words20.txt
```

### 6. Wysłanie na serwer

```
curl -s -X POST http://127.0.0.1:4020/submit -H "Content-Type: application/json" -d  
'{"word":"!24703349#"}'
```

## Zadanie 4.21

4.21 Pod adresem <http://127.0.0.1:4021> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4021/zip> oraz <http://127.0.0.1:4021/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4021:4021 --name ex21 docker.io/mazurkatarzyna/pass-cracking-ex21:latest  
podman run -p 4021:4021 --name ex21 docker.io/mazurkatarzyna/pass-cracking-ex21:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4021:4021 --name ex21 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex21:latest  
podman run -p 4021:4021 --name ex21 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex21:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:4021/zip> używając metody HTTP GET. Otrzymasz w odpowiedzi plik \*.zip zaszyfrowany hasłem.

- (c) Wiedząc, że hasło jest popularnym hasłem znajdującym się w [słownikach popularnych haseł](#), używając:

- [fcrackzip](#)
- [hashcat](#) wraz z [zip2john](#)
- [John the Ripper](#) wraz z [zip2john](#)

złam hasło i rozpakuj plik \*.zip.

- (d) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4021/submit> wartość złamanej hasła (jako password).

- (e) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie.

```
curl -sS -o file.zip http://127.0.0.1:4021/zip
```

```
fcrackzip -v -u -D -p wordlist_pl.txt file.zip
```

Jeśli fcrackzip nic nie znajdzie

```
zip2john file.zip > file.hash
```

```
head -n 5 file.hash
```

```
hashcat -m 13600 -a 0 file.hash wordlist_pl.txt
```

```
hashcat --show -m 13600 file.hash
```

Rozpakowanie pliku po złamaniu hasła:

```
unzip -P 'password123' file.zip -d outdir
```

```
ls -l outdir
```

Wysłanie na serwer:

```
curl -sS -X POST http://127.0.0.1:4021/submit -d "password=password123"
```

## Zadanie 4.22

**4.22** Pod adresem <http://127.0.0.1:4022> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4022/zip> oraz <http://127.0.0.1:4022/submit>.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4022:4022 --name ex22 docker.io/mazurkatarzyna/pass-cracking-ex22:latest  
podman run -p 4022:4022 --name ex22 docker.io/mazurkatarzyna/pass-cracking-ex22:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4022:4022 --name ex22 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex22:latest  
podman run -p 4022:4022 --name ex22 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex22:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:4022/zip> używając metody HTTP GET. Otrzymasz w odpowiedzi plik \*.zip zaszyfrowany hasłem.
- (c) Wiedząc, że hasło jest hasłem o długości pomiędzy 5-6 znaków, i zawiera jedynie cyfry od 0 do 9, używając:
- [fcrackzip](#)
  - [hashcat](#) wraz z [zip2john](#)
  - [John the Ripper](#) wraz z [zip2john](#)
- złam hasło i rozpakuj plik \*.zip. Możesz wygenerować swój własny słownik, używając narzędzia crunch.
- (d) Za pomocą metody HTTP POST, wyślij pod endpoint <http://127.0.0.1:4022/submit> wartość złamaneego hasła (jako password).

---

#### Zadanie 4.23

4.23 Pod adresem <http://127.0.0.1:4023> działa prosty serwer HTTP udostępniający dwa endpointy: <http://127.0.0.1:4023/pdf> oraz <http://127.0.0.1:4023/submit>. Na endpointie <http://127.0.0.1:4023/pdf> serwer zwraca plik \*.pdf zaszyfrowany numerem PESEL.

- (a) Uruchom serwer za pomocą poniższego polecenia:

```
docker run -p 4023:4023 --name ex23 docker.io/mazurkatarzyna/pass-cracking-ex23:latest  
podman run -p 4023:4023 --name ex23 docker.io/mazurkatarzyna/pass-cracking-ex23:latest
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run -p 4023:4023 --name ex23 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex23:latest  
podman run -p 4023:4023 --name ex23 ghcr.io/mazurkatarzynaumcs/pass-cracking-ex23:latest
```

- (b) Wyślij request do endpointa <http://127.0.0.1:4023/pdf> używając metody HTTP GET. Otrzymasz w odpowiedzi plik \*.pdf zaszyfrowany hasłem. Hasło jest numerem pesel.

- (c) Wiedząc, że hasło jest numerem pesel osoby, która urodziła się pomiędzy 01.07.1992 a 31.12.1992 i jest kobietą, oraz używając:

- [pdfrack](#)
- [hashcat](#) wraz z [pdf2john](#)
- [John the Ripper](#) wraz z [pdf2john](#)

złam hasło i odczytaj zawartość pliku \*.pdf.

- (d) Za pomocą metody HTTP POST, wyslij pod endpoint <http://127.0.0.1:4023/submit> wartość złamanej hasła (jako password).

- (e) W odpowiedzi serwer zwróci odpowiednią informację o sukcesie lub błędzie.

- (f) Jeśli nie posiadasz zainstalowanego pdf2john w swoim systemie, możesz skorzystać z jego Dockerowej wersji, którą uruchomisz w następujący sposób:

```
docker run --rm -v $(pwd):/data docker.io/mazurkatarzyna/pdf2john:latest challenge.pdf > hash.txt  
podman run --rm -v $(pwd):/data docker.io/mazurkatarzyna/pdf2john:latest challenge.pdf > hash.txt
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run --rm -v $(pwd):/data ghcr.io/mazurkatarzynaumcs/pdf2john:latest challenge.pdf > hash.txt  
podman run --rm -v $(pwd):/data ghcr.io/mazurkatarzynaumcs/pdf2john:latest challenge.pdf > hash.txt
```

gdzie challenge.pdf to plik \*.pdf pobrany z serwera, a hash.txt to hash wyodrębniony z pliku \*.pdf.

- (g) Jeśli nie posiadasz zainstalowanego pdfrack w swoim systemie, możesz skorzystać z jego Dockerowej wersji, którą uruchomisz w następujący sposób:

```
docker run --rm -v $(pwd):/data docker.io/mazurkatarzyna/pdfrack:latest -f challenge.pdf -w wordlist.txt  
podman run --rm -v $(pwd):/data docker.io/mazurkatarzyna/pdfrack:latest -f challenge.pdf -w wordlist.txt
```

Jeśli Docker Hub nie odpowiada, użyj obrazu zapasowego:

```
docker run --rm -v $(pwd):/data ghcr.io/mazurkatarzynaumcs/pdfrack:latest -f challenge.pdf -w wordlist.txt  
podman run --rm -v $(pwd):/data ghcr.io/mazurkatarzynaumcs/pdfrack:latest -f challenge.pdf -w wordlist.txt
```

gdzie challenge.pdf to plik \*.pdf pobrany z serwera, wordlist.txt to słownik, a hash.txt to hash wyodrębniony z pliku \*.pdf.

- (h) Kilka ciekawych artykułów dotyczących plików zabezpieczonych peselem jako hasłem:

- <https://informatykzakladowy.pl/lamiemy-haslo>
- <https://informatykzakladowy.pl/szyfrowanie-dokumentow-numerem-pesel>
- <https://sekurak.pl/dostalem-pit-a-w-pdfie>
- <https://niebezpiecznik.pl/post/jak-zlamac-haslo-do-zipa-pesel/>