


Uwagi:

- Rozwiązywanie ćwiczeń oznaczonych symbolem flagi  prowadzi do znalezienia "flagi", czyli tajnego hasła w jednym z dwóch formatów: HexA{coś-tutaj} lub HexArcana{coś-tutaj}. Możesz zdobyć punkty za ich rozwiązanie, wpisując flagę w odpowiednim ćwiczeniu na: https://cwiczenia.hackarcana.pl/workshop-session/2025-05-05-pl-praktyczny-python#_tab_exercises
- Jeśli zostanie wspomniany numer slajdu, prosimy pamiętać, że może wystąpić niewielka rozbieżność między numerem na tej liście a numerem w rzeczywistej prezentacji (np. wspomniany slajd 10, podczas gdy w prezentacji był to slajd 12), ponieważ prezentacje często są zmieniane do ostatniej chwili. Przepraszamy za niedogodności.

Informacje różne

1. Dostęp do platformy [cwiczenia.hackArcana.pl](https://cwiczenia.hackarcana.pl) (dostęp jest opcjonalny):
 - a. Dostęp wymaga posiadania lub utworzenia konta na [cwiczenia.hackArcana.pl](https://cwiczenia.hackarcana.pl):
Rejestracja: <https://cwiczenia.hackarcana.pl/register>
Logowanie: <https://cwiczenia.hackarcana.pl/login>
 - b. Szkolenie "Praktyczny Python":
Menu → Realizacja kodu → **162e276989940185**
 - c. Archiwalne szkolenie "Wstęp do programowania i Pythona" (nagrania, itp.)
Menu → Realizacja kodu → **4e5350046a7fe3e6**
2. Niektóre ćwiczenia wymagają stworzenia wirtualnego środowiska Pythona (VENV) – więcej informacji o tym znajdziesz na nagraniu z drugiego dnia szkolenia oraz na pierwszej liście szkoleniowej, a także w oficjalnej dokumentacji (jak i internecie):
<https://docs.python.org/3/library/venv.html>

Ćwiczenia

Pliki do ćwiczeń:

<https://cdn-hackarcana.net/py10-1dc19979d3098465305bb2df6781b461/py10-day04-exercises.zip>

1. Hash in a hashstack

Plik binarny `hashstack.bin` zawiera blok z flagą. Plik niestety jest dość duży, więc trudno flagę od razu wypatrzeć. Na szczęście znamy hasz SHA256 tego bloku:

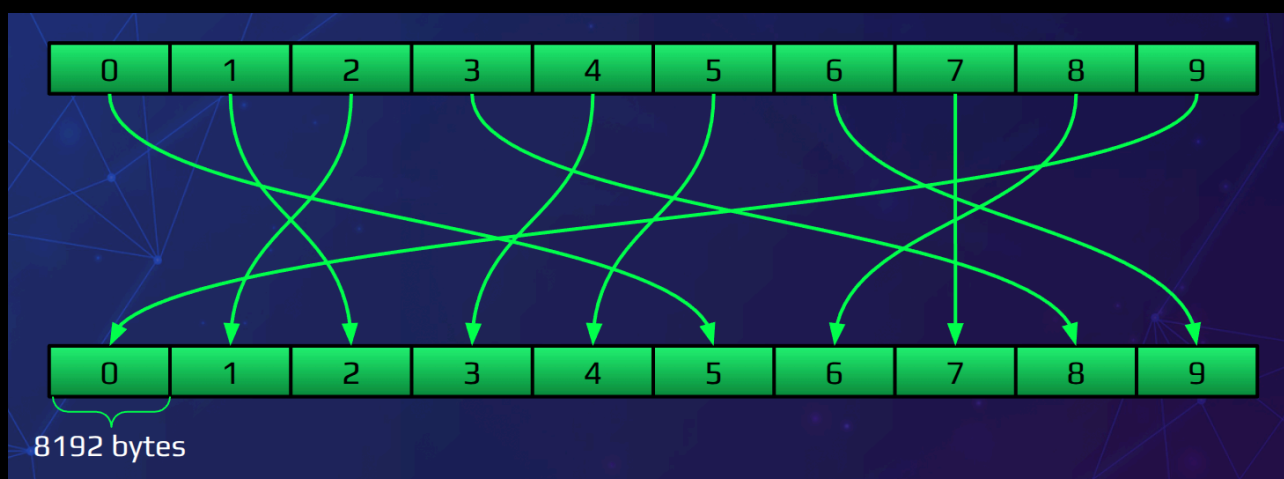
790d88483531ac32a12a57b233818ff698fb4ed7011f5c749f3b7493ba1ac5e1

Aby rozwiązać zadanie po kolei odczytuj bloki o wielkości **512 bajtów**, wyliczaj ich hasz i porównuj z haszem powyżej. Po znalezieniu odpowiedniego bloku zdekoduj go (`block.decode()`), wypisz za pomocą `print` i postępuj zgodnie z wyświetlonymi instrukcjami.

Przydatne funkcje:

- `hashlib.sha256(...).hexdigest()`
- `open(..., "rb")`

2. 🚩 Get shuffled!



Bloki w pliku **binarny shuffled.png.bin** zostały przetasowane!

Każdy blok ma dokładnie **8192 bajtów**, a kolejność jaka została użyta do stworzenia pliku jest widoczna na diagramie powyżej.

"Odtasuj" plik, tj. przywróć bloki do ich oryginalnej kolejności, a następnie odczytaj flagę z wynikowego pliku obrazkowego PNG.

Przydatne funkcje:

- `open(..., "rb")`

3. 🚩 Brute the patch

Niektóre bajty w pliku **binarnym broken.zip.bin** zostały popsute (tj. zmienione na inną wartość)!

Na szczęście mamy listę (`hashes.json`) poprawnych haszy SHA256 wszystkich 32-bajtowych bloków z tego pliku! Wiemy również, że w każdym 32-bajtowym bloku co najwyżej 1 bajt jest popsuty (ale nie wiemy który konkretnie).

Plik można więc naprawić testując wszystkie kombinacje, tj. dla każdego uszkodzonego bloku (którego hasz się nie zgadza) musimy na każdej pozycji przetestować każdą możliwą wartość bajtu (0 - 255), aż hasz całego bloku nie będzie się zgadzał z tym z listy (kombinacji jest tylko $32 * 256$ per blok, czyli niedużo).

Po naprawieniu pliku zapisz go jako ZIP, rozpakuj i odczytaj flagę.

UWAGA: Bardzo częsty błąd, który można zrobić w tym ćwiczeniu, to zmienianie więcej niż jednego bajtu w bloku (np. próba zmiany bajtu numer 8 po tym jak zmieniliśmy już bajt numer 7). W tym ćwiczeniu w danym bloku tylko jeden bajt naraz powinien być zmieniany (tj. trzeba przywrócić poprzedni bajt zanim zaczniemy zmieniać kolejny).

Przydatne funkcje:

- `open(..., "rb")`
- `hashlib.sha256(...).hexdigest()`
- `bytearray(...)` – typ `bytearray` to de facto lista wartości bajtów, w której każdy bajt indywidualnie jest reprezentowany jako liczba od 0 do 255; funkcje takie jak `writeln` na plikach binarnych czy `hashlib.sha256` działają z tym typem poprawnie, tj. nie trzeba konwertować z powrotem na typ `bytes`

4. The Annoying API

Zadanie dwuczęściowe! To zadanie posiada dwie flagi. Do wykonania drugiej części zadania potrzebna będzie implementacja Proof of Work z części pierwszej.

4.1. Proof of Work

Proof of Work (dalej PoW) jest bardzo ciekawym mechanizmem, w którym jedna strona komunikacji chce, żeby druga strona udowodniła, że wykonała pewną czasochłonną pracę (w rozumieniu czasu CPU/GPU/itp.). Jaka to praca jest w zasadzie nieistotne, byle by jej zlecenie i weryfikacja było szybkie i tanie, ale samo wykonanie wymagało sporej ilości obliczeń. Najczęściej "praca" ta ma charakter brutowania jakiegoś hasha, choć czasem jest bardziej sensowna (np. rendering fragmentu sceny 3D). PoW stosuje się np. w:

- ...zabezpieczeniach anti-DoS. W tym przypadku atakujący, który chce zDoSować dany serwis, przed każdym zapytaniem musi zużyć po swojej stronie więcej mocy CPU na wykonanie "pracy", niż serwer zużyje po swojej stronie na wykonanie finalnego zapytania. Przez to DoS staje się bardzo niekorzystny dla atakującego (bo ten zużywa więcej CPU niż serwer), chyba, że atakujący ma dostęp do darmowej mocy obliczeniowej (ale nawet wtedy siła ataku bardzo maleje).
- ..."wykopywaniu" niektórych kryptowalut, jak np. Bitcoin. Kopanie BTC to klasyczny PoW polegający na znalezieniu takich bajtów, które — po dołączeniu do zapisanych binarnie

oczekujących transakcji – dadzą hash, który będzie miał odpowiednią liczbę bitów zerowych z przodu.

- ...zadaniach CTFowych, które po stronie serwera wymagają więcej zasobów (np. każda interakcja z zadaniem wymaga uruchomienie osobnej VM z Windowsem po stronie serwera). W takim przypadku gracze muszą udowodnić wykonanie pewnej czasochłonnej pracy, żeby móc wejść w interakcje z danym zadaniem. Ma to na celu zniechęcenie graczy do prób brute-force'owania samego zadania, jak i ograniczenie liczby połączonych naraz graczy.

"Prace" w PoW wykonuje się oczywiście automatycznie, w sposób oskryptowany.

Niniejsze ćwiczenie, patrząc na nie całościowo, polega na naprawieniu pliku, który został uszkodzony podczas transportu. Klasycznie dla tego projektu, po drugiej stronie mamy serwer, który posiada poprawną wersję pliku, jak i może nam udzielić informacji o tym który fragment pliku ma jaki hash (SHA256).

Niestety, API udostępnione przez serwer wymaga właśnie *Proof of Work*, tj. dowodu na to, że wykonaliśmy pewną pracę.

Konkretniej, pod poniższym adresem znajduje się endpoint API, które podaje nam "wyzwanie", tj. serię zapisanych heksadecymalnie bajtów, oraz instrukcję PoW:

`https://py10-day4-577570284557.europe-west1.run.app/ex4/get-pow`

Przykładowy rezultat wywołania powyższego API będzie wyglądać następująco:

```
{
  "challenge": "807892c60b4822818171eec95815da41fa53448b2c1c4fdf",
  "comment": "To provide a valid Proof of Work token you need to generate
a stream of bytes that start with challenge bytes (you need to decode
hex) and has a SHA256 hash that starts with top 24 bits set to 1.
IMPORTANT: a PoW token is only valid for 120 seconds from when the
challenge was generated."
}
```

Zgodnie z otrzymaną instrukcją, aby wykonać pracę, musimy znaleźć taką sekwencję bajtów zaczynających się od podanego ciągu (80 78 92 ...), której hasz SHA256 będzie miał pierwsze 24 bity zapalone. Inaczej mówiąc, pierwsze 6 znaków hexdigest() hasza to mają być f f f f f f.

Ważne: W tym zadaniu istotne jest kodowanie i dekodowanie bajtów do/z postaci heksadecymalnej. W Pythonie robi się to w następujący sposób:

```
# Dekodowanie:
bytes.fromhex("41414141") # lub b''.fromhex("41414141")
```

Kodowanie:

`data.hex()` # np. `b'\x41\x41\x41\x41'.hex()` → `"41414141"`

Przykładowo, jeśli do bajtów z "wyzwania", tj. 80 78 92 c6 0b 48 22 81 81 71 ee c9 58 15 da 41 fa 53 44 8b 2c 1c 4f df, dołączymy bajty 31 23 36 00 00 00 00 00, to w wyniku dostaniemy zestaw bajtów, którego heksadecymalnie zapisany hasz SHA256 zaczyna się od `ffffff`:

```
>>> chal = b''.fromhex('807892c60b4822818171eec95815da41fa53448b2c1c4fdf')
>>> hashlib.sha256(chal + b'\x31\x23\x36\x00\x00\x00\x00\x00').hexdigest()
'ffffffeeae8f96e29ade369c5e51013f7b34abb12f8c5856ae13863b43538bd7'
```

Skąd wiadomo, że akurat bajty 31 23 36 00 00 00 00 00 dołączone do tego konkretnego wyzwania dadzą hasz o wymaganym formacie? Odpowiedź jest prosta: **nie wiadomo**.

Odpowiedzią jest bruteforce, tj. trzeba po prostu posprawdzać jak najwięcej różnych (losowych lub nie) dołączonych kombinacji bajtów, aż w końcu nie znajdziemy kombinację, która daje odpowiedni hash. Właśnie ten bruteforce jest pracą, którą wykonujemy, a znaleziony ciąg, który daje odpowiedni hasz, jest łatwo sprawdzalnym dowodem, na to, że pracę wykonaliśmy (tj. serwer musi tylko policzyć hash SHA256 z tego co wyślemy, i sprawdzić czy 6 hexcyfr `f` faktycznie jest z przodu).

W praktyce bruteforce tego typu w Pythonie zajmie cokolwiek pomiędzy kilkoma sekundami a minutą (czas trwania bruteforce'owania jest losowy – czasem się szybciej trafi odpowiednia kombinacja, czasem później).

Aby otrzymać pierwszą flagę należy:

1. Poprosić o nowe wyzwanie PoW (endpoint `/ex4/get-pow`).
2. Wykonać pracę, tj. znaleźć ciąg, który zaczyna się od bajtów spod klucza `challenge` i daje hash SHA256 zaczynający się (heksadecymalnie) od `ffffff`.
3. Wysłać zapytanie do endpointu `/ex4/get-flag` podając cały ciąg (`challenge` razem z dołączoną znalezioną sekwencją) w parametrze `pow` (patrz poniżej).

`https://py10-day4-577570284557.europe-west1.run.app/ex4/get-flag?pow=807892c60b4822818171eec95815da41fa53448b2c1c4fdf3123360000000000`

Uwaga 1: Dany *challenge* PoW jest ważny jedynie **120 sekund**. Co za tym idzie, po rozwiązaniu PoW mamy zazwyczaj około minuty na wywołanie API `/ex4/get-flag`

Uwaga 2: Pamiętaj, że o ile na czas transportu bajty są kodowane do heksadecymalnego stringu, to na czas brutowania, w tym liczenia hasza, trzeba je zdekodować z powrotem do postaci tablicy bajtów. Pomieszenie tego to zdecydowanie najczęstszy błąd robiony w tym zadaniu.

Przydatne funkcje:

- `hashlib.sha256(...).hexdigest().startswith('ffffff')`
- `requests.get(...)`
- `bytes.fromhex(...)`
- `some_bytes_object.hex()`

4.2. Popsuty plik z flagą

Plik `brokenflag.png` niestety jest popsuty w kilku miejscach. Na szczęście na serwerze jest jego poprawna kopia! Niestety, ten serwer jest strasznie wolny i do tego wymaga PoW.

Serwer oferuje następujące API:

***** Adres serwera:**

<https://py10-day4-577570284557.europe-west1.run.app>

***** Pobranie "wyzwania" PoW:**

[/ex4/get-pow](#)

UWAGA: wyzwanie jest ważne 120 sekund!

***** Wyliczenie hasza SHA256 z podanego fragmentu poprawnego pliku:**

[/ex4/get-hash?offset=OFFSET&size=SIZE&pow=POW_TOKEN](#)

Parametry:

offset - od którego bajtu w pliku liczyć hasz

size - ile bajtów od początkowego wziąć pod uwagę przy liczeniu hasza

pow - dowód na wykonanie PoW (patrz [/ex4/get-pow](#))

UWAGA: offset i size muszą być podzielne przez 32

***** Pobranie danych (32 bajty) z podanego offsetu:**

[/ex4/get-data?offset=OFFSET&pow=POW_TOKEN](#)

Parametry:

offset - od którego bajtu przesłać 32 bajty danych

pow - dowód na wykonanie PoW (patrz [/ex4/get-pow](#))

UWAGA: offset musi być podzielny przez 32

Ponieważ serwer jest bardzo wolny, to wyliczenie jednego hasza trwa 10 sekund, a pobranie 32 bajtów danych trwa 30 sekund.

Korzystając z powyższego API, które ma dostęp do poprawnego pliku, napraw plik `brokenflag.png` i odczytaj flagę.

Uwaga: To zadanie wymaga cierpliwości (stąd 150 punktów). Nieoptymalne rozwiązanie wymaga bardzo bardzo bardzo dużo cierpliwości, tj. może trwać kilka dni lub nawet tydzień! Zachęcamy do lektury dodatkowych materiałów o optymalizacji udostępnionych do projektu 4, ponieważ optymalne 1-wątkowe rozwiązanie trwa już tylko 30-40 minut (P.S. lepiej nie używać więcej niż 4 wątków – firewall może zablokować dostęp do serwera).

Przydatne funkcje:

- `hashlib.sha256(...).hexdigest().startswith('ffffff')`
- `requests.get(...)`
- `bytes.fromhex(...)`
- `some_bytes_object.hex()`

Powodzenia!