Algorytmy haszujące

Jakub Bukała

30 kwietnia 2024

1 Czym są algorytmy haszujące?

W dobie cyfrowego świata, gdzie dane osobiste i poufne informacje są przechowywane i przesyłane przez sieć, istnieje potrzeba zapewnienia ich bezpieczeństwa. Jednym z podstawowych narzędzi służących do tego celu są algorytmy haszujące. Ich głównym zadaniem jest przekształcenie danych wejściowych, takich jak hasła, w unikalne ciągi znaków o stałej długości. Powielanie tego procesu dla tych samych danych zawsze zwraca ten sam wynik.

Najważniejszą cechą algorytmów haszujących jest to, że nie da się odwrócić ich działania, czyli znaleźć danych wejściowych na podstawie wyniku działania algorytmu.

2 Gdzie są używane algorytmy haszujące?

- Autoryzacja użytkowników w systemach informatycznych.
- Przechowywanie wrażliwych danych, takich jak hasła, w bazach danych.
- Generowanie podpisów cyfrowych, potwierdzających autentyczność dokumentów.
- Zapewnienie integralności danych, na przykład weryfikacja autentyczności pobranego pliku.
- Śledzenie historii zmian w systemach kontroli wersji.
- Generowanie kluczy kryptograficznych.
- Eliminacja duplikatów w zbiorach danych.

3 Najpopularniejsze algorytmy haszujące

3.1 B-Crypt

B-Crypt został zaprojektowany z myślą o zastosowaniu w systemach autoryzacji użytkowników, jako bezpieczny sposób przechowywania haseł w bazach danych.

Działa w oparciu o szyfr blokowy *Blowfish* i wykorzystuje sól (losowy ciąg znaków dodawany do hasła), przez co jest odporny na ataki typu *rainbow table*. Jest bardzo elastyczny, co pozwala na dostosowanie jego parametrów do potrzeb aplikacji. Warto dodać, że w większości implementacji dane wejściowe są ograniczone do 72 bajtów.

Jego złożoność czasowa wynosi $O(2^{cost})$, gdzie *cost* to liczba iteracji algorytmu.

```
1 import bcrypt
2
3 hashed = bcrypt.hashpw(b'super_tajne',
      bcrypt.gensalt())
4 print(hashed)
```

Przykładowa implementacja w języku Python

3.2 SHA-256

SHA-256 jest jednym z algorytmów z rodziny *Secure Hash Algorithm*, zaprojektowanym przez *National Security Agency* (NSA). Jest szeroko stosowany w kryptografii, do generowania podpisów cyfrowych, weryfikacji integralności danych oraz w protokołach bezpiecznej komunikacji, takich jak *HTTPS*. Znajduje on również swoje zastosowanie na rynku kryptowalut.

Po użyciu algorytmu na danych wejściowych, otrzymujemy 256-bitowy skrót. Dzięki swojej długości, jest on odporny na ataki typu *brute force* oraz wystąpienie kolizji.

Jego złożoność czasowa wynosi $O(n)$, gdzie n to długość danych wejściowych.

```
1 import hashlib
2
3 hash_sha256 = hashlib.sha256()
4 hash_sha256.update(b'super_tajne')
5 print(hash_sha256.hexdigest())
```

Przykładowa implementacja w języku Python

3.3 MD5

MD5 jest jednym z najpopularniejszych algorytmów haszujących, zaprojektowanym przez *Ronalda Rivesta* w 1991 roku. Jego zastosowanie jest obecnie ograniczone ze względu na występowanie kolizji, czyli sytuacji, w której dwa różne zestawy danych wejściowych mają ten sam skrót. Mimo to, MD5 jest nadal używany w niektórych systemach, na przykład do weryfikacji integralności plików.

Po użyciu algorytmu na danych wejściowych, otrzymujemy 128-bitowy skrót.

Jego złożoność czasowa wynosi $O(n)$, gdzie n to długość danych wejściowych.

```
1 import hashlib
2
3 hash_md5 = hashlib.md5()
4 hash_md5.update(b"super_tajne")
5 print(hash_md5.hexdigest())
```

Przykładowa implementacja w języku Python

3.4 SHA-1

Kolejnym algorytmem z rodziny *Secure Hash Algorithm* jest SHA-1. Szeroko stosowany w przeszłości (protokoły SSL/TLS, sumy kontrolne), obecnie jest uważany za przestarzały ze względu na występowanie kolizji, podobnie jak MD5.

Po użyciu algorytmu na danych wejściowych, otrzymujemy 160-bitowy skrót.

Jego złożoność czasowa wynosi $O(n)$, gdzie n to długość danych wejściowych.

```
1 import hashlib
2
3 hash_sha1 = hashlib.sha1()
4 hash_sha1.update(b'super_tajne')
5 print(hash_sha1.hexdigest())
```

Przykładowa implementacja w języku Python

3.5 Argon2

Zwycięzca konkursu *Password Hashing Competition* - Argon2 - jest jednym z najbezpieczniejszych algorytmów haszujących. Podobnie jak B-Crypt, został zaprojektowany z myślą o zastosowaniu w systemach autoryzacji użytkowników.

Dzięki automatycznemu zastosowaniu soli, sporemu zużyciu pamięci i długiemu czasowi obliczeń jest niezwykle odporny na ataki typu *brute force*, *rainbow table* oraz *side-channel*.

Dostępne są trzy wersje algorytmu:

- Argon2d - zaprojektowany do zastosowań, gdzie ataki typu *side-channel* są mało prawdopodobne. Nie jest zalecany do przechowywania haseł ze względu na podatności na ataki czasowe.
- Argon2i - zaprojektowany głównie do przechowywania haseł.
- Argon2id - hybrydowe podejście, łączące zalety obu poprzednich wersji.

Jego złożoność czasowa wynosi $O(m \times t)$, gdzie m to zużycie pamięci, a t to liczba iteracji algorytmu.

```
1 import argon2
2
3 hashed = argon2.PasswordHasher().hash("super_tajne")
4 print(hashed)
```

Przykładowa implementacja w języku Python

4 Podsumowanie

Algorytmy haszujące są niezwykle ważnym narzędziem w dzisiejszym świecie cyfrowym. Zapewniają bezpieczeństwo danych, integralność informacji oraz autentyczność dokumentów. W zależności od zastosowania, warto wybrać odpowiedni algorytm, który spełni wymagania aplikacji pod względem bezpieczeństwa i wydajności.

Należy również pamiętać, aby stosować **aktualne** i sprawdzone algorytmy, które są odporne na współczesne ataki.

5 Referencje

1. `https://en.wikipedia.org/wiki/Hash_function`
2. `https://en.wikipedia.org/wiki/Bcrypt`
3. `https://en.wikipedia.org/wiki/Blowfish_(cipher)`
4. `https://en.wikipedia.org/wiki/SHA-2`
5. `https://en.wikipedia.org/wiki/MD5`
6. `https://en.wikipedia.org/wiki/Argon2`
7. `https://en.wikipedia.org/wiki/SHA-1`
8. `https://password-hashing.net`
9. `https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet`
10. `https://www.codecademy.com/resources/blog/what-is-hashing`