Algorytmy Haszowania

Autor: Przemysław Orzechowski i Szymon Kuś

27.04.2024

1 Dlaczego korzystamy z algorytmów haszowania?

Wraz z rozwojem technologii cyfrowej i coraz powszechniejszym przetwarzaniem duzej ilości informacji, istnieje większa potrzeba bezpiecznego zarządzania danymi. Jednym z kluczowych wyzwań jest zapewnienie integralności, poufności i autentyczności danych. Haszowanie, czyli przekształcanie danych o dowolnej długości w wartość o stałej długości za pomocą algorytmów haszujących, stanowi fundamentalne narzędzie w rozwiązywaniu tych problemów.

Integralność danych to w pewien sposób gwarancja, że dane nie zostały zmienione ani uszkodzone w trakcie ich przetwarzania. Algorytmy haszowania umożliwiają wygenerowanie unikalnego "odcisku palca"dla każdego zestawu danych, który można użyć do weryfikacji. Weryfikacja integralności danych jest kluczowym aspektem w dziedzinach takich jak bezpieczeństwo informacji, e-commerce czy cyfrowe podpisy.

Poufność danych to zapewnienie, że tylko uprawnione osoby mają dostęp do informacji w czytelnej formie. Choć algorytmy haszowania same w sobie nie zapewniają poufności danych, są one często wykorzystywane w procesie szyfrowania, gdzie hasze są używane jako klucze do szyfrowania i deszyfrowania danych.

Natomiast dzięki autentyczności danych wiemy, czy dane pochodzą od prawdziwego nadawcy i nie zostały podrobione lub zmienione przez nieuprawnionego użytkownika. Algorytmy haszowania odgrywają kluczową rolę w weryfikacji autentyczności danych poprzez generowanie unikalnych haszy, które mogą być używane do porównania z oryginalnymi danymi.

2 Algorytmy haszowania

W dziedzinie haszowania istnieje wiele różnych algorytmów, z których każdy ma swoje własne cechy, zalety i zastosowania. W niniejszym artykule skupimy się na omówieniu pięciu popularnych algorytmów haszowania, które są szeroko stosowane w praktyce:

SHA-1 (Secure Hash Algorithm 1)

SHA-1 jest algorytmem haszowania, który został opracowany przez National Security Agency (NSA). Generuje hasze o długości 160 bitów i był szeroko stosowany m.in. do weryfikacji integralności danych i generowania cyfrowych podpisów. Jednakże, wraz z postępem technologicznym, SHA-1 stał się podatny na ataki z wykorzystaniem kolizji, co spowodowało, że przestał być uznawany za bezpieczny w zastosowaniach kryptograficznych.

- Wynikiem jest 160-bitowy skrót.
- Działa na blokach danych o długości 512 bitów.
- Zastosowanie funkcji logicznych takich jak XOR, AND, OR.
- Wymaga 80 rund przetwarzania.
- Wykorzystuje 5 32-bitowych rejestrów stanu.
- Operacje na słowach 32-bitowych.

AES (Advanced Encryption Standard)

AES, czyli Advanced Encryption Standard, jest symetrycznym algorytmem szyfrowania blokowego. Charakteryzuje się tym, że ten sam klucz jest używany zarówno do szyfrowania, jak i deszyfrowania danych. AES działa na blokach danych o stałej długości, które mają 128 bitów. Symetryczność oznacza, że jeśli zaszyfrowane dane mają być odszyfrowane, ten sam klucz używany do szyfrowania musi być również używany do deszyfrowania.

Dzięki swojej symetrycznej naturze AES jest stosowany w wielu aplikacjach, gdzie istnieje potrzeba zachowania poufności danych, takich jak komunikacja internetowa, przechowywanie haseł, czy szyfrowanie plików.

- Składa się z trzech wariantów klucza: AES-128, AES-192 i AES-256.
- Działa na blokach danych o długości 128 bitów.
- Wykorzystuje operacje podstawienia (SubBytes), przesunięcia wierszy (ShiftRows), mieszania kolumn (MixColumns) i dodawania klucza rundy (AddRoundKey).
- Składa się z 10 rund dla AES-128, 12 rund dla AES-192 i 14 rund dla AES-256.
- Jest szeroko stosowany w celu zapewnienia poufności danych.

PBKDF2 (Password-Based Key Derivation Function 2)

PBKDF2, czyli Password-Based Key Derivation Function 2, jest algorytmem funkcji kryptograficznej używanym do generowania klucza kryptograficznego z hasła użytkownika. Działa poprzez wielokrotne iteracyjne przetwarzanie hasła przez funkcję haszującą, co sprawia, że uzyskanie klucza wymaga dużego nakładu obliczeniowego.

Jednym z kluczowych aspektów PBKDF2 jest możliwość określenia liczby iteracji, która determinuje stopień trudności przetwarzania hasła. Im większa liczba iteracji, tym dłużej trwa obliczenie klucza, co zwiększa odporność na ataki typu brute force.

- Iteracyjne hashowanie
- Sól (salt)
- Parametryzowalność
- Odporność na ataki brute-force

Argon2

Argon2 to jeden z najnowszych i najbardziej zaawansowanych algorytmów funkcji haszującej, który został specjalnie zaprojektowany do generowania bezpiecznych kluczy pochodzących z haseł. Jest to algorytm oparty na wątkach, co oznacza, że może wykorzystać wielowątkowość procesora do obliczeń, co zwiększa jego wydajność i odporność na ataki brute force.

Jego głównym celem jest zapewnienie bezpiecznego i skutecznego sposobu generowania haszy z haseł, które są trudne do złamania nawet przy użyciu potężnych mocy obliczeniowych.

Jest on szeroko stosowany w różnych aplikacjach, takich jak systemy uwierzytelniania, przechowywanie haseł, czy procesy uwierzytelniania wielopoziomowego. Jego elastyczność pozwala dostosować parametry, takie jak czas obliczeń czy pamięć, co umożliwia optymalne dostosowanie algorytmu do konkretnych wymagań bezpieczeństwa aplikacji.

- Adaptacyjność
- Odporność na ataki z użyciem GPU i ASIC
- Parametryzowalność
- Sól (salt)

B-Crypt

B-Crypt to zaawansowany algorytm haszowania, który jest powszechnie stosowany do bezpiecznego przechowywania haseł użytkowników. Jest oparty na funkcji kryptograficznej Blowfish, co zapewnia mu solidne podstawy bezpieczeństwa. W dziedzinie ochrony haseł jest uważany za jedno z najbezpieczniejszych rozwiązań.

Jedną z głównych zalet algorytmu B-Crypt jest jego zdolność do generowania haszy z opóźnieniami. Proces ten wymaga znacznie więcej zasobów obliczeniowych w porównaniu z prostymi funkcjami haszującymi, co skutecznie utrudnia ataki bruteforce oraz ataki z wykorzystaniem tabel tęczowych. Dzięki temu, nawet w przypadku kradzieży bazy danych, złamanie haseł generowanych przez B-Crypt jest znacznie trudniejsze i bardziej czasochłonne.

Ze względu na swoje zalety B-Crypt stał się popularnym wyborem w aplikacjach, które priorytetowo traktują bezpieczeństwo haseł użytkowników. Jest często stosowany w systemach uwierzytelniania, aplikacjach bankowych czy serwisach e-commerce.

- Używa adaptacyjnej funkcji skrótu.
- Wykorzystuje sól do zwiększenia bezpieczeństwa.
- Zoptymalizowany pod kątem bezpiecznego przechowywania haseł.
- Pozwala na kontrolę kosztu obliczeniowego.
- Jest powszechnie stosowany w aplikacjach do przechowywania haseł.

3 Zastosowanie

Algorytmy haszowania są szeroko wykorzystywane w różnych obszarach informatyki i cyberbezpieczeństwa ze względu na ich użyteczność i niezawodność. Oto kilka najczęstszych zastosowań algorytmów haszowania:

- **Bezpieczne przechowywanie haseł**: Jednym z głównych zastosowań algorytmów haszowania jest bezpieczne przechowywanie haseł użytkowników. Zamiast przechowywać same hasła, system przechowuje jedynie ich hasze, co znacząco zwiększa bezpieczeństwo informacji.
- **Integralność danych**: Algorytmy haszowania są również stosowane do weryfikacji integralności danych. Przykładowo, hasz pliku może być obliczany i porównywany z haszem oryginalnego pliku, aby sprawdzić, czy plik nie został zmieniony.

- **Weryfikacja autentyczności**: W protokołach uwierzytelniania, hasze są używane do weryfikacji autentyczności danych.
- **Podpisy cyfrowe**: Algorytmy haszowania są wykorzystywane do tworzenia podpisów cyfrowych, które potwierdzają autentyczność i integralność dokumentów elektronicznych. Poprzez haszowanie treści dokumentu i podpisywanie wynikowego haszu kluczem prywatnym, podpis cyfrowy może być zweryfikowany za pomocą odpowiadającego klucza publicznego.

4 Jak działają algorytmy?

Algorytmy haszowania przekształcają dane wejściowe na stałą długość bitową, nazywaną haszem. Proces ten jest deterministyczny, czyli dla tych samych danych wejściowych zawsze zostanie wygenerowany ten sam hasz. Jednak nawet mała zmiana danych wejściowych powinna spowodować zupełnie inną wartość hasza.

Algorytmy haszowania opierają się na złożonych operacjach matematycznych, które przekształcają dane wejściowe w sposób, który jest trudny do odwrócenia. W tym celu używane są różnorodne operacje, takie jak przesunięcia bitowe, dodawanie modulo i funkcje logiczne, które zapewniają losowe i nieliniowe przekształcenie danych.

W przypadku szyfrowania haseł, algorytmy haszowania są stosowane do przechowywania haszy użytkowników zamiast samych haseł. W momencie rejestracji lub logowania, system oblicza hasz wprowadzonego hasła i porównuje go z zapisanym haszem w bazie danych. W ten sposób, nawet jeśli baza danych zostanie naruszona, atakujący nie będzie w stanie odczytać oryginalnych haseł użytkowników, ponieważ hasze są trudne do odwrócenia.

5 Przykładowe implementacje

B-Crypt

```
1   import bcrypt
2
3   def register_user(username, password):
4       salt = bcrypt.gensalt()
5
6       hashed_password = bcrypt.hashpw(password.encode('utf-8'),
    salt)
7
8       store_in_database(username, hashed_password, salt)
9
10  def login_user(username, password):
11      stored_password, salt = retrieve_from_database(username)
12
13      hashed_password = bcrypt.hashpw(password.encode('utf-8'),
    salt)
14
```

```
        if hashed_password == stored_password:
            print("Login successful!")
        else:
            print("Incorrect username or password.")

    def store_in_database(username, hashed_password, salt):
        print("Storing user data in the database:")
        print("Username:", username)
        print("Hashed Password:", hashed_password.decode('utf-8'))
        print("Salt:", salt.decode('utf-8'))

    def retrieve_from_database(username):
        dummy_hashed_password = b'
    $2b$12$8nSGH95Jzv7qGALl9MQSGuZTJID.J3hASzR35AyHesvGR5lmF9kU2'
        dummy_salt = b'$2b$12$8nSGH95Jzv7qGALl9MQSGu'
        return dummy_hashed_password, dummy_salt

    if __name__ == "__main__":
        register_user("user123", "password123")
        login_user("user123", "password123")
        login_user("user123", "wrongpassword")

```

Argon2

```
    import argon2

    def hash_password(password):
        salt = b'some_salt'
        time_cost = 16
        memory_cost = 2**16
        parallelism = 1
        hash_length = 32
        hash_bytes = argon2.hash_password_raw(
            password=password.encode('utf-8'),
            salt=salt,
            time_cost=time_cost,
            memory_cost=memory_cost,
            parallelism=parallelism,
            hash_len=hash_length,
            type=argon2.Type.I
        )
        return hash_bytes

    def verify_password(password, hash_bytes):
        try:
            return argon2.verify_password(
                hash_password=hash_bytes,
                password=password.encode('utf-8'),
                type=argon2.Type.I
            )
        except argon2.exceptions.VerificationError:
            return False
        password = "secret_password"
    hashed_password = hash_password(password)
    print("Hashed password:", hashed_password)
```

```
32
33      is_valid = verify_password(password, hashed_password)
34      print("Is password valid:", is_valid)
```

6 Czas łamania poszczególnych algorytmów dla wyrażeń o różnej długości

Czas łamania haszy zależy od wielu czynników, w tym od długości i złożoności hasła, użytego algorytmu haszowania oraz mocy obliczeniowej atakującego. Można przyjąć ogólną zasadę - im dłuższe i bardziej skomplikowane hasło, tym trudniej je złamać.

Dla krótkich haseł czas łamania może być bardzo mały, szczególnie jeśli użyto słabego algorytmu haszowania, który jest podatny na ataki bruteforce lub słownikowe. Przykładowo, dla hasła składającego się z 6 znaków, atakujący korzystający z prostego algorytmu MD5 może go złamać w ciągu zaledwie kilku sekund.

Z drugiej strony, dla długich i silnych haseł, które są generowane losowo i wykorzystują różnorodne znaki, liczby i symbole, czas łamania może być znacznie dłuższy. Nawet najpotężniejsze ataki bruteforce mogą potrzebować godzin, dni, a nawet tygodni, aby przebrnąć przez wszystkie możliwe kombinacje znaków. Przykładowo, dla hasła składającego się z 12 losowo wybranych znaków z alfabetu łacińskiego (małych i dużych liter), liczby oraz znaków specjalnych, czas potrzebny do jego złamania może sięgać nawet kilku milionów lat. W tym przypadku liczba kombinacji do złamania wynosi około

$$146^{12} \approx 8.9906584 \times 10^{23}$$

Dodatkowo, silne algorytmy haszowania, takie jak B-Crypt, posiadają dodatkowe mechanizmy bezpieczeństwa, które dodatkowo zwiększają czas potrzebny do złamania hasła poprzez opóźnienia w obliczeniach.

Przykładowe ataki

- Atak na LinkedIn w 2012 roku: W 2012 roku nastąpił wielki atak na platformę społecznościową LinkedIn, w wyniku którego skradziono ponad 117 milionów haseł. Część haseł została zahaszowana za pomocą algorytmu SHA-1 bez soli, co znacznie ułatwiło ich złamanie. W efekcie wiele haseł zostało odszyfrowanych przez hakerów w stosunkowo krótkim czasie.

- Atak na Yahoo w 2013-2014 roku: W latach 2013-2014 miało miejsce jedno z największych w historii wycieków danych, w którym skradziono ponad 3 miliardy kont użytkowników Yahoo. Hakerzy wykorzystali słabe praktyki haszowania, co umożliwiło im złamanie wielu haseł w stosunkowo krótkim czasie.

- W 2014 roku miało miejsce głośne włamanie na platformę iCloud firmy Apple. Hakerzy wykorzystali atak bruteforce, aby przełamać hasła użytkowników i uzyskać dostęp do ich kont iCloud. W wyniku ataku doszło do kradzieży prywatnych zdjęć i danych użytkowników, które zostały później opublikowane online.

7 Kalkulacja złożoności obliczeniowej

7.1 SHA-1

Algorytm SHA-1 wykonuje 80 rund przetwarzania na blokach danych o długości 512 bitów. Czas potrzebny do wykonania pojedynczej rundy określony jest jako T_r. Wtedy ogólny czas wykonania algorytmu dla danego bloku danych wynosi $80 \times T_r$.

7.2 B-Crypt

Algorytm B-Crypt wykorzystuje funkcję odwrotnie trudną, co oznacza, że jego złożoność obliczeniowa rośnie wraz z wybranym parametrem kosztu. Dla danego kosztu n, możemy określić złożoność obliczeniową algorytmu jako $O(2^n)$, co oznacza, że czas potrzebny do obliczenia hasza rośnie wykładniczo wraz ze wzrostem n.

8 Wnioski

Algorytmy haszowania są niezbędnym narzędziem w dziedzinie informatyki i cyberbezpieczeństwa, zapewniającym integralność danych, bezpieczne przechowywanie haseł oraz weryfikację autentyczności danych. Wybór odpowiedniego algorytmu haszowania zależy od konkretnego zastosowania i poziomu bezpieczeństwa wymaganego przez daną aplikację.

Ważne jest również, aby regularnie aktualizować algorytmy haszowania i praktyki związane z bezpieczeństwem, aby zapewnić ochronę przed nowymi zagrożeniami. Ze względu na rozwój technologii oraz coraz bardziej zaawansowane metody ataków, konieczne jest monitorowanie i adaptacja środków bezpieczeństwa w celu zapewnienia skutecznej ochrony danych.

Referencje

1. https://en.wikipedia.org/wiki/Argon2
2. https://pl.wikipedia.org/wiki/Advanced$_E ncryption_S tandardhttps : //pl.wikipedia.org/wiki/SHA - 1$
3. https://en.wikipedia.org/wiki/PBKDF2
4. https://en.wikipedia.org/wiki/Bcrypt

5. https://www.czarnaowca.it/2020/09/haslo-w-aplikacji-jaki-algorytm-hash-wybrac/

6. https://rmonnetworks.com/what-you-need-to-know-about-the-2014-icloud-hack/

7. https://www.nytimes.com/2017/10/03/technology/yahoo-hack-3-billion-users.html