Algorytmy hashowania

Filip Smol, Dawid Okoń

April 21, 2024

Abstract

Ten dokument omawia pieć popularnych algorytmów hashowania, w tym B-Crypt, oraz ich zastosowania, działanie, przykładowe implementacje i czas łamania hasła.

1 Wstep

Hashowanie jest kluczowa technika stosowana w dziedzinach zabezpieczeń komputerowych, takich jak uwierzytelnianie, integralność danych i inne. Algorytmy hashowania przekształcaja wejściowy ciag danych w skrócona, zwykle wartość o stałej długości , znana jako hash.

2 Algorytmy hashowania

2.1 MD5

2.1.1 Zastosowanie

MD5, czyli Message Digest Algorithm 5, był szeroko stosowany do tworzenia cyfrowych skrótów wiadomości. Głównym zastosowaniem MD5 jest weryfikacja integralności danych, takich jak pliki do pobrania, oraz sprawdzanie, czy dane nie zostały zmienione. Jednak z powodu swojej podatności na ataki, jego zastosowanie w bezpiecznych systemach jest obecnie odradzane.

2.1.2 Jak działa

MD5 przetwarza wejściowy ciag bajtów o dowolnej długości i generuje z niego 128-bitowy (16-bajtowy) skrót. Algorytm dzieli dane wejściowe na bloki o długości 512 bitów, na które każdy jest przetwarzany oddzielnie w serii operacji bitowych, które obejmuja funkcje pomocnicze, takie jak permutacje,

podstawienia i dodawanie. Kluczowym aspektem jest to, że każdy krok jest zależny od poprzedniego, co sprawia, że odwrócenie procesu jest trudne.

2.1.3 Przykładowa implementacja

Poniżej znajduje sie przykładowa implementacja algorytmu MD5 w jezyku Python:

```
import hashlib

def md5_hash(string):
    hash_object = hashlib.md5(string.encode())
    return hash_object.hexdigest()

# Przyk␣lad u̲zycia
print(md5_hash("Hello, world!"))
```

2.1.4 Czas łamania

Czas łamania MD5 zależy od mocy obliczeniowej atakujacego oraz długości hasła. Dzieki postepowi w technologii obliczeń i dostepności mocy obliczeniowej, MD5 jest podatny na ataki brute-force i kolizje. Przykładowo, używajac nowoczesnych GPU, hasła MD5 można złamać w ciagu kilku minut lub nawet sekund, w zależności od ich złożoności i długości.

2.2 SHA-256

2.2.1 Zastosowanie

SHA-256 (Secure Hash Algorithm 256-bit) jest jednym z algorytmów rodziny SHA-2, używanym głównie do weryfikacji integralności danych oraz w systemach bezpieczeństwa, takich jak certyfikaty SSL/TLS. Jest również standardem w wielu rzadowych aplikacjach w USA i stosowany jako cześć protokołu Bitcoin do weryfikacji transakcji.

2.2.2 Jak działa

SHA-256 przekształca wejściowe dane (n-bitowe) w unikalny 256-bitowy (32-bajtowy) skrót. Proces hashowania dzieli sie na kilka etapów: inicjalizacja buforów hash, przetwarzanie bloków danych (każdy o wielkości 512 bitów) w serii iteracji, gdzie każda iteracja zawiera operacje mieszajace dane wejściowe w skomplikowany i nieodwracalny sposób, wykorzystujac funkcje logiczne i operacje na bitach.

2.2.3 Przykładowa implementacja

Przykład implementacji SHA-256 w jezyku Python z wykorzystaniem wbudowanego modułu 'hashlib':

```
import hashlib

def sha256_hash(string):
    hash_object = hashlib.sha256(string.encode())
    return hash_object.hexdigest()

# Przyk␣lad u␣zycia
print(sha256_hash("Hello, world!"))
```

2.2.4 Czas łamania

SHA-256 jest uznawany za bezpieczny przed atakami brute-force ze wzgledu na swoja długość skrótu i konstrukcje. Teoretycznie, liczba możliwych kombinacji w SHA-256 wynosi $2^256, coczyniatakibrute-forceniepraktycznymizobecnymstanemtechno$ $256zależyododpowiedniegoużytkowania(np.stosowaniesoliwhashowaniuhasel), alesamalgorytn$

2.3 SHA-3

2.3.1 Zastosowanie

SHA-3 (Secure Hash Algorithm 3), znany również jako Keccak, stanowi najnowsze rozszerzenie rodziny algorytmów SHA, zatwierdzone przez NIST w 2015 roku. Jest używany w aplikacjach wymagajacych wysokiej odporności na ataki kryptograficzne, takich jak systemy blockchain i kryptowaluty, oraz jako opcja bezpieczeństwa w rzadowych i wojskowych systemach zabezpieczeń.

2.3.2 Jak działa

SHA-3 różni sie od swoich poprzedników (SHA-1 i SHA-2) dzieki zastosowaniu konstrukcji matematycznej zwana sponge construction", która pozwala na absorpcje wejściowych danych o dowolnej długości i wypieranie danych o określonej długości. Algorytm ten może produkować skróty o różnej długości, w zależności od wymagań bezpieczeństwa, poprzez regulowanie ilości danych wypchnietych" z gabki.

2.3.3 Przykładowa implementacja

Poniżej znajduje sie przykładowa implementacja SHA-3 w jezyku Python, korzystajaca z modułu 'hashlib', który obsługuje SHA-3 od wersji Python 3.6:

```
import hashlib

def sha3_256_hash(string):
    hash_object = hashlib.sha3_256(string.encode())
    return hash_object.hexdigest()

# Przyk␣lad u␣zycia
print(sha3_256_hash("Hello, world!"))
```

2.3.4 Czas łamania

Podobnie jak SHA-256, SHA-3 oferuje wysoki poziom bezpieczeństwa przed atakami brute-force, głównie ze wzgledu na duża przestrzeń skrótów i skomplikowana konstrukcje wewnetrzna. SHA-3 jest zaprojektowany tak, aby był odporny na różnego rodzaju ataki kryptograficzne, w tym na ataki z użyciem technik znajdowania kolizji, które były skuteczne przeciwko starszym algorytmom SHA.

2.4 B-Crypt

2.4.1 Zastosowanie

B-Crypt to algorytm hashowania haseł zaprojektowany specjalnie do zabezpieczania danych uwierzytelniajacych. Jest szeroko używany w wielu aplikacjach i systemach do bezpiecznego przechowywania haseł, ponieważ umożliwia łatwe wdrożenie mechanizmów obronnych przeciwko atakom siłowym, takim jak solenie i key stretching.

2.4.2 Jak działa

B-Crypt używa algorytmu Blowfish, symetrycznego szyfru blokowego, do generowania skrótów haseł. Kluczowym elementem jest możliwość określenia czynnika pracy", który określa, jak intensywne i czasochłonne bedzie generowanie skrótu, przez co zwieksza sie odporność na ataki. Proces hashowania obejmuje solenie hasła (dodanie losowego ciagu znaków do oryginalnego hasła przed hashowaniem), co zapobiega atakom słownikowym i użyciu gotowych tablic tzw. rainbow tables.

2.4.3 Przykładowa implementacja

Implementacja B-Crypt w Pythonie za pomoca biblioteki ‘bcrypt‘ jest prosta i bezpośrednia. Poniżej przykład użycia tej biblioteki:

```
import bcrypt

def hash_password(password):
    # Generowanie soli
    salt = bcrypt.gensalt()
    # Hashowanie has␣la
    hashed_password = bcrypt.hashpw(password.encode(), salt)
    return hashed_password

def check_password(password, hashed_password):
    # Weryfikacja has␣la
    return bcrypt.checkpw(password.encode(), hashed_password)

# Przyk␣lad uzycia
password = "securePassword123"
hashed_password = hash_password(password)
print(check_password(password, hashed_password))  # Zwróci True
```

2.4.4 Czas łamania

Czas łamania haseł zabezpieczonych przez B-Crypt zależy głównie od czynnika pracy, który został użyty podczas hashowania. Im wyższy czynnik pracy, tym dłużej trwa przetwarzanie hasła i tym trudniej jest złamać hash metoda brute-force. Dzieki temu mechanizmowi, B-Crypt pozostaje skuteczny nawet w obliczu rosnacej mocy obliczeniowej dostepnej dla potencjalnych atakujacych.

2.5 Argon2

2.5.1 Zastosowanie

Argon2, zwyciezca konkursu Password Hashing Competition w 2015 roku, jest nowoczesnym algorytmem hashowania haseł zaprojektowanym z myśla o maksymalnej odporności na ataki za pomoca sprzetu specjalistycznego, takiego jak FPGA i ASIC. Jest zalecany do stosowania w systemach wymagajacych silnego zabezpieczenia haseł, w tym w aplikacjach internetowych, bazach danych oraz systemach do zarzadzania danymi użytkowników.

2.5.2 Jak działa

Argon2 wykorzystuje mechanizm zwiekszania odporności na ataki siłowe i ataki przy użyciu specjalistycznego sprzetu, poprzez zastosowanie trzech kluczowych elementów: solenia (dodawanie losowego ciagu do hasła przed hashowaniem), mieszania (wykorzystanie dużych ilości pamieci) oraz key stretching (powtórzenia operacji hashowania). Argon2 oferuje trzy warianty: Argon2d, który jest odporny na ataki przez timingowe kanały boczne, Argon2i, zoptymalizowany pod katem hashowania haseł i Argon2id, hybryda zapewniajaca oba rodzaje ochrony.

2.5.3 Przykładowa implementacja

Implementacja Argon2 w jezyku Python może być wykonana za pomoca biblioteki 'argon2-cffi'. Oto przykład jej użycia:

```
from argon2 import PasswordHasher

ph = PasswordHasher()
password = 'secretpassword'
hashed = ph.hash(password)

# Verifying the password
try:
    ph.verify(hashed, password)
    print("Password is correct!")
except:
    print("Password is incorrect!")

# Rehashing check to see if parameters need to be updated
if ph.check_needs_rehash(hashed):
    new_hash = ph.hash(password)
```

2.5.4 Czas łamania

Czas łamania Argon2 jest zależny od parametrów konfiguracyjnych, takich jak użyta ilość pamieci, liczba iteracji i równoległość. Te parametry pozwalaja na dostosowanie Argon2 do oczekiwanego poziomu bezpieczeństwa i dostepnych zasobów sprzetowych, co sprawia, że ataki metoda brute-force staja sie znacznie mniej wykonalne. Argon2 pozwala na elastyczne dostosowanie tych ustawień, aby zabezpieczenia mogły rosnać wraz z postepem technologicznym.

3 Podsumowanie

W tym dokumencie przedstawiono pieć algorytmów hashowania, które sa niezbedne dla zabezpieczania danych. Każdy algorytm ma swoje specyficzne zastosowania i różni sie wydajnościa oraz odpornościa na ataki.