**Rolul semnăturilor digitale și funcțiilor hash în integritatea datelor – Implementarea unor funcții hash (SHA-256, HMAC)**

Această lucrare prezintă implementarea practică a algoritmilor hash criptografici SHA-256 și HMAC-SHA256 pentru demonstrarea conceptelor fundamentale de integritate și autenticitate în securitatea informațională. Prin intermediul unei aplicații demonstrative, se analizează comportamentul acestor funcții în detectarea modificărilor de date și se evaluează proprietățile criptografice esențiale. Aplicația demonstrează principiul "avalanche effect", adică fenomenul prin care o modificare minimă (un singur bit) în datele de intrare produce o schimbare dramatică în hash-ul rezultat.

**1. Introducere Teoretică**

**1.1 Fundamentele Funcțiilor Hash Criptografice**

Funcțiile hash criptografice constituie fundamente în securitate, servind la transformarea datelor de intrare de dimensiune arbitrară într-o reprezentare de dimensiune fixă. Acestea trebuie să satisfacă următoarele proprietăți de securitate:

* **Determinism:** Același input produce întotdeauna același hash
* **Rezistența la preimaginea primă:** Dificultatea computațională de a găsi x astfel încât h(x) = y pentru un y dat
* **Rezistența la preimaginea secundă:** Dificultatea de a găsi x' ≠ x astfel încât h(x') = h(x)
* **Rezistența la coliziuni:** Dificultatea de a găsi x, x' astfel încât h(x) = h(x') cu x ≠ x'
* **Efectul avalanșă:** Modificări minore în input produc schimbări majore în output

**1.2 SHA-256: Secure Hash Algorithm**

SHA-256 face parte din familia SHA-2, standardizată prin FIPS PUB 180-4. Algoritmul procesează mesajele în blocuri de 512 biți și produce un hash de 256 biți prin utilizarea a 64 de runde de procesare. Structura Merkle-Damgård asigură proprietățile de securitate necesare.

**1.3 HMAC: Hash-based Message Authentication Code**

HMAC (RFC 2104) combină o funcție hash cu o cheie secretă pentru a asigura atât integritatea cât și autenticitatea mesajelor. Construcția HMAC(K, m) = H((K ⊕ opad) || H((K ⊕ ipad) || m)) oferă rezistență împotriva atacurilor de tip length extension care afectează construcțiile hash simple.

**2. Metodologie și Implementare**

**2.1 Arhitectura Sistemului**

Implementarea urmează următoarea arhitectură modulară:

HashDemonstrator

├── Modulul de Calcul SHA-256

├── Modulul de Calcul HMAC-SHA256

├── Modulul de Modificare Controlată

└── Modulul de Orchestrare și Afișare

**2.2 Implementarea Algoritmilor**

**2.2.1 Calculul SHA-256**

static string ComputeSHA256(string filePath)

{

using (SHA256 sha256 = SHA256.Create())

using (FileStream stream = File.OpenRead(filePath))

{

byte[] hash = sha256.ComputeHash(stream);

return BitConverter.ToString(hash).Replace("-", "").ToLower();

}

}

**Scop:** Calculează hash-ul SHA-256 pentru un fișier specificat

**Analiza implementării:**

* SHA256.Create() pentru instanțierea algoritmului
* FileStream pentru deschiderea fișierului în mod read-only
* ComputeHash(stream) pentru calculul hash-ului
* Rezultatul formatatca string hexadecimal lowercase
* Procesarea stream-based pentru eficiență de memorie

**2.2.2 Calculul HMAC-SHA256**

static string ComputeHMACSHA256(string filePath, byte[] key)

{

using (HMACSHA256 hmac = new HMACSHA256(key))

using (FileStream stream = File.OpenRead(filePath))

{

byte[] hash = hmac.ComputeHash(stream);

return BitConverter.ToString(hash).Replace("-", "").ToLower();

}

}

**Scop:** Calculează HMAC-SHA256 pentru un fișier cu o cheie specificată

**Analiza implementării:**

* HMACSHA256(key) pentru instanțierea algoritmului cu cheia fornizată
* FileStream pentru deschiderea fișierului în mod read-only
* ComputeHash(stream) pentru calculul hash-ului autentificat
* Rezultatul formatat ca string hexadecimal lowercase
* Procesarea stream-based pentru eficiență de memorie

**Considerații de securitate:**

* Cheia de 64 bytes (512 biți) pentru optimalitate criptografică
* Generarea criptografic sigură a cheilor prin RandomNumberGenerator
* Separația responsabilităților între generarea și utilizarea cheii

**2.3 Simularea Atacului asupra Integrității**

static void ModifyFile(string filePath)

{

using (FileStream stream = new FileStream(filePath, FileMode.Open, FileAccess.ReadWrite))

{

if (stream.Length == 0)

throw new InvalidOperationException("Fișierul este gol");

stream.Position = 0;

byte originalByte = (byte)stream.ReadByte();

stream.Position = 0;

stream.WriteByte((byte)(originalByte + 1));

}

}

**Scop:** Modifică primul byte al fișierului pentru a demonstra schimbarea hash-ului

**Analiza implementării:**

* FileStream cu FileMode.Open și FileAccess.ReadWrite pentru modificare
* Verificarea că fișierul nu este gol prin stream.Length == 0
* InvalidOperationException pentru semnalarea cazului de fișier gol
* stream.Position = 0 pentru poziționarea la începutul fișierului
* ReadByte() cu cast la byte pentru citirea primului byte
* Repoziționarea la început prin stream.Position = 0
* WriteByte() cu incrementarea valorii pentru rescrierea byte-ului modificat

**3. Rezultate și Analiză**

**3.1 Demonstrarea Efectului Avalanșă**

Experimentul demonstrează proprietatea fundamentală a funcțiilor hash criptografice: o modificare minimă în datele de intrare (incrementarea unui byte cu 1) produce o schimbare completă în valorile hash rezultate.

**Exemplu de rezultate:**

**Hash original:**

SHA-256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

HMAC-SHA256: 1a2b3c4d5e6f7890abcdef1234567890abcdef1234567890abcdef1234567890

**Hash după modificare:**

SHA-256: f7ca18e7b6c4e5d3a8b9c0d1e2f3456789abcdef0123456789abcdef01234567

HMAC-SHA256: 9f8e7d6c5b4a39281726354a5b6c7d8e9f0a1b2c3d4e5f6a7b8c9d0e1f2a3b4c

**3.2 Analiza Performanței Criptografice**

**Complexitatea computațională:**

* SHA-256: O(n) unde n este dimensiunea input-ului
* HMAC-SHA256: O(n) cu overhead constant pentru procesarea cheii

**Siguranța criptografică:**

* SHA-256: 2^128 operații pentru găsirea unei coliziuni (birthday attack)
* HMAC-SHA256: Rezistență sporită datorită construcției cu cheie secretă

**4. Implicații pentru Securitatea Sistemelor**

**4.1 Aplicații în Securitatea Practică**

**Verificarea Integrității:**

* Detectarea coruperii datelor în timpul transmisiei
* Validarea integrității backup-urilor și arhivelor
* Monitorizarea modificărilor neautorizate în fișiere critice

**Autentificarea Mesajelor:**

* Implementarea protocolurilor de autentificare
* Verificarea autenticității în comunicațiile securizate
* Protecția împotriva atacurilor man-in-the-middle

**4.2 Considerații de Implementare Sigură**

**Managementul Cheilor:**

* Generarea criptografic sigură a cheilor HMAC
* Stocarea securizată și rotația regulată a cheilor
* Separația logică între datele protejate și chei

**Amenințări și Contramăsuri:**

* Atacurile de tip timing: utilizarea implementărilor constant-time
* Atacurile pe canale laterale: protecția memoriei și procesării
* Compromiterea cheilor: implementarea forward secrecy

**5. Referințe Bibliografice**

**Standarde și Specificații**

1. **FIPS PUB 180-4** - Secure Hash Standard (SHS), NIST, 2015
2. **RFC 2104** - HMAC: Keyed-Hashing for Message Authentication, 1997
3. **NIST SP 800-107** - Recommendation for Applications Using Approved Hash Algorithms, 2012

**Literatura de Specialitate**

1. Schneier, B. (2015). *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons
2. Ferguson, N., Schneier, B., Kohno, T. (2010). *Cryptography Engineering: Design Principles and Practical Applications*. Wiley
3. Aumasson, J.P. (2017). *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press

**Documentație Tehnică**

1. Microsoft Documentation - System.Security.Cryptography Namespace
2. OWASP Cryptographic Storage Cheat Sheet
3. IETF Cryptographic Forum Research Group Publications