Bcrypt

Ce librărie am folosit

Librăria pe care o voi prezenta în cadrul acestei teme este librăria *Bcrypt*. Aceasta se bazează pe cifrul *Blowfish*, folosind o variantă actualizată numită *Eksblowfish* (*Expensive Key Schedule Blowfish*). *Eksblowfish* realizează o serie de runde în care folosește algoritmul *Blowfish*, cu cheia alternând între salt și parola primite ca parametru. Acest algoritm nu este neapărat mai eficient decât *Blowfish,* decât prin faptul că este mai lent, ceea ce crește rezistența la atacuri de tip brute-force.

Cum funcționează:

1. Funcția *EksBlowfishSetup* primește costul, salt-ul și parola ca parametri. Cu aceste date va inițializa și rula funcția *eksblowfish*, prezentată anterior. În cazul în care utilizatorul a introdus o parolă prea scurtă, *bcrypt* o va extinde (*key stretching*).
2. Este rulat algoritmul *eksblowfish* cu cheile setate anterior și textul *OrpheanBeholderScryDoubt.* Acest pas se repetă de 64 de ori

La final hash-ul va avea forma:
$[algoritm]$[cost]$[salt][hash]
unde:

- algoritm: versiunea folosită (2, 2a, 2x, 2y, 2b)
- cost: numărul de runde. Vor fi executate 2^cost runde
- salt: salt-ul pe 16 octeți, 22 de caractere
- hash: hash-ul pe 28 de octeți, 31 de caractere

De ce am ales această abordare

Pentru ce este necesară:

Parolele sunt, în mod normal, păstrate în baze de date. În cazul în care o persoană cu intenții malițioase capătă acces la o astfel de bază de date, ideal ar fi ca aceasta să îi fie indescifrabilă. Mereu vor fi posibile atacuri de tip brute-force, dar am vrea ca acestea să fie prea lente pentru a fi fezabile. De aceea, algoritmii proiectați pentru *password hashing* sunt lenți. Un utilizator, cel mai probabil, nu va avea o problemă să aștepte câteva secunde pentru a se autentifica. În schimb, un atacator ar trebui să aștepte aceste câteva secunde înmulțite cu numărul de utilizatori din baza de date pentru a decripta datele de care are nevoie.

Ce avantaje aduce:

Spre deosebire de alți algoritmi de hashing, Bcrypt:

- Este *preimage resistant*
- Spațiul folosit de salt este suficient de mare pentru a îngreuna atacuri precomputaționale, ca *Rainbow tables*
- Are un cost adaptabil

În trecut era folosită librăria *crypt*. Conform USENIX, în 1976 *crypt* putea cripta mai puțin de 4 parole pe secundă. 20 de ani mai târziu, în urma evoluției tehnologiei, aceeași funcție putea hash-ui 200 000 de parole pe secundă.

În ce context este folosită

După cum am precizat anterior, această librărie este folosită pentru criptarea parolelor. În ziua de azi accesăm foarte multe aplicații sau site-uri de socializare, care necesită un cont. Este evident de ce. Furtul unui cont poate avea efecte cel puțin neplăcute, de aceea parolele trebuie protejate. Întrucât (momentan) este necesar ca acestea să fie păstrate într-o bază de date (care poate fi spartă, la rândul ei), au apărut tot felul de algoritmi care să le ascundă.

Indiferent dacă aplicația este codată în C/C++, Python, Java, JavaScript etc., Bcrypt oferă suportul necesar pentru hashing-ul acestor parole.

Secvența de cod

Eu am folosit această librărie într-un proiect de React, chiar dacă în mod tradițional hashing-ul se face în backend-ul unei aplicații:

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
bcrypt.hash(input.password, saltRounds, function(err: any, hash: string) {
  input.password = hash
});
```

Referințe

- https://www.npmjs.com/package/bcrypt
- https://en.wikipedia.org/wiki/Bcrypt
- https://auth0.com/blog/hashing-in-action-understanding-bcrypt/