Temă – module criptografice

BCrypt

Introducere

Autentificarea este unul din cele mai întâlnite procese în aplicațiile pe care le folosim zilnic. Astfel că dorim o securitate sporită a datelor pe care le stocăm în aceste conturi. Prin urmare, la proiectarea unei astfel de aplicații, decizia în ceea ce privește reținerea acestor date sensibile este foarte importantă.

Cea mai întâlnită modalitate de a reține aceste credențiale este salvarea lor într-un tabel: unui utilizator îi este asociată o parolă. Atunci când user-ul își introduce datele, se va face un request pentru server care va compara parola dată cu cele stocate în baza de date pentru a vedea dacă există vreo potrivire.

Ce este BCrypt?

BCrypt este un modul criptografic, o funcție cu ajutorul căreia se poate aplica un algoritm de hash pe o parolă pentru a o menține în siguranță. Acesta a fost proiectat de Niels Provos și David Mazieres în anul 1999. Funcția bcrypt stă la baza algoritmului de hash pentru parolele din OpenBSD, dar și pentru toate sistemele Linux. "B" vine de la Blowfish care este un cifru bloc cu cheie simetrică, proiectat de Bruce Schneier în 1993. Acest modul poate fi folosit pentru a stoca parolele în baza de date într-un mod foarte sigur, fiind implementat in diverse limbaje de programare (PHP, Python, Ruby, Node.js).

Cum funcționează BCrypt?

Funcție hash criptografica(CHF): este un algoritm care pentru date de dimensiune variabilă generează un output de lungime fixă. Mai exact, aceasta mapează mesajul la o matrice ce are întotdeauna aceeași dimensiune. Aceste funcții nu sunt inversabile. [6]

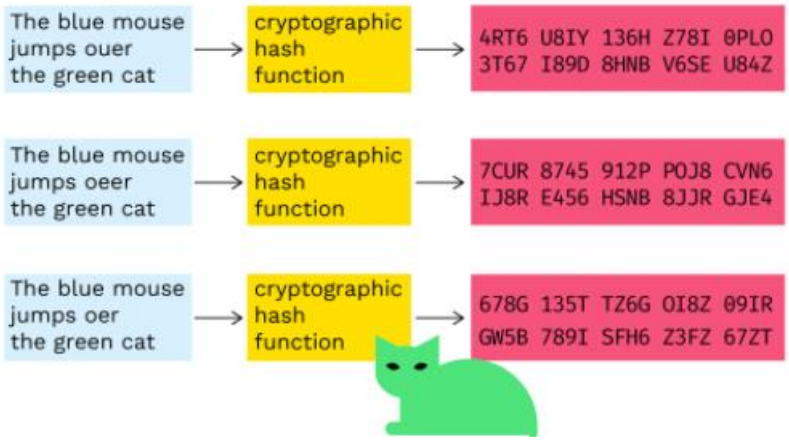

Figura 1

“Salt”: reprezintă un input ce este adăugat de fiecare dată când este stocată o nouă parolă. Este generat aleator și garantează un output unic după aplicarea funcției de hash chiar dacă 2 parole sunt identice. Parola este stocată în baza de date împreună cu acest input.[7]

BCrypt funcționează în 2 pași: 1. Se generează “salt”-ul; 2. Se va aplica funcția de hash pe parola cu salt-ul generat.

Deși BCrypt este foarte lent în comparație cu alte funcții destinate securizării parolelor, output-ul este unul sigur (***Thomas Ptacek*** scrie în articolul “***Enough with the Rainbow Tables”***: *“The better you can optimize your password hash function, the faster your password hash function gets, the weaker your scheme is”*). Rapiditatea calculelor în acest modul depinde de factorul de lucru (***work factor***).

```
andronic@andronic-VirtualBox:~/Desktop/tema to$ python3 gen_work_factor.py
Pentru rounds = 10 timpul de hash este 0.08370113372802734
Pentru rounds = 11 timpul de hash este 0.15593552589416504
Pentru rounds = 12 timpul de hash este 0.3133821487426758
Pentru rounds = 13 timpul de hash este 0.6040244102478027
Pentru rounds = 14 timpul de hash este 1.2008202075958252
Pentru rounds = 15 timpul de hash este 2.422775983810425
Pentru rounds = 16 timpul de hash este 4.859025955200195
Pentru rounds = 17 timpul de hash este 10.532303810119629
Pentru rounds = 18 timpul de hash este 20.18207311630249
```

Se poate observa cum timpul crește foarte repede pentru un cost mai mare.

Cum am folosit eu BCrypt?

Pentru a folosi această librarie criptografică în python trebuie instalat și importat modulul ***bcrypt***. Am utilizat-o într-un script care salvează parola împreună cu username-ul într-un fișier și întoarce parola asociată unui user (criptează și decriptează).

În acestă functie dau ca input username-ul și parola.

- Se generează un *salt*.
- Se aplică algoritmul de hash pe parolă împreuna cu acel ***salt*** care asigură unicitatea criptarii a două parole ce sunt la fel: acest proces se realizează prin funcția ***hashpw(text_clar, salt)***.

```
In [71]: def add_user_pass(user, password):
             password = str.encode(password)
             #     adaug user-ul si parola corespunzatoare(hash) intr-un fisier
             salt = bcrypt.gensalt()
             hashed_pass = bcrypt.hashpw(password, salt)
             hashed_pass = hashed_pass.decode("utf-8")

             user_pass = user + " " + hashed_pass
             f.write(user_pass)
             f.write("\n")
```

La decriptare:

- Primesc ca input username-ul și parola
- Verific daca parola pe care am primit-o ca input corespunde cu cea a username-ului dat ca parametru. Această verificare se face cu funcția ***checkpw(text_clar, parola pe care a fost aplicat algoritmul de hash anterior)***.

```
In [81]: def check_user_pass(user, raw_password):
             bytes_password = str.encode(raw_password)
             lines = read_from_file()
             print(lines)

             check = False
             for line in lines:
         #         print("DA")
                 strs = line.split()
                 file_user = strs[0]
                 hashed_password = strs[1]
                 hashed_password = hashed_password.replace('\n', '')
                 bytes_hashed_password = str.encode(hashed_password)
         #         print(file_user)
         #         print(hashed_password)
                 if file_user == user and bcrypt.checkpw(bytes_password, bytes_hashed_password):
                     print("Da, s-au gasit!")
                     check = True
                     break


             if check == False:
                 print("Nu s-a gasit nicio potrivire in fisier")
```

Avantajele:

1. Proprietatea de cascadă: o mica schimbare în input va genera o schimbare foarte mare în output (*Figura 1*). Acest lucru se datorează funcției hash criptografice.
2. Deoarece necesită și generarea unui *salt* înainte => hashing-ul combinat cu salt-ul protejează împotriva atacurilor de tip ***rainbow***. ***Atacul de tip rainbow*** este bazat pe tabela rainbow, care este o bază de date utilizată pentru atacurile în cazul autentificării. Este sub forma unui dicționar cu parole reprezentate în plaintext și valorile hash asociate acestora. [8]
3. Costul de calcul (***the computation cost***) este parametrizat, deci poate fi crescut odată cu puterea de calcul de care dispunem. Acest cost mai poartă numele de *factor de lucru* (*work factor*). Procesul de hashing este mai lent, acest lucru facând ca un atac prin forță brută să fie mai greu de realizat. Însă în acelasi timp, un factor de lucru ridicat, deși

aduce un plus de securitate, poate face ca experiența utilizatorului să nu fie printre cele mai placute. Folosirea costului ridicat implică și un consum mare de resurse (acest lucru reprezintă un dezavantaj).

Bibliografie

1. https://www.sitepoint.com/why-you-should-use-bcrypt-to-hash-stored-passwords/ (ultima accesare: 20 Aprilie 2021)
2. https://auth0.com/blog/hashing-in-action-understanding-bcrypt/ (ultima accesare: 20 Aprilie 2021)
3. https://zetcode.com/python/bcrypt/ (ultima accesare: 20 Aprilie 2021)
4. https://javascript.plainenglish.io/how-bcryptjs-works-90ef4cb85bf4 (ultima accesare: 20 Aprilie 2021)
5. *Documentație BCrypt:* https://pypi.org/project/bcrypt/ (ultima accesare: 18 Aprilie 2021)
6. https://en.wikipedia.org/wiki/Cryptographic_hash_function (ultima accesare: 20 Aprilie 2021)
7. https://en.wikipedia.org/wiki/Salt_(cryptography) (ultima accesare: 20 Aprilie 2021)
8. https://www.geeksforgeeks.org/understanding-rainbow-table-attack/ (ultima accesare: 20 Aprilie 2021)