Python

◆ Trygg lagring av passord

Skrevet av: Martin Strand

Kurs: Python
Tema: Tekstbasert, Kryptografi
Fag: Programmering
Klassetrinn: 8.-10. klasse, Videregående skole

Introduksjon

Nesten alle programmerere vil på et eller annet tidspunkt være nødt til å håndtere brukernes passord. Du har kanskje hørt mange god tips om hvordan du bør velge dine egne passord, men nå skal du lære hvordan man (ikke) skal behandle andre sine passord.

Tenk det følgende scenario. Du jobber med å lage en ny fantastisk nettside der brukerne må registrere seg. Da velger de også hvilket passord de vil bruke, og du må ta vare på passordet i en database for å sjekke det neste gang brukeren skal logge inn. Utfordringen oppstår idet databasen din kommer på villspor, og listen med brukernavn og passord blir spredt på internett (https://haveibeenpwned.com/PwnedWebsites). Vi skal jobbe oss gjennom noen måter å lagre passord på, og til slutt ende opp med noe som (dessverre) er bedre enn det som brukes på mange nettsider i dag.

Til slutt vil vi kanskje også forstå hvorfor passordtipsene er som de er.

Advarsel

Selv om dagens sluttresultat vil være ganske sikkert, bør du likevel ikke bruke denne koden i ekte systemer. Når det gjelder sikkerhet, er det alltid best å lene seg på noen som vet hva de gjør.

Steg 1: Sette opp systemet

Til denne leksjonen finnes det en påbegynt kodefil (./password_cracker.py). Last den ned og åpne den i IDLE (eller kopier innholdet inn i en ny fil i IDLE).

Vi bruker koden vi laget da vi så på hash-funksjoner, og laster i tillegg inn en ordliste med omtrent 500 000 ord. Vi skal bruke den til å lete etter mulige passord.

Sjekkliste

- [ ] Last ned kodefilen (./password_cracker.py) og ordboken (./ordliste_aspell.txt). Det kan hende du må høyreklikke og velge "Lagre linken som ..." eller lignende. Legg begge filene i samme mappe, og åpne `password_cracker.py` i IDLE. Prøv å forstå de punktene som følger.

- [ ] Linje 18: Her endrer vi litt på ordboka vår. Funksjonen `strip()` fjerner alle mellomrom og linjeskiftstegn, og `lower()` sørger for at alle ord bare består av små bokstaver.

- [ ] Linje 20: Her setter vi opp en *dictionary*, en datastruktur i Python som gjør at dersom vi vet hash-verdien av et ord, så kan vi slå opp det tilhørende ordet veldig lett.

- [ ] Linje 23: Dette er alle brukerne vi har satt opp, og med tilhørende passord. Brukeren "karin" har et veldig langt passord, mens "tjesi" sitt er ganske kort, men ikke akkurat slik ordet står i ordboka.

Steg 2: Enkel hashing

Dersom listen med brukernavn og passord hadde kommet på avveie, kan en skurk lese passordet rett ut. Nå skal vi prøve den første måten å beskytte oss på, nemlig ved å hashe hvert passord direkte. Deretter skal vi se om vi får til å knekke det.

Sjekkliste

- [ ] Legg til den følgende koden i kodefila vår.

```
# Steg 2: Hash-liste
hashed_list = {username: hash(password) for (username, password)
               in users.items()}
```

- [ ] Lagre filen og kjør den med F5. I konsollen kan du skrive `hashed_list` for å se hvordan den ser ut. Du skal da se noe som dette:

```
{'olano': 'ac07ad830f3c420165a140389d5c0c5388b15f15e16f099fcd8c4b818b056b6a',
'karin': '8bf0a05d19e7ee42b2ac61b4a6171d1e69f52dfc41aa3681817ea7f7cb381692',
'agoei': '23504ef356a5aa058d16ee1a05ccb9f2e670310e129ef67b6ad1555fee7cf230',
'tjesi': '2f25b7289a156d59e951eeea0be5ce39b5cc7133c24ac0124106a08d7481de12',
'marst': '23504ef356a5aa058d16ee1a05ccb9f2e670310e129ef67b6ad1555fee7cf230'}
```

Legg merke til at du kan se at to av brukerne har samme passord bare fordi de har samme hash.

- [ ] Legg til følgende kode i kodefila.

```
def crack_hashed_list(hashed_list, hashdict):
    # For tidtaking. Starttidspunktet.
    time_start = datetime.now()

    # Gå gjennom hver bruker/passord i tabellen vår
    for username, password in hashed_list.items():
        # Se om passordet finnes i tabellen vi laget over
        if password in hashdict.keys():
            # Hvis ja, skriv ut det tilhørende passordet
            print(username+": "+hashdict[password])

    # Slutt på tidtaking
    time_end = datetime.now()
    diff_time = time_end-time_start
    print(diff_time.total_seconds(),"sekunder")
```

Prøv å forstå hva koden gjør.

- [ ] Lagre og kjør fila. I konsollen, skriv inn `crack_hashed_list(hashed_list, hashdict)` og trykk Enter. Hvilke passord ble funnet, og hvor lang tid tok det?

Steg 3: En liten klype salt sammen med passordet

En av grunnene til at angrepet i steg 2 gikk så raskt, var at vi kunne lage en liste over hasher av alle ordene i ordboka på forhånd, og så bare slå raskt opp i den. Vår ordbok

hasher av alle ordene i ordboka på forhånd, og så bare slå raskt opp i den. Vår ordbok er "bare" på 500 000 ord, men om man skulle brukt den til et virkelig angrep, ville man kanskje brukt et par dager på forhånd for å lage en ordbok som også inneholdt alle mulige variasjoner av store og små bokstaver, og erstatninger av tall med bokstaver, som i passordet til "tjesi". Bare ekstremt gode passord ville motstått et slikt angrep, resten hadde blitt funnet på under et sekund.

Nå skal vi gjøre en forbedring som gjør at alt det forarbeidet blir til ingen nytte. Angrepet fungerte så godt fordi man alltid lagret samme hash når noen valgte samme passord, for eksempel "appelsin". Hvis vi kan få "appelsin" til å bli lagret som en ny tekst hver gang så har vi plutselig gjort det litt vanskeligere. Det løser vi ved å finne en tilfeldig verdi som vi hasher sammen med passordet. Den tilfeldige verdien kalles *salt* og lagres ved siden av det beskyttede passordet, slik at det blir enkelt å teste senere.

Sjekkliste

- [ ] Legg til følgede kode i fila di.

```
# Steg 3: Saltet hash-liste
salts = [token_hex(16) for i in range(len(users))]
salted_hashed_list = {username: (salt, hash(salt+password))
                      for (username, password, salt)
                      in zip(list(users.keys()), list(users.values
()), salts)}
```

  Denne koden gjør mye på lite plass. Den første linjen bruker `secrets`-biblioteket til å lage en tilfeldig verdi for hver bruker. Den andre linjen lagrer en *dictionary* der hvert brukernavn blir koblet til sitt salt og og en hash av saltet og passordet.

- [ ] Lagre og kjør koden. Skriv først `hashed_list` og deretter `salted_hashed_list`. Ser du at hashene blir forskjellige selv om passordene er de samme?

- [ ] Vi skal nå prøve å finne et angrep også mot denne teknikken. Legg til den følgende koden i Python-fila.

```
def crack_salted_list(salted_hashed_list, wordlist):
    time_start = datetime.now()

    # Gå gjennom alle brukernavn og salt/passordhash
    for username, (salt, password) in salted_hashed_list.items():
        # Gå gjennom alle ordene. Nå kan vi ikke lenger slå opp i
 en
        # preprosessert liste.
        for word in wordlist:
            # Gjør et forsøk: Hash med salt og ordet fra ordboka,
 sammenlign.
            if hash(salt+word) == password:
                print(username+": "+word)
                break

    time_end = datetime.now()
    diff_time = time_end-time_start
    print(diff_time.total_seconds(),"sekunder")
```

Lagre filen og kjør koden. Skriv `crack_salted_list(salted_hashed_list, wordlist)` . Hvilke passord ble funnet, og hvor lang tid tok det denne gangen?

Du har kanskje hørt om Bitcoin? Såkalt *mining* handler om å teste svært mange SHA-256-hasher i sekundet, akkurat som vi gjør her. Da bruker man faktisk skjermkortet til å beregne hashene i stedet, fordi det da kan gå enda mye raskere.

Steg 4: Når koden er ment å være treg

Salt alene var visst ikke nok. Vi kan ikke lenger bruke det samme forarbeidet som vi snakket om over, så vi må teste mot hele listen vår for hvert passord. Problemet er at det går så raskt å sjekke hele listen, så alle enkle passord blir avslørt med én gang likevel. Måten man da prøver å reparere dette på, er å sørge for at det ikke lenger er raskt å gå gjennom listen.

Vi skal bruke en funksjon som er tilgjengelig for alle Python-brukere med versjon 3.4 eller høyere, og funksjonen er laget for å bruke veldig mye prosessorkraft. Når man bare skal sjekke én gang, fordi brukeren vet sitt eget passord, er det ikke noe stort problem at det går litt tregt. "Tregt" i denne sammenhengen betyr kanskje et halvt sekund eller noe

det går litt tregt. "Tregt" i denne sammenhengen betyr kanskje et halvt sekund eller noe

sånt. Brukeren merker ikke et halvt sekunds forsinkelse, men dersom man skal gå gjennom en liste på 500 000 passord for å finne det riktige -- så vil det ta nesten 70 timer!

Det finnes også varianter av denne ideen som handler om å bruke så mye minne som mulig, men da må man ha ekstrabiblioteker installert for å bruke det i Python.

Sjekkliste

- [ ] Kopier den følgende koden til fila di.

```
# Steg 4: Prosessor-intensiv hashing
strong_hashed_list = {username: (salt, hexlify(pbkdf2_hmac('sha25
6',
                        password.encode(), salt.encode(), 100000)).d
ecode())
                        for (username, password, salt)
                        in zip(list(users.keys()), list(users.values
()), salts)}
```

  Legg merke til tallet 100000 midt inni koden. Det betyr at hver runde med denne funksjonen bruker SHA-256-hash-funksjonen 100 000 ganger.

- [ ] Nå skal vi forsøke å angripe den på samme måte som over. Kopier den følgende koden.

```
def crack_strong_list(strong_hashed_list, wordlist):
    # Denne funksjonen er svært lik den forrige, men vi har skifte
t ut
    # hash-algoritmen.

    time_start = datetime.now()

    for username, (salt, password) in strong_hashed_list.items():
        for word in wordlist:
            test = hexlify(pbkdf2_hmac('sha256', word.encode(), sa
lt.encode(), 100000)).decode()
            if (test == password):
                print(username+": "+word)
                break

    time_end = datetime.now()
    diff_time = time_end-time_start
    print(diff_time.total_seconds(),"sekunder")
```

- [ ] Lagre koden og kjør modulen ved å trykke F5. Test angrepet vårt ved å skrive `crack_strong_list(strong_hashed_list, wordlist)`. Hva skjer? (Når du blir lei av å vente kan du trykke `Ctrl+C`. Forfatteren av denne leksjonen testet tålmodigheten sin og lot koden kjøre til den var ferdig. Den fant de samme passordene som før, men denne gangen tok det 28 timer (!), og det var for bare fire-fem passord.

Bonus: Passordtips

Nå har vi jobbet med forskjellige måter å sikre passordene på. *Dersom* du vet at passordet blir lagret like bra eller bedre som det vi gjorde til slutt, så trenger man kanskje ikke et fantastisk passord, så lenge det ikke er blant de vanligste. Om nettsiden du bruker derimot ikke beskytter passordene sine godt (og det er dessverre det vi alltid må gå ut fra), så forstår du kanskje hvor viktig det er å ha et passord som ingen har tenkt på før. Derfor sier mange at man bør ha store og små bokstaver, tall og rare tegn i passordene. Det senker sannsynligheten for at noen har tenkt på passordet og lagt det i lista over kandidater de skal sjekke.

Et vanlig tips nå er å bruke flere vanlige ord som ikke har noe med hverandre å gjøre, og så sette dem sammen til ett passord, som for eksempel "korrekt hest batteri stift" (https://xkcd.com/936/).

Avansert: Hvorfor dette ikke er godt nok

Når det tok over et døgn å knekke disse få og ganske dårlige passordene, så skulle man tro at dette var godt nok. Grunnen til at dette likevel ikke er godt nok til løsninger som krever høy sikkerhet, er at skjermkort er veldig gode til å gjøre denne typen beregninger. Forfatteren av denne leksjonen brukte tilpasset programvare til å angripe de sikrede passordene ved hjelp av skjermkortet i stedet for den vanlige prosessoren. Da gikk kjøretida ned fra 28 timer, til 18 minutter og 13 sekunder.