IZVJEŠTAJ 5.LABORATORIJSKE VJEŽBE

Na 5. laboratorijskoj vježbi smo se dotakli teme *passworda* tj. iterativnog hashiranja istog, kreiranja soli te kako različite hash funkcije daju različita vremena izvođanja u ovisnosti o korištenju memorije. Cilj same vježbe je shvatiti zašto nekada koristimo 'spore' hash funkcije kojima treba više vremena za generiranje samog *hasha* te kako korištenjem tih funkicije demotiviramo napadača da koristi *dictionary* napad jer mu vrijeme samoga napada povećajemo eksponencijalno. Vježbu smo podijelili u 2 dijela.

1.DIO VJEŽBE

U 1.dijeli vježbe smo provjerili koliko nekim kriptografskim hash funkcijama treba vremena za *hashiranje*. Iako su to nama sve jako brze funkcije, postoji znatna razlika u vremenu koje je potrebno da se *hashira* neka informacija ili obična linija teksta. Te nama naizgled sitnice, možemo iskoristiti kada radimo neku vrstu aplikacije koja ima neki oblik *logina* ili registacije korisnika. Zašto i kako? Zašto: gledajući sa sigurnosnog aspekta, ako napadač koristi nekakav riječnik za *brute-force* napad, svaki trenutak je važan. Usporavanjem hash funkcije i uvožđenjem neke vrste *timeouta* (nakon npr. 5 pogrešnih unosa lozinke morate pričekati 10 sekundi) uvelike demoralioziramo napadača. Naravno, ako ste *legit user*, vi želite da se vaš login odvije gotovo neprimjetno te je zbog toga bitan balans između usporavanja napadača te vrijeme koje je *user* voljan čekati za login. Ako čuvamo sigurnost neke naše aplikacije koju radimo trenutno 'iz zabave' tada nećemo toliki naglasak dati sigurnosti tj. koristiti toliko složenu iterativnu hash funkciju (iterativna hash funkcija je ona koja će višestruko hashirati početnu informaciju kako bi došla do konačnog rezultata), ali ako čuvamo sigurnost nekog *walleta* u kojem se potencijalno nalaze milijuni, tada će i korisnik imati razumijevanja ako će njegov login trajati koju sekunudu duže jer je u njegovom najboljem interesu. Kako bismo još dodatno osigurali sigurnost korisnikovih podataka, neke hash funkcije (npr. Argon2) koriste i sol. Sol je 'varijabla' koju kriptografska hash funkcija koristi kao faktor nepredvidljivosti tj. kaka hashiramo neki *password* koji se potencijalno nalazi u napadačevom riječniku dobit ćemo jedan hash, ali ako dodamo još tu sol pa sa njom *hashiramo* taj isti *password*, dobijamo posve drugi hash. To nam omogučava da koristimo isti *password* na više mijesta tj. u više aplikacija jer uvijek imamo tu dodatnu zaštitu u obliku soli. Ove sve spoznaje smo koristili u 1.dijelu vježbe.

CODE

```
from os import urandom

from prettytable import PrettyTable

from timeit import default_timer as time
```

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from passlib.hash import sha512_crypt, pbkdf2_sha256, argon2


def time_it(function):
    def wrapper(*args, **kwargs):
        start_time = time()
        result = function(*args, **kwargs)
        end_time = time()
        measure = kwargs.get("measure")
        if measure:
            execution_time = end_time - start_time
            return result, execution_time
        return result
    return wrapper


@time_it
def aes(**kwargs):
    key = bytes([
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
    ])

    plaintext = bytes([
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    ])

    encryptor = Cipher(algorithms.AES(key), modes.ECB()).encryptor()
    encryptor.update(plaintext)
    encryptor.finalize()


@time_it
def md5(input, **kwargs):
    digest = hashes.Hash(hashes.MD5(), backend=default_backend())
    digest.update(input)
    hash = digest.finalize()
    return hash.hex()


@time_it
def sha256(input, **kwargs):
    digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
    digest.update(input)
    hash = digest.finalize()
    return hash.hex()


@time_it
def sha512(input, **kwargs):
    digest = hashes.Hash(hashes.SHA512(), backend=default_backend())
    digest.update(input)
    hash = digest.finalize()
```

```
    return hash.hex()


@time_it
def pbkdf2(input, **kwargs):
    # For more precise measurements we use a fixed salt
    salt = b"12QIp/Kd"
    rounds = kwargs.get("rounds", 10000)
    return pbkdf2_sha256.hash(input, salt=salt, rounds=rounds)


@time_it
def argon2_hash(input, **kwargs):
    # For more precise measurements we use a fixed salt
    salt = b"0"*22
    rounds = kwargs.get("rounds", 12)            # time_cost
    memory_cost = kwargs.get("memory_cost", 2**10) # kibibytes
    parallelism = kwargs.get("rounds", 1)
    return argon2.using(
        salt=salt,
        rounds=rounds,
        memory_cost=memory_cost,
        parallelism=parallelism
    ).hash(input)


@time_it
def linux_hash_6(input, **kwargs):
    # For more precise measurements we use a fixed salt
```

```
    salt = "12QIp/Kd"
    return sha512_crypt.hash(input, salt=salt, rounds=5000)


@time_it
def linux_hash(input, **kwargs):
    # For more precise measurements we use a fixed salt
    salt = kwargs.get("salt")
    rounds = kwargs.get("rounds", 5000)
    if salt:
        return sha512_crypt.hash(input, salt=salt, rounds=rounds)
    return sha512_crypt.hash(input, rounds=rounds)


@time_it
def scrypt_hash(input, **kwargs):
    salt = kwargs.get("salt", urandom(16))
    length = kwargs.get("length", 32)
    n = kwargs.get("n", 2**14)
    r = kwargs.get("r", 8)
    p = kwargs.get("p", 1)
    kdf = Scrypt(
        salt=salt,
        length=length,
        n=n,
        r=r,
        p=p
    )
    hash = kdf.derive(input)
```

```
    return {
        "hash": hash,
        "salt": salt
    }


if __name__ == "__main__":
    ITERATIONS = 100
    password = b"super secret password"

    MEMORY_HARD_TESTS = []
    LOW_MEMORY_TESTS = []


    TESTS = [
        {
            "name": "AES",
            "service": lambda: aes(measure=True)
        },
        {
            "name": "HASH_MD5",
            "service": lambda: sha512(password, measure=True)
        },
        {
            "name": "HASH_SHA256",
            "service": lambda: sha256(password, measure=True)
        },
        {
            "name": "HASH_SHA512",
```

```
        "service": lambda: sha512(password, measure=True)
    },
    {
        "name": "Linux CRYPT_6",
        "service": lambda: linux_hash_6(password, measure=True)
    },
    {
        "name": "Linux CRYPT_10k",
        "service": lambda: linux_hash(password, salt = "test", rounds = 10**3, measure=True)
    },
    {
        "name": "ARGON2_20",
        "service": lambda: argon2_hash(password, rounds = 20, measure=True)
    },
    {
        "name": "SCRYPT_N_2_16",
        "service": lambda: scrypt_hash(password, lenght = 64, salt = urandom(16), n = 2**16,
mesure = True)
    }
]

table = PrettyTable()
column_1 = "Function"
column_2 = f"Avg. Time ({ITERATIONS} runs)"
table.field_names = [column_1, column_2]
table.align[column_1] = "l"
table.align[column_2] = "c"
table.sortby = column_2

for test in TESTS:
```

```
name = test.get("name")
service = test.get("service")


total_time = 0
for iteration in range(0, ITERATIONS):
   print(f"Testing {name:>6} {iteration}/{ITERATIONS}", end="\r")
   _, execution_time = service()
   total_time += execution_time
average_time = round(total_time/ITERATIONS, 6)
table.add_row([name, average_time])
print(f"{table}\n\n")
```

2.DIO VJEŽBE

Koristeći spoznaje iz 1.dijela vježbe, kreiramo SQLite bazu podataka u koju ćemo spremati *username* i *password* korisnika. Ovdije isto upozajemo nekoliko stvari koje su bitne za čuvanje sigurnosti podataka kao što su: koristimo getpass za dohvaćanje lozinke (getpass neće ispisati lozniku na konzolu kada je unosimo), kako kreirati bazu da nam ne dozvoli unošenje korisnika sa istim *usrenamemo*, traženje unosa i *usernamea* i *passworda* kako bismo dodatno usporili samog napadača itd. Iako zahtjevamo unos *usernamea* i *passworda*, tu postoji način da napadač zna da li je *username* ili *password* pogriješan. Naime, naš program funkcionira na slijedeći način: tražemo unos podaraka u konzolu i tada prvo preko SQl upita provjeravamo da li postoji taj user te ako postoji provjerajemo *password* Argon2 funkcijom i ako jedan od to dvoje nije ispravan, ispisujemo na konzolu da *usernamea* ili *password* nije točan. Za razliku od Argon2 provjere, SQL upit je izrazito brz te ako je vrijeme odgovora bilo kratko, tada napadač zna da je pogriješio u *usernameu* te tada prilagođava parametre napada, a ako je odgovor trajao duže, onda je pogriješna lozinka. Sve ove spoznaje smo primjenili za kôd 2.dijela vježbe.

CODE

```
import sys

from InquirerPy import inquirer
from InquirerPy.separator import Separator

import sqlite3
from sqlite3 import Error
from passlib.hash import argon2

import getpass

def verify_password(password: str, hashed_password: str) -> bool:
    # Verify that the password matches the hashed password
    return argon2.verify(password, hashed_password)



def get_user(username):
    try:
        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM users WHERE username = ?", (username,))
        user = cursor.fetchone()
        conn.close()
        return user
    except Error:
        return None
```

```
def register_user(username: str, password: str):
    # Hash the password using Argon2
    hashed_password = argon2.hash(password)     #creates salt on its own behaf

    # Connect to the database
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()

    # Create the table if it doesn't exist
    cursor.execute(
        "CREATE TABLE IF NOT EXISTS users (username TEXT PRIMARY KEY UNIQUE, password TEXT)"
    )

    try:
        # Insert the new user into the table
        cursor.execute("INSERT INTO users VALUES (?, ?)", (username, hashed_password))

        # Commit the changes and close the connection
        conn.commit()
    except Error as err:
        print(err)
    conn.close()

def do_register_user():
    username = input("Enter your username: ")

    # Check if username taken
    user = get_user(username)
    if user:
```

```
        print(f'Username "{username}" not available. Please select a different name.')
        return

    password = getpass.getpass("Enter your password: ")
    register_user(username, password)
    print(f'User "{username}" successfully created.')


def do_sign_in_user():
    username = input("Enter your username: ")
    password = getpass.getpass("Enter your password: ")
    user = get_user(username)

    if user is None:
        print("Invalid username or password.")
        return

    password_correct = verify_password(password=password, hashed_password=user[-1])

    if not password_correct:
        print("Invalid username or password.")
        return
    print(f'Welcome "{username}".')


if __name__ == "__main__":
    REGISTER_USER = "Register a new user"
    SIGN_IN_USER = "Login"
    EXIT = "Exit"
```

```
while True:
    selected_action = inquirer.select(
        message="Select an action:",
        choices=[Separator(), REGISTER_USER, SIGN_IN_USER, EXIT],
    ).execute()

    if selected_action == REGISTER_USER:
        do_register_user()
    elif selected_action == SIGN_IN_USER:
        do_sign_in_user()
    elif selected_action == EXIT:
        sys.exit(0)
```