Laboratorijska vježba 5 - Password-hashing (iterative hashing, salt, memory-hard functions)

U prvom dijelu ove laboratorijske vježbe smo pokrenuli izborni kod kako bi mogli usporediti brze i spore kriptografske hash funkcije na osnovi vremena izvršavanja.

Zaključili smo da je vrijeme izvršavanja kod sporih hash funkcija malo iako mislimo da je to nedovoljno za demotivirati napadača, ali u usporedbi s brzim hash funkcijama i kada još dodamo broj iteracija koje napadač mora iterirati, spore hash funkcije poprilično demotiviraju napadača do te mjere da mu se ne isplati niti pokušati izvršiti napad zbog ekonomske neisplativosti, a i jer možda nema potrebne resurse memorije i CPU-a kako bi neometano izvršio napad.

Izvorni kod koji smo pokrenuli :

```
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: sha512(password, measure=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")
```

U drugom dijelu vježbe smo implementirali jednostavan sustav autentikacije korisnika. Pokretanjem koda, imamo izbornik sa 3 opcije; registracija novog korisnika, prijava već postojećeg korisnika i izlaz. Prilikom registracije korisnika, moramo provjeriti postoji li već korisnik sa unesenim username-om. Svaki korisnik ima jedinstveni username. Sama lozinka korisnika se hashira i uz sami hash ima i sol pa tako neka dva korisnika, ako i imaju identičnu lozinku, sigurnost neće biti narušena jer će se lozinke međusobno razlikovati po soli. Prilikom prijave već postojećeg korisnika, sustav traži da se unese username i lozinka, a tek onda provjerava postoji li taj username u bazi. Ako bi tražili samo unos username-a te odmah provjeravali postoji li, napadač bi mogao znati koji sve točno username-ovi postoje u bazi pa iz tog razloga, sustav odmah traži oba podatka.

Pitanja

1. Koliko korisnika je registrirano u bazu podataka? - 3.
2. Usporedite hash vrijednosti zaporki korisnika `jdoe` i `jean_doe`. Što možete zaključiti? - Oba korisnika imaju istu lozinku, ali im je hash vrijednost različita zbog soli koja je dodana u hash vrijednost.
3. Zašto pri provjeri unesene zaporke `argon2` funkcija treba oboje, zaporku i njenu `hash` vrijednost? - Kako bi mogli usporediti unesenu lozinku sa hash vrijednošću ispravne lozinke.

4. Koji još važan element treba `argon2` za ispravnu provjeru unesene zaporke? - Treba sol koja se nalazi na kraju hash vrijednosti lozinke.
5. Zašto u funkciji `do_sign_in_user()` tražimo od korisnika da uvijek unese oboje, `username` i `password`, čak iako `username` potencijalno nije ispravan? - Kada bi nakon unosa username-a odmah provjeravali postoji li korisnik ili ne, napadač bi sa sigurnošću znao koji username-ovi postoje u bazi podataka, a kada provjeravamo nakon unosa username-a i lozinke, napadač ne zna je li greška u username-u ili lozinci.

Izvršavanje koda ;

Prilikom registracije, svaka lozinka se hashira i ima jedinstvenu vrijednost. Kod provjere lozinke, argon2 iz unesene lozinke uz pomoć javne soli generira hash vrijednost i onda uspoređuje sa ispravnom koja je pohranjena u bazi podataka. Za prijavu već postojećeg korisnika, kao što je već rečeno, od korisnika se traži i username i lozinka kako napadač ne bi znao je li greška u username-u ili lozinci.

Kod za sustav autentikacije korisnika :

```
from passlib.hash import argon2
from sqlite3 import Error
import sqlite3
import sys
from InquirerPy import inquirer
from InquirerPy.separator import Separator
import getpass


def register_user(username: str, password: str):
    # Hash the password using Argon2
    hashed_password = argon2.hash(password)

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


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)
```