Sigurnost računala i podataka - lab 5

Lab 5: Password-hashing (iterative hashing, salt, memory-hard functions)

U laboratorijskoj vježbi pet upoznali smo se s osnovnim konceptima vezanim za sigurnu pohranu lozinki. U prvom dijelu vježbe smo pokrenuli kodove kako bi usporedili vrijeme brzih odnosno klasičnih i sporih odnosno specijaliziranih kriptografskih funkcija za sigurnu pohranu lozinki i izvođenje enkripcijskih ključeva. Vrijeme hashiranja kod sporih hash funkcija je jako malo i nedovoljno za demotivirati napadača, no u usporedbi s brzim hash funkcijama koje su i do 1000 puta brže i kada uzmemo u obzir broj pokušaja hashiranja koje bi napadač trebao izvesti shvaćamo da bi to vjerojatno bilo popriličnо dovoljno za demotivirati napadača. Također, napad je i ekonomski neisplativ i napadaču su potrebni veliki resursi memorije i CPU-a kako bi napad bio moguć, što može dodatno demotivirati napadača.

Korišteni kod za usporedbu brzih i sporih hash funkcija

```
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")
```

Nakon usporedbe hash funkcija implementirali smo sustav za autentikaciju korisnika. Sustav nam je davao tri mogućnosti pri pokretanju: registracija novog korisnika, prijavu postojećeg korisnika i izlaz. Prilikom registracije morali smo paziti na to je li se netko registrira s već postojećim usrename-om jer svaki korisnik treba imati jedinstven username . Pomoću SQLite-a smo implementirali bazu podataka u koju su se spremali svi registrirani korisnici. Lozinka se spremala hashirano i ima sol što je dodatna zaštita koja sprječava da napadač zaključi da korisnici imaju istu šifru na osnovu iste hash vrijednosti. Dodatna zaštita je i činjenica da sustav ako dođe do pogreške prilikom log-in-a javlja samo da je log-in neispravan bez specificiranja u kojem dijelu je greška; username-u ili password-u. Odnosno sustav traži unos i lozinke i username-a da bi javio uspješnu ili neuspješnu prijavu, jer ako bi tražio prvo username, napadaču bi to pomoglo pri otkrivanju username-a te bi trebao samo pronaći pripadajuće lozinke.

Korišteni kod za log in i registraciju korisnika:

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

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)
```

Odgovori na pitanja:

1. Koliko korisnika je registrirano u bazu podataka?
   U bazi su registrirana 3 korisnika.

2. Usporedite hash vrijednosti zaporki korisnika `jdoe` i `jean_doe`. Što možete zaključiti?
   Iako korisnici imaju istu lozinku, hash vrijednost im je drugačija te zaključujemo da je dodana sol te su stoga hash vrijednosti različite.

3. Zašto pri provjeri unesene zaporke `argon2` funkcija treba oboje, zaporku i njenu `hash` vrijednost?
   Kako bi mogao usporediti hash vrijednost ispravne lozinke sa unesenom lozinkom.

4. Koji još važan element treba `argon2` za ispravnu provjeru unesene zaporke?
   Potrebna mu je i sol koje se nalazi unutar hash vrijednosti.

5. 1. Zašto u funkciji `do_sign_in_user()` tražimo od korisnika da uvijek unese oboje, `username` i `password`, čak iako `username` potencijalno nije ispravan?
   Da ne bi olakšali posao napadaču, ako sazna ispravan username iz baze ostaje mu samo pronaći odgovarajuću lozinku.