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

- Analizirali smo korištenje hash funkcija za pohranu lozinki
- Lozinke pohranjujemo u hashiranom obliku
- Uspoređivali smo brzinu AES, SHA512 i SHA256
- SHA512 i i SHA256 imaju otprilike jednaka vremena izvršavanja
- Ovisno o brzini hashiranja funkcije imaju različite funkcije (npr. ekstremno spore funkcije se koriste za zaštitu vrijednih informacija)

```
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",
```

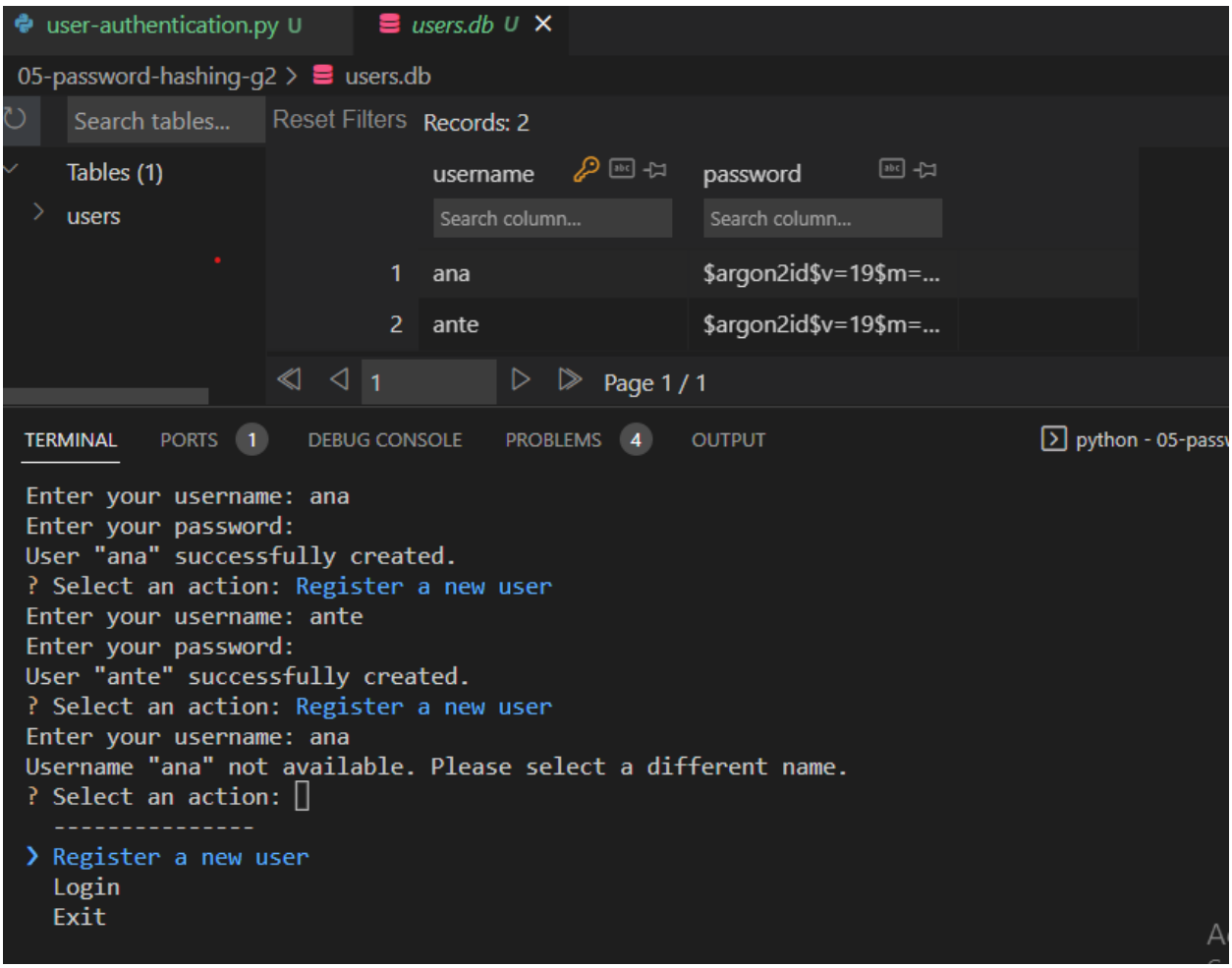
```
            "service": lambda: aes(measure=True)
        },
        {
            "name": "HASH_MD5",
            "service": lambda: sha512(password, measure=True)
        },
        {
            "name": "HASH_SHA256",
            "service": lambda: sha256(password, measure=True)
        }
    ]

    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")
```

- Implementirali smo autentikaciju korisnika (koristili smo SQLite bazu podataka i Argon2 password hashing funkcije koja koristi salt u hashiranju)
- Šifre svih korisnika imaju drukčije hash vrijednosti zbog jedinstvene vrijednosti salt-a
- Kod verifikacije password-a unešene vrijednosti se hashiraju i uspoređuju s postojećim hash vrijednostima u bazi
- Korisnici ne mogu imati ista korisnička imena, ali mogu imati iste šifre (zbog salt-a bit će pohranjenje kao različite hash vrijednosti)
- U funkciji do_sign_in_user() se traži unos username-a i password-a jer napadač neće znati što je pogrešno

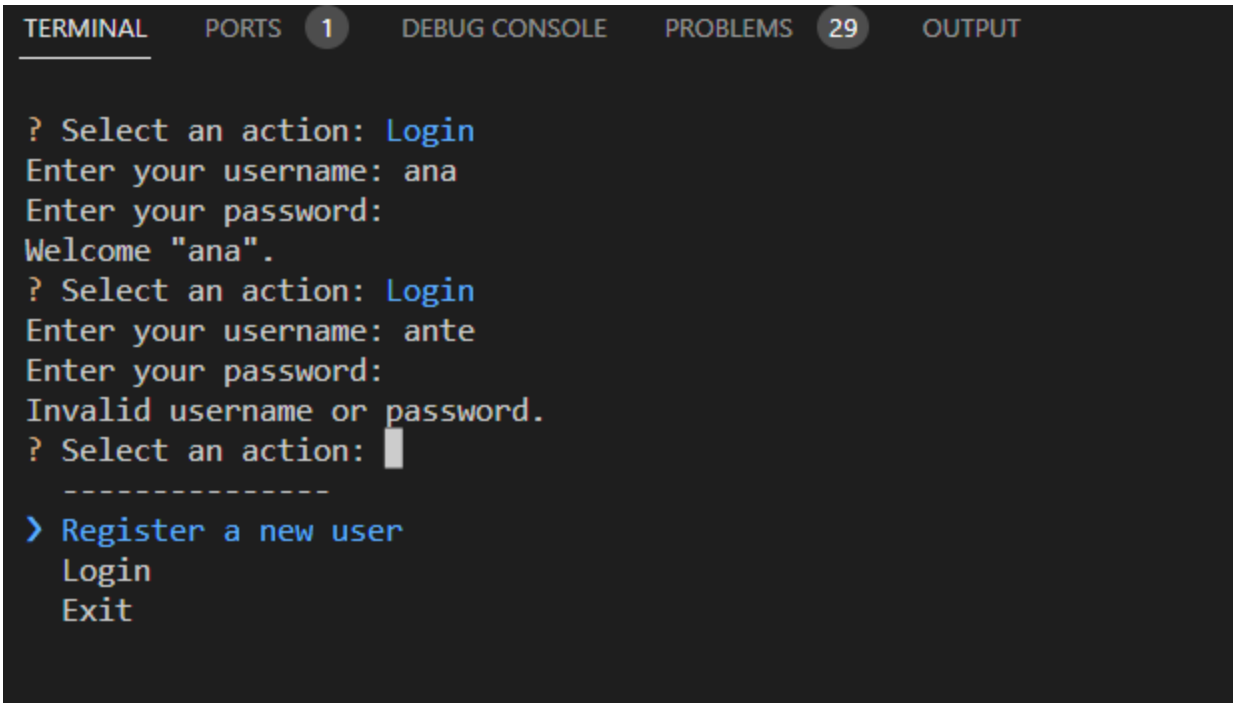
- Što je pogrešno unešeno može zaključiti na osnovu razlike u vremenu potrebnom da odgovor dođe do njega

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


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 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)
```


user-authentication.py U
users.db U
05-password-hashing-g2 > users.db
Search tables...
Reset Filters
Records: 2
Tables (1)
users
username
password
Search column...
Search column...
1 ana $argon2id$v=19$m=...
2 ante $argon2id$v=19$m=...
Page 1 / 1
TERMINAL
PORTS 1
DEBUG CONSOLE
PROBLEMS 4
OUTPUT
python - 05-passv
Enter your username: ana
Enter your password:
User "ana" successfully created.
? Select an action: Register a new user
Enter your username: ante
Enter your password:
User "ante" successfully created.
? Select an action: Register a new user
Enter your username: ana
Username "ana" not available. Please select a different name.
? Select an action:

Register a new user
Login
Exit


TERMINAL
PORTS 1
DEBUG CONSOLE
PROBLEMS 29
OUTPUT
? Select an action: Login
Enter your username: ana
Enter your password:
Welcome "ana".
? Select an action: Login
Enter your username: ante
Enter your password:
Invalid username or password.
? Select an action:

❯ Register a new user
Login
Exit