SRP Vježba 4

Prvi dio:

Svaki student dobio je dvije slike i dva potpisa. Jedna od tih slika je lažna, cilj je ustvrditi koja.

Na raspolaganju imamo javni ključ profesora, obe slike i oba potpisa.

```
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.exceptions import InvalidSignature

PUBLIC_KEY_FILE = "public.pem"
IMAGE_1_SIGNATURE = "challenges\prezime_ime\public_key_challenge\image_1.sig"
IMAGE_1_PNG = "challenges\prezime_ime\public_key_challenge\image_1.png"
IMAGE_2_SIGNATURE = "challenges\prezime_ime\public_key_challenge\image_2.sig"
IMAGE_2_PNG = "challenges\prezime_ime\public_key_challenge\image_2.png"

with open(IMAGE_1_SIGNATURE, "rb") as file:
  SIGNATURE1_Obj = file.read()

with open(IMAGE_1_PNG, "rb") as file:
  IMAGE1_Obj = file.read()

with open(IMAGE_2_SIGNATURE, "rb") as file:
  SIGNATURE2_Obj = file.read()

with open(IMAGE_2_PNG, "rb") as file:
  IMAGE2_Obj = file.read()

def load_public_key():
    with open(PUBLIC_KEY_FILE, "rb") as f:
        PUBLIC_KEY = serialization.load_pem_public_key(f.read(),backend=default_backend())
    return PUBLIC_KEY

def verify_signature_rsa(signature, message):
    PUBLIC_KEY = load_public_key()
    try:
        PUBLIC_KEY.verify(signature, message, padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH), hashes.SH
    except InvalidSignature:
        return False
    else:
        return True

if __name__ == "__main__":
  print(load_public_key())
  print(verify_signature_rsa(SIGNATURE1_Obj, IMAGE1_Obj))
  printverify_signature_rsa(SIGNATURE2_Obj, IMAGE2_Obj))
```

Zaključak: Primjetimo da za provjeru uzimamo i sliku i potpis. Zašto nam je potrebna i slika a ne samo potpis? Zato što sam potpis ne sadrži originalnu informaciju u sebi nego je on produkt *hash* izlaza. I samim time moramo lokalno generirati naš potpis da ga možemo usporediti sa originalom, naravno ovo radi funkcija iz biblioteke.

Drugi dio:

Upoznajemo se s brzim i sporim te “*memory-hard”* kriptografskim hash funkcijama. Te funkcije koristimo za generiranje sekvenci koje kasnije pohranjujemo u memoriju. Same zaporke nikad ne pohranjujemo u *plaintext-u*.

```
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"tajna zaporka"
    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")
```

Ovaj kod ispisuje vremena izvršavanja pojedinih kripografskih hash funkcija. Vidimo da povećavanjem broja iteracija raste vrijeme izvršavanja. Za jednu zaporku to nije toliko bitno za krajnjeg korisnika koji se prijavi na servis par puta dnevno. No za nekoga napadača koji mora provjeriti milijune lozinki ovo mu zadaje ogromne vremenske probleme. Kasnije testiramo Argon2 funkciju koja je memory-hard. Prosljedimo joj koliko radne memorije želimo koristiti. Što više radne memorije samo generiranje će biti brže. No ako nemamo dovoljno memorije vrijeme generiranja značajno raste. Ovime smo značajno usporili napadača i u financijskom (hardver) i vremenskom smislu. Još jedan benefit je onemogućili smo paralelizaciju.