LAB 4

Sigurnost računala i podataka

Password-hashing (iterative hashing, salt, memory-hard functions)

Ovo nam je ostalo od LABA 3 , jer nismo verifyali key protekli put

```
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
def load_public_key():
  with open("public.pem", "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.SHA256()
    )
  except InvalidSignature:
      return False
  else:
      return True
if **name** == "**main**":
    # Reading from a file
    with open("image_2.sig", "rb") as file:
      signature = file.read()

    with open("image_2.png", "rb") as file:
      image = file.read()
```

```
    is_authentic = verify_signature_rsa(signature, image)
    print(is_authentic)
```

Ovdje kreće LAB 4.Ovdje smo radili na pohrani lozinke. Usporedili smo brze kriptografske hash funkcije i specijalizirane, sporije hash funkcije . Instalirali smo pakete potrebne za vježbu naredbom :

```
pip install -r requirements.txt
```

Kopirali smo kod:

```
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")
```

Gledali smo AES, HASH_MD5 i HASH_SHA256 vremena izvršavanja i uspoređivali ih

Potom smo testirali Linux Crypt koji koristi 5000, 10 000, 50 000 i 100 000 iteracija hashiranja.

```
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)

        },

        {

            "name": "Linux CRYPT",

            "service": lambda: linux_hash(password, measure=True)
```

```
        },
        {
            "name": "Linux CRYPT 10k",
            "service": lambda: linux_hash(password, rounds = 10**4, measure=True)
        },
        {
            "name": "Linux CRYPT 50k",
            "service": lambda: linux_hash(password, rounds = 5*10**4, measure=True)
        },
        {
            "name": "Linux CRYPT 100k",
            "service": lambda: linux_hash(password, rounds = 10**5, measure=True)
        }
    ]
```

Zaključili smo da veći broj iteracija znaci i veće vrijeme izvršavanja i s time možemo usporiti napad.

Potom smo testirali linux_hash i vidjeli da ona nadodaje salt tijekom hashiranja.