Sigurnost računala i podataka

Lab 4 - Public-key crpyptography and Password Hashing

Prvo smo u pythonu otvorili virtualno okruženje

`python -m venv srp`

Onda smo otišli u direktorij /scripts i pokrenuli /activate

`cd env`

`cd Scripts`

`activate`

Public-key crpytography

Otvorili smo visual studio code sa naredbom `code .` i napravili python dokument public_key.py

U ovom izazovu trebali smo odrediti autentičnu sliku (između dvije ponuđene) koju je profesor potpisao svojim privatnim ključem. Odgovarajući javni ključ bio je dostupan na serveru.

U datoteku gdje se nalazi kod smo preuzeli naše dvije slike te i njihove odgovarajuće potpise.

Slike i odgovarajući digitalni potpisi nalaze se u direktoriju `prezime_ime\public_key_challenge`. Kao i u prethodnoj vježbi, za rješavanje ove koristite Python biblioteku `cryptography`.

Prvo smo ubacili kod za učitavanje javnog ključa te vidjeli da li je to ispravno napravljeno pomoću printa:

```
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend

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

#------------------------
print(load_public_key()) #isprinta javni kljuc nakon deserijalizacije, tada je nasta nekakav objekt PUBLIC_KEY
```

Sveukupni kod nam je izgledao ovako:

```
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( //verify sluzi za usporedbu kao kod verificiranja certifikata
            signature,
            message,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
    except InvalidSignature:
        return False
    else:
        return True


#------------------------
#print(load_public_key())

 # Reading from a file
 with open("image_1.sig", "rb") as file:
     signature = file.read()

with open("image_1.png", "rb") as file:
     image = file.read()

is_authentic = verify_signature_rsa(signature, image)
print(is_authentic) //vraca ako je slika autenticna ili ne
```

Funkcija verify_signature prima poruku tj. u ovom slučaju sliku, i potpis te poruke. Uspoređuje taj potpis sa onim koji generira na sličan način kao kod certifikata te na kraju vraća true ako su jednaki i false ako su različiti.

Password Hashing

U okviru ove vježbe upoznali smo se pobliže sa osnovnim konceptima relevantnim za sigurnu pohranu lozinki. Usporedili smo klasične (*brze*) kriptografske *hash* funkcije sa specijaliziranim (*sporim* i *memorijski zahtjevnim*) kriptografskim funkcijama za sigurnu pohranu zaporki i izvođenje enkripcijskih ključeva (*key derivation function (KDF)*).

Za potrebe ove vježbe trebali smo instalirati requierments `pip install -r requirements.txt`.

Kopirali smo slijedeći kod:

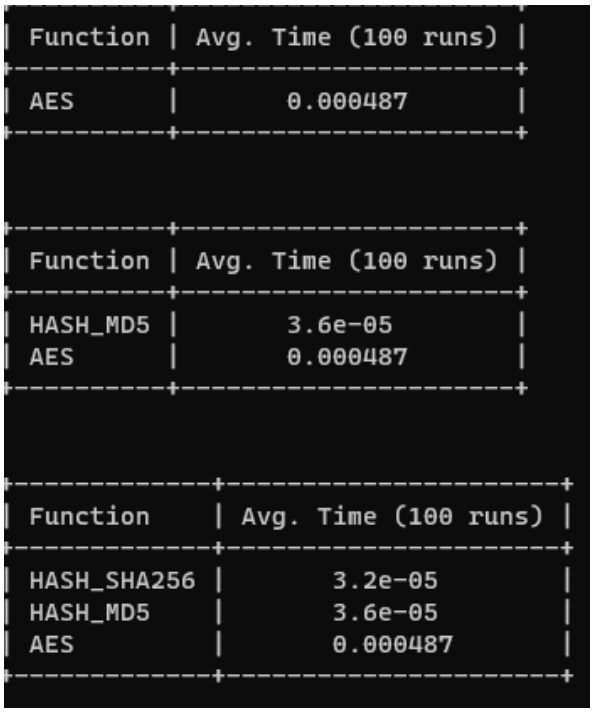
```
from os import urandom
from prettytable import PrettyTable
from timeit import default_timer as time
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
from cryptography.hazmat.primitives.ciphers 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")
```

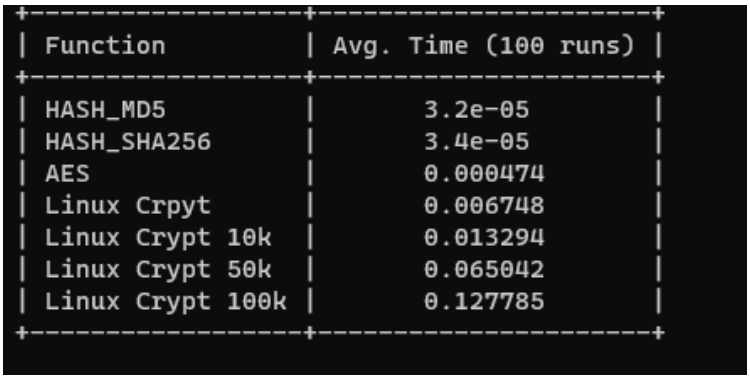
Prvo smo testirali slijedeće hash funkcije: AES, HASH_MD5, HASH_SHA256.

Dobili smo slijedeća prosječna vremena izvršavanja (100 puta su se izvršavali algoritmi pa se uzimalo prosječno vrijeme svih):



Nakon toga dodali smo još par testova u kojima smo testirali vrijeme iteracija Linux Crypta koji koristi 5000 iteracija hashiranja, onda 10k, pa 50k i na kraju 100k 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 Crpyt", #koristi 5000 iteracija
            "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)
        }
    ]
```



Function	Avg. Time (100 runs)
HASH_MD5	3.2e-05
HASH_SHA256	3.4e-05
AES	0.000474
Linux Crpyt	0.006748
Linux Crypt 10k	0.013294
Linux Crypt 50k	0.065042
Linux Crypt 100k	0.127785

Vidimo iz rezultata da se povećanjem iteracija hashiranja povećava i prosječno vrijeme izvršavanja.

Na taj način se povećava razina sigurnosti jer treba i samom napadaču dulje vremena da napravi svoje napade.

Također smo napravili ovu izmjenu u kodu da vidimo da neke hash funkcije koriste i sol pri stvaranju hash vrijednosti. Zato nam ove dvije print funkcije ne izbacuju istu vrijednost jer funkcija linux_hash koristi sol pri stvaranju hash vrijednosti.

```
# 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")

    print(linux_hash(password))
    print(linux_hash(password))
#necemo dobit iste hash vrijednosti jer se nadodaje sol
```

Na kraju smo i testirali hash funkciju koja je ne samo vremenski zahtjevna već i memorijski zahtjevna. To jest, koristili smo argon2_hash funkciju u kojoj možemo postaviti koliko radne memorije je potrebno pri generiranju hash vrijednosti. U donjem primjeru smo postavili da je za to potrebno 1 GB memorije te smo pri pokretanju funkcije, promatrali u task manageru stanje memorije i uistinu vidjeli kako se korištenje poveća za 1 GB.

```
argon2_hash(password, rounds = 20, memory_cost=2**20) #testirali argon- da zauzme 1gb radne memorije
```