Vježba 4: Password-hashing (iterative hashing, salt, memory-hard functions)

Precomputed dictionary attack

Dictionary napad je tip brute force napada pri kojem se pokušava pronaći šifra tako da se napravi lista šifri kojom se pokušava pristupiti nečijem računu. Takva lista sadrži milione najčešćih šifri. Razlog zašto ovakav napad funkcionira je zbog toga što ljudi često koriste šifre s malom entropijom tj. neke šifre se ponavljaju češće nego druge. S obzirom da se šifre spremaju hashirane u bazu podataka i šifre u dictionariju su hashirane. Pri napadu se pokušavaju naći dvije iste hash vrijednosti.

Postoje više sigurnosnih postupaka kako bi se obranili od precomputed dictionary napada:

- iterativno hashiranje
- sol
- memory-hard funkcije

Iterativno hashiranje

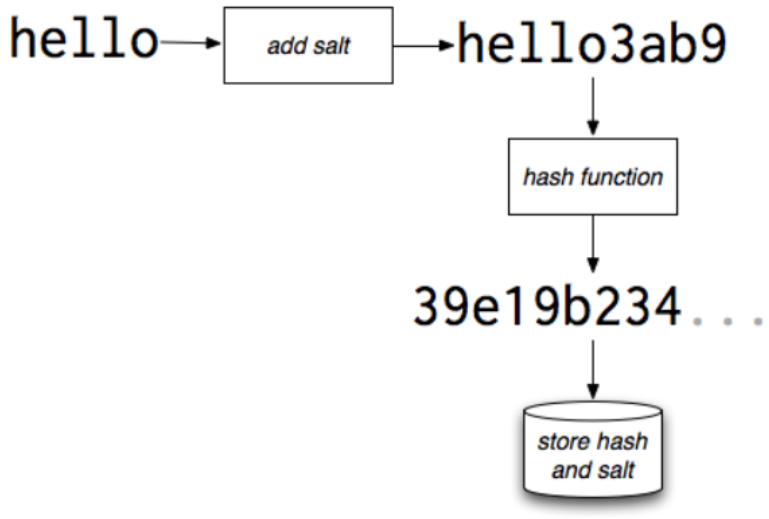
Iterativno hashiranje je jednostavno postupak uzostopnog hashiranja poruke koje hashiramo.

- $H^n(p) = H(...H(H(p))...)$

Cilj iterativnog hashiranja je usporiti cijeli proces napada.
Cijeli proces će se usporiti za *n* puta, ovisno koliko puta vršimo iterativno hashiranje.
To znači i da će se pri korisničkom pokušaju loggiranja vrijeme odužiti, ali neznatno s obzirom da se radi o milisekundama. S druge strane napadač pokušava brute forceat s milijunama šifri, s tim da svaku treba hashirati *n* puta. Za **SHA-256**, preporučen minimalan broj iteracija (*n*) je 10 000.

Sol

Sol je random generirani string koji se dodaje šifri prije iterativnog hashiranja.
Pošto većina naših šifri predstavljaju prave riječi, napadaču je lakše pogoditi samu šifru.
Smisao soli je dodati niz random simbola koje nemaju neko značenje u našu šifru, te nakon toga iterativno hashirati kako bi se smanjila vjerovatnost pogađanja šifre.

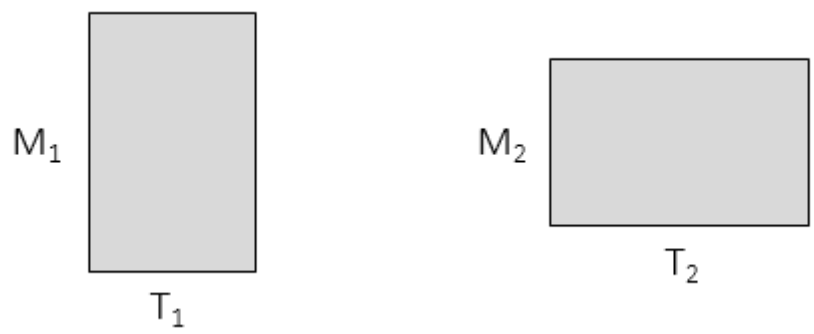


Sol se sprema u bazu podataka, te se pri pokušaju loggiranja poziva, dodaje šifri i hashira.
Usporedbom dvaju hash vrijednosti gledamo je li unesena šifra ispravna.

Memory-hard funkcije

Memory-hard funkcije poput **Argon2** i **Scrypt** su hash algoritmi koje pri izračunu hash vrijednosti koriste puno radne memorije računala. U slučaju napada, napadač mora imati enormno mnogo računalne snage kako bi se uspio loggirati. (Ili vremena, što

napadaču nije isplativo)



Vježba

Pri pisanju koda koristio se dekorator *@time_it,* što je funkcija koja kao argument prima funkciju, te isto tako i vraća funkciju. Pri korištenju se koristi kao ekstenzija ili modifikacija funkciji ne mijenjajući kôd početne funckije. U ovom slučaju, kako ne bi ponavljali kôd računanja vremena, pozovemo samo dekorator *@time_it* tj. služi kao dodatak pri pozivanju funkcija.

Kôd:

password_hashing.py > argon2_hash

```python
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()
```

```
@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)
```

```python
@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
    }

```

```python
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:
```

```python
    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")
```

Prvi rezultat:

Uspoređujemo algoritme **AES, MD5 i SHA-256**

```python
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)
    }
]
```

```
(marko_kusacic) C:\Users\A507\marko_kusacic\marko_kusacic\Scripts>python password_hashing.py
+----------+----------------------+
| Function | Avg. Time (100 runs) |
+----------+----------------------+
| AES      |       0.003413       |
+----------+----------------------+


+----------+----------------------+
| Function | Avg. Time (100 runs) |
+----------+----------------------+
| HASH_MD5 |       3.3e-05        |
| AES      |       0.003413       |
+----------+----------------------+


+-------------+----------------------+
| Function    | Avg. Time (100 runs) |
+-------------+----------------------+
| HASH_SHA256 |        3e-05         |
| HASH_MD5    |       3.3e-05        |
| AES         |       0.003413       |
+-------------+----------------------+
```

Primjećujemo da je prosječno vrijeme potrebno za hashiranje **SHA-256** i **MD5** algoritmima brže nego **AES-om** za otprilike **100 puta.** Zato je **AES** prikladniji pri zaštiti od napada.

Drugi rezultat:

Sada uspoređujemo istu funckiju hashiranu 5 tisuća i milion puta.

```
{
    "name": "Linux CRYPTO 5k",
    "service": lambda: linux_hash(password, measure=True)
},
{
    "name": "Linux CRYPTO 1M",
    "service": lambda: linux_hash(password, rounds=10**6, measure=True)
}
```

```
+----------------+----------------------+
| Function       | Avg. Time (100 runs) |
+----------------+----------------------+
| HASH_MD5       |        3e-05         |
| HASH_SHA256    |        3e-05         |
| AES            |       0.00047        |
| Linux CRYPTO 5k |       0.006847       |
+----------------+----------------------+



+----------------+----------------------+
| Function       | Avg. Time (100 runs) |
+----------------+----------------------+
| HASH_MD5       |        3e-05         |
| HASH_SHA256    |        3e-05         |
| AES            |       0.00047        |
| Linux CRYPTO 5k |       0.006847       |
| Linux CRYPTO 1M |       1.264356       |
+----------------+----------------------+
```

Uspoređivanjem funkcije **Linux CRYPTO 5k** (hashirane 5 tisuća puta) i
Linux CRYPTO 1M (hashirane milion puta) vidimo da je razlika prosječnog vremenea hashiranja
185 puta, tj. onoliko puta koliko hashiramo, toliko produžujemo vrijeme da se hashira.
Cilj programera je naći ravnotežu između toga da se dovoljno zaštiti i da produženo vrijeme neometa rad s podacima.