0x14 Data Security Exercise Session

07.10.2024

hexhive

EPFL

Recap

Access Control in Databases

Least privilege: Each actor can only access the resources it strictly needs

Defense in depth: Multiple layers of security

- Hardware:
  - Physical protection: Locks, cameras, alarms
  - Protection in the cloud: Strong authentication
- OS: Access control for the files of the database
- Database:
  - Discretionary access control: Grant user access to objects
  - Role-based access control: Grant access to roles → Grant roles to users
- Network: Accept connections only from machines that should talk to the DB
- Application: Use different DB users for different accesses

Data Encryption: An Efficient Way to Protect Data

We have the same layers as with DB access control

At rest

Layer	Function	Protect against
Hardware / OS	Data is encrypted when read/write to disk	Stealing of disk/cloning virtual machines
Database	DB encrypts when read/write to file	Access by OS users/admins
Network	DB encrypts when read/write to network (e.g., TLS)	Hackers cannot sniff data in transit
Application	Application encrypts when read/write to the DB	Access by DB admins, memory dumps by OS admins

In motion

In use

Password Storage

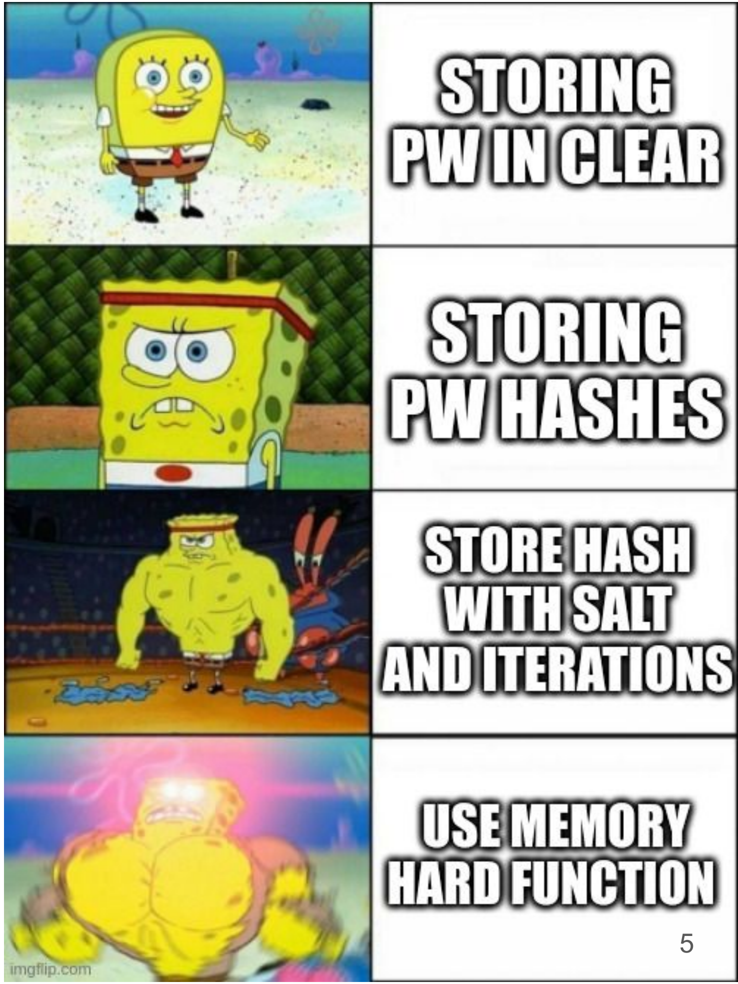
NEVER store passwords in cleartext

Always use **salt** to store passwords. Otherwise:

- Multiple hashes can be cracked at once
- Hashes can be calculated in advance
  → Rainbow tables can efficiently store this information

Salt is insufficient, must **slow down hash function**

- Iterations are a good start
- Memory hard functions are better



Demo: Password Cracking

Part 1: Brute Force Attack

Task: Brute force passwords

- Randomly generated
- Character set: Lowercase letters and digits (“`abcd...xyz0123...9`”)
- Length: 4 or 5 characters

Question

- How can you estimate the computational time required?
- How can you parallelize this attack?

Part 2: Dictionary Attack with Rules

Task: Crack passwords with the help of dictionaries and modification rules

- Base: A random word from a large dictionary
- Modification: Randomly apply some common user modifications, e.g.,
  - Capitalize the first letter and every letter which comes after a digit
  - Change ‘e’ to ‘3’
  - Change ‘o’ to ‘0’
  - Change ‘i’ to ‘1’

Question

- What do you observe compared to part 1?

Part 3: Dictionary Attack with Salt

Task: Crack salted passwords with the help of dictionaries

- Password: A random word from a large dictionary
- Salt: Provided, two random hexadecimal characters

Question

- Can you estimate the complexity required in this part if the salts were not provided?

Solution of Part 1: Brute Force Attack

```
# create a pool of processes
pool = multiprocessing.Pool(processes=multiprocessing.cpu_count())
total_matches = []
for length in range(4, 6):
    # create the set of passwords for this length
    combinations_generator = itertools.product(charset, repeat=length)

    # compute number of possible passwords
    total_combinations = len(charset)**length

    #read the generator lazily and map combinations to the function
    for i, x in enumerate(pool.imap_unordered(block_func, combinations_generator, 1000)):
        if x is not None:
            total_matches.append(x)
            if len(all_hashes) == len(total_matches):
                break
        if i % 100 == 0:
            # display your progress
            print("Progress for length {}: {:.3f}%".format(length,100*i/total_combinations), end="\r")
```

- Get cartesian product, equivalent to a nested for-loop
- Processing elements ***one at a time*** rather than bringing the whole password set into memory ***all at once***

- pool.imap_unordered: distribute the work to each process in the pool
- block_func: take one password, compute its hash, and return if a match is found

Solution of Part 2: Dictionary Attack with Rules

```python
def modif1(p: str):
    return p.title()
def modif2(p: str):
    return p.replace("e", "3")
def modif3(p: str):
    return p.replace("o", "0")
def modif4(p: str):
    return p.replace("i", "1")
```

```python
# generating all possible combinations of password modifications
all_modifs_combinations = set()
all_modifs = [modif1, modif2, modif3, modif4]
for length in range(1, len(all_modifs)+1):
    for comb in itertools.permutations(all_modifs, length):
        all_modifs_combinations.add(comb)

file = open("rockyou.txt", encoding="latin-1")
pool = multiprocessing.Pool(multiprocessing.cpu_count()) # define a pool of workers
results = []

# read the file by chunks of 10k rows at a time, then feed one
# rows one after the other to a worker
for r in pool.imap_unordered(modif_and_hash, file, 10000):
    if r is not None:
        print(r)
        results.append(r)

file.close()
```

All possible orderings
No repeated elements

Take a base password from the dictionary, try all modifications, hash them, and return if a match is found

Solution of Part 3: Dictionary Attack with Salt

```python
def salt_and_hash(p):
    """Take one password, and hash it using all the possible salts."""
    salts = ["b9", "be", "72"]
    p = p.replace("\n", "") # remove possible trailing \n
    for s in salts:
        salted = p+s
        hash = hashlib.sha256(salted.encode()).hexdigest()
        if hash in all_hashes:
            print("{} === {} (salt {})".format(hash, p, s))
            return p, hash

file = open("rockyou.txt", encoding="latin-1")
pool = multiprocessing.Pool(multiprocessing.cpu_count()) # define a pool of workers
results = []

# read the file by chunks of 10k rows at a time, then feed one
# rows one after the other to a worker
for r in pool.imap_unordered(salt_and_hash, file, 10000):
    if r is not None:
        print(r)
        results.append(r)

file.close()
```

Compute the hashes of the password + salt

Part 4: A CTF Challenge

Task: Crack a password hashed with multiple hash algorithms for 32 iterations

- Password: A random word from the “rockyou” dictionary
- Salt: Random characters
- Total length of the password and salt: 128 bytes

Part 4: A CTF Challenge

Task: Crack a password hashed with **multiple hash algorithms** for **32 iterations**

- Password: A random word from the "rockyou" dictionary
- **Salt: Random characters**
- **Total length of the password and salt: 128 bytes**

Infeasible to crack with brute force or rainbow tables

Hint: Observe the hash calculation algorithm

```
for i in range(0, 32):
```

$$salt_{i+1} = salt_i \oplus hash\_round_{[31-i]}(hash_i)$$
$$hash_{i+1} = hash_i \oplus hash\_round_{[i]}(salt_{i+1})$$

Solution of Part 4: A CTF Challenge

The hash calculation algorithm can be reversed!

```
for i in range(0, 32):
```

$$salt_{i+1} = salt_i \oplus hash\_round_{[31-i]}(hash_i)$$
$$hash_{i+1} = hash_i \oplus hash\_round_{[i]}(salt_{i+1})$$

```
for i in range(0, 32):
```

$$salt_i = salt_{i+1} \oplus hash\_round_{[31-i]}(hash_i)$$
$$hash_i = hash_{i+1} \oplus hash\_round_{[i]}(salt_{i+1})$$

when i = 31:

$$hash_{31} = hash_{32} \oplus hash\_round_{[31]}(salt_{32})$$
$$salt_{31} = salt_{32} \oplus hash\_round_{[0]}(hash_{31})$$

```
for i in range(31, -1, -1):
```

$$hash_i = hash_{i+1} \oplus hash\_round_{[i]}(salt_{i+1})$$
$$salt_i = salt_{i+1} \oplus hash\_round_{[31-i]}(hash_i)$$

Solution of Part 4: A CTF Challenge

hw04_password_cracking > demo > solution > ex4_sol.py > main

```python
def main():
    # reverse the hash calculation
    in_salt = password_hash[:64]
    in_hash = password_hash[64:]
    for pos, neg in zip(methods, methods[::-1]):
        '''
            interim_salt = xor(interim_salt, hash_rounds[-1-i](interim_hash))
            interim_hash = xor(interim_hash, hash_rounds[i](interim_salt))
        '''
        in_hash = xor(in_hash, eval("{}(in_salt)".format(neg)))
        in_salt = xor(in_salt, eval("{}(in_hash)".format(pos)))
    print(in_hash, in_salt)
```

Hash calculation algorithms should never be reversible

Good practices need to be combined carefully