

# **BASIC CRYPTOGRAPHY & HASH CRACKING**



# Prepared by

Kayla Putri Maharani

5026231158

PAI B

# OVER -----

# VIEW

Praktikum 4 berfokus pada pemahaman **dasar-dasar kriptografi modern** serta penerapan praktis dari beberapa teknik pengamanan data seperti **XOR Encryption, RSA, Diffie–Hellman Key Exchange, dan Hash Cracking**.

Tujuan utamanya adalah untuk memperlihatkan bagaimana algoritma dasar kriptografi bekerja, sekaligus menunjukkan potensi kelemahan bila penerapannya tidak aman.

# PRAKTIKUM 4 ...

# REPORT

## Table of Content

<b>Q1- XORXOR</b>	<b>3</b>
Analysis	3
Approach and Methodology	4
Complete Solver	6
Result	6
<b>Q2 - diffie-rsa</b>	<b>7</b>
Analysis	9
Approach and Methodology	10
Complete Solver	12
Results	12
<b>Q3 - Hash Cracker</b>	<b>13</b>
Analysis	13
Approach and Methodology	14
Complete Solver	17
Results	17
<b>Complete Code GitHub</b>	<b>17</b>

## Q1- XORXOR

Remember XOR property? The flag is XORed again with a secret 4-byte key (KEY5) for extra security.

```
k1=KEY1  
k21=KEY2 ^ KEY1  
k23=KEY2 ^ KEY3  
k1234=KEY4 ^ KEY1 ^ KEY3 ^ KEY2  
f45=FLAG ^ KEY4^KEY5
```

```
k1=3c3f0193af37d2ebbc50cc6b91d27cf61197  
k21=ff76edcad455b6881b92f726987cbf30c68c  
k23=611568312c102d4d921f26199d39fe973118  
k1234=91ec5a6fa8a12f908f161850c591459c3887  
f45=0269dd12fe3435ea63f63aef17f8362cdba8
```

### Analysis

#### 1. **KEY5 sangat pendek**

Kunci KEY5 hanya berukuran 4 byte, sehingga ruang kuncinya kecil. Ukuran ini membuat KEY5 mudah untuk ditebak atau di-brute-force. Jika kunci ini digunakan berulang, maka pola hasil XOR menjadi mudah ditebak dan bisa dieksplorasi menggunakan teknik known-plaintext.

#### 2. **Kemungkinan kunci diulang**

Panjang data sekitar 18 byte, jauh lebih panjang dari 4 byte KEY5. Biasanya, sistem seperti ini akan mengulang kunci pendek tersebut agar sesuai dengan panjang data. Pola pengulangan ini membuat struktur data menjadi tidak acak dan dapat dianalisis untuk memulihkan isi pesan.

#### 3. **XOR bersifat linier**

Operasi XOR memiliki sifat yang mudah dibalik. Dengan kombinasi nilai k1, k21, k23, dan k1234, kita bisa langsung menghitung KEY2, KEY3, dan KEY4 tanpa menebak nilai apa pun. Sifat ini menyebabkan sebagian besar kunci bisa dipulihkan secara pasti hanya dengan operasi dasar.

#### 4. **Tidak ada perlindungan integritas**

Skema ini hanya menggunakan XOR tanpa mekanisme autentikasi seperti MAC atau HMAC. Artinya, siapa pun bisa memodifikasi ciphertext dan menghasilkan perubahan pada plaintext tanpa ada cara untuk mendeteksinya. Ini membuat pesan mudah dimanipulasi.

#### 5. **Tidak ada elemen acak atau nonce**

Semua nilai tampak deterministik tanpa adanya inisialisasi acak (IV) atau nonce. Ketika sistem enkripsi tidak memiliki komponen acak, pola pada ciphertext menjadi konsisten, sehingga mempermudah analisis dan perbandingan antar pesan.

## 6. Format flag mudah ditebak

Format flag yang umum digunakan, seperti `cry{...}`, dapat dimanfaatkan sebagai petunjuk (crib). Dengan menebak beberapa byte awal flag, kita bisa menghitung bagian pertama dari KEY5. Jika kunci ini diulang, maka seluruh flag dapat dipulihkan dengan sangat cepat.

## Approach and Methodology

### Understanding the Problem

Soal ini adalah bentuk *XOR cryptography challenge*, di mana sebuah flag telah dienkripsi menggunakan kombinasi beberapa kunci berlapis (KEY1, KEY2, KEY3, KEY4, KEY5).

Kita diberi hubungan antar kunci, bukan kuncinya langsung. Dengan memanfaatkan sifat-sifat XOR berikut, kita dapat menurunkan semua kunci:

- Commutative:  $A \oplus B = B \oplus A$
- Associative:  $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
- Self-Inverse:  $A \oplus A = 0$
- Identity:  $A \oplus 0 = A$

Tujuan akhirnya adalah untuk mendapatkan FLAG asli dari ekspresi akhir  $\text{FLAG} \wedge \text{KEY4} \wedge \text{KEY5}$ .

### Step 1 – Converting the Given Hex Values

```
❸ xorSolver.py > ...
1   k1  = bytes.fromhex("3c3f0193af37d2ebbc50cc6b91d27cf61197")
2   k21 = bytes.fromhex("ff76edcad455b6881b92f726987cbf30c68c") # KEY2 ^ KEY1
3   k23 = bytes.fromhex("611568312c102d4d921f26199d39fe973118") # KEY2 ^ KEY3
4   k1234 = bytes.fromhex("91ec5a6fa8a12f908f161850c591459c3887") # KEY4 ^ KEY1 ^ KEY3 ^ KEY2
5   f45 = bytes.fromhex("0269dd12fe3435ea63f63aef17f8362cdba8") # FLAG ^ KEY4 ^ KEY5
```

Semua data (kunci dan hasil XOR) diberikan dalam format hexadecimal, sehingga perlu diubah menjadi bytes sebelum dioperasikan dengan XOR bitwise.

### Step 2 – Defining the XOR Function

```
7   bxor = lambda a,b: bytes(x ^ y for x,y in zip(a,b))
```

Fungsi ini melakukan XOR byte per byte antara dua input (a dan b), yang kemudian menghasilkan output dalam bentuk bytes lagi. Fungsi `zip()` memastikan operasi dilakukan sepanjang pasangan byte terpendek dari keduanya.

### Step 3 – Deriving the Hidden Keys (KEY2, KEY3, KEY4)

Menggunakan sifat *self-inverse* ( $A \oplus A = 0$ ), kita bisa menurunkan kunci berikut

```
9   k2 = bxor(k1, k21)           # KEY2 = (KEY2 ^ KEY1) ^ KEY1
10  k3 = bxor(k2, k23)           # KEY3 = (KEY2 ^ KEY3) ^ KEY2
11  k4 = bxor(k1234, bxor(k1, bxor(k3, k2))) # KEY4 = (KEY4^KKEY1^KKEY3^KKEY2) ^ KEY1 ^ KEY3 ^ KEY2
```

#### Penjelasan logika:

- Dari  $KEY2 \wedge KEY1$ , kita XOR kembali dengan  $KEY1$ , hasilnya  $KEY2$ .
- Dari  $KEY2 \wedge KEY3$ , XOR dengan  $KEY2$ , hasilnya  $KEY3$ .
- Untuk  $KEY4$ , karena bentuknya melibatkan empat kunci, kita XOR ulang dengan seluruh kombinasi yang menyertakan  $KEY1$ ,  $KEY2$ , dan  $KEY3$  untuk “menghapus” mereka.

### Step 4 – Simplifying the FLAG Relationship

Diketahui dari soal :

$$f45 = FLAG \wedge KEY4 \wedge KEY5$$

Dengan menggunakan sifat XOR, kita bisa mendapatkan :

$$FLAG \wedge KEY5 = f45 \wedge KEY4$$

Untuk memudahkan, hasil ini dinamai **k5**.

```
k5 = bxor(f45, k4)           # FLAG ^ KEY5 = f45 ^ KEY4
```

### Step 5 – Crib Method

Karena  $KEY5$  tidak diketahui, kita gunakan crib method, yaitu menebak bagian awal flag. Biasanya format flag diawali dengan "cry{" (atau "flag{"), sehingga bagian ini dijadikan acuan. Kemudian  $k5\_first$  diulang agar panjangnya sama dengan flag. Lalu gunakan kembali XOR untuk menemukan flag sebenarnya

```
14  # ---- crib method ----
15  prefix = b"cry{"
16  k5_first = bxor(k5[:len(prefix)], prefix)
17  k5_full = (k5_first * ((len(k5)//len(k5_first))+1))[:len(k5)]
18  flag = bxor(k5, k5_full)
```

### Step 6 – Output Decoding

Langkah terakhir adalah menampilkan hasil flag dalam bentuk teks (ASCII). Jika hasilnya berupa teks bermakna seperti cry{...}, berarti dekripsi berhasil. Jika tidak terbaca, kemungkinan prefix (crib) salah dan perlu diubah.

```
print(flag.decode(errors="ignore"))
```

## Complete Solver

```
● ● ●  
1 k1 = bytes.fromhex("3c3f0193af37d2ebbc50cc6b91d27cf61197")  
2 k21 = bytes.fromhex("ff76edcad455b6881b92f726987cbf30c68c") # KEY2 ^ KEY1  
3 k23 = bytes.fromhex("611568312c102d4d921f26199d39fe973118") # KEY2 ^ KEY3  
4 k1234 = bytes.fromhex("91ec5a6fa8a12f908f161850c591459c3887") # KEY4 ^ KEY1 ^ KEY3 ^ KEY2  
5 f45 = bytes.fromhex("0269dd12fe3435ea63f63aef17f8362cdba8") # FLAG ^ KEY4 ^ KEY5  
6  
7 bxor = lambda a,b: bytes(x ^ y for x,y in zip(a,b))  
8  
9 k2 = bxor(k1, k21) # KEY2 = (KEY2 ^ KEY1) ^ KEY1  
10 k3 = bxor(k2, k23) # KEY3 = (KEY2 ^ KEY3) ^ KEY2  
11 k4 = bxor(k1234, bxor(k1, bxor(k3, k2))) # KEY4 = (KEY4^KEY1^KEY3^KEY2) ^ KEY1 ^ KEY3 ^ KEY2  
12 k5 = bxor(f45, k4) # FLAG ^ KEY5 = f45 ^ KEY4  
13  
14 # ---- crib method ----  
15 prefix = b"cry{"  
16 k5_first = bxor(k5[:len(prefix)], prefix)  
17 k5_full = (k5_first * ((len(k5)//len(k5_first))+1))[:len(k5)]  
18 flag = bxor(k5, k5_full)  
19  
20 print(flag.decode(errors="ignore"))
```

## Result

```
PS C:\Users\kayla\Documents\Kuliah\Semester 5\PAI\Praktikum>  
ester 5/PAI/Praktikum/xorSolver.py"  
cry{xorxor_is_fun}
```

## Q2 - diffie-rsa

```
from Crypto.Util.number import getPrime, bytes_to_long, long_to_bytes
import gmpy2

p = getPrime(1024)
q = getPrime(1024)
p_dh = getPrime(2048)
g = getPrime(512)
a = getPrime(512)
b = getPrime(512)

def generate_public_int(g, a, p):
    return g ^ a % p

def generate_shared_secret(A, b, p):
    return A ^ b % p

n = p * q
e = 3
flag = SECRET
flag_int = bytes_to_long(flag)
A = generate_public_int(g,a,p_dh)
B = generate_public_int(g,b,p_dh)
shared_int = generate_shared_secret(A, b, p_dh)
flag2 = flag_int ^ shared_int
c = pow(flag2, e, n)

print(f"e = {e}")
print(f"n = {n}")
print(f"c = {c}")
print(f"p_dh = {p_dh}")
print(f"g = {g}")
print(f"A = {A}")
print(f"B = {B}")
```

e = 3	=
n =	
157596150421806490415966582876684144891890771122358601002119111525043130284968	
3276285731106548115937638289979087969524234208611870710633515079991373663766	
423874055118662541824187955220440397928304236310603796796313625606897016057	
5767446773599434863923445031457095796786743483187822292976477141742175160186	
94539007160608944430937034544250573079612716803553232903441662736871400867	
227470549648830376257722171953827226464889796516771624865677860133545288603	
94219682870901232117577347287566718833816940026413458546850105609903347261912	
726437507166375156175930347443033427979086283458174970895428510469964404661	
3927076737	
c =	
2344291097829188983579466594189698282383664262684774986896537805572522683	
1280550477483002476963942424904220271884548598076644573379786587930007779	
364593090093728561666710754310961567739554445059702634720756180532383222112	
7755063294497295212857461948115330332104551166898770688310681640876971305869	
0096813388100618801583589770113565361941160389675526436558849335309158887629	
2221883846233135618241090566288411446312195640148976020337278023363734087932	
74102135597864	
p_dh =	
2741819365577231802183656080185141605069290343260176193232259063127085639884	
665322333876983160164816070633827245576412868669464132928736305712470188483	
265595042921304475622984913358377816994866331000194017228529032491373620665	
699353230407187438926470693984281961019402022577585308848288070157267406196	
4002978113499344301009342814017383033244358886218297201525801832550215845980	
216778276302227039948212464281320312461226020648259134986770846870958550047	
6032516424555208566095641726437828019424079857347627497208896043827962564	
36877614065358000405891185557651690273462831344090087006816576227532248090	
921032709043449811	
g =	
120509552677757675375684702420645727618408160421138777389762735074984441998	
7188776740318017009728925151725056345074192389201064445866806510457212267706	
9707	
A =	
4454616487663596538806517973478271079131182042349146941797659062793031468845	
4925902502209726569648458488623582347798516866556354250339767805114246461	
36324	
B =	
4124156168261670138441671835404469967345593457632202994459118983978680881665	
9017319984353478142906481190837266137596857955780803840553445191890249081479	
76	

## Analysis

### 1. Salah pakai tanda XOR ( $\wedge$ ) di Python

Dalam kode ini, tanda  $\wedge$  dipakai untuk operasi yang seharusnya menghitung pangkat (exponentiation). Padahal di Python,  $\wedge$  bukan pangkat, tapi bitwise XOR (operasi logika antar-bit). Karena itu, rumus Diffie-Hellman yang seharusnya aman berubah jadi perhitungan biasa yang gampang ditebak. Akibatnya, nilai kunci rahasia bisa dihitung langsung tanpa perlu tahu kunci privat.

### 2. Nilai kunci rahasia (shared key) bisa ditebak dari data publik

Biasanya, Diffie-Hellman dibuat supaya hanya dua pihak tertentu yang tahu kunci rahasia. Tapi karena di sini semua pakai XOR, siapa pun yang melihat nilai publik ( $g, A, B$ ) bisa langsung menghitung nilai rahasia bersama itu. Jadi, sistem ini gagal menjaga kerahasiaan datanya.

### 3. RSA pakai nilai $e = 3$ tanpa padding sangat berbahaya

Nilai  $e = 3$  memang membuat enkripsi jadi cepat, tapi kalau tidak diberi padding, hasil enkripsi bisa dibalik dengan mudah. Kalau nilai hasil pangkat ( $\text{flag}^2 \wedge 3$ ) lebih kecil dari  $n$ , maka cukup ambil akar kubiknya untuk tahu isi aslinya. Itulah sebabnya attacker bisa langsung mendapatkan isi flag.

### 4. Langsung XOR antara flag dan kunci rahasia (tanpa pengamanan tambahan)

Di sini flag langsung di-XOR dengan shared key tanpa proses penguatan (seperti KDF atau hashing). Masalahnya, kalau shared key bisa ditebak (seperti di kasus ini), maka flag juga langsung bisa diketahui. Biasanya, shared key harus diolah dulu jadi kunci yang aman sebelum dipakai.

### 5. Tidak ada padding atau nonce, jadi hasil enkripsi mudah ditebak

Enkripsi di kode ini selalu menghasilkan hasil yang sama untuk data yang sama, karena tidak ada nilai acak (nonce). Akibatnya, kalau seseorang melihat hasil enkripsi beberapa kali, dia bisa menebak isinya. Sistem seperti ini juga tidak punya perlindungan terhadap perubahan data.

### 6. Pemilihan angka acak yang tidak sesuai standar

Kode membuat  $g$ ,  $a$ , dan  $b$  dari bilangan prima acak tanpa aturan tertentu. Dalam Diffie-Hellman yang benar, nilai-nilai itu harus berasal dari grup yang sudah diuji keamanannya. Kalau asal pilih, bisa saja nilainya punya pola yang membuat sistem lebih mudah diserang.

### 7. Kurang pemahaman tentang cara kerja operator Python

Kesalahan utama (pakai  $\wedge$  bukan  $\text{pow}$ ) bisa dihindari kalau pembuat kode tahu perbedaan fungsi di Python atau melakukan pengecekan. Dalam kriptografi, salah tanda sedikit saja bisa bikin seluruh sistem bocor. Karena itu, sebaiknya pakai library kriptografi resmi (misalnya `cryptography.hazmat`) daripada menulis algoritma sendiri.

## Approach and Methodology

### Understanding the Problem

Soal ini mencampur dua konsep kriptografi: Diffie-Hellman untuk membuat shared secret dan RSA untuk mengenkripsi flag. Namun, terdapat dua kesalahan besar:

1. Tanda  $\wedge$  di Python digunakan sebagai XOR, bukan pangkat.
2. RSA memakai nilai  $e = 3$  tanpa padding, yang rentan terhadap low exponent attack.

Dengan dua hal ini, attacker bisa menghitung nilai rahasia bersama dan kemudian menemukan flag tanpa harus mengetahui kunci privat

### Step 1 – Deriving the Shared Secret

Langkah pertama adalah menghitung shared key atau shared\_int. Karena operator  $\wedge$  adalah XOR, maka nilai rahasia dapat dihitung hanya dari nilai publik  $g$ ,  $A$ , dan  $B$ .

```
# 1) turunkan shared secret (eksplorasi XOR bug)
shared_int = g ^ (A ^ B)
```

Di Python operator  $\wedge$  = XOR, bukan pangkat. Soal mendefinisikan  $A = g \wedge a$  dan  $B = g \wedge b$  sehingga:

$$A \wedge B = (g \wedge a) \wedge (g \wedge b) = (g \wedge g) \wedge (a \wedge b) = 0 \wedge (a \wedge b) = a \wedge b.$$

Dengan itu, shared secret yang mereka gunakan di soal didefinisikan  $shared\_int = g \wedge (A \wedge B)$ . Karena  $A \wedge B = a \wedge b$ , maka  $shared\_int = g \wedge (a \wedge b)$  – nilai ini dapat dihitung hanya dari publik  $g$ ,  $A$ ,  $B$ .

### Step 2 – Recovering flag<sup>2</sup> (flag2) using Cube Root

RSA dipakai dengan  $e = 3$ . Jika  $flag2^{1/3} < n$  (atau secara praktis  $c$  adalah pangkat-3 sempurna), maka  $c = flag2^{1/3}$  dan kita bisa ambil akar kubik integer langsung untuk mendapatkan flag2. Kita menggunakan `gmpy2.iroot(c, 3)` karena aman dan efisien untuk bilangan besar, mengembalikan (`root, exact`) dimana `exact true` kalau `root^3 == c`.

Jika  $c$  bukan pangkat 3 sempurna secara langsung, kita masih mencoba `root` dan `root+1` dengan cek `pow(root, 3, n) == c` untuk menangani kemungkinan operasi modulo (kasus tertentu). Jika tidak ditemukan, kita hentikan karena asumsi soal tidak terpenuhi.

```
# 2) ambil integer cube root dari c => flag2
root, exact = gmpy2.iroot(c, 3)
if exact:
    flag2 = int(root)
elif pow(int(root), 3, n) == c:
    flag2 = int(root)
elif pow(int(root)+1, 3, n) == c:
    flag2 = int(root) + 1
else:
    raise SystemExit("gagal menemukan flag2 (c bukan pangkat-3 sempurna)")
```

### Step 3 - Recover original flag integer and convert to bytes

Dari soal kita tahu `flag2 = flag_int ^ shared_int` (karena `flag2` dibuat sebagai `flag_int ^ shared_int` sebelum dipangkatkan). Maka `flag_int = flag2 ^ shared_int` (XOR adalah self-inverse).

Setelah mendapat `flag_int` sebagai integer, kita konversi ke bytes dengan `long_to_bytes` dan decode ke string teks. Gunakan `errors="ignore"` atau akhiri dengan handling bila ada byte non-UTF8.

```
# 3) balik XOR untuk dapat flag, lalu cetak
flag = long_to_bytes(flag2 ^ shared_int)
print(flag.decode(errors="ignore"))
```

## Complete Solver

```
q2solver.py > ...
1  from Crypto.Util.number import long_to_bytes
2  import gmpy2
3
4  # nilai
5  n = 157596150421806490415966582876684144891890771122358601002119111152504313028496832762
6  c = 234429109782918898357946659418969828238366426268477498689653780557252268312805504774
7  g = 120509552677757675375684702420645727618408160421138777389762735074984441998718877674
8  A = 445461648766359653880651797347827107913118204234914694179765906279303146884549259025
9  B = 412415616826167013844167183540446996734559345763220299445911898397868088166590173199
10
11 # 1) turunkan shared secret (eksplorasi XOR bug)
12 shared_int = g ^ (A ^ B)
13
14 # 2) ambil integer cube root dari c => flag2
15 root, exact = gmpy2.iroot(c, 3)
16 if exact:
17     flag2 = int(root)
18 elif pow(int(root), 3, n) == c:
19     flag2 = int(root)
20 elif pow(int(root)+1, 3, n) == c:
21     flag2 = int(root) + 1
22 else:
23     raise SystemExit("gagal menemukan flag2 (c bukan pangkat-3 sempurna)")
24
25 # 3) balik XOR untuk dapat flag, lalu cetak
26 flag = long_to_bytes(flag2 ^ shared_int)
27 print(flag.decode(errors="ignore"))
28 |
```

## Results

```
PS C:\Users\kayla\Documents\Kul:kum\Praktikum 4\q2solver.py"
cry{aarrghh}
```

## Q3 – Hash Cracker

Imagine you've just come across some confidential data that has been hashed as shown below:

- 28cc09d8d8959871a97b24a07d87bcb05b9f3e7ac6d9f20ff82196ca5f908b2c
- 6c569aabbf7775ef8fc570e228c16b98

Find the original plaintexts for the hashes above. You are allowed to use any tools.

## Analysis

### 1. Lack of metadata

Soal cuma memberikan dua nilai hash mentah tidak ada info apa algoritma yang dipakai, tidak ada salt, tidak ada versi format.

### 2. Use of fast hash functions (practical weakness)

Bentuk hash (32 hex) cocok MD5; 64 hex bisa SHA-256/Keccak. MD5/SHA-256 adalah fungsi sangat cepat serta dapat membuat brute-force/dictionary (rockyou) sangat efektif

### 3. No rate limiting or server-side defenses (implicit)

Soal memberi hash mentah, kalau ini skenario nyata dan endpoint verifikasi ada, tanpa rate limit penyerang bisa coba online brute force.

## Approach and Methodology

### Download & prepare tools and wordlist

Siapkan Hashcat v7.1.2 ke folder Documents dan rockyou.txt tersedia di folder yang sama.

### Identifikasi tipe hash (trial / confirmation) untuk soal 3A

```
C:\Users\kayla\Documents\hashcat-7.1.2>hashcat.exe q3a.txt --show
The following 10 hash-modes match the structure of your input hash:

# | Name | Category
-----+-----+-----
34600 | MD6 (256) | Raw Hash
1400 | SHA2-256 | Raw Hash
17400 | SHA3-256 | Raw Hash
11700 | GOST R 34.11-2012 (Streebog) 256-bit, big-endian | Raw Hash
6900 | GOST R 34.11-94 | Raw Hash
17800 | Keccak-256 | Raw Hash
31100 | ShangMi 3 (SM3) | Raw Hash
1470 | sha256(utf16le($pass)) | Raw Hash
20800 | sha256(md5($pass)) | Raw Hash salted and/or iterated
21400 | sha256(sha256_bin($pass)) | Raw Hash salted and/or iterated
```

Melakukan percobaan mode dengan Hashcat (trial-and-error) untuk menentukan mode yang benar; hasilnya q3a cocok dengan mode **17800 (Keccak-256)**.

## Jalankan cracking untuk q3a menggunakan Keccak-256 (mode 17800)

```
C:\Users\kayla\Documents\hashcat-7.1.2\hashcat-7.1.2>hashcat.exe -m 17800 -a 0 q3a.txt rockyou.txt
hashcat (v7.1.2) starting

./OpenCL/m17800_a0-optimized.cl: Pure kernel not found, falling back to optimized kernel
OpenCL API (OpenCL 3.0 ) - Platform #1 [Intel(R) Corporation]
=====
* Device #01: Intel(R) Iris(R) Xe Graphics, 3553/7106 MB (1776 MB allocatable), 8MCU

./OpenCL/m17800_a0-optimized.cl: Pure kernel not found, falling back to optimized kernel
Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 31

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Optimizers applied:
* Optimized-Kernel
* Zero-Byte
* Not-Iterated
* Single-Hash
* Single-Salt
* Raw-Hash
* Uses-64-Bit

Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.

Host memory allocated for this attack: 991 MB (2622 MB free)

Dictionary cache built:
* Filename..: rockyou.txt
* Passwords.: 14344391
* Bytes.....: 139921497
* Keyspace..: 14344384
* Runtime...: 1 sec

28cc09d8d8959871a97b24a07d87bcb05b9f3e7ac6d9f20ff82196ca5f908b2c:Password

Session...........: hashcat
Status.........: Cracked
Hash.Mode.....: 17800 (Keccak-256)
Hash.Target....: 28cc09d8d8959871a97b24a07d87bcb05b9f3e7ac6d9f20ff82...908b2c
Time.Started....: Thu Oct 16 21:32:40 2025 (0 secs)
Time.Estimated...: Thu Oct 16 21:32:40 2025 (0 secs)
Kernel.Feature...: Optimized Kernel (password length 0-31 bytes)
Guess.Base.....: File (rockyou.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#01.....: 22171.1 kH/s (10.82ms) @ Accel:192 Loops:1 Thr:511 Vec:1
Recovered.....: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
```

-m 17800 memilih hash mode Keccak-256, -a 0 memilih attack mode straight (dictionary). Perintah ini menjalankan dictionary attack menggunakan rockyou.txt sebagai sumber kandidat password.

## Identifikasi tipe hash (trial / confirmation) untuk soal 3B

```
C:\Users\kayla\Documents\hashcat-7.1.2\hashcat-7.1.2>hashcat.exe q3b.txt --show
The following 12 hash-modes match the structure of your input hash:

# | Name | Category
=====+=====+=====
 900 | MD4 | Raw Hash
   0 | MD5 | Raw Hash
   70 | md5(utf16le($pass)) | Raw Hash
 2600 | md5(md5($pass)) | Raw Hash salted and/or iterated
 3500 | md5(md5(md5($pass))) | Raw Hash salted and/or iterated
 4400 | md5(sha1($pass)) | Raw Hash salted and/or iterated
20900 | md5(sha1($pass).md5($pass).sha1($pass)) | Raw Hash salted and/or iterated
32800 | md5(sha1(md5($pass))) | Raw Hash salted and/or iterated
 4300 | md5(strtoupper(md5($pass))) | Raw Hash salted and/or iterated
 1000 | NTLM | Operating System
 9900 | Radmin2 | Operating System
 8600 | Lotus Notes/Domino 5 | Enterprise Application Software (EAS)
```

Dari percobaan saya, q3b terdeteksi sebagai **mode 0 (MD5)**.

## Jalankan cracking untuk q3b menggunakan MD5 (mode 0)

```
C:\Users\kayla\Documents\hashcat-7.1.2\hashcat-7.1.2>hashcat.exe -m 0 -a 0 q3b.txt rockyou.txt
hashcat (v7.1.2) starting

OpenCL API (OpenCL 3.0 ) - Platform #1 [Intel(R) Corporation]
=====
* Device #01: Intel(R) Iris(Xe Graphics, 3553/7106 MB (1776 MB allocatable), 8MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Optimizers applied:
* Zero-Byte
* Early-Skip
* Not-Salted
* Not-Iterated
* Single-Hash
* Single-Salt
* Raw-Hash

ATTENTION! Pure (unoptimized) backend kernels selected.
Pure kernels can crack longer passwords, but drastically reduce performance.
If you want to switch to optimized kernels, append -O to your commandline.
See the above message to find out about the exact limits.

Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.

Host memory allocated for this attack: 652 MB (3435 MB free)

Dictionary cache hit:
* Filename...: rockyou.txt
* Passwords.: 14344384
* Bytes.....: 139921497
* Keyspace...: 14344384

6c569aabbf7775ef8fc570e228c16b98:password!

Session.....: hashcat
Status.....: Cracked
Hash.Mode....: 0 (MD5)
Hash.Target...: 6c569aabbf7775ef8fc570e228c16b98
Time.Started...: Thu Oct 16 21:34:56 2025 (1 sec)
Time.Estimated...: Thu Oct 16 21:34:57 2025 (0 secs)
Kernel.Feature...: Pure Kernel (password length 0-256 bytes)
Guess.Base.....: File (rockyou.txt)
Guess.Queue.....: 1/1 (100.00%)
```

## Complete Solver

**# check type for q3a**

```
C:\Users\kayla\Documents\hashcat-7.1.2\hashcat-7.1.2>hashcat.exe q3a.txt --show
```

**# crack q3a (Keccak-256)**

```
C:\Users\kayla\Documents>.\hashcat-7.1.2\hashcat-7.1.2\hashcat.exe -m 17800 -a 0  
q3a.txt rockyou.txt
```

**# check type for q3b**

```
C:\Users\kayla\Documents\hashcat-7.1.2\hashcat-7.1.2>hashcat.exe q3b.txt --show
```

**# crack q3b (MD5)**

```
C:\Users\kayla\Documents\hashcat-7.1.2\hashcat-7.1.2>hashcat.exe -m 0 -a 0 q3b.txt  
rockyou.txt
```

## Results

### Answer 3A

```
28cc09d8d8959871a97b24a07d87bcb05b9f3e7ac6d9f20ff82196ca5f908b2c:Password
```

### Answer 3B

```
6c569aabbf7775ef8fc570e228c16b98:password!
```

## Complete Code GitHub

<https://github.com/kaylapm/PAl>