🔐 Tutorial 6: Hashes and MACs

In this tutorial, we will complete several tasks related to **cryptographic hash functions** and **message authentication codes (MACs)**.

🔎 1) CRC and Its Suitability for Message Authentication

What is CRC?
Cyclic Redundancy Check (CRC) is a widely used error-detection technique in digital communication and storage. It treats the message as a polynomial over a finite field, divides it by a fixed generator polynomial, and appends the remainder to the message. The receiver performs the same division to verify data integrity.

This use of polynomial arithmetic may remind you of operations in **AES**, where multiplication in the finite field GF(2^8) is also defined using polynomials. In AES, however, the focus is on defining field multiplication rules for cryptographic transformations (e.g., MixColumns), not on detecting accidental errors. While both involve polynomials over finite fields, **CRC is linear and lacks cryptographic strength**, making it unsuitable for authentication purposes.

Tasks:

- ❓ **Is CRC a suitable function to be used in a MAC? Why or why not?**
  Consider how CRC works and whether it offers security against intentional manipulation.
- 🔍 **Research Task — MD5:**
  Why is the MD5 hash algorithm no longer considered secure?
  What essential property of cryptographic hash functions does it fail to uphold?
- 🔍 **Research Task — SHA-1:**
  Why is SHA-1 being phased out?
  Is it still safe to use today? What are the potential risks?

> **Solution:**
>
> **Is CRC a suitable function to be used in a MAC?**
> No, CRC (Cyclic Redundancy Check) is not suitable for use in a MAC. Although CRC is good at detecting accidental changes in data, it is not designed to be resistant to deliberate tampering. CRC is linear and predictable, meaning an attacker can easily manipulate the input and produce a valid CRC without knowing any secret key. Therefore, it lacks the collision resistance and unpredictability required for secure MACs.
>
> ---

> **Why is MD5 no longer used? What property of hash functions does it violate?**
> MD5 is no longer considered secure because it is vulnerable to **collision attacks**—different inputs can produce the same hash value. This violates the essential property of **collision resistance**, which is critical for cryptographic hash functions. These vulnerabilities make MD5 unsuitable for applications like digital signatures and message integrity.
>
> ---
>
> **Why is SHA-1 being phased out? Is it still safe to use?**
> SHA-1 is being phased out because researchers have demonstrated practical **collision attacks** against it. The famous "SHAttered" attack by Google and CWI Amsterdam in 2017 showed that it is feasible to generate two different inputs that produce the same SHA-1 hash. This undermines the integrity of systems relying on SHA-1.
>
> - https://shattered.io/
> - While it may still be used in legacy systems, it is considered **risky** and should be replaced with stronger algorithms like SHA-256 or SHA-3.

🔍 2) Understanding Hashing Methods

Next, we will perform some basic tasks to explore common hashing methods.

The most widely used hashing methods include:

- **MD5** (128-bit)
- **SHA-1** (160-bit)
- **SHA-256** (256-bit)
- **SHA-3** (256-bit)
- **bcrypt** (192-bit)
- **PBKDF2** (256-bit)

Recall from our previous tutorials that, some methods like **bcrypt**, **scrypt**, and **PBKDF2** incorporate a configurable number of **rounds or iterations**, which slows down the hashing process. This intentional slowdown significantly reduces the hashing rate, making brute-force or dictionary attacks more difficult—an important feature for securely storing passwords.

a) Matching MD5 Hashes

Match each MD5 hash signature to its corresponding word from the list: "Sydney", "Melbourne", "Brisbane", "Hobart", "Darwin", and "Perth".

i) a4bd01593487c956f68d360c18cb68b3

ii) 1fc91e86962825bb745de53d1657b3e4

iii) 759302fb84c5f72078746e7358ba44a0

iv) 4bc44fa25c7afaf63618e6586084052e

v) 9f610f074be03799cf1727a85127ab55

vi) 7c885b9c7c703a77befcabeea54944d5

In [1]:
```
from hashlib import md5

for i in {"Sydney", "Melbourne", "Brisbane", "Hobart", "Darwin","Perth"}:
    ### TODO: Print the string and its MD5 hash as a hexadecimal value.
    ### TODO: Remember to encode() the string before hashing, then use he
    print(i,md5(i.encode()).hexdigest())
```

```
Darwin a4bd01593487c956f68d360c18cb68b3
Melbourne 7c885b9c7c703a77befcabeea54944d5
Perth 4bc44fa25c7afaf63618e6586084052e
Brisbane 759302fb84c5f72078746e7358ba44a0
Sydney 1fc91e86962825bb745de53d1657b3e4
Hobart 9f610f074be03799cf1727a85127ab55
```

b) Generate MD5 Hashes Using OpenSSL

OpenSSL is a useful tool for analyzing hashes and certificates. For more information, see:
https://help.ubuntu.com/community/OpenSSL

- Use the `echo` command together with OpenSSL to generate MD5 hashes for the given strings.

In [2]:
```
!echo -n 'Sydney' | openssl md5
```

```
MD5(stdin)= 1fc91e86962825bb745de53d1657b3e4
```

In [3]:
```
!for var in Sydney Melbourne Brisbane Hobart Darwin Perth; do echo -n "$v
```

```
MD5(stdin)= 1fc91e86962825bb745de53d1657b3e4
MD5(stdin)= 7c885b9c7c703a77befcabeea54944d5
MD5(stdin)= 759302fb84c5f72078746e7358ba44a0
MD5(stdin)= 9f610f074be03799cf1727a85127ab55
MD5(stdin)= a4bd01593487c956f68d360c18cb68b3
MD5(stdin)= 4bc44fa25c7afaf63618e6586084052e
```

c) Check the Hash Lengths

- Determine the number of hexadecimal characters in the given hash signatures.
- How does the number of hex characters relate to the length (in bits) of the hash?
- Check the hash types: MD5, SHA-1, SHA-256, SHA-384, and SHA-512.

In [4]:
```
from hashlib import md5
from hashlib import sha1
from hashlib import sha256
from hashlib import sha384
from hashlib import sha512
```

```
plaintext = "Sydney"
print(plaintext,sha512(plaintext.encode()).hexdigest(), len(sha512(plaint
```

```
Sydney 0e0d9cdfe40a716fbef25e70be757c800bce3a82e082c842d58e066aa28c4f1b220
11e738d83f8dd5a75cde05ba18c499c0ec2bf2729dc4f2b2bf57a35333bf4 128
```

Solution:

- MD5 produces **32** hex characters, corresponding to **128 bits**
- SHA-1 produces **40** hex characters, corresponding to **160 bits**
- SHA-256 produces **64** hex characters, corresponding to **256 bits**
- SHA-384 produces **96** hex characters, corresponding to **384 bits**
- SHA-512 produces **128** hex characters, corresponding to **512 bits**

> One hexadecimal character represents **4 bits**.

d) Identifying File Modifications Using Hashes

You will find five files in the `Tutorial_6` folder: `a.txt` to `e.txt`.

The MD5 hashes of these files have been provided. Identify which file has been modified since the hashes were computed.

MD5(a.txt)= 47712f2371c49ed3f1b0086cf1ef16bb

MD5(b.txt)= eae6cb73251401f731306f79d7a6d7c6

MD5(c.txt)= 3dbaab35b18015be90c7cceb71a8b6a4

MD5(d.txt)= 418d6d2d7ccdfbf2bddffa7f91a1bcba

MD5(e.txt)= 3907aa1785f078dfd0b32b1e7d3ec78d

You can use either `openssl dgst -md5 <filename>` or `md5sum <filename>` to compute the MD5 hash of a file.

In [5]:
```
!openssl dgst -md5 a.txt
!openssl dgst -md5 b.txt
!openssl dgst -md5 c.txt
!openssl dgst -md5 d.txt
!openssl dgst -md5 e.txt
```

```
MD5(a.txt)= 47712f2371c49ed3f1b0086cf1ef16bb
MD5(b.txt)= eae6cb73251401f731306f79d7a6d7c6
MD5(c.txt)= 3dbaab35b18015be90c7cceb71a8b6a4
MD5(d.txt)= 3f6788e675355ea33244910e01181fe8
MD5(e.txt)= 3907aa1785f078dfd0b32b1e7d3ec78d
```

Solution:

File `d.txt` has been modified.

🔐 What is HMAC?

HMAC (Hash-based Message Authentication Code) is a cryptographic technique used to ensure **message integrity** and **authenticity**. It combines a cryptographic hash function (like SHA-256) with a secret key to produce a fixed-size MAC (Message Authentication Code). This MAC verifies that:

- The message has not been altered.
- The sender knows the shared secret key.

> ⚠️ HMAC does **not** provide confidentiality — it does not hide the message content.

How HMAC Differs from AE and AEAD

- **AE (Authenticated Encryption)** ensures both **confidentiality** and **authenticity** of a message.
  It encrypts the message and attaches a tag (like a MAC) to verify integrity.
  Examples: `AES-GCM`, `ChaCha20-Poly1305`.

- **AEAD (Authenticated Encryption with Associated Data)** extends AE by allowing parts of the data
  (called *associated data*) to remain **unencrypted** but still **authenticated**.
  For example, metadata like packet headers can be authenticated without being encrypted.

- **HMAC**, in contrast:
  - Does **not encrypt** the message.
  - Provides **authentication** and **integrity only**, not confidentiality.
  - Can be used **alongside encryption** (e.g., *encrypt-then-MAC*) to form secure protocols.

🆚 HMAC vs AE/AEAD

Feature	HMAC	AE (e.g., AES-GCM)	AEAD (e.g., ChaCha20-Poly1305)
Integrity	✅	✅	✅
Authenticity	✅	✅	✅
Confidentiality	❌	✅	✅
Associated Data	❌	❌	✅ (authenticated but not encrypted)

---

3) Blueprint of HMAC

Implement the HMAC scheme as a Python function `hmac(k, m)`.

- Use a block length of 256 bits (i.e., 32 bytes).
- A fixed value is given for the key `k`.
- Use SHA-2 (also called SHA-256) as the hash function.
- Use `0x5c` repeated 32 times as the outer padding (`opad`).
- Use `0x36` repeated 32 times as the inner padding (`ipad`).

Note: This is only a conceptual implementation to help you understand the ideas. **Never use such a homemade HMAC in production.** Finally, check if your result matches the HMAC generated by PyCryptodome.

In [6]:
```python
import random
from hashlib import sha256
from Crypto.Hash import HMAC as pycryptodome_hmac
from Crypto.Hash import SHA256 as pycryptodome_sha256

# The Block length for SHA256 is 512 bits, which is 64 bytes.
BLOCK_LENGTH_BYTES = 64

def hmac_custom(m, key):
    """
    Custom implementation of HMAC-SHA256.

    Args:
        m (str): The message to be hashed.
        key (str): The secret key.

    Returns:
        str: The hexadecimal representation of the HMAC value.
    """

    # 1. Convert key and message to bytes
    key_bytes = key.encode('utf-8')
    m_bytes = m.encode('utf-8')

    # 2. Key Handling:
    # If the key is longer than the block length, hash it first, so that
    if len(key_bytes) > BLOCK_LENGTH_BYTES:
        key_bytes = sha256(key_bytes).digest()

    # 3. Key Padding:
    # Pad the key with zeros to the block length.
    padded_key = key_bytes.ljust(BLOCK_LENGTH_BYTES, b'\x00')

    # 4. Define opad and ipad (as bytes)
    opad = b'\x5c' * BLOCK_LENGTH_BYTES
    ipad = b'\x36' * BLOCK_LENGTH_BYTES

    # 5. XOR the padded key with opad and ipad
    okeypad = bytes(a ^ b for a, b in zip(padded_key, opad))
    ikeypad = bytes(a ^ b for a, b in zip(padded_key, ipad))

    # 6. Apply the HMAC formula: H(okeypad || H(ikeypad || m))
    # Inner hash: H(ikeypad || m)
    inner_hash = sha256(ikeypad + m_bytes).digest()
```

```
    # Outer hash: H(okeypad || inner_hash)
    final_hmac = sha256(okeypad + inner_hash).hexdigest()

    return final_hmac

# --- Verification ---

# Sample data
message = 'Security engineering is cool'
secret_key = 'MySecretKey123'

# Calculate HMAC using the custom function
custom_hmac_value = hmac_custom(message, secret_key)
print(f"Custom HMAC:   {custom_hmac_value}")

# Calculate HMAC using pycryptodome for verification
pycryptodome_hmac_obj = pycryptodome_hmac.new(secret_key.encode('utf-8'),
pycryptodome_hmac_obj.update(message.encode('utf-8'))
pycryptodome_hmac_value = pycryptodome_hmac_obj.hexdigest()
print(f"Pycryptodome:  {pycryptodome_hmac_value}")

# Verify that the values match
if custom_hmac_value == pycryptodome_hmac_value:
    print("\nVerification successful: The custom HMAC matches the pycrypt
else:
    print("\nVerification failed: The custom HMAC does not match.")
```

```
Custom HMAC:   01367a2812c570f88cb9e94b21bbaf96fe1986d7c7bf528913a3eb10f5a
b4a45
Pycryptodome:  01367a2812c570f88cb9e94b21bbaf96fe1986d7c7bf528913a3eb10f5a
b4a45

Verification successful: The custom HMAC matches the pycryptodome HMAC.
```

🔐 4) AE and AEAD in Practice

As mentioned above, **AE (Authenticated Encryption)** and **AEAD (Authenticated Encryption with Associated Data)** ensure that **encryption** and **message authentication** happen **together**, rather than as separate steps.

This design became necessary because, in the past, developers often **misused or incorrectly combined** separate encryption and MAC functions, leading to **serious security flaws**. AEAD addresses this by integrating both operations into a single, well-defined primitive.

Today, **AEAD is considered the state-of-the-art** and is used in most modern communication protocols, including **TLS (HTTPS)**, **QUIC**, and **Signal**. It reduces the chance of implementation errors, improves performance, and guarantees that both the **confidentiality** and **integrity** of the message are protected in a single operation.

AEAD also supports *associated data* (like headers) that need to be authenticated but not encrypted—useful in protocols where some information must remain visible but still protected against tampering.

> 🔎 **Note:** AES-GCM is a widely used AEAD mode that combines AES in counter mode for encryption with a MAC called **GHASH**, which is based on finite field multiplication. Unlike traditional MACs such as HMAC or CMAC (more on this later), GHASH is optimized for performance and parallelizability, making AES-GCM especially efficient in high-speed networking.

🔐 Exercise: AES-GCM as AEAD with Integrity Verification

✅ Task: Complete the following Python code using **PyCryptodome** to:

- Encrypt a plaintext message using **AES-GCM**.
- Add *Additional Authenticated Data (AAD)* that will not be encrypted, but whose integrity is protected.
- Print the ciphertext and the authentication tag (MAC).
- Simulate tampering by modifying the ciphertext.
- Attempt decryption and ensure that integrity verification **fails** when the ciphertext is altered.

In [7]:
```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

# Step 1: Generate a key and nonce
key = get_random_bytes(16)  # AES-128
nonce = get_random_bytes(12)  # Recommended size for GCM

data = b'This is a secret message.'
aad = b'authenticated-data'

# Step 2: Encrypt using AES-GCM
cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
cipher.update(aad)  # Add AAD
ciphertext, tag = cipher.encrypt_and_digest(data)  # Encrypt and get tag

print("Ciphertext:", ciphertext.hex())
print("Tag:", tag.hex())

# Step 3: Modify the ciphertext (simulate tampering)
tampered_ciphertext = bytearray(ciphertext)
tampered_ciphertext[0] ^= 0x01  # Flip a bit

# Step 4: Try to decrypt and verify tag
try:
    cipher_dec = AES.new(key, AES.MODE_GCM, nonce=nonce)
    cipher_dec.update(aad)  # Add the same AAD
    plaintext = cipher_dec.decrypt_and_verify(tampered_ciphertext, tag)
    print("Plaintext:", plaintext)
except ValueError:
    print("Integrity check failed — ciphertext has been tampered with.")
```

```
Ciphertext: a37f805559510961355 3364e77d5b6750fb353e5f78278d684
Tag: b034bca99a3cd834d389191e0e8f9d5f
Integrity check failed — ciphertext has been tampered with.
```