Encryption and Integrity

0 Learning Goals

Goal	Commands / Topics
Create an RSA key-pair	openssl genpkey, openssl rsa -pubout
Do hybrid encryption	openssl enc -aes-256-cbc + openssl pkeyutl
Sign & verify	openssl dgst -sha256 -sign / -verify
Add integrity after CBC	openssl dgst -mac HMAC
Use helper scripts & environment variables	shell basics

1 Story Board

A partner sends you a directory named vault/:

- msg.enc — customer data encrypted with AES-256-CBC
- key.enc — the 256-bit AES key, itself encrypted with **your** RSA public key
- msg.sig — detached RSA signature over the plaintext
- README — contains the IV (Initialization Vector) you'll need

Your job:

- Generate (or import) an RSA-2048 key-pair.
- Decrypt the AES key.
- Decrypt the message.
- Verify the signature.
- Confirm integrity with an HMAC.

2 Setup the lab

0. Create the directory

mkdir vault

```
tiago-paquete@Linux:~$ mkdir vault
tiago-paquete@Linux:~$ ls -l | grep vault
==========================================================================
drwxrwxr-x 2 tiago-paquete tiago-paquete 4096 May  5 14:56 vault
==========================================================================
```

1. RSA key-pair

```
openssl genpkey \
    -algorithm RSA -pkeyopt rsa_keygen_bits:2048 \
    -out ~/.ssh/id_rsa
```

Component	Explanation
`openssl`	Command-line tool for using the OpenSSL cryptography library.
`genpkey`	OpenSSL subcommand to generate a private key.
`-algorithm RSA`	Specifies the algorithm to use for key generation, in this case, RSA.
`-pkeyopt rsa_keygen_bits:2048`	Option passed to `genpkey` to specify that the RSA key should be 2048 bits in length.
`-out ~/.ssh/id_rsa`	Output path for the generated private key file. The file is written to the standard SSH location for RSA private keys.

```
tiago-paquete@Linux:~$ openssl genpkey \
        -algorithm RSA -pkeyopt rsa_keygen_bits:2048 \
        -out ~/.ssh/id_rsa
==========================================================================
........+.+..+...+....+........+..................+......+.+..
+..........+.....+...+....++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++*....+.++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++*.......+......+........+......+.......+........+....
==========================================================================

tiago-paquete@Linux:~$ ls -l .ssh/id_rsa
==========================================================================
-rw------- 1 tiago-paquete tiago-paquete 1704 May  5 15:08 .ssh/id_rsa
==========================================================================
```

Visual Characters (e.g., +, ., *)

These characters are progress indicators printed by OpenSSL to show the internal steps during RSA key generation, especially while generating large prime numbers, which are essential for building secure RSA keys.

Character meanings:

- . – Indicates that OpenSSL is testing a number for primality.
- + – A prime number has been successfully found.
- * – A strong prime or a key component has passed an important validation check.

openssl rsa -pubout -in ~/.ssh/id_rsa -out ~/.ssh/id_rsa.pub

Component	Explanation
`openssl`	Command-line tool for cryptographic operations using OpenSSL.
`rsa`	OpenSSL subcommand to process RSA keys (e.g., convert formats, extract public key).
`-pubout`	Flag that tells `openssl rsa` to output the public key corresponding to the input private key.
`-in ~/.ssh/id_rsa`	Specifies the input file to read the private RSA key from.
`-out ~/.ssh/id_rsa.pub`	Specifies the output file where the public key will be saved. Commonly used as an SSH public key file.

```
tiago-paquete@Linux:~$ openssl rsa -pubout -in ~/.ssh/id_rsa -out
~/.ssh/id_rsa.pub
================================================================================
writing RSA key
================================================================================

tiago-paquete@Linux:~$ cat ~/.ssh/id_rsa
================================================================================
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQC5mEVvpyBCa4et
ZlqLHS0OiNHecqf/hf852T6g/G8kjDFc+h9RmdLqLezL4fgp5DqjEIh2hkwWBhT1
…
-----END PRIVATE KEY——
================================================================================

tiago-paquete@Linux:~$ cat ~/.ssh/id_rsa.pub
================================================================================
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAuZhFb6cgQmuHrWZaix0t
DojR3nKn/4X/Odk+oPxvJIwxXPofUZnS6i3sy+H4KeQ6oxCIdoZMFgYU9b3OQeaq
…
-----END PUBLIC KEY——
================================================================================
```

2. Create a secret message

echo 'Supply-chain security is everyone's job 📦🔐' > secret.txt

```
tiago-paquete@Linux:~$ echo 'Supply-chain security is everyone's job 📦🔐' > secret.txt

tiago-paquete@Linux:~$ cat secret.txt
================================================================================
Supply-chain security is everyone's job 📦🔐
================================================================================
```

3. Generate AES material

openssl rand -hex 32 > vault/aes.key # 256-bit key
openssl rand -hex 16 > vault/aes.iv # 128-bit IV (CBC wants 16 bytes)

Component	Explanation
`openssl`	A command-line tool for using the OpenSSL cryptography library, which provides various cryptographic operations.
`rand`	Subcommand to generate cryptographically strong pseudo-random bytes.
`-hex`	Option that tells `openssl rand` to output the generated random bytes in hexadecimal format (two hex characters per byte).
`32` (first command)	Number of bytes to generate. Since output is in hex, this will result in 64 hex characters (32 bytes × 2 hex chars). Used as a **256-bit key**.
`16` (second command)	Number of bytes to generate. Outputs 32 hex characters. Used as a **128-bit IV** (Initialization Vector), which is required for AES-CBC mode.
`>`	Shell redirection operator. Writes the command output into the specified file instead of printing it to stdout.
`vault/aes.key`	Path to the file where the random 256-bit AES key will be saved.
`vault/aes.iv`	Path to the file where the random 128-bit Initialization Vector will be saved.

```
tiago-paquete@Linux:~$ openssl rand -hex 32 > vault/aes.key
tiago-paquete@Linux:~$ openssl rand -hex 16 > vault/aes.iv
tiago-paquete@Linux:~$ cat vault/aes.key
================================================================================
4efecbd6155be1aad05f7db7610d67f56cc173c5534f383b60fa1083a0611feb
================================================================================
tiago-paquete@Linux:~$ cat vault/aes.iv
================================================================================
b6c14b92d27f1c8e93df1f54da1245c7
================================================================================
```

4. Encrypt the message with AES-256-CBC

```
KEY=$(cat vault/aes.key)
IV=$(cat vault/aes.iv)

openssl enc -aes-256-cbc -K "$KEY" -iv "$IV" -nosalt \
      -in secret.txt -out vault/msg.enc
```

Component	Explanation
`KEY=$(cat vault/aes.key)`	This assigns the contents of the file `vault/aes.key` to the shell variable `KEY`. The `cat` command reads the file, and `$(...)` is command substitution, capturing the output of the command inside it.
`IV=$(cat vault/aes.iv)`	Similarly, this assigns the contents of the file `vault/aes.iv` to the variable `IV`, using command substitution.
`openssl`	A powerful toolkit for SSL/TLS and cryptography, used here for file encryption.
`enc`	A subcommand of `openssl` used for symmetric cipher encryption and decryption.
`-aes-256-cbc`	Specifies the encryption cipher: AES (Advanced Encryption Standard) with 256-bit key size in CBC (Cipher Block Chaining) mode.
`-K "$KEY"`	Supplies the raw hexadecimal key (not a passphrase) to use for encryption. It uses the previously defined `KEY` variable.
`-iv "$IV"`	Supplies the initialization vector in raw hexadecimal, using the previously defined `IV` variable. Required in CBC mode to randomize encryption.
`-nosalt`	Tells OpenSSL not to use a salt when encrypting. By default, OpenSSL adds an 8-byte salt to help protect against dictionary attacks when using passphrases. Here, no salt is used because a raw key (`-K`) is provided.
`-in secret.txt`	Specifies the input file to encrypt (`secret.txt`).
`-out vault/msg.enc`	Specifies the output file (`vault/msg.enc`) that will contain the encrypted content.

```
tiago-paquete@Linux:~$ KEY=$(cat vault/aes.key)
tiago-paquete@Linux:~$ IV=$(cat vault/aes.iv)

tiago-paquete@Linux:~$ echo “$KEY"
================================================================================
4efecbd6155be1aad05f7db7610d67f56cc173c5534f383b60fa1083a0611feb
================================================================================

tiago-paquete@Linux:~$ echo “$IV"
================================================================================
b6c14b92d27f1c8e93df1f54da1245c7
================================================================================
```

```
tiago-paquete@Linux:~$ openssl enc -aes-256-cbc -K "$KEY" -iv "$IV"
-nosalt \
> -in secret.txt -out vault/msg.enc


tiago-paquete@Linux:~$ cat vault/msg.enc
==========================================================================
?*tiago-paquete@Linux:~$ ???Bw??q3??&
==========================================================================
```

Why cat vault/msg.enc Shows Gibberish

Reason 1: Encrypted file is binary
The openssl enc -aes-256-cbc command encrypts the plaintext into binary format, not into readable ASCII. This is expected behavior, as encryption produces seemingly random bytes.

Reason 2: Terminal can't display binary cleanly
When you use cat on a binary file, the terminal attempts to interpret raw bytes as characters. Since the binary content doesn't conform to readable text formats, you see gibberish or control characters.

5. Sign the *plaintext*

```
openssl dgst -sha256 -sign ~/.ssh/id_rsa \
      -out vault/msg.sig secret.txt
```

Component	Explanation
`openssl`	The command-line tool for using the OpenSSL library to perform various cryptographic operations such as hashing, encryption, signing, and certificate generation.
`dgst`	Short for "digest"; this subcommand tells OpenSSL to compute message digests (hashes) or to perform digital signing and verification.
`-sha256`	Specifies the digest algorithm to use. In this case, SHA-256 (Secure Hash Algorithm 256-bit). This is used to compute a secure hash of the input file.
`-sign ~/.ssh/id_rsa`	Signs the resulting hash using the specified private key. `~/.ssh/id_rsa` is the path to the RSA private key file.
`-out vault/msg.sig`	Specifies the output file where the digital signature will be saved. In this case, it saves the signature to the `msg.sig` file inside the `vault` directory.
`secret.txt`	The input file whose content will be hashed and then signed. This is the file being authenticated.

```
tiago-paquete@Linux:~$ openssl dgst -sha256 -sign ~/.ssh/id_rsa \
> -out vault/msg.sig secret.txt

tiago-paquete@Linux:~$ cat secret.txt
========================================================================
Supply-chain security is everyone's job 📦🔐
========================================================================

tiago-paquete@Linux:~$ cat vault/msg.sig
========================================================================
*?xS??eQœ?!??8??AZ?|?????V?W?:x
                                ???w?A?z?,?O}?(?|??m(j??ik:PR?E?~??σ1?
H9(3???/3M~t??<p?[,L=?t?<"z??l??      c>[???!%.?L?|?6?,X????4??AH??????
tiago-paquete@Linux:~$ ??9*ɜ??M?3T??????????z???
========================================================================
```

6. Wrap the AES key with your RSA *public* key

```
openssl pkeyutl -encrypt -pubin -inkey ~/.ssh/id_rsa.pub \
      -in vault/aes.key -out vault/key.enc
```

Component	Explanation
`openssl`	The command-line tool for using the OpenSSL cryptography library.
`pkeyutl`	A utility within OpenSSL for public key algorithm operations such as encryption, decryption, signing, and verification.
`-encrypt`	Specifies that the operation is to encrypt data (asymmetric encryption using a public key).
`-pubin`	Indicates that the input key provided with `-inkey` is a **public key**.
`-inkey ~/.ssh/id_rsa.pub`	Specifies the path to the **public key file** used to encrypt the input data. Here, it points to the default RSA public key typically used for SSH.
`-in vault/aes.key`	Specifies the **input file** to encrypt. In this context, it's the raw AES key stored in `vault/aes.key`.
`-out vault/key.enc`	Specifies the **output file** to store the encrypted result (i.e., the wrapped AES key).

```
tiago-paquete@Linux:~$ openssl pkeyutl -encrypt -pubin -inkey ~/.ssh/
id_rsa.pub \
        -in vault/aes.key -out vault/key.enc

tiago-paquete@Linux:~$ ls -l vault
========================================================================
total 20
-rw-rw-r-- 1 tiago-paquete tiago-paquete  33 May  5 15:33 aes.iv
-rw-rw-r-- 1 tiago-paquete tiago-paquete  65 May  5 15:33 aes.key
-rw-rw-r-- 1 tiago-paquete tiago-paquete 256 May  5 17:08 key.enc
-rw-rw-r-- 1 tiago-paquete tiago-paquete  64 May  5 15:42 msg.enc
-rw-rw-r-- 1 tiago-paquete tiago-paquete 256 May  5 15:50 msg.sig
========================================================================
```

7. Helper file for the recipient

cat <<EOF > vault/README
To recover the data:

*** Decrypt key.enc with your RSA private key.**
*** IV = $IV**
EOF

Component	Explanation
`cat`	Short for "concatenate", this command is used here to output a block of text. When used with a here-document (`<<`), it takes input until a specific delimiter is encountered (in this case `EOF`) and sends that input to standard output.
`<<EOF`	This is a **here-document** syntax. It tells the shell to read input until the word `EOF` is seen on a line by itself. The input between `<<EOF` and `EOF` is treated as if it were typed at the command line.
`>`	This is the redirection operator. It directs the output from the `cat` command into a file, **overwriting** it if it exists.
`vault/README`	This is the path and filename where the output from the here-document will be written. It creates (or overwrites) the file `README` inside the `vault/` directory.
`To recover the data:`	This is part of the content being written to the file. It is plain text and will appear in `vault/README`.
`* Decrypt key.enc with your RSA private key.`	A bullet point line being added to the README file as instructional text.
`* IV = $IV`	Another line of text. `$IV` is a shell variable, and its value will be expanded (replaced with its actual value) before writing to the file.
`EOF`	This is the delimiter that **ends** the here-document input. No more content is added after this line.

```
tiago-paquete@Linux:~$ cat <<EOL > vault/README
To recover the data:
* Decrypt key.enc with your RSA private key.
* IV = $IV
EOL

tiago-paquete@Linux:~$ cat vault/README
=======================================================================
To recover the data:

* Decrypt key.enc with your RSA private key.
* IV = b6c14b92d27f1c8e93df1f54da1245c7
=======================================================================
```

8. Destroy plaintext traces

shred -u secret.txt vault/aes.key

You now have **exactly** the artefacts you'll meet in the tasks.

`shred -u secret.txt vault/aes.key`

Component	Explanation
`shred`	A command used to securely delete files by overwriting their contents to make recovery difficult.
`-u`	Stands for "unlink" – tells `shred` to delete the file after overwriting it.
`secret.txt`	First file to be securely deleted.
`vault/aes.key`	Second file to be securely deleted. This is a relative path inside the `vault` directory.

```
tiago-paquete@Linux:~$ shred -u secret.txt vault/aes.key
```

3 Hands-On Tasks

Task 3.1 Check / create your key-pair

[-f ~/.ssh/id_rsa] || openssl genpkey -algorithm RSA -out ~/.ssh/id_rsa

```
tiago-paquete@Linux:~$ [ -f ~/.ssh/id_rsa ] || openssl genpkey
-algorithm RSA -out ~/.ssh/id_rsa
```

Component	Explanation
`[ -f ~/.ssh/id_rsa ]`	A **test command** to check if the file `~/.ssh/id_rsa` exists and is a regular file.
`-f`	Test operator that returns true if the file exists and is a **regular file** (not a directory or device).
`~/.ssh/id_rsa`	Path to the default **private SSH key** file in the user's home directory.
`openssl`	Command-line tool for using the **OpenSSL** cryptography library.
`genpkey`	OpenSSL subcommand to **generate a private key**.
`-algorithm RSA`	Specifies the **key algorithm** to use, in this case **RSA**.
`-out ~/.ssh/id_rsa`	Specifies the **output file** for the private key. Saves it as `~/.ssh/id_rsa`.

openssl rsa -pubout -in ~/.ssh/id_rsa -out ~/.ssh/id_rsa.pub

```
tiago-paquete@Linux:~$ openssl rsa -pubout -in ~/.ssh/id_rsa -out
~/.ssh/id_rsa.pub
writing RSA key
```

Component	Explanation
`openssl`	Command-line tool for working with OpenSSL.
`rsa`	Subcommand used to process **RSA keys**.
`-pubout`	Tells OpenSSL to output the **public key** part of the RSA key.
`-in ~/.ssh/id_rsa`	Specifies the **input file** containing the RSA **private key**.
`-out ~/.ssh/id_rsa.pub`	Specifies the **output file** where the **public key** will be saved.

Task 3.2 Recover the AES key (hybrid step)

```
cd ~/vault
openssl pkeyutl -decrypt -inkey ~/.ssh/id_rsa \
      -in key.enc -out aes.key
```

Component	Explanation
`openssl`	A command-line tool for using the OpenSSL cryptography library. It provides utilities for cryptographic operations such as encryption, decryption, certificate handling, etc.
`pkeyutl`	Stands for "Public Key Utility." A subcommand of `openssl` used to perform public key algorithm operations like encryption, decryption, signing, and verifying using private/public keys.
`-decrypt`	Specifies the operation mode. This flag tells `pkeyutl` to decrypt the input using the provided private key.
`-inkey ~/.ssh/id_rsa`	Specifies the private key file to use for the operation. `~/.ssh/id_rsa` is the default private RSA key used by SSH.
`-in key.enc`	Defines the input file containing the encrypted data (in this case, `key.enc`).
`-out aes.key`	Specifies the output file where the decrypted result will be saved (here, `aes.key`).

```
tiago-paquete@Linux:~$ cd vault

tiago-paquete@Linux:~/vault$ openssl pkeyutl -decrypt -inkey ~/.ssh/
id_rsa \
> -in key.enc -out aes.key

tiago-paquete@Linux:~/vault$ cat aes.key
================================================================================
4efecbd6155be1aad05f7db7610d67f56cc173c5534f383b60fa1083a0611feb
================================================================================
```

Task 3.3 Decrypt the message (AES-256-CBC)

```
IV=$(grep 'IV' README | awk '{print $4}')
KEY=$(cat aes.key)

openssl enc -d -aes-256-cbc -K "$KEY" -iv "$IV" -nosalt \
  -in msg.enc -out msg.txt

cat msg.txt
```

Before:

```
tiago-paquete@Linux:~$ cat vault/msg.enc
==========================================================================
?*tiago-paquete@Linux:~$ ???Bw??q3??&
==========================================================================

tiago-paquete@Linux:~$ cat vault/README
==========================================================================
To recover the data:

* Decrypt key.enc with your RSA private key.
*IV = b6c14b92d27f1c8e93df1f54da1245c7
==========================================================================
```

Component	Explanation
`IV=$(...)`	Assigns the output of the enclosed command to the variable `IV`. This is **command substitution**in Bash.
`grep 'IV' README`	Searches the file `README` for lines containing the string `"IV"`.
`awk '{print $3}'`	Extracts and prints the **third field** (column) from each line output by `grep`. Fields are delimited by whitespace by default.
`KEY=$(...)`	Assigns the output of the enclosed command to the variable `KEY`.
`openssl enc -d -aes-256-cbc`	Invokes OpenSSL to perform encryption/decryption. **`enc`**: Subcommand for symmetric encryption. **`-d`**: Decrypt mode. **`-aes-256-cbc`**: Use AES with 256-bit key in CBC (Cipher Block Chaining) mode.
`-K "$KEY"`	Supplies the **encryption key** in hexadecimal (without a leading `0x`). The key comes from the `KEY`variable, which was generated using `xxd`.
`-iv "$IV"`	Supplies the **initialization vector** (IV) in hexadecimal. Extracted from the `README` file.
`-nosalt`	Disables the use of a salt in encryption/decryption. This is important when the key and IV are manually supplied.
`-in msg.enc`	Specifies the **input file** (`msg.enc`) that contains the encrypted data.
`-out msg.txt`	Specifies the **output file** (`msg.txt`) where decrypted plaintext will be written.

After:

```
tiago-paquete@Linux:~/vault$ IV=$(grep 'IV' README | awk '{print $4}’)

tiago-paquete@Linux:~/vault$ KEY=$(cat aes.key)

tiago-paquete@Linux:~/vault$ echo “$IV"
==========================================================================
b6c14b92d27f1c8e93df1f54da1245c7
==========================================================================


tiago-paquete@Linux:~/vault$ echo “$KEY"
==========================================================================
3465666563626436313535626531616164303566376462373631306436376635366363316
373363335333346633383362363066613130383361303631316665620a
==========================================================================

tiago-paquete@Linux:~/vault$ openssl enc -d -aes-256-cbc -K "$KEY" -iv
"$IV" -nosalt \
  -in msg.enc -out msg.txt

tiago-paquete@Linux:~/vault$ cat msg.txt
==========================================================================
Supply-chain security is everyone’s job 📦🔐
==========================================================================
```

Task 3.4 Verify the detached signature

```
openssl dgst -sha256 -verify ~/.ssh/id_rsa.pub \
    -signature msg.sig msg.txt
```

Expected output: Verified OK

Component	Explanation
`openssl`	A versatile command-line tool used to perform various cryptographic operations such as encryption, decryption, key generation, and message digests.
`dgst`	Stands for **digest**. It invokes the message digest command in OpenSSL, used for hashing or signing/verification using digests.
`-sha256`	Specifies the digest algorithm to use, in this case, **SHA-256** (Secure Hash Algorithm 256-bit). It produces a 256-bit hash of the input data.
`-verify ~/.ssh/id_rsa.pub`	Verifies the digital signature using the specified **public key** file. This path points to a typical SSH public key, used here for demonstration.
`-signature msg.sig`	Specifies the **signature file** (`msg.sig`) that was generated during the signing process. This is the digital signature to be verified.
`msg.txt`	The **original message file**. This is the content whose hash will be recomputed and compared against the signed hash in the signature file.

```
tiago-paquete@Linux:~/vault$ openssl dgst -sha256 -verify ~/.ssh/
id_rsa.pub \
> -signature msg.sig msg.txt
==========================================================================
Verified OK
==========================================================================
```

Task 3.5 Compute & compare an HMAC (integrity after CBC)

`openssl rand -hex 64 > hmackey` # simulate a shared secret

Component	Explanation
`openssl`	A command-line tool for using the OpenSSL cryptographic library.
`rand`	Subcommand to generate pseudo-random bytes.
`-hex`	Outputs the random bytes in hexadecimal format.
`64`	Specifies the number of bytes to generate (64 bytes = 128 hex characters).
`>`	Shell redirection operator to write the output to a file.
`hmackey`	The output file where the generated hex key (shared secret) is stored.

```
tiago-paquete@Linux:~/vault$ openssl rand -hex 64 > hmackey

tiago-paquete@Linux:~/vault$ cat hmackey
================================================================================
6c0d27b401e8187cdfb6afddceda2945761ad588f5e57e9f46122c2225c83415696f999e
20553adb13223596e6fab614ee63e36eedc4a4e32c2e55b2260c16d7
================================================================================
```

`openssl dgst -sha256 -mac HMAC -macopt hexkey:$(cat hmackey) msg.txt`

Component	Explanation
`openssl`	OpenSSL command-line tool.
`dgst`	Subcommand used to compute message digests (cryptographic hashes).
`-sha256`	Specifies the hash algorithm to use; here, SHA-256.
`-mac HMAC`	Indicates that the digest should be computed using HMAC (keyed-hash message authentication code).
`-macopt hexkey:$(cat hmackey)`	Supplies the MAC key in hexadecimal format. The `$(cat hmackey)` substitutes the contents of the file.
`msg.txt`	The input file whose integrity is being checked with the HMAC.

```
tiago-paquete@Linux:~/vault$ openssl dgst -sha256 -mac HMAC -macopt
hexkey:$(cat hmackey) msg.txt
================================================================================
HMAC-SHA2-256(msg.txt)=
72b74475c86ea9a5457588e2c5d395ca5773f30ad95d7b4c303ead2f0c8674bc
================================================================================
```

Use Case Explanation

Step	What Happens	Purpose / Use Case
1. `openssl rand -hex 64 > hmackey`	Generates a 64-byte **hexadecimal secret key**.	**Simulates a shared secret key** between two parties for use in HMAC (used to protect integrity).
2. View of `hmackey`	Example output shows the secret key.	This key must remain secret and is used to authenticate messages.
3. `openssl dgst -sha256 -mac HMAC -macopt hexkey:$(cat hmackey) msg.txt`	Computes an **HMAC using SHA-256** on the file `msg.txt`using the secret key in `hmackey`.	Produces an **HMAC tag** which can later be used to **verify the integrity** of the file. If even a single bit in `msg.txt` changes, the HMAC will be different.

Real-World Use Cases of HMAC After CBC

Scenario	Purpose of HMAC
Encrypted Backups	After encrypting a file using AES-CBC, you compute an HMAC and store it alongside the ciphertext. Later, when decrypting, you recompute the HMAC and compare it to ensure the data wasn't altered.
Secure Messaging	Both parties share a secret key. Each message sent includes an HMAC computed over the plaintext or ciphertext. The receiver verifies the HMAC to confirm message integrity.
Software Distribution	When distributing sensitive files, even if encrypted, an HMAC ensures the file wasn't tampered with during download.
Digital Vaults / Encrypted Archives	After encrypting a file, an HMAC validates that the contents weren't modified when the file is decrypted later.