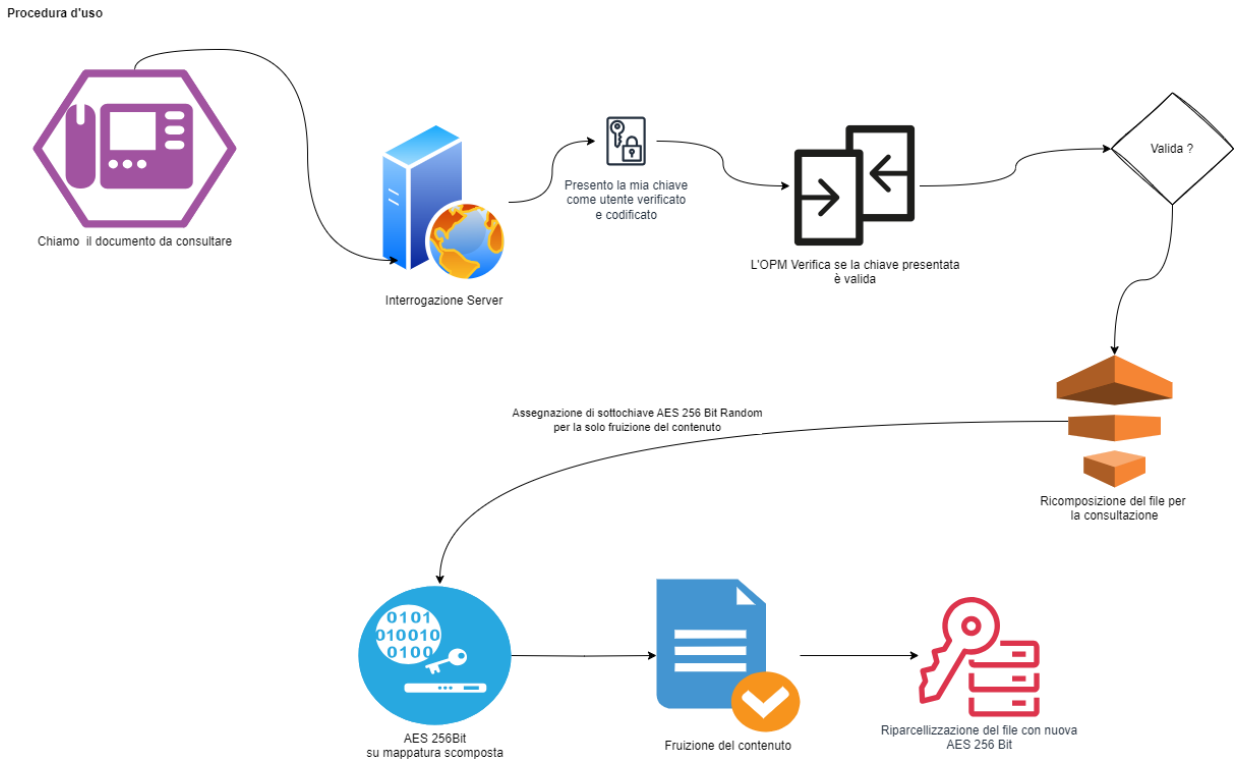
OPM CHIMERA

Protezione dei file da azioni di prelievo non autorizzato da parte di algoritmi LLM

MASSIMILIANO NICOLINI
WWW.OLIMAINT.TECH
Fondazione Olitec Caritate Christi Compulsi

Descrizione della procedura di protezione dell'opm chimera

Il dispositivo software agisce nel momento in cui viene effettuata la chiamata ad un documento presente su un server collegato alla rete, una volta che si effettua l'interrogazione del server l'utente deve presentare La sua chiave crittografica, crittografia che è scritta in as a 256 bit, la chiave Fra l'altro viene fornita all'inizio dell'accesso al sistema e quindi l'utente sostanzialmente effettua una vera e propria richiesta di consultazione del documento che è seguita dall'attribuzione della chiave di accesso di apertura del documento stesso, cosa questa Fra l'altro non possibile dall'eseguirsi da parte di un sistema llm che non è in grado di effettuare delle azioni di autenticazione tipiche del soggetto umano utilizzatore.



Olitec ®© Laboratorio di Ricerca e Sviluppo presso
Fondazione Olitec Caritate Christi - olimaint©® is a trade mark of Olimaint Company
Brescia Via XX Settembre 52 - Italy (EU)
Valmontone via Colle S.Angelo 2/O - Italy (EU)

www.olimaint.tech - desk@oliverso.it

☏ +39 030 364332 int 5 +39 345 563 0496

Va specificato che il documento protetto secondo questa modalità è un documento che deve essere autorizzato in maniera preventiva, l'utente deve sempre munirsi di una chiave di accesso automatica per l'apertura e la lettura dei documenti, in questo caso il protocollo dell'avatar biometrico avendo un identificativo univoco non clonabile Non duplicabile è il solo ed unico strumento che può permettere all'utente di andare a leggere un documento crittografato ottenere in automatico una chiave provvisoria di decrittografazione del documento e procedere nella sua consultazione.

Difatti l'opm software integrato in questa procedura effettua una verifica tra la chiave in possesso dell'avatar biometrico e quella che è la chiave di accesso al sistema ovvero alla possibilità di leggere il documento, il documento di suo viene archiviato in maniera scomposta ovvero una volta completato e pubblicato lo PM lo prende lo divide in tante parti e distribuisce queste parti in posizioni che solo ed unicamente lo stesso PM è in grado di andare a recuperare e ricostruire, una volta effettuata questa operazione l'utente ha la possibilità di leggere consultare e utilizzare normalmente il documento al termine dell'utilizzazione il documento ritorna nella posizione iniziale ovvero si divide ancora in piccole parti va a riposizionarsi nelle allocazioni che solo l'opm conosce e viene riprodotto con due nuove chiavi di crittografia a 256 bit.co

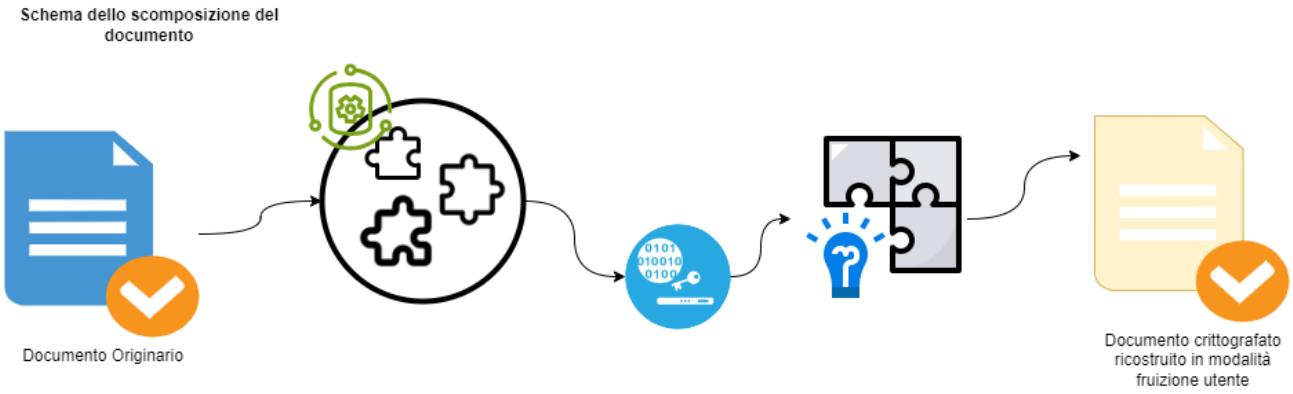
Parcellizzazione del file

Olitec ®© Laboratorio di Ricerca e Sviluppo presso
Fondazione Olitec Caritate Christi - olimaint©® is a trade mark of Olimaint Company
Brescia Via XX Settembre 52 - Italy (EU)
Valmontone via Colle S.Angelo 2/O - Italy (EU)

www.olimaint.tech - desk@oliverso.it

☎ +39 030 364332 int 5 +39 345 563 0496

il file subisce un processo di parcellizzazione, ovvero una volta completato ed integrato nella procedura di protezione dell'opm chimera viene suddiviso in tante parti e quindi non più raggruppato in un'unica parte e quindi facilmente individuabile e leggibile soprattutto, ma dicevamo viene suddiviso in tante piccole parti e queste piccole parti vengono sparpagliate sulla distribuzione della rete, all'interno l'opm Chimera conserva la mappa di posizionamento di tutte le parcelle distribuite del file, come un puzzle solo l'algoritmo o PM Chimera è in grado di ricomporre quel file ovvero di ricomporre quel puzzle.



una volta ricomposto il puzzle il documento è leggibile tranquillamente dall'utilizzatore e viene poi riportato nella condizione originale del momento in cui quel documento non viene utilizzato da nessuno, quindi viene riparcellizzato ridistribuito secondo nuove posizioni perché l'opm Chimera cambia ogni utilizzo del file e per ogni utilizzatore sia le chiavi di apertura e di chiusura dell'utilizzo del file ma soprattutto il posizionamento delle parcelle dello stesso e quindi diventa impossibile se per caso si vada ad identificare una parcellizzazione distribuita e riuscire a ricostruirla in un secondo momento perché sarà cambiata.

La lunghezza della chiave crittografica AES determina il numero totale di possibili chiavi uniche. Per AES a 256 bit, la chiave stessa è lunga 256 bit, che corrisponde a 2^{256} combinazioni possibili. Questo valore è estremamente grande e offre una resistenza molto elevata contro gli attacchi di forza bruta.

Olitec ®© Laboratorio di Ricerca e Sviluppo presso
Fondazione Olitec Caritate Christi - olimaint©® is a trade mark of Olimaint Company
Brescia Via XX Settembre 52 - Italy (EU)
Valmontone via Colle S.Angelo 2/O - Italy (EU)

www.olimaint.tech - desk@oliverso.it

☏ +39 030 364332 int 5 +39 345 563 0496

Per avere un'idea della vastità di 2^{256}, si può confrontare con il numero approssimativo di atomi nell'universo osservabile, che è stimato essere dell'ordine di 10^{80} La dimensione di 2^{256} supera enormemente il numero di atomi nell'universo, rendendo praticamente impossibile un attacco di forza bruta per esaminare tutte le possibili chiavi. La sicurezza di AES a 256 bit è considerata molto robusta e adatta per la maggior parte delle applicazioni crittografiche.

Violabilità

Attualmente, non esiste alcun metodo noto che possa violare direttamente una chiave AES a 256 bit attraverso attacchi crittanalitici praticabili. La sicurezza di AES (Advanced Encryption Standard) è basata sulla resistenza di un algoritmo a una serie di attacchi noti, tra cui l'attacco di forza bruta, l'analisi differenziale, l'analisi lineare, ecc.

La lunghezza della chiave AES a 256 bit offre una vasta quantità di possibili combinazioni 2^{256}, rendendo virtualmente impraticabile un attacco di forza bruta, anche con le risorse computazionali più avanzate conosciute oggi.

Va notato che la sicurezza di un sistema crittografico non dipende solo dall'algoritmo utilizzato, ma anche dall'implementazione e dalla gestione delle chiavi. Possibili debolezze possono sorgere da errori nella progettazione, implementazione o dall'uso negligente delle chiavi.

AES a 256 bit è considerato uno degli algoritmi di crittografia più sicuri e ampiamente utilizzati. Tuttavia, la sicurezza potrebbe essere compromessa in futuro con l'avanzare della tecnologia e l'emergere di nuovi approcci crittanalitici. Pertanto, è sempre consigliabile seguire le best practice di sicurezza, aggiornare regolarmente i sistemi e rimanere informati sulle ultime sviluppi in crittografia.

Olitec ®© Laboratorio di Ricerca e Sviluppo presso
Fondazione Olitec Caritate Christi - olimaint©® is a trade mark of Olimaint Company
Brescia Via XX Settembre 52 - Italy (EU)
Valmontone via Colle S.Angelo 2/O - Italy (EU)

www.olimaint.tech - desk@oliverso.it

☎ +39 030 364332 int 5 +39 345 563 0496

ESEMPI FUNZIONALI[2]

Algoritmo di esempio di integrazione di una chiave crittografica in un file[1]

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes
import os

def encrypt_file(input_file, output_file, key):
    with open(input_file, 'rb') as f:
        plaintext = f.read()

    cipher = Cipher(algorithms.AES(key), modes.CFB(os.urandom(16)), backend=default_backend())
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(plaintext) + encryptor.finalize()

    with open(output_file, 'wb') as f:
        f.write(ciphertext)

def decrypt_file(input_file, output_file, key):
    with open(input_file, 'rb') as f:
        ciphertext = f.read()

    cipher = Cipher(algorithms.AES(key), modes.CFB(os.urandom(16)), backend=default_backend())
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()

    with open(output_file, 'wb') as f:
        f.write(plaintext)

# Esempio di utilizzo:
key = os.urandom(32)  # 32 byte per AES 256-bit

encrypt_file('plaintext.txt', 'encrypted.txt', key)
decrypt_file('encrypted.txt', 'decrypted.txt', key)
```


Olitec ®© Laboratorio di Ricerca e Sviluppo presso
Fondazione Olitec Caritate Christi - olimaint©® is a trade mark of Olimaint Company
Brescia Via XX Settembre 52 - Italy (EU)
Valmontone via Colle S.Angelo 2/O - Italy (EU)

www.olimaint.tech - desk@oliverso.it

☎ +39 030 364332 int 5 +39 345 563 0496

ESEMPIO 2

```
import os

from cryptography.hazmat.primitives.ciphers import (
    Cipher, algorithms, modes
)

def encrypt(key, plaintext, associated_data):
    # Generate a random 96-bit IV.
    iv = os.urandom(12)

    # Construct an AES-GCM Cipher object with the given key and a
    # randomly generated IV.
    encryptor = Cipher(
        algorithms.AES(key),
        modes.GCM(iv),
    ).encryptor()

    # associated_data will be authenticated but not encrypted,
    # it must also be passed in on decryption.
    encryptor.authenticate_additional_data(associated_data)

    # Encrypt the plaintext and get the associated ciphertext.
    # GCM does not require padding.
    ciphertext = encryptor.update(plaintext) + encryptor.finalize()

    return (iv, ciphertext, encryptor.tag)

def decrypt(key, associated_data, iv, ciphertext, tag):
    # Construct a Cipher object, with the key, iv, and additionally the
    # GCM tag used for authenticating the message.
    decryptor = Cipher(
        algorithms.AES(key),
        modes.GCM(iv, tag),
    ).decryptor()

    # We put associated_data back in or the tag will fail to verify
    # when we finalize the decryptor.
    decryptor.authenticate_additional_data(associated_data)

    # Decryption gets us the authenticated plaintext.
    # If the tag does not match an InvalidTag exception will be raised.
    return decryptor.update(ciphertext) + decryptor.finalize()

iv, ciphertext, tag = encrypt(
    key,
    b"a secret message!",
    b"authenticated but not encrypted payload"
)

print(decrypt(
    key,
    b"authenticated but not encrypted payload",
    iv,
    ciphertext,
    tag))
```


Olitec ®© Laboratorio di Ricerca e Sviluppo presso
Fondazione Olitec Caritate Christi - olimaint©® is a trade mark of Olimaint Company
Brescia Via XX Settembre 52 - Italy (EU)
Valmontone via Colle S.Angelo 2/O - Italy (EU)

www.olimaint.tech - desk@oliverso.it

☎ +39 030 364332 int 5 +39 345 563 0496

Chiave crittografica a 256 bit[3]

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
import os

def encrypt_AES_256(plaintext, key):
    # Genera un vettore di inizializzazione (IV) casuale
    iv = os.urandom(16)

    # Crea l'oggetto Cipher con l'algoritmo AES, la chiave e la modalità di operazione CBC
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default_backend())

    # Crea un oggetto encryptor
    encryptor = cipher.encryptor()

    # Cifra il testo in chiaro
    ciphertext = encryptor.update(plaintext) + encryptor.finalize()

    # Restituisce il vettore di inizializzazione e il testo cifrato
    return iv + ciphertext

def decrypt_AES_256(ciphertext, key):
    # Estrae il vettore di inizializzazione (IV) dal ciphertext
    iv = ciphertext[:16]
    ciphertext = ciphertext[16:]

    # Crea l'oggetto Cipher con l'algoritmo AES, la chiave e la modalità di operazione CBC
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default_backend())

    # Crea un oggetto decryptor
    decryptor = cipher.decryptor()

    # Decifra il ciphertext
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()
```


Olitec ®© Laboratorio di Ricerca e Sviluppo presso
Fondazione Olitec Caritate Christi - olimaint©® is a trade mark of Olimaint Company
Brescia Via XX Settembre 52 - Italy (EU)
Valmontone via Colle S.Angelo 2/O - Italy (EU)

www.olimaint.tech - desk@oliverso.it

☏ +39 030 364332 int 5 +39 345 563 0496

```
    # Restituisce il testo in chiaro
    return plaintext

# Esempio di utilizzo:
key = os.urandom(32)  # 32 byte per AES 256-bit
plaintext = b'Hello, world! This is a test message.'

# Crittografia
ciphertext = encrypt_AES_256(plaintext, key)
print("Ciphertext:", ciphertext)

# Decrittografia
decrypted_text = decrypt_AES_256(ciphertext, key)
print("Decrypted Text:", decrypted_text.decode('utf-8'))
```

(1)(2)(3) Strutture di esempio indicative che non rappresentano necessariamente la scrittura completa dell'algoritmo OPM Chimera