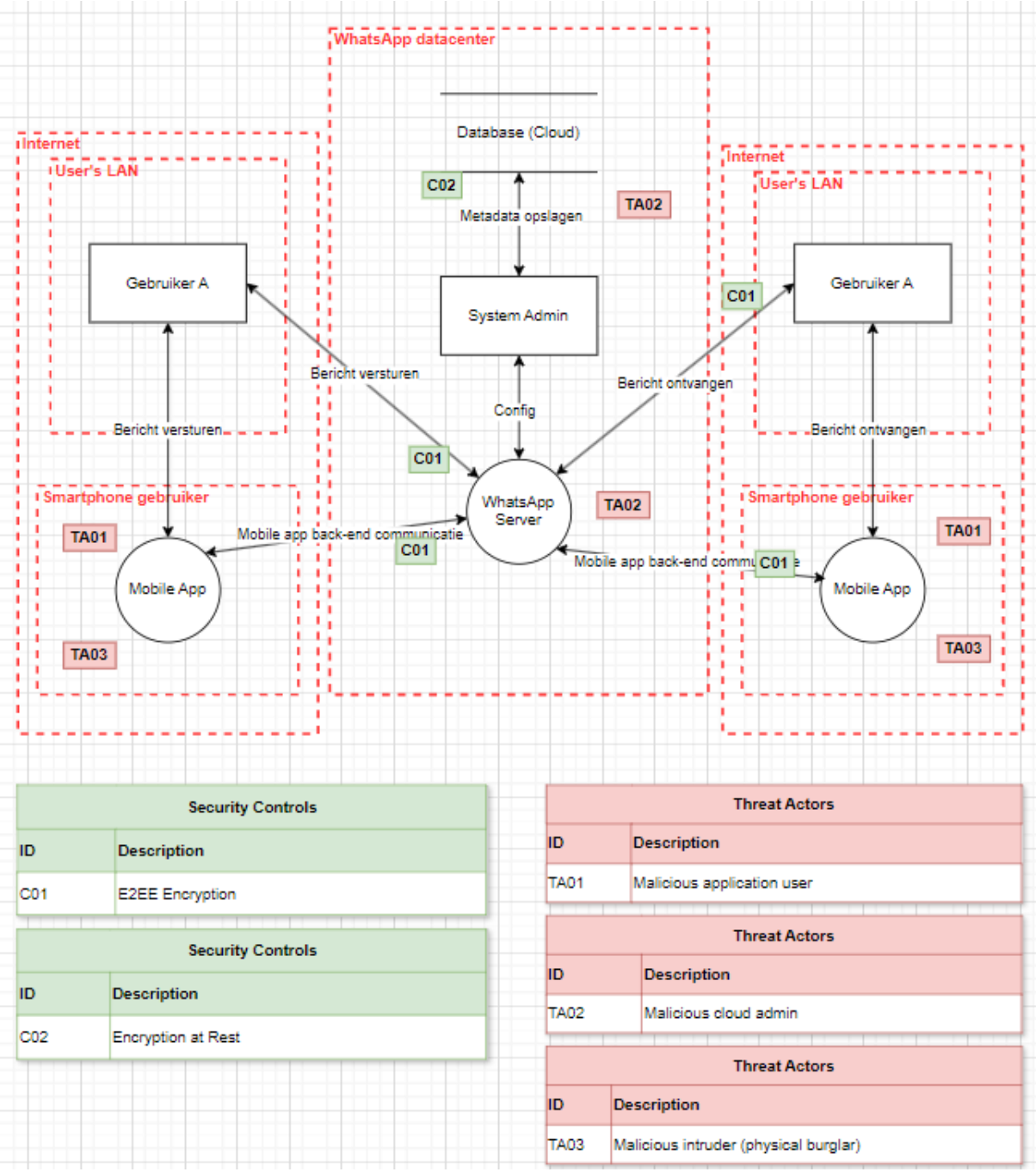
Verslag opdracht 2

Opdracht 2.1



Security Controls	
ID	Description
C01	E2EE Encryption

Security Controls	
ID	Description
C02	Encryption at Rest

Threat Actors	
ID	Description
TA01	Malicious application user

Threat Actors	
ID	Description
TA02	Malicious cloud admin

Threat Actors	
ID	Description
TA03	Malicious intruder (physical burglar)

Het threat model diagram toont de communicatie tussen gebruikers van WhatsApp en hoe verschillende threat de beveiliging van de app kunnen beïnvloeden.

Gebruikers A en B: ze sturen berichten naar elkaar via de mobiele app en de WhatsApp-server. End-to-end encryptie (C01) zorgt ervoor dat alleen de verzender en ontvanger de inhoud van de berichten kunnen lezen, zelfs WhatsApp zelf heeft geen toegang tot de inhoud.

WhatsApp Server: de server functioneert als tussenpersoon voor het verzenden van berichten. Het slaat geen berichten op, maar beheert configuratie- en communicatieprocessen en slaat metadata (informatie over berichten zoals timestamps) op in een cloud-database.

Cloud Database: deze slaat metadata op (geen berichteninhoud) en is beveiligd met encryptie bij Rest (C02). Dit betekent dat de metadata versleuteld is wanneer het niet actief wordt gebruikt, waardoor het beschermd is tegen onbevoegde toegang.

Dreigingsactoren:

- TA01 (Malicious application user): deze persoon kan proberen via de app zelf toegang te krijgen tot berichten of kwetsbaarheden in de app uit te buiten.
- TA02 (Malicious cloud admin): een admin van de cloudinfrastructuur zou potentieel toegang kunnen hebben tot opgeslagen metadata, hoewel de gegevens versleuteld zijn.
- TA03 (Malicious intruder): dit is een fysieke threat waarbij een indringer probeert toegang te krijgen tot een apparaat (zoals een gestolen smartphone) om berichten in te zien.
- Beveiligingsmaatregelen: end-to-end encryptie (C01) beschermt de inhoud van berichten, zelfs als een dreigingsactor toegang krijgt tot de server of app. Encryptie bij Rest (C02) beschermt metadata wanneer deze is opgeslagen in de cloud.

Zou je op basis van vorig threat model encryption at rest aanraden? **Ja, het beschermt metadata die in de cloud worden opgeslagen, wat essentieel is omdat deze informatie gevoelige gegevens kan bevatten over het gebruik van de applicatie, zoals timestamps en afzender/ontvanger-informatie. Zonder encryptie zouden threat actors, zoals een malicious cloud admin (TA02), gemakkelijk toegang kunnen krijgen tot deze metadata en deze kunnen misbruiken.**

Zou je op basis van vorig threat model homomorphic encryption aanraden? **Homomorfe encryptie is theoretisch goed, maar ik zou het in dit specifieke threat model misschien niet direct gebruiken. Het biedt de mogelijkheid om berekeningen uit te voeren op versleutelde gegevens zonder ze te ontsleutelen, wat betekent dat een systeembeheerder (TA02) niet direct toegang heeft tot de inhoud van de berichten. Echter, vanwege de complexiteit en de prestaties van homomorfe encryptie, kan het moeilijk zijn om dit in de praktijk te implementeren. E2EE (C01) biedt al een sterke bescherming voor de inhoud van berichten, wat in dit geval meer effectief is.**

Whatsapp en Signal gebruiken end-to-end encryption, wat betekent dat? Leg uit op basis van je threat model. **End-to-end encryptie (E2EE) betekent dat alleen de verzender en de ontvanger van een bericht de inhoud kunnen lezen; zelfs de serviceprovider (in dit geval WhatsApp) heeft geen toegang tot de berichten. In het threat model is dit zichtbaar, omdat het voorkomt dat threat actors de inhoud van de communicatie kunnen onderscheppen of lezen. Dit biedt een sterke beveiliging tegen afluisteren en garandeert dat, zelfs als de server of het netwerk wordt gecompromitteerd, de inhoud van de berichten veilig blijft.**

Opdracht 2.2

```python
import tenseal as ts

context = ts.context(
    ts.SCHEME_TYPE.BFV,
    poly_modulus_degree=8192,
    plain_modulus=1032193
)

context.global_scale = 2**40
context.generate_galois_keys()

encrypted_75 = ts.bfv_vector(context, [75])
encrypted_326 = ts.bfv_vector(context, [326])

encrypted_result = encrypted_75 + encrypted_326

print("\nVersleutelde representatie van de som van 75 en 326:")
print(encrypted_result.serialize())

decrypted_result = encrypted_result.decrypt()

print(f"\nDe ontsleutelde som van 75 en 326 is: {decrypted_result[0]}")
```

Deze code gebruikt de TenSEAL-bibliotheek voor homomorfe encryptie met het BFV-schematiek. Het maakt een encryptiecontext aan en versleutelt de waarden 75 en 326. Vervolgens worden deze versleutelde waarden opgeteld en de versleutelde som wordt afgedrukt. Ten slotte wordt de som ontsleuteld en het resultaat wordt weergegeven.

```
3\xa3gs\xe4\xa4\xc1\xf2m\x00\\0\x9e\xdc}\xba\xaa\x97\x93*L\xfe3o"\x045\xfa\xe6\xe0\\\x00\x10,\xbeY\x010\x12\x9a\xf7]+7\xc2\x08\xd2{W1\x16\x0f_O\xfe\xf4\x83\xc0
oE\xdf+oY\xc9\xb2E\xf0_#\x84\xb4\xe8H\xf2\'\x06\x0b\x8b\xea\x90|:\xc8\x83\x18\x045\xdf\xe2R\xf1\'\\\xdf\x94\x14\x87\x85\x99!\xaf\xc4q\x02@\xde\x97_\xeb\x02\xb9
<\xc6\xf85\x80\xa2Z?\xc6\x1eM\xdbR\xe4\x11\xd9\xaf\x9a\xdaXI\xde=\x12\x86(\x0b%\x16\xfci+\xc5\x8d/!\x1e\xe5\x93\xcaT\xa0\xdb\xab\x90XA\xb0O\xbf\x06\xa0q\x01\xc
5\xfaL\nl\nJ\r~\xb3\x81b\xd8z\xe1\x95,PaI$\xf6\xcb\xf3T,!f\xbf\x04P\x08\x12\x9c\x1a\x8f\xa6z4b\xd7\xa5_\x03\x82*s2=K\'\xcc\x18\x9d\xa4G&1\xfd\xad\xb7\xfe5=\x13
KD\x8f&\x06\xc8\xe3\x91\xea\x97\x11\x0eY\x88\x80\xe1\xd1\x11d\xfc\xd2\xc0\xf9\xe6\x9a2s\xa9.\xfdrF\x81\x1e\x80<\x1e\xed\xfb5\xab\x8d\xda7\x01\xed\xe0\xe8A\xebW
^\x80^L\xad~US\xe6-\x07\x84gs\xa1\xf8\xd5\x84\xd1\xa3JHd\xf1\xc0\xe80$\x82)\xc2\xccz\xd6\x89\x85!\x8cZ\xfd\xc9\x05N\x9d\xae\xc7o\x12\xb2\x96&u\xf9\xf4G\x9a\x9b
O\xab\x1eY\x07B\x166?\x8f\\\xa5J\x14\x15)\x9f\xfa\x8a#F\xc7\xf7J\x1d\x8d\x0c\xca,|\xab\xa6U_\x1e\xe4\xafJ\\$\xf0l\xf9\xb4\x80\xda\x0f[C\xfeL+dE!\xc5\xb7hk\x0bd
\xe1}\xcai\xc0\x12D\x86\x7f]\xd4H#(\xd6\xbb\x83XXd\x12\xfc\xb4Y\xe5Q\xaa5\xef6\x98t\xae1y%%0\xa0\xb2 y\xa5\xab\n\x9d\xfa\x80|z\xb7\xca\r\x8b\x9aW\xca\x1d\x893#
\xe2\x95\xb2\x1e\xfd\xc8T\xe0\xd16%j\x15Z\xf0h\x85/O\x98L\xbe]\x95X@\x08\xa3?O\x8d\xb1A\xb2\xf5\'\xa2-\x00\xa2Tx\x95\x94\x18*\x15$}\xd2\xd2\xe0\xaa\x0f^\x1eu\x
83"\x02\x00B\xb7LA\x92z{\xc1\xb0\x0b\xa9\x01\xad\x02\x00\x13F\x14\x8f\x94\xf3e\x1d\x90\xfe\x8drk\xcfA\x99g]\x80\xa1\xd0\x85\xe7\xdf\x15\x17R\xd9i\xf5\xe7\x830\
x7f\xb9\xf8<2\xacG\x9f&\x8b_7\x11\xf0\xe0\xe3\xc9/]A\xbf\x9e\xd4\xf8y\xfd\x98r\xc2\xcc7\xcb<i\xf0\x04\xd6+\x1f\x9fdA \xe2[\xbc\x16\x8d\xe8\x18\x1e\x00'

De ontsleutelde som van 75 en 326 is: 401
```

Hier zie je een klein deel van de versleutelt som en daaronder de ontsleutelde som.

Opdracht 2.3

```python
import binascii
from cryptography.fernet import Fernet
from secretsharing import SecretSharer
import requests

sentence = "My name is Adrian."
key = Fernet.generate_key()
cipher_suite = Fernet(key)
ciphertext = cipher_suite.encrypt(sentence.encode())

hex_key = binascii.hexlify(key).decode()

shares = SecretSharer.split_secret(hex_key, 3, 5)

share_to_paste = shares[0]
paste_content = f"Ciphertext: {ciphertext.decode()}\nShare: {share_to_paste}"

url = "https://api.paste.ee/v1/pastes"
headers = {
    "Content-Type": "application/json",
    "X-Auth-Token": "aDMZT7E9A9BgNtTRulDvDEG5YsgjvFklj8DzuC2os"
}
```

```python
data = {
    "description": "Shamir Secret Sharing Example",
    "sections": [{"name": "Secret", "syntax": "text", "contents": paste_content}],
    "expire": "6m"
}
response = requests.post(url, json=data, headers=headers)

if response.status_code == 201:
    paste_url = response.json().get("link")
    print(f"Paste URL: {paste_url}")
else:
    print(f"Failed to upload to paste.ee: {response.text}")

print(f"Share verslag: {shares[1]}")
```

Voor deze opdracht heb ik gebruik gemaakt van de cryptography en secretsharing bibliotheken om een zin te versleutelen en de nodige encryptiesleutel op te splitsen. Ten eerste genereert het een Fernet-sleutel en versleutelt de zin. Vervolgens wordt de sleutel omgezet naar een hexadecimale string en opgedeeld in 5 delen, waarvan er 3 nodig zijn om de sleutel te herstellen. De versleutelde zin en een van de shares worden vervolgens geüpload naar de paste.ee API, en de link naar de geüploade inhoud wordt weergegeven. Ten slotte print de code een tweede share voor om in dit verslag toe te voegen. Ik heb ook geprobeerd om de Shamir of secretsharing bibliotheek te gebruiken, maar dat lukte niet. Om deze opdracht uit te voeren heb je namelijk een methode nodig om je secret in verschillende delen te splitsen. Helaas laten deze bibliotheken het niet toe om de secrets te splitsen, waardoor het niet mogelijk was om dit te realiseren.

Paste URL: https://paste.ee/p/toU78

Share verslag: 2-a55370c73425453f10aa3a2322dabf29581497fca846a9ed4b566fc4ecf52339f6556c48135a5bf6a00c09f8080cd697

Opdracht 2.4

```python
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import os

def derive_key(secret: bytes, salt: bytes) -> bytes:
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA512(),
        length=32,
        salt=salt,
        iterations=100000,
    )
    return kdf.derive(secret)

secret = b'my_shared_secret_passphrase'
salt = os.urandom(16)

key = derive_key(secret, salt)

print(f"Derived AES key (256-bit) using SHA-512: {key.hex()}")

message = b"Hi my name is Adrian."

iv = os.urandom(12)

cipher = Cipher(algorithms.AES(key), modes.GCM(iv))
encryptor = cipher.encryptor()
ciphertext = encryptor.update(message) + encryptor.finalize()

auth_tag = encryptor.tag

print(f"\nCiphertext: {ciphertext.hex()}")
print(f"IV: {iv.hex()}")
print(f"Auth Tag: {auth_tag.hex()}")
```

```python
cipher = Cipher(algorithms.AES(key), modes.GCM(iv, auth_tag))
decryptor = cipher.decryptor()
decrypted_message = decryptor.update(ciphertext) + decryptor.finalize()

print(f"\nDecrypted message: {decrypted_message.decode()}")
```

Aan het begin van deze opdracht had ik geen idee hoe ik post-quantum cryptografie moest implementeren. Door de gegeven link te volgen, ontdekte ik in de comments belangrijke informatie die me heeft geholpen dit script te maken. De eerste belangrijke informatie die ik vond, was dat je cryptografische technieken met elkaar kunt combineren om te voldoen aan de eisen van post-quantum encryptie. Daarnaast realiseerde ik me dat het essentieel is om een sterke hashfunctie, zoals SHA-512, te gebruiken bij het hash-en, zodat deze bestand is tegen quantumaanvallen. Met deze inzichten ben ik aan de slag gegaan en heb ik mijn oplossing ontwikkeld.

Ik gebruik symmetrische encryptie met behulp van AES in Galois/Counter Mode (GCM). Voor te maken van de sleutel de functie “derive_key” maakt gebruik van PBKDF2HMAC (Password-Based Key Derivation Function 2) om een AES-sleutel te generen op basis van een gedeelde geheim een salt value. De sleutel wordt afgeleid met behulp van SHA-512.

De code genereert een willekeurige initialisatievector (IV) van 12 bytes en gebruikt deze samen met de afgeleide sleutel om de te versleutelen met AES-GCM. De versleuteling produceert een ciphertext en authenticatietag (auth_tag) om de integriteit van de data te waarborgen.

De ciphertext kan worden gedecrypteerd door dezelfde sleutel, IV en authenticatietag te gebruiken. Dit garandeert dat de boodschap weer in zijn oorspronkelijke vorm kan worden hersteld.

Waarom is dit nu exact post-quantum cryptografie? Sommige methodes zelf zijn niet echter post-quantum crypto maar die gebruiken concepten die relevant zijn:

Sleutel: De aanpak van het afleiden van een sleutel met een sterke hashfunctie (zoals SHA-512) en met veel iteraties kan deel uitmaken van een hybride cryptografisch systeem dat weerstand biedt tegen quantumaanvallen.

Gebruik van AES: AES kan worden beschouwd als een potentieel post-quantum algoritme, vooral in combinatie met andere technieken die de beveiliging kunnen versterken.

Als de aanvaller de gegevens zou willen decrypteren moet hij dan toegang hebben tot de versleutelde gegevens, IV en authenticatietag verkrijgen.

Zonder toegang tot de oorspronkelijke gedeelde geheim of het wachtwoord dat is gebruikt voor de sleutelafleiding, is het vrijwel onmogelijk om de sleutel te reconstrueren, vooral als het wachtwoord sterk en moeilijk te raden is.

Natuurlijk kan een aanvaller proberen brute-force aanvallen uit te voeren of gebruik te maken van quantumcomputers, maar dit zal enorm veel tijd kosten. Als de gebruiker regelmatig van boodschap wisselt, wordt het vrijwel onmogelijk om deze te ontcijferen voordat de gebruiker overschakelt naar een nieuwe.

Link Github: