UE Authentication and Registration Process

Objective-

The objective is to create a simulation that mimics the real-world process a UE (like a smartphone or IoT device) undergoes to authenticate itself and register with a 5G network. We'll use the IMSI as the UE's unique identifier and incorporate authentication tokens to ensure secure communication. This.is going to represent complex cryptographic information (such as 5G-AKA or EAP-AKA) in a simple flow, emphasizing the functions.of network functions and the interaction sequence. Students will implement a system that authenticates credentials and initiates a registration process, mimicking the behaviour of the 5G core network.

Concepts Covered-

Let's break down the main participants and ideas involved:

IMSI (International Mobile Subscriber Identity)

The IMSI is a special identifier contained within the UE's SIM card that is employed to identify the subscriber within the network. It's where the process of authentication begins, enabling the network to search for the UE's subscription information.

AMF (Access and Mobility Management Function)

The AMF is the 5G core network gatekeeper. It processes registration requests from the UE, coordinates the authentication, and provides secure NAS signalling. In our simulation, the AMF will accept the UE's first request and coordinate the authentication.

UDM (Unified Data Management)

The UDM is the central database for holding subscriber profiles, including IMSI and its relevant authentication keys. It produces authentication vectors (tokens) used to authenticate the UE's identity.

AUSF (Authentication Server Function)

The AUSF is the authentication expert. It collaborates with the UDM to create authentication tokens and checks the UE's responses during the authentication challenge. It authenticates the UE as who he/she says he/she is.

NAS Authentication (Non-Access Stratum)

NAS is the layer between the UE and the AMF, employed for secure signalling. NAS authentication is the process by which the network sends a challenge to the UE in the form of an authentication token, and the UE returns a response to authenticate its identity.

Simulation Design

To make this tangible, let's design a step-by-step simulation that students can implement (e.g., in Python or a similar language). We'll simplify the cryptography by using basic token generation and verification, while preserving the 5G flow.

Step 1: UE Initiates Registration

Action:-

The UE sends a registration request to the AMF, including its IMSI.

Simulation:-

Model the UE as a client that provides its IMSI (e.g., a hardcoded value such as "123456789012345") to the AMF module.

Why: -

This simulates the UE registering with a 5G base station (gNodeB) and providing its identity to the core network.

Step 2: AMF Sends Request to AUSF and UDM

Action: -

The AMF gets the IMSI and asks for an authentication vector from the UDM through the AUSF.

Simulation: -

The AMF module forwards the IMSI to a UDM function, which creates a basic authentication token (e.g., a random string or hashed value). The AUSF retrieves this token.

Why: -

In an actual 5G network, the UDM creates a vector containing a RAND (random number), AUTN (authentication token), and keys, which the AUSF processes.

Step 3: NAS Authentication Challenge

Action: -

The AMF forwards the authentication token to the UE through NAS signalling, presenting the challenge for it to reply.

Simulation: -

The AMF forwards the token (for example, "abc123") to the UE. The UE needs to calculate a reply from a shared secret (simplified into a pre-agreed key or formula).

Why: -

This validates the UE with the right credentials, blocking unauthorized users.

Step 4: UE Responds to Challenge

Action:-

The UE computes a response based on the token and its secret key and returns it to the AMF.

Simulation: -

The UE creates a response (e.g., by adding its IMSI to the token and hashing it) and sends it back to the AMF.

Why: -

This verifies the authenticity of the UE to the network.

Step 5: Verification by AUSF

Action: -

The UE's response is passed on by the AMF to the AUSF, which checks it against the UDM's expected value.

Simulation: -

The AUSF checks the UE's response against its own computation based on the same token and secret. If they are equal, authentication is successful.

Why: -

This ensures mutual authentication—both UE and network trust both.

Step 6: Registration Completion

Action: -

When authenticated successfully, the AMF registers the UE and allots it a temporary identity (e.g., GUTI - Globally Unique Temporary Identifier).

Simulation: -

The AMF informs the UE of successful registration and gives a temporary ID (e.g., "GUTI-001").

Why: -

Registration permits the UE to utilize network services, and the GUTI supplants the IMSI for privacy in subsequent interactions.

Expected Outcome-

Students will, by the conclusion of this exercise, have built a minimal authentication mechanism that:

Checks Credentials: -

The network validates the UE response against the predicted value, replicating safe identification confirmation.

Swarms Registration: -

The UE is taken through the change in status from being unauthenticated to being registered, issued with an ejection timer.

Practical Training: -

Practical experience in the role each component of the 5G network plays in operation.

Example Pseudocode-

Here’s a simplified outline to guide implementation:

```
UE:
  IMSI = "123456789012345"
  SendRegistrationRequest(IMSI) to AMF

AMF:
  Receive(IMSI)
  AuthVector = RequestAuthVector(IMSI) from UDM via AUSF
  SendChallenge(AuthVector.Token) to UE
  Response = ReceiveResponse() from UE
  If AUSF.Verify(Response, AuthVector):
    RegisterUE(IMSI)
    SendSuccess(GUTI) to UE

UDM:
  GenerateAuthVector(IMSI):
    Token = RandomString()
    ExpectedResponse = Hash(Token + SecretKey)
    Return {Token, ExpectedResponse}

AUSF:
  Verify(Response, AuthVector):
    Return Response == AuthVector.ExpectedResponse

UE:
  ReceiveChallenge(Token)
  Response = Hash(Token + SecretKey)
  SendResponse(Response) to AMF
  ReceiveSuccess(GUTI)
```

Step-by-Step Explanation-

1. Create Subscriber Database (UDM)

- **What Happens:** The UDM is the centralized repository in the 5G core network that stores subscriber data, including the UE's International Mobile Subscriber Identity (IMSI) and long-term secret keys (e.g., K, the subscriber key). In the simulation, this step involves setting up a mock database that holds these credentials for each UE.
- **Why It Matters:** The UDM is the source of truth for subscriber authentication. Without this database, the network can't verify who the UE claims to be.
- **In the Code:** This could be a simple data structure (e.g., a dictionary or list) where each entry maps an IMSI to a secret key. For example:

```
udm_database = {
   "123456789012345": {"key": "secretK123", "registered": False}
}
```

- **Real-World Note**: In 5G, the UDM also manages subscription profiles and interacts with other functions like the AUSF to provide authentication data.

2. Generate Authentication Vectors (AUSF)

- **What Happens**: The AUSF, in collaboration with the UDM, generates an authentication vector (AV) for the UE based on its IMSI. This vector typically includes:
  - **RAND**: A random number used as a challenge.
  - **AUTN**: An authentication token proving the network's legitimacy to the UE.
  - **XRES**: The expected response from the UE.
  - **K_AUSF**: A key for further security derivations.

- **Why It Matters**: The AV ensures mutual authentication—both the network and UE verify each other. The RAND challenges the UE, while AUTN proves the network isn't fake.
- **In the Code**: The AUSF might generate a random RAND (e.g., using random.randbytes) and compute AUTN and XRES using a simplified function (e.g., a hash of RAND and the secret key). Example:

```
import hashlib, os
RAND = os.urandom(16)  # 16-byte random challenge
key = udm_database[imsi]["key"]
AUTN = hashlib.sha256(RAND + key.encode()).hexdigest()[:16]  #
Simplified AUTN
XRES = hashlib.sha256(RAND + key.encode()).hexdigest()  #
Expected response
auth_vector = {"RAND": RAND, "AUTN": AUTN, "XRES": XRES}
```

- **Real-World Note**: In 5G-AKA, this involves complex cryptography (e.g., MILENAGE algorithms), but we simplify it here for clarity.

3. Challenge UE with Random Data (RAND, AUTN)

- **What Happens**: The AMF receives the authentication vector from the AUSF and sends the RAND and AUTN to the UE over the Non-Access Stratum (NAS) protocol. This is the network's challenge to the UE: "Prove you know the secret key."
- **Why It Matters**: This step initiates the authentication process, ensuring only a legitimate UE with the correct key can respond correctly.
- **In the Code**: The AMF acts as the intermediary, forwarding RAND and AUTN to the UE simulation. Example:

```
ue.receive_challenge(auth_vector["RAND"], auth_vector["AUTN"])
```

- **Real-World Note**: NAS signaling is encrypted and integrity-protected in a real network, but we assume a basic transmission here.

4. UE Computes a Response (RES)

- **What Happens**: The UE, using its SIM card's secret key (shared with the UDM), computes a response (RES) to the RAND challenge. It also verifies the AUTN to ensure the network is legitimate.
- **Why It Matters**: The RES proves the UE's identity, while AUTN verification prevents man-in-the-middle attacks.
- **In the Code**: The UE mirrors the network's computation using RAND and its key. Example:

```
class UE:
    def __init__(self, imsi, key):
        self.imsi = imsi
        self.key = key
    def receive_challenge(self, RAND, AUTN):
        # Compute RES
        self.RES = hashlib.sha256(RAND + self.key.encode()).hexdigest()
        # Simplified AUTN check (in reality, more complex)
        expected_autn = hashlib.sha256(RAND +
self.key.encode()).hexdigest()[:16]
        if expected_autn == AUTN:
            return self.RES
        else:
            raise Exception("Network authenticity failed")
```

- **Real-World Note**: The UE's SIM uses algorithms like f1-f5 (MILENAGE) to compute RES and verify AUTN, but we use a hash for simplicity.

5. Authenticate UE Response

- **What Happens**: The AMF receives the UE's RES and forwards it to the AUSF, which compares it to the XRES from the authentication vector. If they match, the UE is authenticated.
- **Why It Matters**: This confirms the UE's legitimacy, ensuring only authorized devices access the network.
- **In the Code**: The AUSF performs a simple comparison:

```
def authenticate_ue(res, auth_vector):
    return res == auth_vector["XRES"]
if authenticate_ue(ue_response, auth_vector):
    print("UE Authenticated")
```

- **Real-World Note**: Successful authentication triggers key agreement and session establishment.

6. If Successful, Derive Security Keys (K_SEAF)

- **What Happens**: Upon authentication, the AUSF derives an anchor key (K_AUSF), which is used to compute the Session and Mobility Management Key (K_SEAF). The AMF uses K_SEAF to secure NAS signalling with the UE.
- **Why It Matters**: K_SEAF protects subsequent communication, ensuring confidentiality and integrity.
- **In the Code**: A simplified key derivation might concatenate the key and IMSI:

```
K_SEAF = hashlib.sha256((key + imsi).encode()).hexdigest()
print(f"Derived K_SEAF: {K_SEAF}")
```

- **Real-World Note**: In 5G, K_SEAF is derived via a Key Derivation Function (KDF) using K_AUSF, SUPI (Subscription Permanent Identifier), and other parameters.

7. Register UE in the AMF

- **What Happens**: With authentication complete and keys derived, the AMF registers the UE, assigning it a temporary identifier (e.g., GUTI - Globally Unique Temporary Identifier) and marking it as active.
- **Why It Matters**: Registration allows the UE to access network services, and GUTI enhances privacy by replacing the IMSI in future interactions.
- **In the Code**: Update the UDM database and notify the UE:

```
udm_database[imsi]["registered"] = True
GUTI = f"GUTI-{imsi[-4:]}"
```

```
print(f"UE Registered with GUTI: {GUTI}")
```

- **Real-World Note**: The AMF maintains the UE's context, including mobility and session states.

How It All Fits Together-

The simulation follows the 5G AKA flow:

1. **UE → AMF**: Sends IMSI to initiate registration.
2. **AMF → AUSF/UDM**: Requests authentication vector.
3. **UDM/AUSF → AMF**: Provides RAND, AUTN, XRES.
4. **AMF → UE**: Challenges with RAND and AUTN.
5. **UE → AMF**: Responds with RES.
6. **AMF → AUSF**: Verifies RES against XRES.
7. **Success**: Derives K_SEAF and registers the UE.

Final Code With Each Segment Elaboration-

1. Importing Necessary Libraries

```
import ns.core

import ns.network

import ns.internet

import ns.mobility

import ns.lte

import ns.applications

import ns.point_to_point

import ns.config_store

import hashlib

import hmac
import random
import time
import uuid
```

- **NS-3 Modules (ns.core, ns.network, ns.internet, etc.)** → Used for **network simulation** in NS-3 (but not actually used in this script).
- **hashlib, hmac** → Used for computing cryptographic **hash-based message authentication codes (HMAC)** for security.
- **random, time, uuid** → Used for generating random numbers, time-based sequences, and unique identifiers.

---

2. Authentication Vector Class

```
class AuthenticationVector:
    """Class representing the 5G Authentication Vector"""
    def __init__(self, rand, autn, xres, k_ausf):
        self.rand = rand          # Random challenge
        self.autn = autn          # Authentication token
        self.xres = xres          # Expected response
        self.k_ausf = k_ausf      # Key for AUSF
```

Purpose

- Represents an **authentication challenge** generated for a UE.
- Stores:
    - **RAND** → A random number for authentication.
    - **AUTN** → Authentication token to verify the network.
    - **XRES** → Expected response from UE.
    - **K_AUSF** → A key derived for secure communication.

---

3. User Equipment (UE) Class

```
class UE:
    """User Equipment class"""
    def __init__(self, imsi, ki, op):
        self.imsi = imsi         # International Mobile Subscriber Identity
        self.ki = ki             # Subscriber authentication key
        self.op = op             # Operator code
        self.registered = False
        self.k_seaf = None       # Security anchor function key
        self.supi = f"imsi-{imsi}"  # Subscription Permanent Identifier
```

Purpose

- Represents a **mobile device** that attempts to register with the 5G network.
- Stores:
  - **IMSI** → Unique identifier for the mobile subscriber.
  - **KI** → Secret key used for authentication.
  - **OP** → Operator-specific key.
  - **Supi** → Derived from IMSI (e.g., imsi-123456789012345).
  - **Registered** → Tracks whether the UE is successfully authenticated.
  - **K_SEAF** → A security key derived after successful authentication.

---

4. UE Computes Authentication Response

```
def compute_response(self, rand, autn):
    """
    Compute authentication response using MILENAGE algorithm (simplified)
    In real 5G systems, this would use proper 5G-AKA or EAP-AKA' algorithms
    """

    mac = hmac.new(self.ki.encode(),
                   (rand + autn + self.op).encode(),
                   hashlib.sha256).hexdigest()
    return mac[:32]  # Return the response (RES)
```

How it Works

- **Takes RAND and AUTN as input** (sent by the network).
- **Computes a response (RES)** using an HMAC-based hash function.
- **Returns the first 32 characters** (simplified version).

---

5. UE Derives Security Keys

```
def derive_keys(self, k_ausf):
    """Derive keys for secure communication"""
    self.k_seaf = hmac.new(k_ausf.encode(),
                           self.supi.encode(),
                           hashlib.sha256).hexdigest()

    return self.k_seaf
```

- **Takes K_AUSF as input** (provided by the network).
- **Derives K_SEAF (Security Anchor Key)** using HMAC for secure communication.

---

6. Unified Data Management (UDM) Class

```
class UDM:
    """Unified Data Management class"""
    def __init__(self):
        self.subscriber_db = {}  # Database of subscribers
```

- **Maintains a database of subscribers (subscriber_db)**.

---

7. Adding a Subscriber

```
def add_subscriber(self, imsi, ki, op):
    """Add a subscriber to the database"""
    self.subscriber_db[imsi] = {"ki": ki, "op": op}
```

- Stores **IMSI, KI, OP** in the UDM database.

---

8. Generating an Authentication Vector

```
def generate_auth_vector(self, imsi):
    """Generate authentication vector for a subscriber"""
    if imsi not in self.subscriber_db:
        return None

    subscriber = self.subscriber_db[imsi]

    rand = uuid.uuid4().hex  # Random challenge
    seq = int(time.time())  # Sequence number
    autn = hmac.new(subscriber["ki"].encode(), str(seq).encode(), hashlib.sha256).hexdigest()[:32]
    xres = hmac.new(subscriber["ki"].encode(), (rand + autn + subscriber["op"]).encode(), hashlib.sha256).hexdigest()[:32]
    k_ausf = hmac.new(subscriber["ki"].encode(), (rand + str(seq) + subscriber["op"]).encode(), hashlib.sha256).hexdigest()

    return AuthenticationVector(rand, autn, xres, k_ausf)
```

- Generates **RAND, AUTN, XRES, and K_AUSF** for authentication.

---

9. Authentication Server Function (AUSF)

```
class AUSF:
    def __init__(self, udm):
        self.udm = udm
        self.auth_contexts = {}  # Store active authentication contexts
```

- **Stores authentication contexts for ongoing authentication sessions**.

---

10. Authenticating UE

```
def authenticate(self, supi, res):
    """Authenticate the UE based on response"""
    if supi not in self.auth_contexts:
        return {"result": "failed", "reason": "No authentication context"}

    context = self.auth_contexts[supi]
    auth_vector = context["auth_vector"]

    if res == auth_vector.xres:
        context["status"] = "authenticated"
        return {"result": "success", "k_ausf": auth_vector.k_ausf}
    else:
        return {"result": "failed", "reason": "Authentication failed"}
```

- **Checks if UE's response matches the expected response**.
- If **matches** → UE is authenticated.
- If **mismatch** → Authentication fails.

---

11. AMF Handles Registration

```
class AMF:
    def __init__(self, ausf):
        self.ausf = ausf
        self.registered_ues = {}
```

- **Manages UE registration and security keys**.

---

12. Handling UE Registration

```
def handle_registration_request(self, ue):
    auth_data = self.ausf.initiate_authentication(ue.supi)
    res = ue.compute_response(auth_data["rand"], auth_data["autn"])
    auth_result = self.ausf.authenticate(ue.supi, res)

    if auth_result["result"] == "success":
        k_seaf = ue.derive_keys(auth_result["k_ausf"])
        self.registered_ues[ue.supi] = {"k_seaf": k_seaf}
        ue.registered = True
        return True
    return False
```

- **Triggers authentication**, receives response, verifies it, and registers UE if successful.

Final Execution

```
if __name__ == "__main__":
    setup_5g_network()
```

Runs the setup, adds subscribers, and performs UE authentication.