Host Identity Protocol Implementation Report

Authentication, Authorization, Encryption & Replay Attack Protection

Team members

1. Farida Ahmed Salem 2206160
2. Hadir Amr 22010450
3. Martin Maher 22010445
4. Youssef Ahmed Ragaee 2202126
5. Youssef Tamer Mohamed Ahmed 2206172
6. Rewan Salah Mahmoud Shiha 20221447143

Overview

In today's digital world, we often send and receive sensitive information like login credentials, financial data, or private messages. But what if someone pretends to be us? Or intercepts our messages as they travel through the internet? Worse, what if someone reuses an old message to trick the system?
These threats are real, and that's where cybersecurity steps in using tools like authentication, authorization, encryption, and replay attack protection to keep communications safe and trustworthy.
To go a step further, there's also a protocol designed specifically to boost security and flexibility at the network level it's called the Host Identity Protocol (HIP).
HIP separates a device's identity from its IP address using public-key cryptography. Instead of identifying a machine by a potentially changing IP, HIP assigns a unique identity based on cryptographic keys. This helps build stronger, more mobile, and more secure communication by creating encrypted connections using IPsec.
In our simulation, we bring these ideas to life using Python showing how secure communication can be achieved by combining these core security principles with concepts inspired by HIP. From verifying who a sender really is to blocking replay attacks, we simulate a simplified version of how secure systems work behind the scenes.

Our code simulates a communication scenario between two users, User A and User B. The goal is to make sure that:
The message is really from who they say they are (Authentication) RSA for Digital Signatures
The user is allowed to perform the action they requested (Authorization) Role-Based Access Control
The message is hidden from anyone else using encryption (Confidentiality) AES Encryption
Old messages can't be resent to trick the system (Replay Attack Detection)

Dependencies :

```python
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes
from cryptography.fernet import Fernet
import time
```

1. Key Generation

The system generates two pairs of RSA keys for Entity A as sender and Entity B as receiver each entity has a private key used for signing and a public key used for verifying the sender’s identity think of it like a signature only you can sign it and others can check if it’s really yours

PrivateKey = rsa.generate_private_key(public_exponent=65537, key_size=2048) generates a new RSA private key. The key_size=2048 means the key will be 2048 bits long, which is currently considered secure for most applications. The public_exponent=65537 is a commonly used value in RSA because it strikes a balance between security and performance. It is a prime number, large enough to avoid certain attacks, but small enough to make encryption and signature verification efficient. Together, this configuration ensures the generated keys are strong, safe from common cryptographic vulnerabilities, and fast enough for real-world use.

```python
def generate_key_pair():
    PrivateKey = rsa.generate_private_key(public_exponent=65537, key_size=2048)
    PublicKey = PrivateKey.public_key()  # Fixed method name
    return PrivateKey, PublicKey
```

2. Access Control List

Access control is handled via an Access Control List (ACL) that maps roles to allowed actions:
Admin can read, write, delete
Analyst can read and write
Guest can read

```python
ACL = {
    "admin": ["read", "write", "delete"],
    "analyst": ["read", "write"],
    "guest": ["read"]
}
```

3. Authentication Digital Signature

The authenticate() function allows one entity to sign a message using its private RSA key. The receiver verifies the authenticity using the sender's public key.
This process ensures message authenticity confirms the sender's identity and message integrity confirms the message was not tampered with.

```python
def authenticate(signerPrivate, verifierPublic, Message):
    try:
        signature = signerPrivate.sign(
            Message,
            padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH),
            hashes.SHA256()
        )
        verifierPublic.verify(
            signature,
            Message,
            padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH),
            hashes.SHA256()
        )
        return True, signature
    except Exception:
        return False, None
```

4. Authorization

Once we trust who sent the message, we ask is this user allowed to do this action?
This prevents unauthorized actions for example "analyst" trying to perform a "delete" operation will be denied.

```python
def authorize(role, action):
    allowed_actions = ACL.get(role.lower(), [])
    return action in allowed_actions
```

5. Confidential Messaging AES Encryption

After authentication, secure communication is established using AES encryption via Fernet:
A random symmetric key is generated.
The message is encrypted and then decrypted to simulate secure communication.
Benefits confidentiality only entities with the AES key can read the message and simplicity fernet handles encryption, decryption, and key management securely.

```python
def aes_communication(shared_key, message):
    cipher = Fernet(shared_key)
    encrypted = cipher.encrypt(message)
    decrypted = cipher.decrypt(encrypted)
    return encrypted.decode(), decrypted.decode()
```

6. Replay Attack Detection

A replay attack happens when someone captures a valid message and tries to send it again later to trick the system. To prevent this: Each message includes a timestamp to show when it was created, the receiver compares this timestamp with the current time and if the message is older than 30 seconds, it's considered suspicious and automatically rejected. This ensures message is new and recent, integrity and replay protection

```python
# Global timestamp for replay protection
last_timestamp = 0

def run_scenario(title, sender_priv, receiver_priv, role, action, message, spoofed=False, replay=False):
    global last_timestamp

    # Use old timestamp if simulating replay attack
    if replay:
        timestamp = last_timestamp
    else:
        timestamp = time.time()

    timestamped = message + b'||' + str(timestamp).encode()

    # Authentication
    if spoofed:
        authAtoB, _ = authenticate(sender_priv, receiver_priv.public_key(), timestamped)
        authBtoA, _ = authenticate(receiver_priv, sender_priv.public_key(), timestamped)
    else:
        authAtoB, _ = authenticate(sender_priv, sender_priv.public_key(), timestamped)
        authBtoA, _ = authenticate(receiver_priv, receiver_priv.public_key(), timestamped)

    # Authorization
    if authAtoB and authBtoA:
        authz = authorize(role, action)
    else:
        authz = "Skipped (authentication failed)"
```

last_timestamp is initialized to zero because it serves as a baseline reference before any message is processed. Starting with 0 ensures that the first incoming message's timestamp—always a large current time value—will be accepted. It avoids mistakenly flagging the first valid message as a replay attack and allows the program to update this value only after successful authentication and replay checks.

We write authBtoA, _ to indicate that the function authenticate() returns two values, but we only care about the first one (authBtoA). The underscore _ is a common Python convention used to ignore the second return value, which might be extra data like a signature or log info that isn't needed at that point in the code. This keeps the code clean and focused on the important result — whether authentication succeeded.

```python
# AES
aesKey = Fernet.generate_key()
encrypted, decrypted = aes_communication(aesKey, message)
if not (authAtoB and authBtoA and authz == True):
    decrypted = "Not allowed"
```

This part of the code is responsible for handling the encryption and decryption of the message using AES (Advanced Encryption Standard) with the Fernet symmetric encryption scheme.
a new AES key is generated using Fernet.generate_key().
The aes_communication function encrypts and decrypts the message using this key.
The if not (authAtoB and authBtoA and authz == True) check ensures that if the authentication (both from A to B and B to A) or authorization fails, the message is not decrypted and instead, it is marked as "Not allowed". This ensures that only authorized and authenticated users can access the content of the message.

```python
# Replay Check
if authAtoB and authBtoA:
    try:
        msg_parts = timestamped.split(b'||')
        msg = msg_parts[0]
        ts = float(msg_parts[1].decode())
        current_time = time.time()

        if ts < last_timestamp:
            replay_valid = False
            replay_result = "Detected replay attack! Hacker reused an old message."
        elif abs(current_time - ts) > 30:
            replay_valid = False
            replay_result = "Replay attack detected (expired)."
        else:
            replay_valid = True
            replay_result = msg.decode()
            last_timestamp = ts
    except:
        replay_valid = False
        replay_result = "Invalid timestamp format!"
else:
    replay_valid = False
    replay_result = "Authentication failed – Replay check not performed"
```

The "Replay Check" section is crucial for detecting and preventing replay attacks, where an attacker intercepts and reuses a legitimate message to gain unauthorized access or cause damage. The process begins by extracting and decoding the timestamp attached to the message (ts = float(msg_parts[1].decode())), then comparing it with the last valid timestamp (last_timestamp). If the timestamp is older, it indicates a replay attack, as the message was reused. Additionally, if the timestamp is more than 30 seconds old, it could mean the message was delayed or stolen and replayed later, which is also flagged as a replay attack. If the timestamp is recent and within an acceptable range, the message is considered valid, and the system updates the last_timestamp for future checks. If an error occurs during decoding or parsing the timestamp, the system flags the message as invalid. In cases where authentication fails, the replay check is skipped entirely, ensuring that only authenticated messages are subjected to replay checks. This system helps protect against attackers trying to reuse old or expired messages to bypass security mechanisms.

```python
print(f"{title}:")
print(f"Authentication A -> B: {authAtoB}\n")
print(f"Authentication B -> A: {authBtoA}\n")
print(f"Role: {role}\n")
print(f"Requested Action: {action}\n")
print(f"Authorization Result: {authz}\n")
print(f"Encrypted AES Message: {encrypted}\n")
print(f"Decrypted AES Message: {decrypted}\n")
print(f"Replay Check Valid: {replay_valid}\n")
print(f"Replay Check Result: {replay_result}\n")
```

This block of code is used to display the results of the authentication, authorization, and encryption process. It prints detailed information such as whether authentication was successful between A and B, the role and action requested, the result of the authorization check, the encrypted and decrypted messages, and the result of the replay attack check. This helps in debugging, tracking, and understanding the flow of the security process in the system.

Scenarios We Tested

```python
# Scenarios
run_scenario("Scenario 1", APrivate, BPrivate, "analyst", "write", b"Hello")
print()
run_scenario("Scenario 2", APrivate, BPrivate, "analyst", "delete", b"Hello 2")
print()
run_scenario("Scenario 3", BPrivate, APrivate, "guest", "read", b"Hello 3", spoofed=True)
print()
run_scenario("Scenario 4 (Replay Attack)", APrivate, BPrivate, "analyst", "write", b"Hello", replay=True)
```

In Scenario 1 (Normal Communication), a valid user with the role "Analyst" sent a legitimate write request. The system successfully authenticated the user, verified their permission to perform the action, encrypted the message, and confirmed that it was fresh (not a replay). All security checks passed, and the message was allowed.

```
PS C:\Users\Administrator> & C:/Users/Administrator/AppData/Local/Programs/Python/Python313/python.exe c:/Users/Administrator/Downloads/hip.py
Scenario 1:
Authentication A -> B: True

Authentication B -> A: True

Role: analyst

Requested Action: write

Authorization Result: True

Encrypted AES Message: gAAAAABoE2RnXlwb9nV7DQe9iHR0e_KTCpGtxXAMe7vJCHbC4PiwbZ4WmoIvMzB4hFEqM9zyQuaSiFwRDPjGaR2B6-QEA6e1cA==

Decrypted AES Message: Hello

Replay Check Valid: True

Replay Check Result: Hello
```

In Scenario 2 (Spoofed Communication), an attacker tried to forge a message using mismatched cryptographic keys. This caused the authentication step to fail. As a result, the system skipped further checks like authorization and encryption and immediately blocked the message. This demonstrates how digital signatures prevent identity forgery.

```
Scenario 2:
Authentication A -> B: True

Authentication B -> A: True

Role: analyst

Requested Action: delete

Authorization Result: False

Encrypted AES Message: gAAAAABoE2RnynG9lU3axppNyNDsnIV3r5PA_TrDXsgRYKRcgsfsvZ2Th3I9F3CnNFjS6kYmyZDaD7VB_HD5pp6aXIx-nndx9w==

Decrypted AES Message: Not allowed

Replay Check Valid: True

Replay Check Result: Hello 2
```

In Scenario 3 (Unauthorized Action), a user with the "Guest" role attempted to perform a write operation. While the authentication step was successful (proving the user's identity), the system denied the request during the authorization check due to lack of required permissions. The message was encrypted and passed the replay check, but was ultimately not allowed.

```
Scenario 3:
Authentication A -> B: False

Authentication B -> A: False

Role: guest

Requested Action: read

Authorization Result: Skipped (authentication failed)

Encrypted AES Message: gAAAAABoE2RnV2L0B-9w_N_0QUeaPSMagfZKK2t-1_RQhJXARxJ60O1mVDHIxDbe2zVz9DVTxdFqzPEJGDZa1Oc0y-cBKQHzXA==

Decrypted AES Message: Not allowed

Replay Check Valid: False

Replay Check Result: Authentication failed – Replay check not performed
```

In Scenario 4 (Replay Attack) a valid message was sent again using a previously used timestamp. Although authentication and authorization were correct, and the encryption worked as intended, the system identified the reused timestamp and rejected the message to prevent a replay attack. This highlights the importance of ensuring message freshness in secure communications.

```
Scenario 4 (Replay Attack):
Authentication A -> B: True

Authentication B -> A: True

Role: analyst

Requested Action: write

Authorization Result: True

Encrypted AES Message: gAAAAABoE70MuOKBxkC6O8mNFY5_kOgpx1WhQMBqPY0yr3DjfeQQSnQTfYuqXwe2XLcb2IWVs5NGqp23Wf05-0TpNF3ze4JqhA==

Decrypted AES Message: Hello

Replay Check Valid: False

Replay Check Result: Detected replay attack! Hacker reused an old message.
```

Conclusion

This simulation brings together key cybersecurity principles authentication, authorization, encryption, and replay protection into one practical, working model. Inspired by the ideas behind the Host Identity Protocol (HIP), it demonstrates how separating identity from network location (IP address) and using strong public-key cryptography can enhance both security and flexibility in communication.

By combining RSA-based identity verification with AES encryption for data confidentiality and timestamp-based replay protection, the code offers a simple yet realistic approach to secure communications.

Though designed as an educational prototype, this implementation reflects many best practices used in real-world secure systems making it a great foundation for further exploration or development in modern, identity-driven network security.

Resources

https://nordvpn.com/cybersecurity/glossary/host-identity-protocol/

https://www.ericsson.com/en/reports-and-papers/research-papers/host-identity-protocol-hip-connectivity-mobility-multi-homing-security-and-privacy-over-ipv4-and-ipv6-networks#:~:text=The%20Host%20Identity%20Protocol%20(HIP,layer%20and%20the%20transport%20protocols.

https://www.sangfor.com/glossary/cybersecurity/what-is-replay-attack#:~:text=In%20the%20realm%20of%20cybersecurity,or%20fraudulently%20repeated%20or%20delayed.

https://me-en.kaspersky.com/resource-center/definitions/replay-attack

https://aarafat27.medium.com/understanding-the-underscore-in-python-f274d600b880