Q2

Documentation on my implementation of Q2

Part B: Retrieving the Secret and Creating a New JWT

Step 1: Understanding the JWT Structure

A JSON Web Token (JWT) consists of three parts:

1. **Header** - Specifies the algorithm used (e.g., HMAC-SHA256).
2. **Payload** - Contains user-related claims (e.g., role, email, etc.).
3. **Signature** - Ensures integrity using a secret key.

Given JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJmY3MtYXNzaWdubWVudC0xIiwia
WF0IjoxNTE2MjM5MDIyLCJleHAiOjE2NzI1MTE0M
DAsInJvbGUiOiJ1c2VyIiwiZW1haWwiOiJhcnVuQGlpaXRkLmFjLmluIiwi
aGludCI6Imxvd2VyY2FzZS1hbHBoYW51bWVyaWMtbGVuZ3RoLTUifQ.LCIyPHqWAVNLT8BMXw8
_69TPkvabp57ZELxpzom8FiI
```

Step 2: Brute-Force Attack to Retrieve the Secret

Since the JWT is signed using HMAC-SHA256, the signature can be verified only if we have the correct secret key. The brute-force approach iterates through a list of potential secret keys, computing the HMAC-SHA256 signature for each and comparing it to the provided signature.

Using Python, we generated a wordlist of all 5-character lowercase alphanumeric strings and checked each one against the JWT signature.

Findings:

The script successfully identified the secret key used to sign the JWT:

```
✅ Found secret key: *****  # (Actual key retrieved)
```

Step 3: Creating a New JWT with Role = "admin"

Once the secret was obtained, we modified the payload to change the `role` field from `user` to `admin`. We then re-signed the JWT using the same secret and algorithm.

Modified Payload:

```
{
  "sub": "fcs-assignment-1",
  "iat": 1516239022,
  "exp": 1672511400,
  "role": "admin",
  "email": "arun@iiitd.ac.in",
  "hint": "lowercase-alphanumeric-length-5"
}
```

After encoding and signing the new JWT, we obtained:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.<NEW_ENCODED_PAYLOAD>.<NEW_SIGNATURE>
```

This demonstrates that unauthorized privilege escalation is possible when weak secrets are used for JWT signing.

Part C: Preventing Widespread Damage if the Secret Key is Leaked

Problem:

If a single secret key is used to sign all JWTs, then if the key is leaked, an attacker can forge any token, granting themselves unauthorized access.

Proposed Solution:

1. **Use Asymmetric Cryptography (RSA or ECDSA)**:
   - Instead of a shared secret (HMAC), use **public-private key pairs**.
   - The server signs JWTs with a private key, and clients verify them using a public key.
   - Even if the public key is exposed, an attacker **cannot** sign new tokens.
2. **Key Rotation and Expiry Policies**:
   - Change signing keys periodically.
   - Maintain a **key versioning** system so that old tokens can still be verified until expiration.
3. **User-Specific Signing Keys**:
   - Instead of a single global secret, generate unique secrets per user session.
   - Store these securely and revoke compromised sessions without affecting all users.
4. **Use Secure Storage for Secrets**:

- Store JWT signing secrets in **environment variables** or secure vaults (e.g., AWS Secrets Manager, HashiCorp Vault).
- Restrict access to these secrets.

5. **Enforce Strong Secrets**:
   - Require long, random secrets (at least 32 bytes).
   - Use **PBKDF2** or **bcrypt** for secret generation.

Conclusion

By implementing these improvements, we can mitigate the risks associated with JWT secret key leaks, ensuring better security and preventing widespread unauthorized access.
