Section1 – Question 1

Assumptions:

1. Alice wants to send a non-confidential message to Bob, so encryption is not required.
2. Alice and Bob have pre-shared a secret key (symmetric key).
3. Alice and Bob have synchronized clocks.

Flow, the Message, Actions Involved:

1. Alice creates a secret key that she will share with Bob. **K and K_2**. Fig 1.
2. Alice creates a private key **Pr** and a public key **Pu**. Alice would keep the private key to herself whilst the public key could be used by someone else, in this instance Bob will use it to validate the signature sent with the message later in the sequence.
3. Alice generates the plaintext message **M**. "The company website has not limited the number of transactions a single user or device can perform in a given period of time. The transactions/time should be above the actual business requirement, but low enough to deter automated attacks."
4. Alice creates a timestamp **t** and adds this to the message **(t, M)**.
5. Alice uses secret key **K** to create a **hash** of the combined timestamp and message **(t, M, K) = hash(t, M, K) = hash_code**.
6. Alice creates a digital signature using their private key **Pr** from step 2 above and the **hash_code**, containing a hash of the timestamp, plaintext message, and secret key, to create a digital signature **Pr(hash_code)**.
7. The message **Pr(hash_code) = Pr(hash(t, M, K))** is sent to Bob.
8. Bob received the transmitted message in the form of 64 bytes, each byte contains 8 bits = 512 bits. Bob uses the public key **Pu** to validate the signature **Pu(Pr(hash(t, M, K)) => hash(t, M, K)**.
9. Bob now uses the copy of the shared secret key **K_2** to verify the hash that Alice created earlier with their own **K** secret key. **K_2(hash(t, M, K)) => (t, M)**.
10. Bob would now be able to view the timestamp and the message.

How this Protocol Avoids Modification of Message and Replay Attacks:

1. **Avoids Modification:** Alice signs the message using her private key **Pu**. So long as only Alice possesses **Pu** this digital signature can only be produced by Alice using **Pu**, **and** Alice cannot later repudiate (deny) signing the message. The message could be read by Bob and a 'man in the middle' because the key used to verify the digital signature is public. But if the man in the middle were to modify the message in any way, they would not be able to replicate Alice's signature without the private key. Since the **hash_code** is generated by Alice using a symmetric secret key **K**, and Bob recalculates the HMAC using the shared secret key **K_2**. The **hash_code** in the received message and the one recalculated by Bob should match, if they do not, then Bob would be alerted to the possibility of tampering.

2. **Replay Attack Mitigation:** Alice includes a timestamp in the message sent, Bob can compare this with the current time and if the message is too old, Bob could choose to disregard the message, mitigating possible replay attack. Alice and Bob agree suitable parameters for this beforehand. Also, the time stamp is added to the message before the message is hashed by Alice with secret key **K**, if the timestamp were tampered with, when Bob recalculates the received hash, the result would not match the new calculated hash, alerting Bob to a possible attack.

Section 2 – Question 2

Implementation Process:

```python
# 1. Generate a random secret key K
def generate_secret_key():
    return os.urandom(32)  # 32 bytes for a 256-bit key
# Alice generates a secret key and shares it with Bob
def share_secret_key(secret_key):
    print("Shared Secret Key:", secret_key.hex())
if __name__ == "__main__":
    # Alice generates the secret key
    secret_key = generate_secret_key()
    # Alice shares the secret key with Bob
    share_secret_key(secret_key)
```

```
Shared Secret Key: 1ad422dc68d87d12c5e8be7b970c435cf641b283474019631591f842bf0132cc
```

Fig 1. shows the function returning a 32-byte string corresponding to the 256-bit key. The secret key **K** is then shared with Bob **K_2**, here it is simply printed but in a real-world scenario it would be sent to Bob securely.

```python
# 2. Generate a private key Pr and public key Pu
def generate_private_key():
    return rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
def extract_public_key(private_key):
    return private_key.public_key()
def serialize_private_key(private_key):
    return private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )
def serialize_public_key(public_key):
    return public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
if __name__ == "__main__":
    private_key = generate_private_key()
    public_key = extract_public_key(private_key)
    private_key_pem = serialize_private_key(private_key)
    public_key_pem = serialize_public_key(public_key)
    print("Private Key:")
    print(private_key_pem.decode())
    print("\nPublic Key:")
    print(public_key_pem.decode())
```

Fig 2. Alice generates a private key **Pr** using the RSA algorithm and a key size of 2048 bits, then extracts the corresponding public key **Pu** from the private key object, then serialises the keys into 'Privacy-Enhanced Mail' (PEM) format, this is often used to define the structure of the file used to store a bit of data, in this case the keys [2]. The keys are very long so will not be shown here but are shown in Appendix 1.

```python
plain_text_message = """The company website has not limited the number of transactions a single user or device
can perform in a given period of time. The transactions/time should be above the actual business requirement,
but low enough to deter automated attacks."""
print(plain_text_message)
```

```
The company website has not limited the number of transactions a single user or device
can perform in a given period of time. The transactions/time should be above the actual business requirement,
but low enough to deter automated attacks.
```

```python
# 4. Function to add a timestamp to the plaintext message
def add_timestamp(plain_text_message):
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    message_with_timestamp = f"{timestamp}: {plain_text_message}"
    return message_with_timestamp
if __name__ == "__main__":
    plain_text_message = """The company website has not limited the number of transactions a single user or device can p
    message_with_timestamp = add_timestamp(plain_text_message)
    print("Message with Timestamp:")
    print(message_with_timestamp)
```

```
Message with Timestamp:
2024-03-12 22:06:50: The company website has not limited the number of transactions a single user or device can perform in a
given period of time. The transactions/time should be above the actual business requirement, but low enough to deter automat
ed attacks.
```

Fig 3. The first of the two code snippets above shows the plain text message **M** that Alice wants to send to Bob, and this is printed. Alice then creates and adds a timestamp **t** to the message formatted as a string and concatenated with the message and printed to demonstrate. **(t, M)**

```python
# 5. Function to create a hash code of the message with timestamp using the secret key
def create_hash(message_with_timestamp, secret_key):
    hmac_hash = hmac.new(secret_key, message_with_timestamp.encode(), 'sha256')
    return hmac_hash.digest()
if __name__ == "__main__":
    secret_key = b'MySecretKey123'
    message_with_timestamp = "2024-03-12 15:30:00: The company website has not limited the number of transactions a sing
    hash_code = create_hash(message_with_timestamp, secret_key)
    print("Hash Code:")
    print(hash_code.hex())
```

```
Hash Code:
81b03996830fd3e897be5c858f11504daa9000941cc01c9bfa011ad70a513d0f
```

```python
# 6. Function to create a signature using Alice's secret key
def create_signature(private_key, hash_code):
    signature = private_key.sign(
        hash_code,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature

if __name__ == "__main__":
    signature = create_signature(private_key, hash_code)
    signed_hash_code = hash_code + signature
    print("Signed Hash Code:")
    print(signed_hash_code.hex())
```

```
Signed Hash Code:
cdc6663732840c98c3ee7451482e5f52073158972e0a2198d30c4279de6fec194df8da58ba72bdc7592acb8de3eaa5da60a81d709b0155001df92af86fb1
432179569eeddb3d8eeae24440604db1b2f1c36759a84f3178482bfad6aabdb6d86c1f63ccde0793ad02f21fbd69869369cf79e788227e24593da1882a60
2d56dcfbe82141723abd134f5ca1cce97d0c190026179aa5eaa549392e60090ccb3486a05daa43aeab6328ffd0ff4b1eca36ca1365ec15b6edf19bccbe7b
cbcd9954e046d19c076c5f64dae5d8b9f26bfae03d61c73cb26a3db04d4fcb5ae8fcc619ad4dbe010bfc1fa02c8457ec2302687af3f3a9d7e1de2b2583a1
caf15e478371c224ce81dd2b618024bf9f6cc97dc3e1ec1728467ec5009c8e46b4c78688436f9e84
```

Fig 4. The first of the two code snippets above shows how a hashed message authentication code 'HMAC' using the hash function algorithm SHA-256 is created from Alice's secret key **K** and **(t, M) => hash(t, M, K) = hash_code** printed Any modification to the message or the timestamp would result in a different hash. The second code snippet is where Alice creates a digital signature using their private key **Pr** and appends this to **hash_code**. **Pr(hash(t, M, K)) => Pr(hash_code)**, signed hash is printed.

```python
# 7. Code for Alice to generate the hash and transmit the message and hash to Bob S(hash(T,M,K))
import hmac
def generate_hash(message_with_timestamp, shared_secret_key):
    return hmac.new(shared_secret_key, message_with_timestamp.encode(), 'sha256').digest()
if __name__ == "__main__":
    message_with_timestamp = "2024-03-12 15:30:00: The company website has not limited the number of transactions a sing
    shared_secret_key = b'MySecretKey123'
    hash_code = generate_hash(message_with_timestamp, shared_secret_key)
    transmitted_message = message_with_timestamp
    transmitted_hash = hash_code
    print("Transmitted Message:", transmitted_message)
    print("Transmitted Hash:", transmitted_hash.hex())
```

```
Transmitted Message: 2024-03-12 15:30:00: The company website has not limited the number of transactions a single user or de
vice can perform in a given period of time. The transactions/time should be above the actual business requirement, but low e
nough to deter automated attacks.
Transmitted Hash: 81b03996830fd3e897be5c858f11504daa9000941cc01c9bfa011ad70a513d0f
```

Fig 5. Above simulates the printed output being sent to Bob. Bob receives **Pr(hash(t, M, K))**.

```python
# 8. Bob receives message and verifies the signature
from cryptography.exceptions import InvalidSignature
# Function to verify the signature using public key
def verify_signature(public_key, hash_code, signature):
    try:
        public_key.verify(
            signature,
            hash_code,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True
    except InvalidSignature:
        return False
if __name__ == "__main__":
    is_signature_valid = verify_signature(public_key, hash_code, signature)
    if is_signature_valid:
        print("Signature Verification Result: VALID")
    else:
        print("Signature Verification Result: INVALID")
```

```
Signature Verification Result: VALID
```

Fig 6. The code snippet above shows how Alice's signature is verified using Alice's public key **Pu**. Inputs are the hash code of the message and the signature. Successful verification returns a valid signature 'true'. This verifies that the message was signed by Alice the private key holder.

Pu(Pr(hash(t, M, K)) => hash(t, M, K).

```python
import hmac
# 9. Function to verify the hash using the secret key
def verify_hash(message_with_timestamp, shared_secret_key, received_hash):
    computed_hash = hmac.new(shared_secret_key, message_with_timestamp.encode(), 'sha256').dig
    print("Computed Hash:", computed_hash.hex())
    print("Received Hash:", received_hash.hex())
    if hmac.compare_digest(computed_hash, received_hash):
        return True
    else:
        return False
if __name__ == "__main__":
    message_with_timestamp = "2024-03-12 15:30:00: The company website has not limited the num
    shared_secret_key = b'MySecretKey123'
    received_hash = bytes.fromhex("81b03996830fd3e897be5c858f11504daa9000941cc01c9bfa011ad70a5
    is_hash_valid = verify_hash(message_with_timestamp, shared_secret_key, received_hash)
    if is_hash_valid:
        print("Hash verification successful. The message is authentic.", message_with_timestam
    else:
        print("Hash verification failed. The message may have been tampered with.")
```

```
Computed Hash: 81b03996830fd3e897be5c858f11504daa9000941cc01c9bfa011ad70a513d0f
Received Hash: 81b03996830fd3e897be5c858f11504daa9000941cc01c9bfa011ad70a513d0f
Hash verification successful. The message is authentic. 2024-03-12 15:30:00: The company website h
as not limited the number of transactions a single user or device can perform in a given period of
time. The transactions/time should be above the actual business requirement, but low enough to det
er automated attacks.
```

Fig 7. Above is the verify hash function where Bob can verify that the message within **hash(t, M, K)** has not been tampered with. Bob computes a hash reversing the process that Alice used to create the hash, Bob computes a hash using the secret key **K_2** that Alice shared.

The result should equal the hash Alice created in step 5., of the ‘Flow’ section above. The output here shows they are the same. With this extracted, Bob would be left with the message and the timestamp **(t, M)**. Bob would be able to see the timestamp and so long as it was within acceptable parameters he could accept the message had not been replayed.

Critical Analysis and Justification:

```python
import hmac
# 9. Function to verify the hash using the secret key
def verify_hash(message_with_timestamp, shared_secret_key, received_hash):
    computed_hash = hmac.new(shared_secret_key, message_with_timestamp.encode(), 'sha256').digest()
    print("Computed Hash:", computed_hash.hex())
    print("Received Hash:", received_hash.hex())
    if hmac.compare_digest(computed_hash, received_hash):
        return True
    else:
        return False
if __name__ == "__main__":
    message_with_timestamp = "2024-03-12 15:30:00: The company website has not limited the number of
    shared_secret_key = b'MySecretKey'
    received_hash = bytes.fromhex("81b03996830fd3e897be5c858f11504daa9000941cc01c9bfa011ad70a513d0f")
    is_hash_valid = verify_hash(message_with_timestamp, shared_secret_key, received_hash)
    if is_hash_valid:
        print("Hash verification successful. The message is authentic.", message_with_timestamp)
    else:
        print("Hash verification failed. The message may have been tampered with.")
```

```
Computed Hash: 40712e39daedb3d6c51424f2acb4cc897d6fb267a1b93e12c2838f37f241c8e4
Received Hash: 81b03996830fd3e897be5c858f11504daa9000941cc01c9bfa011ad70a513d0f
Hash verification failed. The message may have been tampered with.
```

Fig 8. In fig 8 above the secret key is incorrect and the output shows that the message may have been tampered with, the two hashes do not match. Distinct from Fig 7., above where the correct secret key has been used to recalculate the hash and the two hashes match and the plaintext message has also been printed.

Vulnerabilities:

Key management is crucial if the private key is compromised an attacker could forge the signature. A breach in confidentiality for either of the secret keys (used to generate the HMAC) or the HMAC could seriously jeopardize the security of this protocol and could lead to unauthorized tampering and replay. It is imperative that the secret key remains secret between Alice and Bob, this is crucial to protect against modification of the message by an attacker. If an attacker somehow got hold of the secret key, they could intercept and modify the message, albeit Alice’s private key would still be required to properly sign the message. Also, if any of the keys **K, K_2, Pr** became known to an attacker all past and future communications could be compromised and messages could be replayed, tampered with, or forged.

The protocol depends upon cryptographic algorithms that could weaken over time increasing the possibility of attack, these should be reviewed, for vulnerabilities, and updated regularly to keep security measures current and relevant.

Conclusion: The protocol described uses digital signature, and timestamp validation to protect against man in the middle attacks of unauthorized tampering and replay. Despite its strengths, the protocol has limitations particularly surrounding key management. Alice would need to keep the private key safe, anyone in possession of the private key could not only forge Alice’s signature but also tamper with the message and timestamp before signing it, and the message recipient (Bob) would very likely not realise that the message received had been compromised. The protocol provides mechanisms for ensuring message integrity and protects against replay and would be a good foundation for secure communication practices.

References:

1. Stallings, W., 2011. *Cryptography and network security, 5/E*.
2. [What Is a PEM File and How Do You Use It? (howtogeek.com)]
3. Link to full code at GitHub: https://github.com/Settings2022/Comp2006_CW1.git

Appendices:

Appendix 1:

Private Key:

-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcw
ggSjAgEAAoIBAQCfbVRNnqGoDN/d
Nv6uIWIpXW3W/bMUpHNkc67fwqQWwSLAgjWm
muXILQQxS0OPy6momVhyMuyitZYu
wuGNvN0HbRHsZ8GfbpwNQfi6Bo3yQpv2kQY5
gr9A+DR0BIFnUqikC0ZSnHGsDGvX
WcGv0FDf9FIO5YCUzmkE44tD+wcxNrlpYBSg
5XpT3W9gIC2MXZH2LmzcElsHBaJa
ySPvmwCeswLMa1vDV30kA6Ln2zUkNC/n3BAs
xLX8dSGIEB0c7H4hXOeaMvylaEnm
8JlMkarfb/SChr3YtFwrAs3MYYL//1ob7Bp7
fhT/a4wgnsUHgi6ZzOrX9KaEKIKo
Q6vM4KbPAgMBAAECggEAOWyx7vtykwuAHzhN
5/7SUr/6bnQXaxoJDme015hObnvG
vryoAojMt5uvZOoNPeBIDhieCRRatAjgPZuo
jts2ahRZXmPJFFpND2Fx69+shVSE
xP1b1p9HsGk2y4tL6SGW6O++tsbvEt+ugHUu
FNveJazlWcRE2lS28zHa2fOzlX/k
XdMa7AxZ7/SIBp5uh5ZdHcG35yGPhkG6qfyl
Tg70fsq4Zj+wZ9dVhbDKFzMxgrKM
aeVJE8ySig2TeH8N1K9nKRMSbFEU+WRCVn5/
xfSqOZ/FMo+ayZrEIk9YDZHOSSRF
UFJzx+80LhbljZFHTp6KrSWYcPASqcLEaNO4
qa7W0QKBgQDf8Io9sElB+np3v/De
BNufwxl3TtiOfotsKWjzDmFjpt37K5xzs2im
v/s7eB1fZhM4/H4+ruKMepZG/73q
vXZ3V7MM44j/tQZN7M4Yook1LY3XRA5TytpW
hKmq6qm6LgRGUzGulXsVi6VJ0eck
+qwLl43p28Z5LkgUo9xXjRmIKwKBgQC2QGIv
Ag5vyLEHE+ob6IlKigaPpUj/AZ7r
/xtxdpNQhATtw42+k1SGNzKMhoXvAxQn3hiI
Jeo/pTkWMwLPXRVhVfg+y4jatT57
TrFTC+QQHskpmnfV8h2hefoyo9bul8g18D+z
pJBpWPny4DjzGZdrLDvdBK1THsUk
hCccwJ5F7QKBgDblvt30k2DrSI6GrUOgKT73
Ew0edRQpjYBMfn/nLJTDWXOzcz0h
5CvMsIgZoAm8+kVkEIbJVJxfiOuK0kHzhFEp
XKlyNimJdSwxOyzq23v/2N/GvURp
XDENgJJ3yHftw/qBdpJ37p6Ph0ube3CjSv3k
f1OvHu6iG+WDbgbAflvVAoGBAKu4
YkqUj3G4EUTv+KevJJz9DE2QmQTdTBZk2kDA
TvGQUuyMUyP7wapsm85Yeh3IMteV
pluyDdNGJFHMYptrw1dhH3RbZmlcWLDqZp4v
GAYwW649gygs5spdGedZBIzuqpBX
/E2Rgxgf4/J6Xm5/8HHkzcrkO0OUPIC5m/i1
bOvJAoGAORsKTjro/uhtpmRQlx26
zIwxUn/+8ptPPxsHif34vNX7k0NRZUkQs66G
lUNxjFqtZdcV5hVEYvtuJv+6QIru
CjM072+deNT3L7FCH+OpRaJwYbigsQL8Ox3s
5Le+31igqhbdcKTCY6UrE1n6M29M
HZEKL0ec+7agCslTYN1V1Eg=
-----END PRIVATE KEY-----

Public Key:

-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIB
CgKCAQEAn21UTZ6hqAzf3Tb+riFi
KV1t1v2zFKRzZHOu38KkFsEiwII1pprlyC0E
MUtDj8upqJlYcjLsorWWLsLhjbzd
B20R7GfBn26cDUH4ugaN8kKb9pEGOYK/QPg0
dASBZ1KopAtGUpxxrAxr11nBr9BQ
3/RSDuWAlM5pBOOLQ/sHMTa5aWAUoOV6U91v
YCAtjF2R9i5s3BJbBwWiWskj75sA
nrMCzGtbw1d9JAOi59s1JDQv59wQLMS1/HUh
iBAdHOx+IVznmjL8pWhJ5vCZTJGq
32/0goa92LRcKwLNzGGC//9aG+wae34U/2uM
IJ7FB4Iumczq1/SmhCiCqEOrzOCm
zwIDAQAB
-----END PUBLIC KEY-----