

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 **K₂**. 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 **K₂** to verify the hash that Alice created earlier with their own **K** secret key. **K₂(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 **K₂**. 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:

```

1 # 1. Generate a random secret key K
2 def generate_secret_key():
3     return os.urandom(32) # 32 bytes for a 256-bit key
4 # Alice generates a secret key and shares it with Bob
5 def share_secret_key(secret_key):
6     print("Shared Secret Key:", secret_key.hex())
7 if __name__ == "__main__":
8     # Alice generates the secret key
9     secret_key = generate_secret_key()
10    # Alice shares the secret key with Bob
11    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 **K₂**, here it is simply printed but in a real-world scenario it would be sent to Bob securely.

```

1 # 2. Generate a private key Pr and public key Pu
2 def generate_private_key():
3     return rsa.generate_private_key(
4         public_exponent=65537,
5         key_size=2048,
6         backend=default_backend()
7     )
8 def extract_public_key(private_key):
9     return private_key.public_key()
10 def serialize_private_key(private_key):
11     return private_key.private_bytes(
12         encoding=serialization.Encoding.PEM,
13         format=serialization.PrivateFormat.PKCS8,
14         encryption_algorithm=serialization.NoEncryption()
15     )
16 def serialize_public_key(public_key):
17     return public_key.public_bytes(
18         encoding=serialization.Encoding.PEM,
19         format=serialization.PublicFormat.SubjectPublicKeyInfo
20     )
21 if __name__ == "__main__":
22     private_key = generate_private_key()
23     public_key = extract_public_key(private_key)
24     private_key_pem = serialize_private_key(private_key)
25     public_key_pem = serialize_public_key(public_key)
26     print("Private Key:")
27     print(private_key_pem.decode())
28     print("\nPublic Key:")
29     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.

```

1 plain_text_message = ""The company website has not limited the number of transactions a single user or device
2 can perform in a given period of time. The transactions/time should be above the actual business requirement,
3 but low enough to deter automated attacks.---
4 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.

```

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

```

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

```

Hash Code:
81b03996830fd3e897be5c858f11504daa9000941cc01c9fa011ad70a513d0f

```

1 # 6. Function to create a signature using Alice's secret key
2 def create_signature(private_key, hash_code):
3     signature = private_key.sign(
4         hash_code,
5         padding.PSS(
6             mgf=padding.MGF1(hashes.SHA256()),
7             salt_length=padding.PSS.MAX_LENGTH
8         ),
9         hashes.SHA256()
10    )
11    return signature
12 if __name__ == "__main__":
13     signature = create_signature(private_key, hash_code)
14     signed_hash_code = hash_code + signature
15     print("Signed Hash Code:")
16     print(signed_hash_code.hex())

```

Signed Hash Code:
cdc663732840c9c3e7451482e5f52073158972e0a218d38c4279defec194df8da58ba72bdc7592acbd8e3ea5da60a81d709b01550801df92af86b143217959edd3d8ee2a4460404b12f1c3675984f3178482b6fda6ab06d86c1f63cde0793ad0721f4d6989369cf79e788227e24593da1882a602056dcfbed2141723ab0134f5ca1cc070bc108025179a5a54939260909c3b486a065aa43eaa0323ff0ff401eca36ca1365c150a0ef19b2c0b70cbcd9954e4d419c076c5f6daec5d809f26f0a63d6c73cb26a3d0d4dfc5ae8fcc618ad4d0e10bfc1fa02c8457ec2302687ef3f3a9d7e12db25831ca15e478371c24c8e1d2b6180240f9f6cc97dc3e1c1728467ec5009c0e46b4c78088436f9e84

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.

```

1 # 7. Code for Alice to generate the hash and transmit the message and hash to Bob S(hash(t,M,K))
2 import hmac
3 def generate_hash(message_with_timestamp, shared_secret_key):
4     return hmac.new(shared_secret_key, message_with_timestamp.encode(), 'sha256').digest()
5 if __name__ == "__main__":
6     message_with_timestamp = "2024-03-12 15:30:00: The company website has not limited the number of transactions a sing
7     shared_secret_key = b'MySecretKey123'
8     hash_code = generate_hash(message_with_timestamp, shared_secret_key)
9     transmitted_message = message_with_timestamp
10    transmitted_hash = hash_code
11    print("Transmitted Message:", transmitted_message)
12    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 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.
Transmitted Hash: 81b03996830fd3e897be5c858f11504daa9000941cc01c9fa011ad70a513d0f

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

```

1 # 8. Bob receives message and verifies the signature
2 from cryptography.exceptions import InvalidSignature
3 # Function to verify the signature using public key
4 def verify_signature(public_key, hash_code, signature):
5     try:
6         public_key.verify(
7             signature,
8             hash_code,
9             padding.PSS(
10                mgf=padding.MGF1(hashes.SHA256()),
11                salt_length=padding.PSS.MAX_LENGTH
12            ),
13            hashes.SHA256()
14        )
15        return True
16    except InvalidSignature:
17        return False
18 if __name__ == "__main__":
19     is_signature_valid = verify_signature(public_key, hash_code, signature)
20     if is_signature_valid:
21         print("Signature Verification Result: VALID")
22     else:
23         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))**.

```

1 import hmac
2 # 9. Function to verify the hash using the secret key
3 def verify_hash(message_with_timestamp, shared_secret_key, received_hash):
4     computed_hash = hmac.new(shared_secret_key, message_with_timestamp.encode(), 'sha256').digest()
5     print("Computed Hash:", computed_hash.hex())
6     print("Received Hash:", received_hash.hex())
7     if hmac.compare_digest(computed_hash, received_hash):
8         return True
9     else:
10        return False
11 if __name__ == "__main__":
12     message_with_timestamp = "2024-03-12 15:30:00: The company website has not limited the num
13     shared_secret_key = b'MySecretKey123'
14     received_hash = bytes.fromhex("81b03996830fd3e897be5c858f11504daa9000941cc01c9fa011ad70a513d0f")
15     is_hash_valid = verify_hash(message_with_timestamp, shared_secret_key, received_hash)
16     if is_hash_valid:
17         print("Hash verification successful. The message is authentic.", message_with_timestamp)
18     else:
19         print("Hash verification failed. The message may have been tampered with.")

```

Computed Hash: 81b03996830fd3e897be5c858f11504daa9000941cc01c9fa011ad70a513d0f
Received Hash: 81b03996830fd3e897be5c858f11504daa9000941cc01c9fa011ad70a513d0f
Hash verification successful. The message is authentic. 2024-03-12 15:30:00: 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.

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 **K₂** 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:

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**, **K₂**, **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? \(howtogeek.com\)](https://howtogeek.com/what-is-a-pem-file-and-how-do-you-use-it/)
3. Link to full code at GitHub: https://github.com/Settings2022/Comp2006_CW1.git

Appendices:

Appendix 1:

```
Private Key:
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBCcw
ggSjAgEAAoIBAQCfbVRNnqGoDN/d
Nv6uIWIpXW3W/bMUpHNkc67fwqQWwSLAgjWm
muXILQQxS0OPy6momVhyMuyitZYu
wuGNvN0HbRHsZ8GfbpwNQfi6Bo3yQpv2kQY5
gr9A+DR0BIFnUqikC0ZSnHGSDGvX
WcGv0FDf9FIO5YCUzmKE44tD+wcxNrlpYBSg
5XpT3W9gIC2MXZH2LmzcElSHBaJa
ySPvmwCeswLMalvDV30kA6Ln2zUkNC/n3BAS
xLX8dSGIEB0c7H4hXOeaMvylaEnm
8JlMkarfb/SChr3YtFwrAs3MYYL//1ob7Bp7
fhT/a4wgnSUHgi6ZzOrX9KaEKIKo
Q6vM4KbPAgMBAAECgGEAOWyx7vtykwuAHzhN
5/7SUr/6bnQXaxoJDme015h0bnvG
vryoAojMt5uvZOoNPeBIDhieCRRatAjpgPZuo
jts2ahRZXmPJFFpND2Fx69+shVSE
xPlb1p9HsGk2y4tL6SGW6O++tsbvEt+ugHUu
FNveJazlWcRE2lS28zHa2fOzlX/k
```

```

XdMa7AxZ7/SIBp5uh5ZdHcG35yGPhkG6qfyl
Tg70fsq4Zj+wZ9dVhbDKFzMxgrKM
aeVJE8ySig2TeH8N1K9nKRMSbFEU+WRCVn5/
xfSqOZ/FMo+ayZrEIk9YDZHOSRF
UFJzx+80LhbljZFHTp6KrSWYcPASqcLEaNO4
qa7W0QKBgQDf8Io9sElB+np3v/De
BNufwxl3TtiOfotsKWjzDmFjpt37K5xzs2im
v/s7eBlfZhM4/H4+ruKMepZG/73q
vXZ3V7MM44j/tQZN7M4YooklLY3XRA5TytpW
hKmq6qm6LgRGUzGulXsVi6VJ0eck
+qwLl43p28Z5LkgUo9xXjRmIKwKBgQC2QGIv
Ag5vyLEHE+ob6IlKigaPpUj/AZ7r
/xtxdpNQhATtw42+k1SGNzKMhoXvAxQn3hiI
Jeo/pTkWMwLPXRvHvfg+y4jatT57
TrFTC+QQHskpmnfV8h2hefoyo9bul8gl8D+z
pJBpWPny4DjzGZdrLDvdBK1THsUk
hCccwJ5F7QKBgDblvt30k2DrSI6GrUOGKT73
Ew0edRQpjYBMfn/nLJTDWXOzc0h
5CvMsIgZoAm8+kVKEIbJVJxfiOuK0kHzhFEp
XKlyNimJdSwxOyzq23v/2N/GvURp
XDENGJJ3yHftw/qBdpJ37p6Ph0ube3CjSv3k
f10vHu6iG+WDbgbaflvVAoGBAKu4
YkqUj3G4EUTv+KevJJz9DE2QmQTdTBZk2kDA
TvGQUuyMUyP7wapsm85Yeh3IMteV
pluyDdNGJFHYptrwldhH3RbZmlcWLDqZp4v
GAYwW649gygs5spdGedZBIzuqpBX
/E2Rgxgf4/J6Xm5/8HHkzcrkO0OUPIC5m/i1
bOvJAoGAORsKTjro/uhtpmRQlx26
zIwxUn/+8ptPPxsHif34vNX7k0NRZUKQs66G
lUNxjFqtZdcV5hVEYvtuJv+6QIru
CjM072+deNT3L7FCH+OpRaJwYbigSQL80x3s
5Le+3ligqhbdcKTCY6UrElN6M29M
HZEKL0ec+7agCslTYN1VlEg=
-----END PRIVATE KEY-----

```

Public Key:

```

-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIB
CgKCAQEAn2lUTZ6hqAzf3Tb+riFi
KV1t1v2zFKRzZHOu38KkFsEiwI1lpprlyC0E
MUTdj8upqJlYcjLsorWWLsLhjbzd
B20R7GfBn26cDUH4ugaN8kKb9pEGOYK/QPg0
dASBZ1KopAtGUpxxrAxr1lnBr9BQ
3/RSDuWAlM5pBOOLQ/sHMTa5aWAUoOV6U91v
YCAtjF2R9i5s3BJbBwWiWskj75sA
nrMCzGtbwld9JAoi59s1JDQv59wQLMS1/HUH
iBAH0x+IVznmjL8pWhJ5vCZTJGq
32/0goa92LRcKwLNzGGC//9aG+wae34U/2uM
IJ7FB4Iumczq1/SmhCiCqEOrzOCm
zwIDAQAB
-----END PUBLIC KEY-----

```