



## TEB2093 Computer Security - Lab 07

### Members

- Ammar Farhan Bin Mohamad Rizam (22006911)
- Amisya Fareezan Binti Mohd Fadhil (22007082)
- Ahmad Anas Bin Azhar (22005996)
- Muhammad Hanis Afifi Bin Azmi (22001602)

### Table of Contents

- TEB2093 Computer Security - Lab 07
  - Members
  - Table of Contents
  - Part 1: Understanding Digital Signatures
    - Part 1 Activity 1 - Generating RSA Key Pairs
      - Part 1 Activity 1 Tasks
      - Part 1 Activity 1 Questions
    - Part 1 Activity 2 - Creating a Digital Signature
      - Part 1 Activity 2 Tasks
      - Part 1 Activity 2 Questions
    - Part 1 Activity 3 - Verifying a Digital Signature
      - Part 1 Activity 3 Tasks
      - Part 1 Activity 3 Questions
    - Part 1 Activity 4 - Experimenting with Different Hash Algorithms
      - Part 1 Activity 4 Tasks
      - Part 1 Activity 4 Questions
  - Part 2: Using Digital Signatures in Email Communications
    - Part 2 Activity 1 - Installing and Configuring GnuPG and Thunderbird
      - Part 2 Activity 1 Tasks
      - Part 2 Activity 1 Questions
    - Part 2 Activity 2 - Generating and Importing Keys
      - Part 2 Activity 2 Tasks
      - Part 2 Activity 2 Questions
    - Part 2 Activity 3 - Signing an Email
      - Part 2 Activity 3 Tasks
      - Part 2 Activity 3 Questions
    - Part 2 Activity 4 - Verifying a Signed Email
      - Part 2 Activity 4 Task
      - Part 2 Activity 4 Questions
  - Helper Script
    - Usage



Extract public key:

```
$ openssl rsa -pubout -in private_key.pem -out public_key.pem  
writing RSA key
```

```
$ cat public_key.pem  
-----BEGIN PUBLIC KEY-----  
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAuLujSHfoi7xKEXHi06ya  
jwnFyN/Mh/IAeDI8Nokdm0rYkynLn1kaU5b0Lk014ELJR6agQN2Xd7tUPN2meBIe  
U1D0o3uRRf8oAbXhrt0bAga2k6xdQIZxtT+oBnu4RMDN6MOR6CdHyYwLgB9mw6DN  
9xcFaMnGnYtmER8Ewy9YIwTX1Ixbqtv1jjXhu4qQ4QmDb0BLTSwSx/52c/SmyLXW  
YczMYmqIw3lgl1hg1UUVLv0VnJ5yP8ZVHdxjj6ZsUVCAoFS1+ETNvFPxexqZIma0  
oUgz+T7a8R5U8bbUUmf0zR1nR91ReGqmJfhLbPxUFv+LCfxiuuuuLz47f3zdXECV  
MQIDAQAB  
-----END PUBLIC KEY-----
```

### *Part 1 Activity 1 Questions*

1. Explain the role of the private key and the public key in digital signatures.

Private key is used to create a digital signature by encrypting the hash of a document. Public key is used to verify the authenticity of the digital signature by decrypting the hash and comparing it with the newly generated hash of the document.

2. Why is it important to keep your private key secure?

Private key is essential for signing documents. If it is compromised, an attacker could forge digital signatures in your name, leading to identity fraud and unauthorized access to sensitive data.

## Part 1 Activity 2 - Creating a Digital Signature

### Part 1 Activity 2 Tasks

Save message to message.txt:

```
echo "This is a test message for digital signature" > message.txt
```

Generate SHA-256 Hash & sign message:

```
openssl dgst -sha256 -sign private_key.pem -out message.sig message.txt
```

The generated signature contains unprintable characters. When message.sig is converted to hex:

```
246f018fcd8c2eb45b2645b8ef7980a644d83a3cdcac8eb0b8b810421a9ab72cdd9bef698085989e475e79a
5f869bb72f20fe95e292c6fd5e5e2426dadd30ddd18b76977d27c3037c8cfedde0889d45acb73fa2a4f3dc
c59ac37d0557132ed6e63f4c7dc898e232989b12661a0b6f38573a3e393bab3f7d5297b0e1f417e027d9c79
89a8abb66199a3e67cc46068d143c24ce7b057de25eff2ed94dd985b10dba42c1b03a31b0963c8e6540fa1e
ada028790ebc44ccf60cddf73825ae0171db130f7c99de7a00e52f685248cb7fcd5fc434f0527228db7d330
e3783595e2b9720d6bcf86eabb1fa1bcdd564ecffcb4a09944ea53a3fdf0b8a5f4bf5927daa61
```

To learn more how to convert signature hex to .sig file, or to print .sig file to hex, check out [this section](#).

### Part 1 Activity 2 Questions

1. Describe the purpose of the -sha256 option in the command.

The -sha256 option specifies the SHA-256 hashing algorithm, which is used to create a unique fingerprint (hash) of the message before signing it.

2. What does the message.sig file represent?

The message.sig file contains the digital signature, which is the encrypted hash of message.txt. This signature can be used to verify the authenticity and integrity of the message.

## Part 1 Activity 3 - Verifying a Digital Signature

### Part 1 Activity 3 Tasks

Verify the signature:

```
$ openssl dgst -sha256 -verify public_key.pem -signature message.sig message.txt
Verified OK
```

If the message has been tampered with during delivery:

```
$ echo "You owe me $10 million" > message.txt && openssl dgst -sha256 -verify
public_key.pem -signature message.sig message.txt
Verification failure
400899FC01000000:error:02000068:rsa routines:ossl_rsa_verify:bad
signature:crypto/rsa/rsa_sign.c:442:
400899FC01000000:error:1C880004:Provider routines:rsa_verify_directly:RSA
lib:providers/implementations/signature/rsa_sig.c:1041:
```

### Part 1 Activity 3 Questions

1. What does the verification process confirm about the message and signature?

The verification process confirms two things:

- Authenticity – The message was signed by the expected sender.
- Integrity – The message has not been altered since it was signed.

2. What are some possible reasons for a verification failure?

- Incorrect public key used for verification.
- The message has been altered after signing.
- The digital signature does not correspond to the message.
- Corruption of the signature file (message.sig).

## Part 1 Activity 4 - Experimenting with Different Hash Algorithms

### Part 1 Activity 4 Tasks

Generate a signature using SHA-512:

```
openssl dgst -sha512 -sign private_key.pem -out message_sha512.sig message.txt
```

The generated signature contains unprintable characters. When message\_sha512.sig is converted to hex:

```
a7675de5c3426e1ab9a653a05cc1325b7425949a92423f7523a2b97df86364933d76807cc0215bee7d91a61
700d197746234d11aa2dda4b4ea40f542b3e4d18878fe390ec28da81e2e84e6144cac567cd1343d550213b4
7fc05812c70e663269658c97617a64a74c755d47ba8174f0f7cf469249154aaa8a300770030979a4bba3ca6
5c127856de5969748d7ce11cc0db06af3d89304c1d0ecc42644fabe072af01294419ba91dcfccb131a4b5d1
a6fc798349f12eb00bc62ae23b4fa8a13bea267e17aff19c76ed85f81087661619357d25f87ab05d5c3693b
8b4ab7162dc261c448ebee21eeb16c64709c35acf2dca5692ef155379dc0cb0ba91e8eaedebf7
```

To learn more how to convert signature hex to .sig file, or to print .sig file to hex, check out [this section](#).

Verify the SHA-512 signature:

```
$ openssl dgst -sha512 -verify public_key.pem -signature message_sha512.sig message.txt
Verified OK
```

Discuss any differences observed in file size, performance, and security implications:

- File size

The hash output in SHA-512 is larger (64 bytes) compared to SHA-256 (32 bytes). However, this does not affect the signature file size since the private key encryption step determines the final size. The digital signature files message.sig and message\_sha512.sig remain the same size, as the file size depends on the RSA key length (256 bytes).

- Performance

To determine the performance of each hash function, we can use hyperfine to measure execution time.

```
$ hyperfine --warmup 5 "openssl dgst -sha256 -sign private_key.pem -out message.sig message.txt"
Benchmark 1: openssl dgst -sha256 -sign private_key.pem -out message.sig
```

```

message.txt
  Time (mean ± sigma):   4.4 ms ±   0.9 ms    [User: 3.5 ms, System: 0.7 ms]
  Range (min ... max):   3.3 ms ... 14.3 ms    340 runs

$ hyperfine --warmup 5 "openssl dgst -sha512 -sign private_key.pem -out
message_sha512.sig message.txt"
Benchmark 1: openssl dgst -sha512 -sign private_key.pem -out message_sha512.sig
message.txt
  Time (mean ± sigma):   4.5 ms ±   0.6 ms    [User: 3.6 ms, System: 0.7 ms]
  Range (min ... max):   3.6 ms ... 10.1 ms    439 runs

$ hyperfine --warmup 5 "openssl dgst -sha256 -verify public_key.pem -signature
message.sig message.txt"
Benchmark 1: openssl dgst -sha256 -verify public_key.pem -signature message.sig
message.txt
  Time (mean ± sigma):   3.2 ms ±   0.8 ms    [User: 2.5 ms, System: 0.5 ms]
  Range (min ... max):   2.5 ms ... 16.1 ms    487 runs

$ hyperfine --warmup 5 "openssl dgst -sha512 -verify public_key.pem -signature
message_sha512.sig message.txt"
Benchmark 1: openssl dgst -sha512 -verify public_key.pem -signature
message_sha512.sig message.txt
  Time (mean ± sigma):   3.3 ms ±   0.4 ms    [User: 2.5 ms, System: 0.6 ms]
  Range (min ... max):   2.4 ms ...   5.3 ms    522 runs

```

In theory, SHA-512 requires more processing power due to its longer hash. However, the difference is usually negligible for small messages.

- Security implications

In theory, SHA-512 is more secure than SHA-256 because SHA-512 has 512-bit security, while SHA-256 has 256-bit security. Despite that, SHA-256 is still considered secure, but SHA-512 offers better protection against brute-force attacks.

A collision attack occurs when two different inputs produce the same hash. SHA-512 provides a higher collision resistance. While SHA-256 is still secure today, advances in quantum computing may make SHA-512 a better long-term choice.

### Part 1 Activity 4 Questions

1. What are the advantages of using a stronger hash algorithm like SHA-512?
  - Higher security – SHA-512 produces a longer hash, making it more resistant to brute-force and collision attacks.
  - Better integrity – Less likely to have two different inputs producing the same hash.
  - Future-proofing – More secure against advances in computing power.
2. How does the choice of hash algorithm affect the security of the digital signature?

A weaker hash algorithm, such as MD5, is more vulnerable to attacks, which can compromise the security of the digital signature. A stronger hash algorithm, such as SHA-512, provides better resistance against cryptographic attacks.

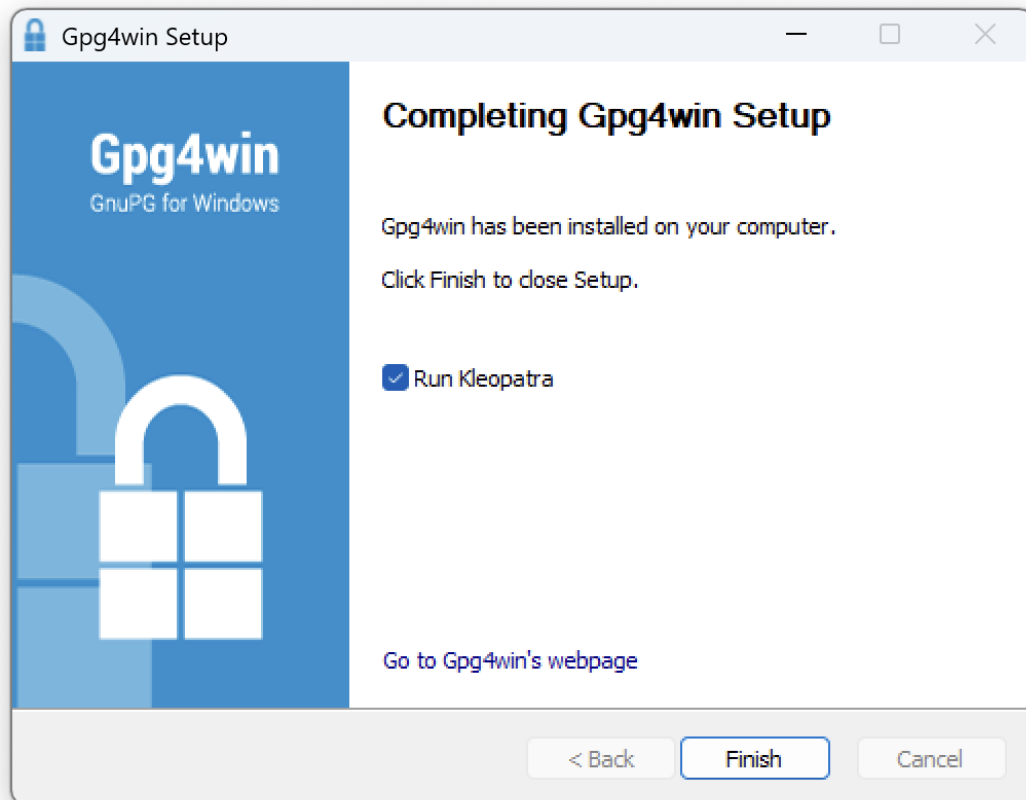


## Part 2: Using Digital Signatures in Email Communications

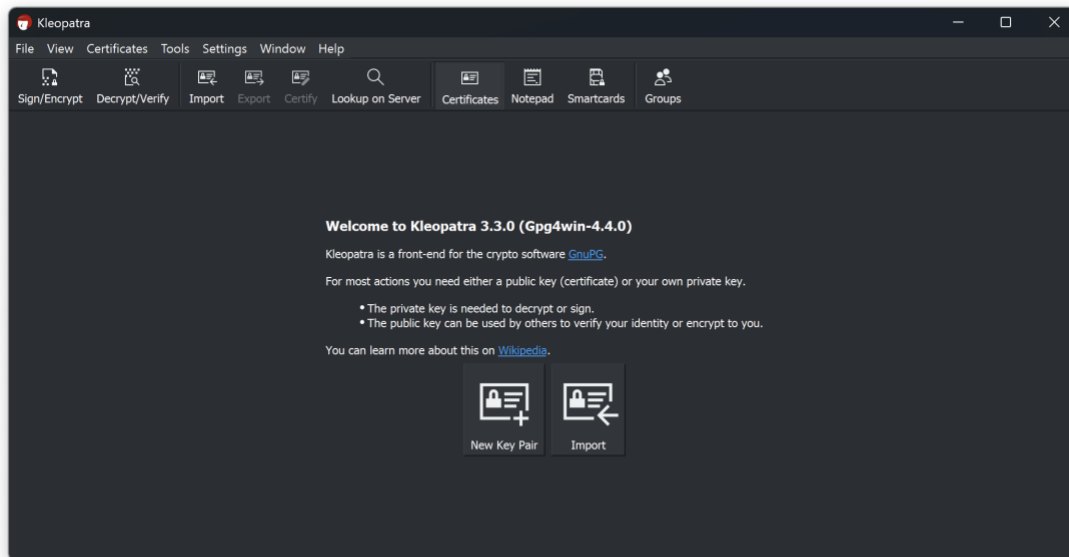
### Part 2 Activity 1 - Installing and Configuring GnuPG and Thunderbird

#### *Part 2 Activity 1 Tasks*

Installing GnuPG:

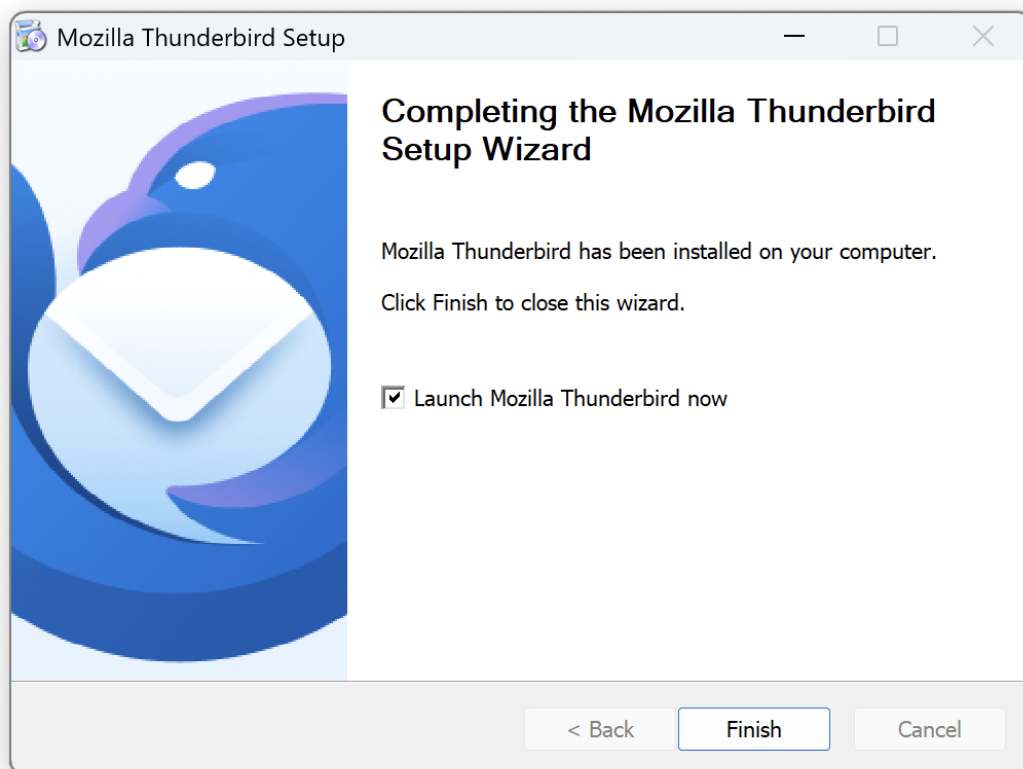


*GnuPG Installation Successful*

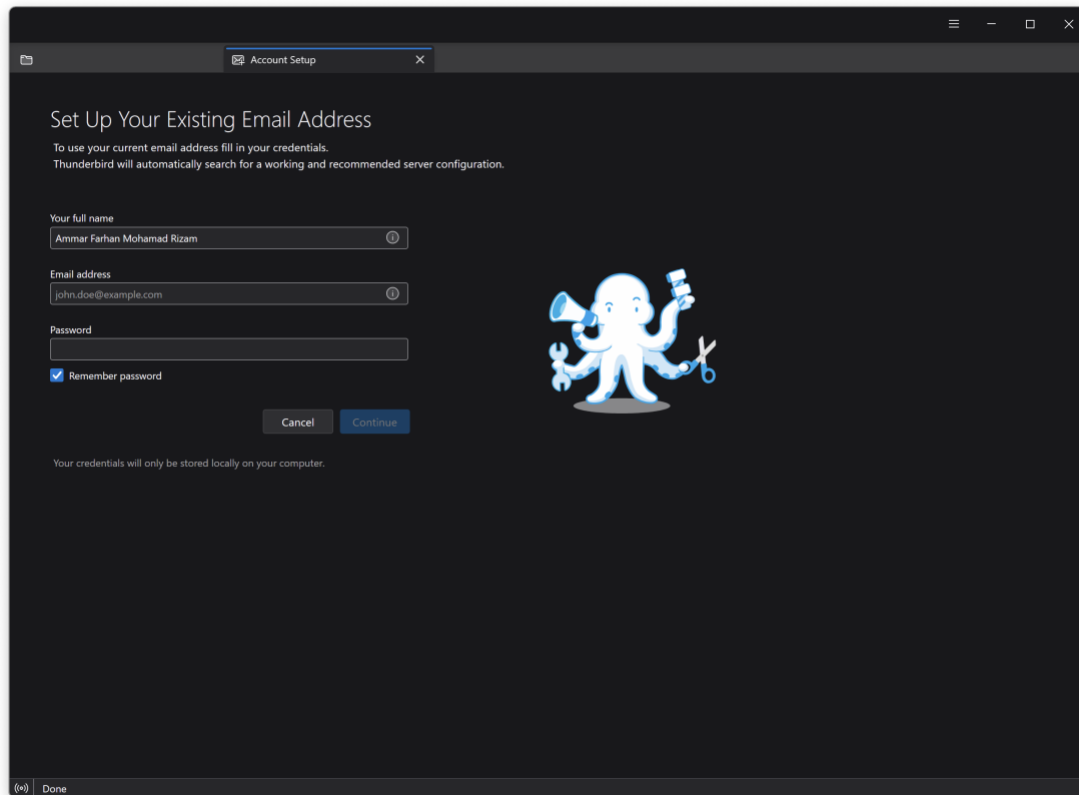


*Kleopatra Running*

Installing Thunderbird:



*Thunderbird Installation Successful*



*Thunderbird Running*

### *Part 2 Activity 1 Questions*

1. What is the purpose of using GnuPG in email communications?

GnuPG (GNU Privacy Guard) provides encryption and signing for email communications, ensuring messages are originally from the sender and they are not tampered with.

2. How does Enigmail enhance the functionality of Thunderbord for secure emails?

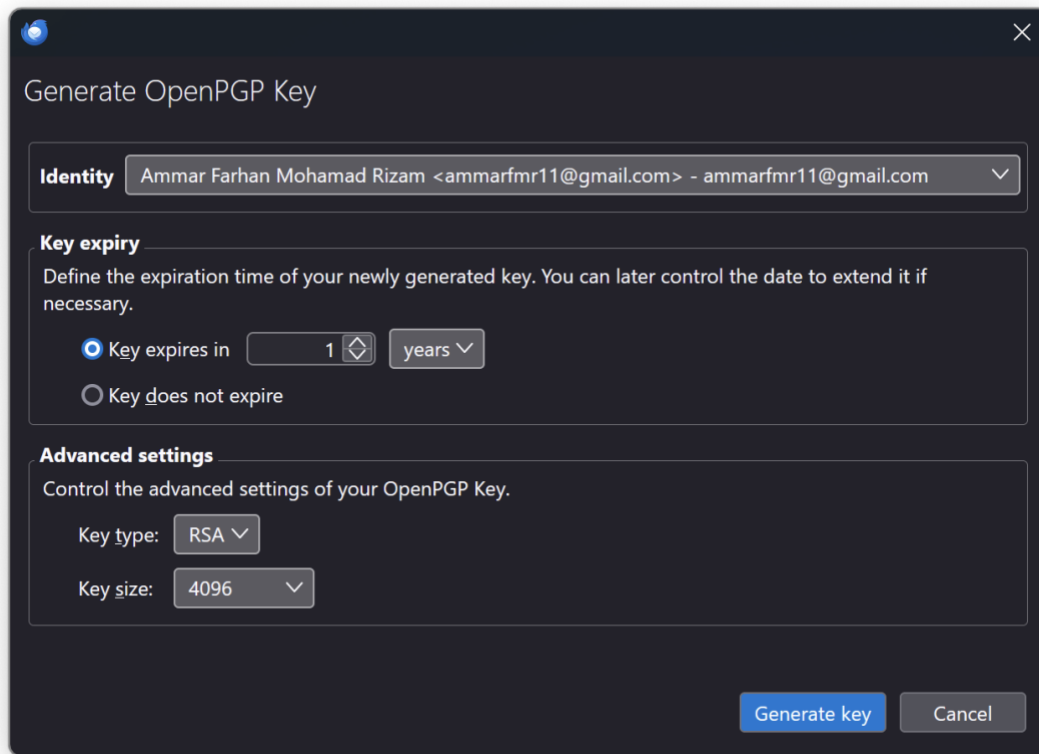
Enigmail integrates GnuPG into Thunderbird. It allows users to encrypt, decrypt, sign, and verify emails directly within Thunderbird. P.S. Microsoft Outlook has a similar feature.

## Part 2 Activity 2 - Generating and Importing Keys

### Part 2 Activity 2 Tasks

At the time of writing this, Enigmail add-on does not support Thunderbird version 128.7. Different steps are taken to achieve the activity objectives.

Generating keys:

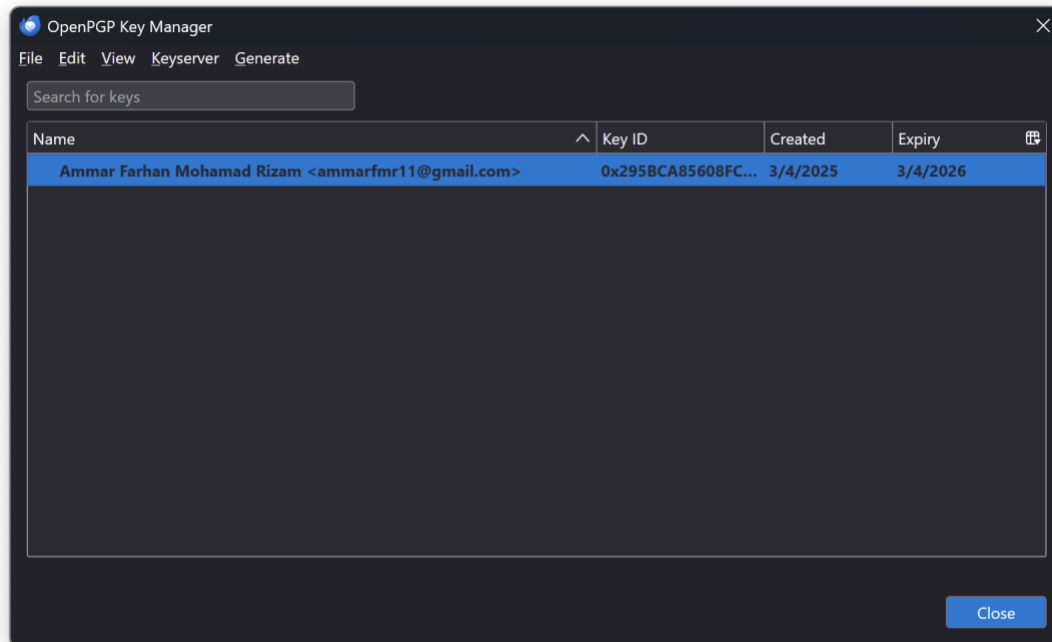


The screenshot shows a dark-themed dialog box titled "Generate OpenPGP Key". It contains the following sections:

- Identity:** A dropdown menu showing "Ammar Farhan Mohamad Rizam <ammarfmr11@gmail.com> - ammarfmr11@gmail.com".
- Key expiry:** A section with the text "Define the expiration time of your newly generated key. You can later control the date to extend it if necessary." It has two radio buttons: "Key expires in" (selected) and "Key does not expire". The "Key expires in" option is followed by a numeric input field set to "1" and a "years" dropdown menu.
- Advanced settings:** A section with the text "Control the advanced settings of your OpenPGP Key." It contains two dropdown menus: "Key type:" set to "RSA" and "Key size:" set to "4096".

At the bottom right, there are two buttons: "Generate key" (highlighted in blue) and "Cancel".

*Setting Up Keys*



*Keys Setup Successful*

Public key generated:

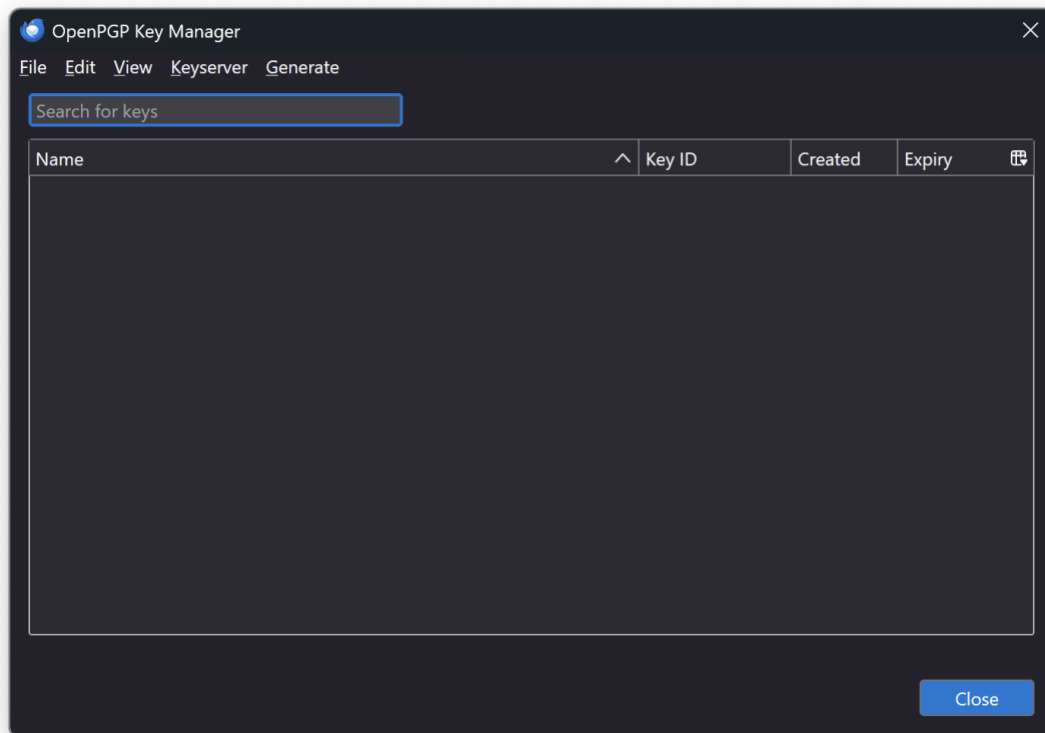
-----BEGIN PGP PUBLIC KEY BLOCK-----

```
xsFNBGfMmNEBEADENmkYk4TDSE9JgaIPh+R9HQguMLH+ANGnKmoXNPvuvRYQ315h
4Bw0DnhDo3ufLpf3DBv5LY0uXS2Bf3a3VKDAGNsRKLTDfzPc+iGPPfajzTzvb0QW
e7sSqV7cQ+Q2TH5mLf0LrsRjhFQ2UKguKxmAQJBUfQkunNo6Kp0021GGsNY5KzR9
bzhHk6Xui30qVYbvG0hH5HaeIjc6+YGxyyiDHv6kx0Y1xpovXeZN6HZp3nXGU96Z
e11a0bZq10Y1d1FYvOXuBICFIz521iElngSkfw6/hNAsldV0pYwwNzQna8c0hwKh
l82CY3BYVzEb9SJlMbBLj7YeAtRkF0ohEvrMjihUq8dti+Rn20BwfH6D+0VcyQjs
7gNlqyImaZiU6lKtm8/FTfLmq2oqjRqyB7ESdSimgX7S4CC7fiRQGSF08ogQmING
yHJfTnToAVb7jRe+BNbZAKGviHvWwJSnPWwkgzoq+0rNRNlzb+BhPnQFdGGe0taW
Sb/Gayk629xSFidW0Tu/sJBnmKGVmldQVHqJxfU6iBZ9WZ+ahP2Wxv0r9BmKgYa
VK9xpzc0llTz0p0GGdC/g0YInK1K6CTaJ7Vqewheu4og9mdmX6cdYHMXVYT+EF/b
TTFJRper10z1LnF2rINPslagwic1dXm9ZkH4e+8+JAYQ46R0LQzmW1u0wARAQAB
zTFBbW1hciBGYXJoYW4gTW9oYW1hZCBSaXphbSA8YW1tYXJmbXlXmUBnbWVpbC5j
b20+wsGHBBMBCAAxFiEE2Kmiqa4T49EW0rDXCf9r6q/mMEFamfMmNECGwMECwKI
BwUVCakKCcUWAgMBAAAKCRAJ/2vqr+YwwbyLEACSGsj0z8m9VC35szJbY47010Xp
7kK/rBqBATp0NmTPLK6MU3Jh91txKcIAMtHHk6WT/rjZVjtneJbs0H1bj0iMasL
1axJl20ZDdX2ryCWS7Aa+vTndOSBL6mAvACxYq94LLT6QLm53k3ii8He1uRP9L1B
LdUht4ZKM9G9E+yzgkogLj1DmPvuhVN0mMSTTJn9m6Y7VIA76AG5uETwpLkjyMys
v6p00YBrHIbYr/qRnCxKQ6RZ+mMC+9W2/KhwUA1He43+jqEdxlr6xicVxf9J0pEp
iL8UF5GqlzaU12odf5UAJz7MR9732rGU1lujo8rkCNTkeD0zmyFbGS6G7XoIKcRG
M8WlSTZ0APMkgePjd0wJPAmAWZddzeC4390T7MKCtT0mwzEPKj4bfzWB4vZVDjSK
P7jReZ+1XPt2I6pb4XfIz5Rrv00LvhwGevim2d20y0uSqIP7g4fE6dNH14ZFBXcA
```

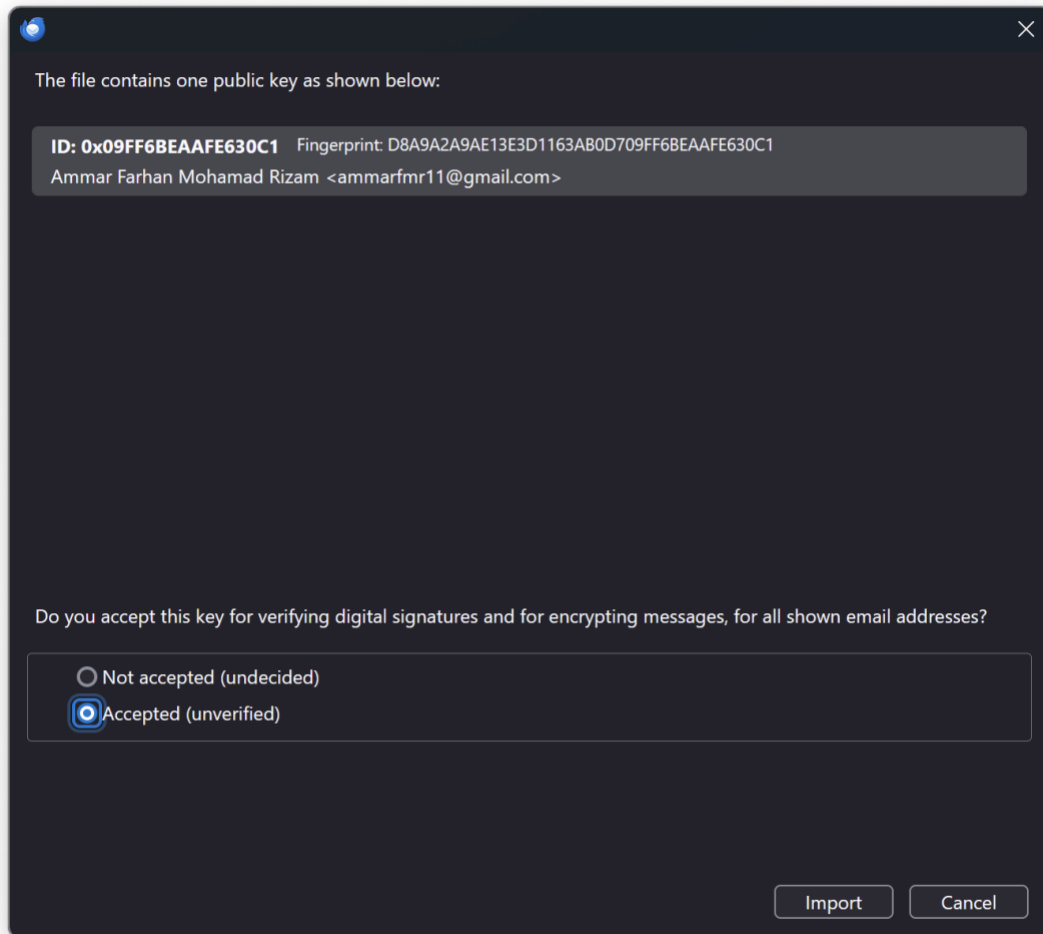
kymYdVFG4wry0jIabExy8iEdHUhiBYTzj1f3FVLpFD0rAFuy5kCFwymXzWEViDoL  
YUyWM9K2l1YPxZUhn9KYJDMV7l1RdI/+pXFAdr5G3U70oNWl1Na2hYCaUYiSVBeJ  
Gwa7C0XP6nFoaKa3K6yWnRtjmo0Lg8HBy6TK/U5GSN1tDgrb2p4RuFMs8mSBbonP  
IaROL0QHEgv78tnDaM7BTQRnzJjSARAAtt5+BUG3/bNTQUGBYfQt4RTI6v7m18Nr  
poqthvJeLZyKBQtoBskh0s2xu0iUdhi2mJg9WUHx+HXA9RxYeGgz/3Kg9YVNlune  
26A2yYnERiq5bnCq7mHb3+HGVB9PbsMMYfvT3QjOHK/jgY2tkQeWw17XFPxFQYU  
bfUJnlrBA0Q/hQN9ITRh0c5Xgc4eVSbF4J3jgKuGmVIqcYJaslku43HDx12wUAR6  
Cb00iEnu80UmkdIxG6JwN6HguZqMv3eDj1I4LND3o0Zfxyxij9W0k0s2bS5UJek+  
1klwGL68MGzgmJnb09ABuS8KxD27p7/Z3Nbm+DDU0xoYbw8aFkUV7jxv4I0d+8wV  
MOOCfhQ9Mm0rcl3H+ciqer8boAdV/Sw9p4Mg723bHK42cWkQ57Cgm+7e8f+YcB6j  
6CJ4pr33I9IviBQ9ITnz3EGC1JBBrwR+RcP/cYvoQhWngEaLRcYvB0NrtEnqbBE8B  
KfX/rbXXddD5sbA4Dg8IW96mW0BDb3RB0cE+Uw2lBKt8bzLEtqXQZEez65cnwwh6  
fZEZEiaL9umTNx0zp/DhQ9DqYhhal+0tOHmLNWGEpaVG+ZizpqNJonhnC9HiiIG3  
ZhHEBc7BVVoQfTS5AGcrB8X+T9uZdX+qGc70jLcyLBGvocTTxgtRBpT55JKs4eNB  
+13ksBlM6/cAEQEAACLBdgQYAQgAIBYhBNipoqmuE+PRFjqw1wn/a+qv5jDBBQJn  
zJjTAhsMAAoJEAn/a+qv5jDBfYMQAJnh9I/jNMu6QxiSDVGgqakXjvNVLQLWIXIx  
ctur5b4FFViv8L7r/n0Mqis4mu8hF/LKBNGvLjAE2I1Gs19iPKlN1dbckIGDLH9E  
2jHw8u4Cfa6ViMZSlXRyynSZB6iryZJF00hoa/vwrXqmWI3k1YuarjAUAH6GCKaM  
7cokF8pGVGngVibvSTc6EP6/+qexkMB8y2tPf27DNST8Pal0IXUdnNC8o8QmzjgV  
A04MK3sDcPMFECpuTnGsgqdbAfAqQa3AwZSxhtQ61525RhXq7LGraUHye6Db6nls  
+6nSg0fGIFpJKu85W8b69lQsCNw7bWZsi4hRDjNkD0BH+XW0orsWCtd//PAm+OSL  
WnckKiKpgFturz24lZf4Nju0Sfa0fJ8h80dLc0gOKktRSGHHWqnZ7p2p0JUzhiqN  
qHR2uLNjYByR2p50na4x/+UcOfdC0aVW3tsnoD9PTPPirmp0uiw8R1W3Af/FcuzV  
8217qFanEJs/BccBH/djTLR8IWkQQZvS7/cWEsEvva/5Vvp+4Khisd6hkXYyBZv0  
eEOw+B1c+1ojos9xvoGI4Yb7VFIfo6PgCGQUmv0Q7D2XBGEYeA/YZKJ4r2NOXAKK  
vqywZI9yltGSEj8KxqU01V6sza96PhqrMq6W5EsIYFrdC7Mfz1GxwKHACHnY7GBU  
qDyRstpY  
=N115

-----END PGP PUBLIC KEY BLOCK-----

Importing public keys:

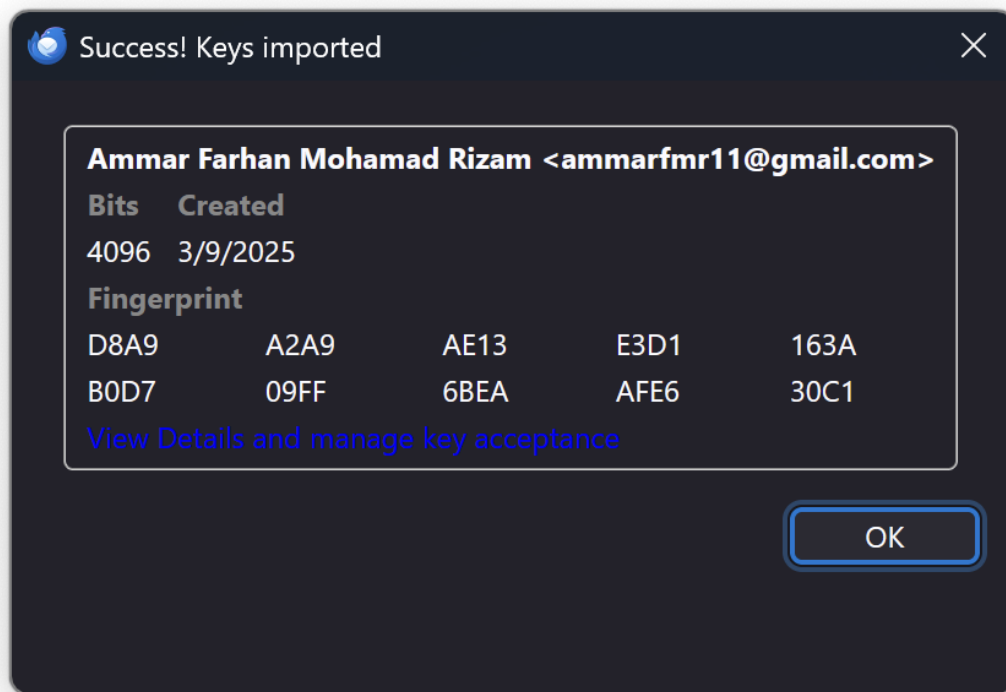


*Before Adding Public Key on Recipient Side*



*Adding Public Key on Recipient Side*





*Added Public Key on Recipient Side*

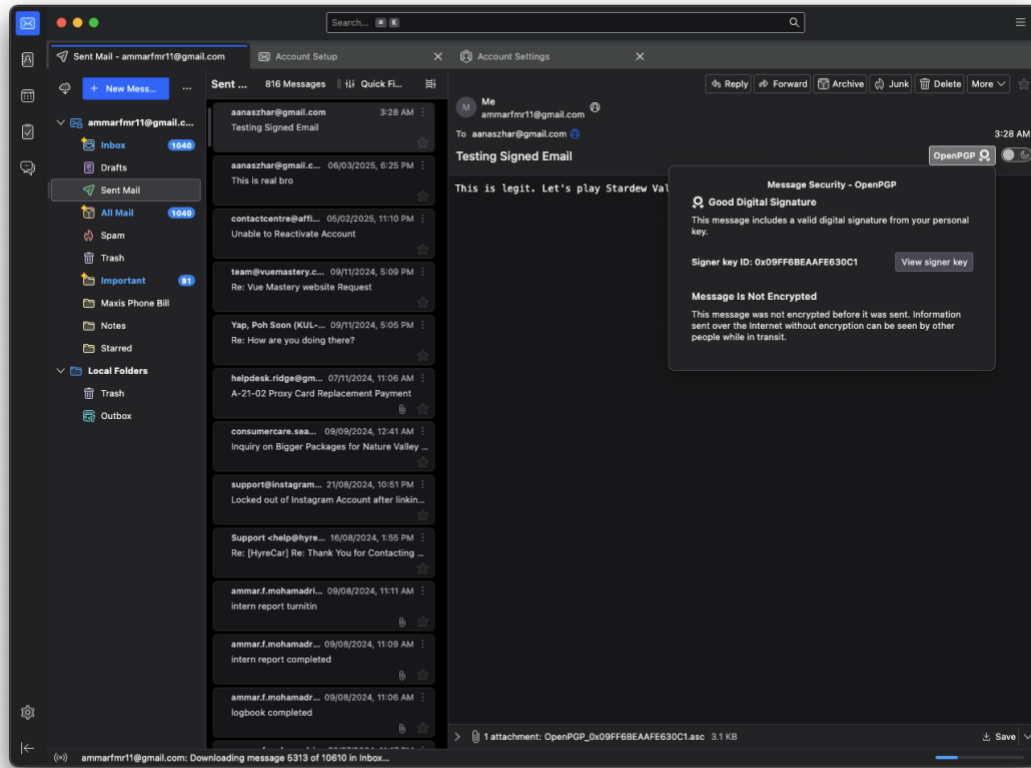
### *Part 2 Activity 2 Questions*

1. Why is key generation important in digital signatures? Key generation creates a unique pair of cryptographic keys (using large random numbers) which are necessary for signing and verifying digital messages securely.
2. Explain the process of importing a public key and its importance. Importing a public key allows a recipient to verify signatures sent by the key's owner. Without importing the correct public key, signature verification will fail.

## Part 2 Activity 3 - Signing an Email

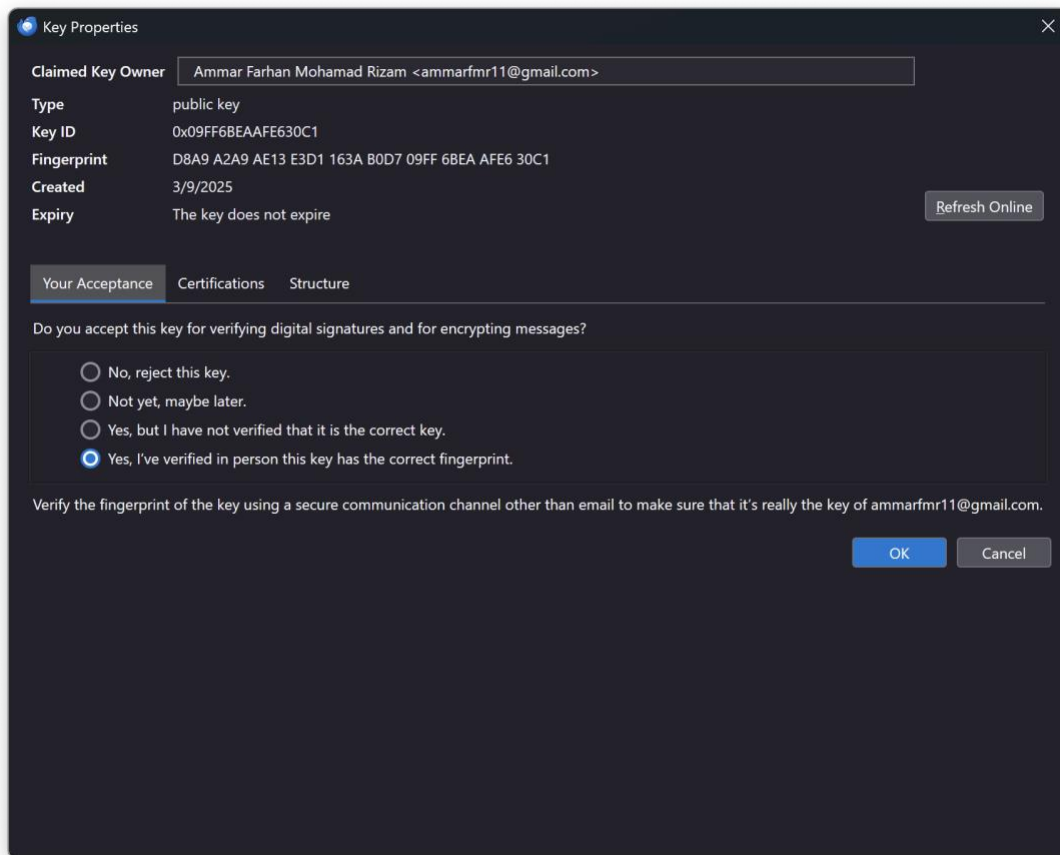
### Part 2 Activity 3 Tasks

Composing a Signed Email:



*Email Signed and Sent by the Key's Owner*

## Verifying the Signature:



### *Verifying Imported Public Key on Recipient Side*

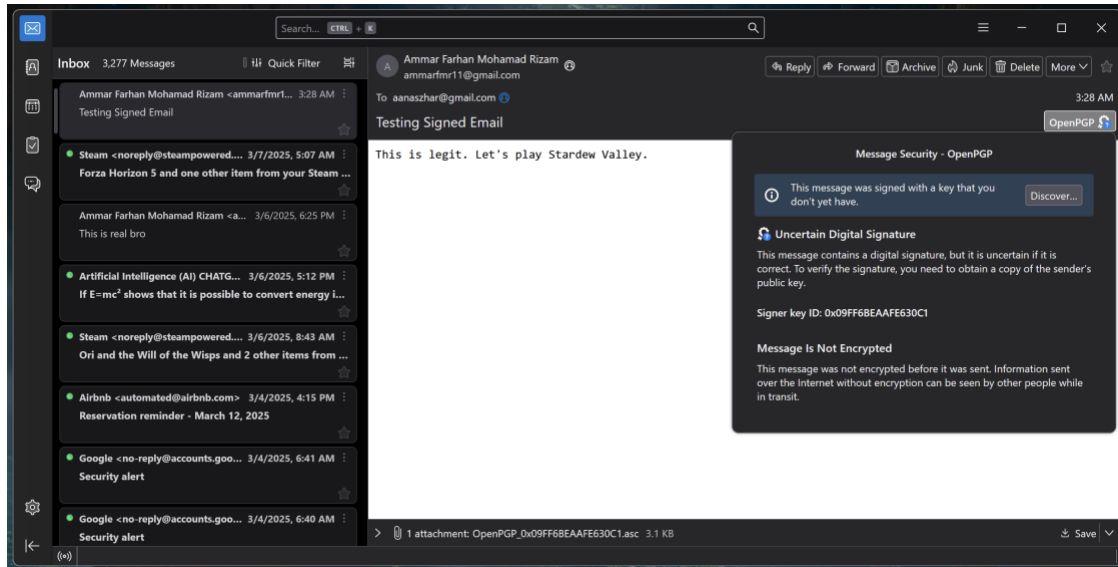
## Part 2 Activity 3 Questions

1. Describe the steps to sign an email in Thunderbird.
  1. Generate private and public keys in Account Settings > End-To-End Encryption > Add key.
  2. Optional: Publish the public key to `vks://keys.openpgp.org` by clicking on the Publish button when your key is selected on the End-To-End Encryption page.
  3. Compose a new email.
  4. Click on the dropdown list for OpenPGP, and check Digitally Sign.
  5. Send the email.
2. What benefits does a signed email provide to the recipient?
  - Authenticity: Confirms the sender's identity.
  - Integrity: Ensures the message has not been altered.
  - Non-repudiation: Prevents the sender from denying the email was sent.

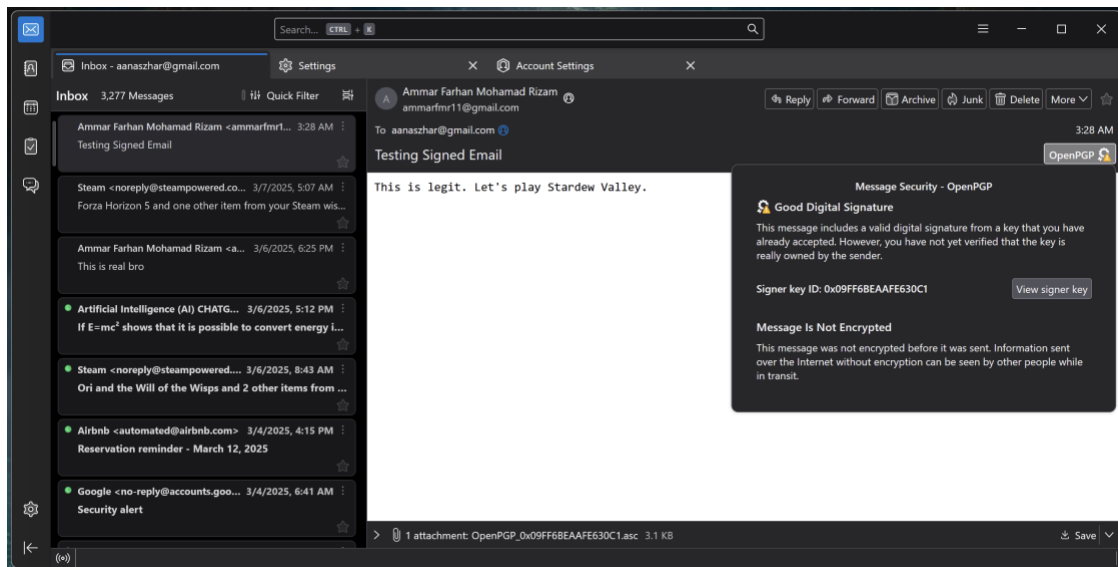
## Part 2 Activity 4 - Verifying a Signed Email

### Part 2 Activity 4 Task

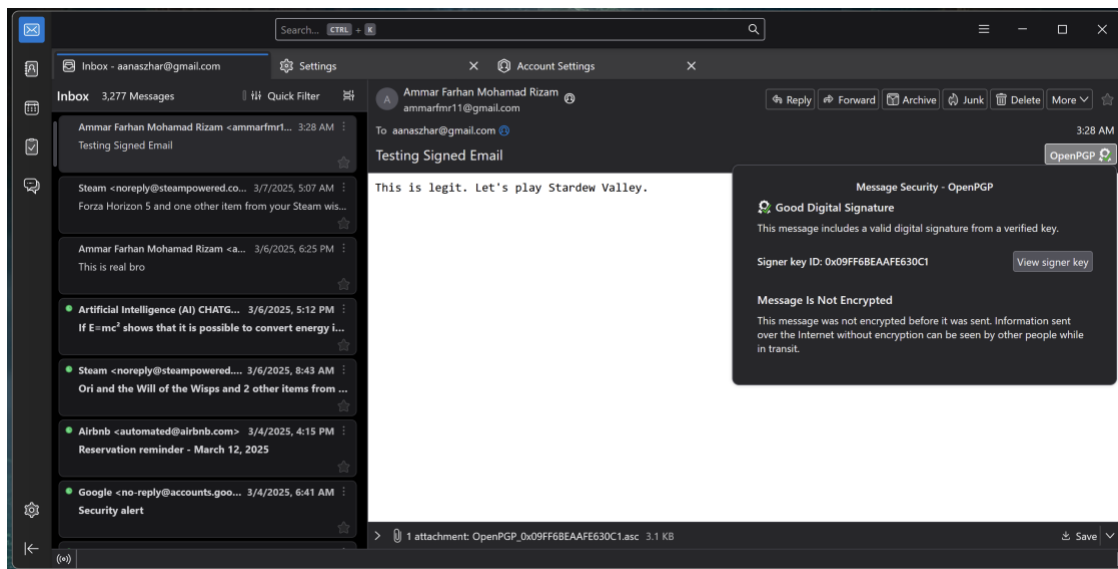
Receiving and Verifying:



*Email Before Adding Public Key*



*Email Before Verifying Public Key*



*Email After Verifying Public Key*

### Part 2 Activity 4 Questions

1. Explain how the verification of a signed email is performed. On Thunderbird version 128, the verification is done automatically. However, to perform manual checks, the recipient uses the sender's public key, decrypts the signature part, and compares the hash of the decrypted hash with the hash of the received message.
2. What are the possible reasons for a verification failure?
  - Recipient has not imported the correct public key.
  - Recipient has not verified the public key.
  - Email has been modified after signing.
  - Signature corruption during transmission.
  - Incorrect hashing algorithm used for verification.

## Helper Script

This script was written to print .sig file to hex, or to convert signature hex to .sig file.

```
#!/usr/bin/env python3
```

```
import argparse
import os
```

```
def is_valid_file(parser: argparse.ArgumentParser, file_path: str) -> str:
    if not os.path.exists(file_path):
        parser.error(f"The file {file_path} does not exist!")
    return file_path
```

```
def parse_arguments() -> argparse.Namespace:
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--signature-file-path",
        required=True,
        metavar="FILE",
        help="signature file to read/write",
    )
    parser.add_argument(
        "--hex-to-file",
        help="convert hex to binary file",
        action="store_true",
    )
    parser.add_argument(
        "--hex",
        help="signature hex to convert to binary",
    )
    args = parser.parse_args()
    if args.hex_to_file and args.hex is None:
        parser.error(f"signature hex needs to be supplied")
    elif args.hex and not args.hex_to_file:
        parser.error(
            f"signature hex is only read when --hex-to-file is chosen")
    elif not args.hex_to_file:
        is_valid_file(parser=parser, file_path=args.signature_file_path)
    return args
```

```
if __name__ == "__main__":
    args = parse_arguments()
```

```
    if not args.hex_to_file:
```

```

    with open(args.signature_file_path, "rb") as signature_file_descriptor:
        print(signature_file_descriptor.read().hex())
else:
    with open(args.signature_file_path, "wb") as signature_file_descriptor:
        print(f"[*] Writing binary to {args.signature_file_path}...")
        signature_hex = args.hex
        if signature_hex.startswith("0x"):
            signature_hex = signature_hex[2:]
        signature_file_descriptor.write(bytes.fromhex(signature_hex))
        print(
            f"[+] Successfully wrote binary to {args.signature_file_path}!")

```

## Usage

Allow executable for the script above using the following command (assuming that you save the above script as signature\_converter.py):

```
chmod +x signature_converter.py
```

To print message.sig file to hex:

```

$ ./signature_converter.py --signature-file message.sig
246f018fcd8c2eb45b2645b8ef7980a644d83a3cdcac8eb0b8b810421a9ab72cdd9bef698085989e475e79a
5f869bb72f20fe95e292c6fd5e5e2426dadd30ddd18b76977d27c3037c8cfedde0889d45acb73fa2a4f3dc
c59ac37d0557132ed6e63f4c7dc898e232989b12661a0b6f38573a3e393bab3f7d5297b0e1f417e027d9c79
89a8abb66199a3e67cc46068d143c24ce7b057de25eff2ed94dd985b10dba42c1b03a31b0963c8e6540fa1e
ada028790ebc44ccf60cddf73825ae0171db130f7c99de7a00e52f685248cb7fcd5fc434f0527228db7d330
e3783595e2b9720d6bcf86eabb1fa1bcdd564ecffcb4a09944ea53a3fdf0b8a5f4bf5927daa61

```

To convert signature hex to custom.sig:

```

$ ./signature_converter.py --signature-file custom.sig --hex-to-file --hex
246f018fcd8c2eb45b2645b8ef7980a644d83a3cdcac8eb0b8b810421a9ab72cdd9bef698085989e475e79a
5f869bb72f20fe95e292c6fd5e5e2426dadd30ddd18b76977d27c3037c8cfedde0889d45acb73fa2a4f3dc
c59ac37d0557132ed6e63f4c7dc898e232989b12661a0b6f38573a3e393bab3f7d5297b0e1f417e027d9c79
89a8abb66199a3e67cc46068d143c24ce7b057de25eff2ed94dd985b10dba42c1b03a31b0963c8e6540fa1e
ada028790ebc44ccf60cddf73825ae0171db130f7c99de7a00e52f685248cb7fcd5fc434f0527228db7d330
e3783595e2b9720d6bcf86eabb1fa1bcdd564ecffcb4a09944ea53a3fdf0b8a5f4bf5927daa61
[*] Writing binary to custom.sig...
[+] Successfully wrote binary to custom.sig!

```