Information Security IA-1

Group No.	Roll Number	Name of student	Security Tool	Module Number of Tool from syllabus
G3	16010122275	Aahan Shetye	GnuPG	Module 1
	16010122269	Vihan Kumbhare		
	16010122280	Rudra Shinde		

Part A: Detailed Project Write-Up on GnuPG

1. Introduction and Background

In today's digital environment, securely exchanging information is crucial. Modern businesses, government agencies, and private individuals rely on strong encryption tools to protect sensitive data—from personal emails to confidential corporate documents. One of the most trusted tools for this purpose is **GnuPG (GNU Privacy Guard)**.

GnuPG, often shortened to "GPG," is a free, open-source software package that implements the **OpenPGP** standard described in RFC 4880. It uses **public-key cryptography** to protect data in transit or at rest, and it enables users to verify the authenticity and integrity of messages and documents through **digital signatures**. Because of its reliability and flexibility, GnuPG is considered a de facto standard for many cryptographic operations on Linux, macOS, and Windows systems.

2. Concepts and Features

Below are key aspects of GnuPG that underpin its widespread usage:

1. **Public-Key Cryptography**
    - **Key Pair**: GnuPG generates two mathematically linked keys—one public key (shared widely) and one private key (kept secret).
    - **Encryption/Decryption**: A message encrypted with a recipient's public key can only be decrypted with that recipient's corresponding private key, ensuring that only the intended recipient can view the information.
2. **Digital Signatures**
    - **Signing**: GnuPG allows you to sign files or messages with your private key. This signature uniquely identifies you as the author (or signer).
    - **Verification**: Recipients use your public key to verify the signature and confirm the message was not altered in transit. This underpins the integrity and authenticity of communications.
3. **Key Management**

    - Users can generate, revoke, and manage their keys through the GnuPG suite.
    - Keys can be annotated (e.g., with user names, emails, comments), set to expire after a certain period, and published to public keyservers to facilitate broad distribution.
4. **Support for Multiple Algorithms**
    - GnuPG supports RSA (Rivest–Shamir–Adleman), DSA (Digital Signature Algorithm), and Elliptic Curve cryptography.
    - Users can select preferred key sizes and algorithms, giving them flexibility in balancing performance and security.
5. **Cross-Platform Compatibility**
    - GnuPG runs natively on Linux and macOS (often preinstalled), and is easily installable on Windows.
    - Multiple graphical front-ends and email integrations exist (e.g., Enigmail for Thunderbird), making it user-friendly beyond the command line.

3. Importance in Information Security

Why is GnuPG integral to modern security practices?

1. **Confidentiality**
    - Confidential data—such as business plans, financial information, or personal details—must be protected from unauthorized access. By encrypting files or emails with GnuPG, you ensure that only the holder of the corresponding private key can access the content.
2. **Authentication and Integrity**
    - Digital signatures prevent data tampering and forgery. Any alteration to a signed document will cause verification to fail, thus immediately indicating potential breaches.
    - This feature is essential for verifying code releases, software packages, and official documents.
3. **Widespread Adoption and Trust**
    - GnuPG's open-source nature means its codebase is publicly reviewable. Security experts worldwide contribute to its scrutiny, patching vulnerabilities and maintaining trust.
    - It forms the backbone for other security tools, including password managers, secure email platforms, and document-signing systems.
4. **Regulatory and Compliance Standards**
    - Many regulations (e.g., GDPR for data protection in the European Union, HIPAA for patient data in the U.S.) highlight encryption as a key measure to protect sensitive information.
    - Organizations often adopt GnuPG to align with best-practice guidelines for data security and cryptography.

4. Real-World Applications

1. **Secure Email**
    - GnuPG works with email clients like Thunderbird via extensions (e.g., Enigmail) to encrypt messages end-to-end. Recipients then decrypt the email on their own devices.
2. **Software Package Signing**
    - Many Linux distributions (e.g., Debian, Ubuntu, Fedora) sign their software packages using GPG.
    - End-users verify these signatures to ensure downloaded programs come from an official source.
3. **Code Signing in Git**
    - Developers can sign commits with GPG, adding a layer of trust to their version-control workflow. This prevents malicious commits from being mistaken for genuine updates.
4. **Encrypted Backups**
    - Sensitive files or databases can be encrypted locally with GnuPG before being transferred to cloud storage, ensuring data is protected at rest and in transit.
5. **Document Integrity**
    - GnuPG can sign important documents (e.g., agreements, memos) so that any changes post-signature are detectable.

5. Project Scope and Objectives

For the **Information Security Internal Assessment 1**, our primary aim is to demonstrate a working knowledge of GnuPG by implementing it in a simplified workflow. Specifically:

1. **Understanding Key Pair Generation**
    - You will learn how to create and manage GPG keys (public and private).
    - You will set an appropriate key type, length, and expiry to suit best practices.
2. **Public Key Exchange**
    - You will distribute your public key to group members (or a designated partner).
    - This mimics real-world scenarios where individuals or parties share public keys so they can send each other encrypted data.
3. **Encrypting and Decrypting Files**
    - As a group, you will practice encrypting documents with a partner's public key.
    - The partner will then decrypt those documents using their corresponding private key.
4. **Signing and Verifying**
    - You will sign documents/files to prove authenticity.
    - Other group members will verify these signatures, learning how to detect tampering or fraudulent files.
5. **GitHub Repository Integration**
    - You will upload relevant scripts, your public keys, and demonstration files to a GitHub Classroom repository.
    - This fosters version control best practices and provides a central location for your instructor to review your work.

6. Detailed Demonstration Plan

A structured plan to show our proficiency:

1. **Key Generation Demonstration**
    - Demonstrate in class or via screenshots how you generate a new key pair using GnuPG's `--full-generate-key` command.
    - Explain your choice of algorithm (e.g., RSA 4096 bits) and passphrase considerations.
2. **Public Key Import/Export**
    - Show how to export your public key in ASCII armor format (`.asc`).
    - Let your teammate import that key, verifying it with GnuPG's `--list-keys` command.
3. **Encryption/Decryption of a Test File**
    - Provide a simple "Hello, this is a secret project file" text file.
    - Encrypt it with your teammate's public key, then have them decrypt it with their private key (show the correct passphrase usage).
4. **Signing and Verification**
    - Sign a separate file or message to illustrate how recipients can ensure it wasn't tampered with.
    - Peers in your group verify the signature using your public key, confirming both authenticity and integrity.
5. **Scripts and Documentation**
    - (Optional) Create small shell scripts (e.g., `encrypt_file.sh`) that automate encryption/decryption for demonstration or class usage.
    - Provide thorough comments in your scripts and a `README.md` in the GitHub repository explaining each step.
6. **Security Considerations**
    - Discuss passphrase complexity and why you selected a particular key size.
    - Highlight what happens if a private key is compromised, and show how a revocation certificate would be used.

7. Benefits and Outcomes of Your Implementation

By the end of this project, our group was able to:

1. **Confidently Use GnuPG Commands**
    - Perform key management, encryption, decryption, signing, and verification from the command line (or via a GUI).
2. **Appreciate Real-World Cryptographic Concepts**
    - Grasp how public-key cryptography underpins secure communication on the internet.
3. **Practice Secure Collaboration**
    - Exchange data in an encrypted fashion, establishing trust that your messages are safe from eavesdroppers and tampering.
4. **Build a Foundation for Advanced Security Topics**

- GnuPG is just one piece of the broader cryptographic puzzle. With a firm grasp of it, you are better prepared for advanced encryption solutions, certificate authorities, and PKI (Public Key Infrastructure).

8. Challenges and Potential Solutions

1. **Entropy Shortages**
   - Generating strong cryptographic keys may require substantial system entropy. On some systems, you may need to perform keyboard or mouse interactions to speed up random data generation.
2. **Key Compromise / Revocation**
   - If a private key is suspected to be compromised, immediate revocation is critical. Creating a revocation certificate at the outset is best practice.
   - You can demonstrate how a key is revoked and re-issued in your project (if time permits).
3. **Trust Model**
   - In large organizations or open communities, the concept of a "web of trust" emerges, where people cross-sign keys to vouch for each other's identities.
   - You can highlight the difference between a hierarchical Certificate Authority (CA) model (like those used for HTTPS) and the decentralized trust web used by OpenPGP.
4. **User Adoption**
   - Command-line tools can be intimidating. While GnuPG has GUIs, you may highlight how you overcame or simplified any user interface hurdles (scripts, instructions, etc.).

9. Conclusion and Next Steps

GnuPG stands out as a cornerstone in the realm of open-source security tools due to its **robust cryptographic functionality**, **open and transparent development**, and **wide adoption** across various platforms. By implementing GnuPG in your Internal Assessment project, you and your team will gain **hands-on experience** with the fundamentals of encryption and digital signatures—skills that are increasingly essential in every sector of IT and cybersecurity.

Your demonstration will highlight how a simple workflow—**key generation**, **public key exchange**, **file encryption/decryption**, and **digital signing/verification**—can protect sensitive data and ensure authenticity. With these insights, you can further explore:

- **Advanced GnuPG configurations** (using subkeys, custom trust policies, etc.).
- **Integration into CI/CD pipelines** for code signing.
- **Expanding a personal 'web of trust'** by cross-signing keys.

Ultimately, mastering GnuPG provides a solid foundation in modern cryptographic practices and sets you on the right path for deeper cybersecurity studies.

Reference Materials & Further Reading

- Official GnuPG Website
- GNU Privacy Handbook
- OpenPGP Standard (RFC 4880)
- Email Self-Defense Guide by FSF

Part B: Implementation of GnuPG

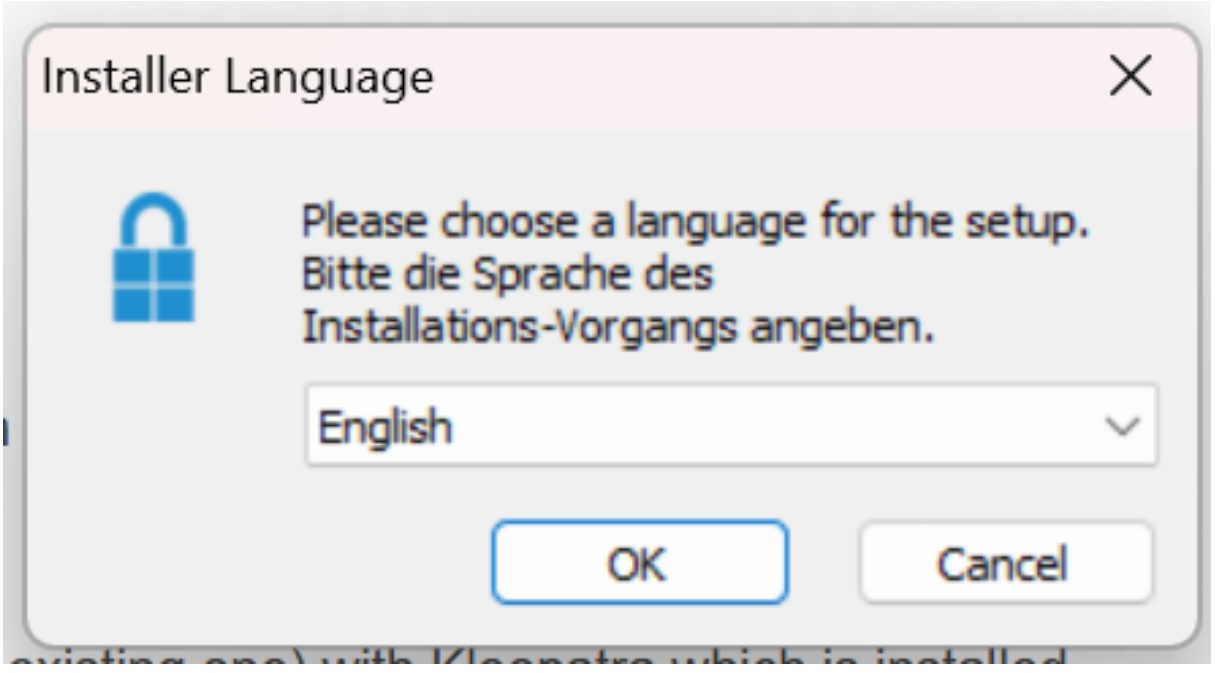
In this section, we describe **how our team implemented GnuPG** to meet our project objectives of securing files and demonstrating public-key cryptography in action. These steps capture the actual procedures we followed, supported by screenshots (where indicated), and reflect our experience on Windows, macOS, and Linux platforms.

---

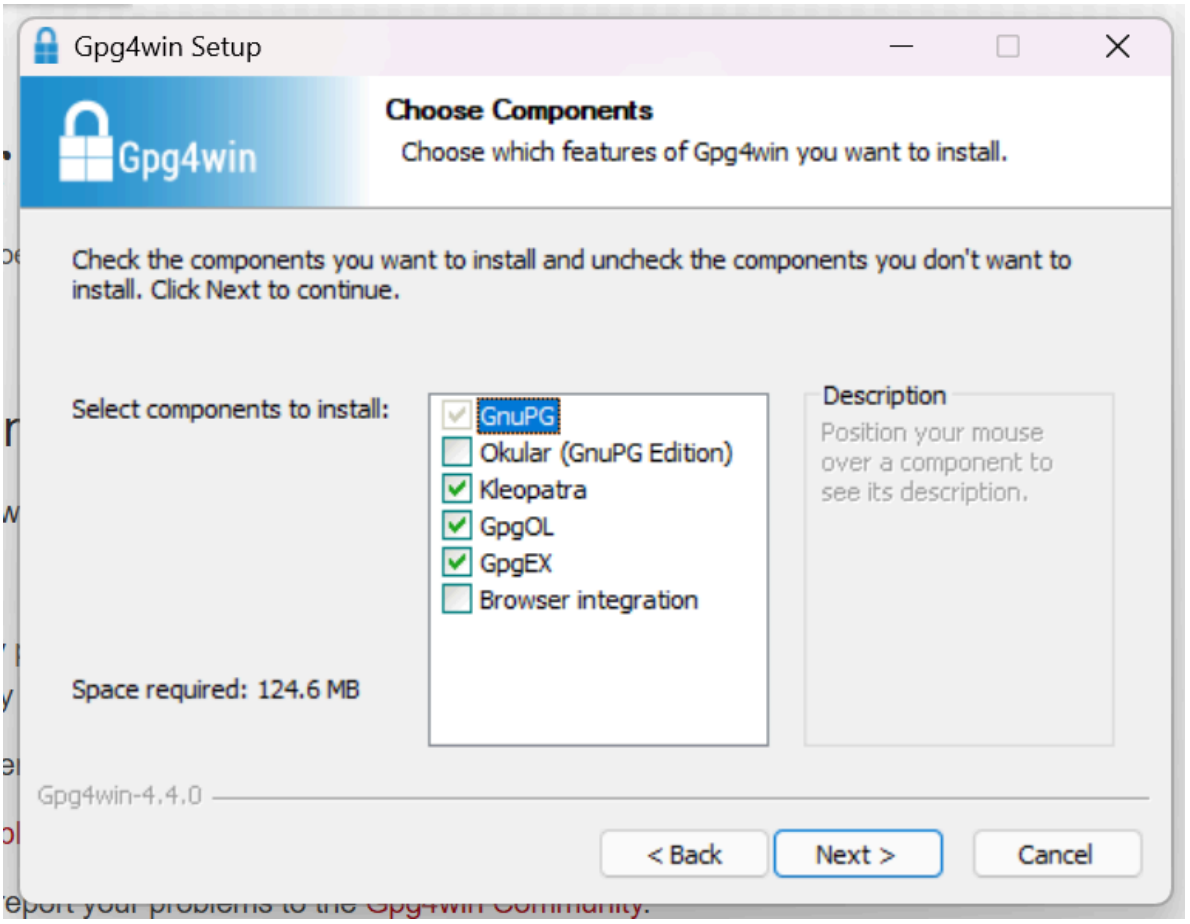
1. Installation of GnuPG

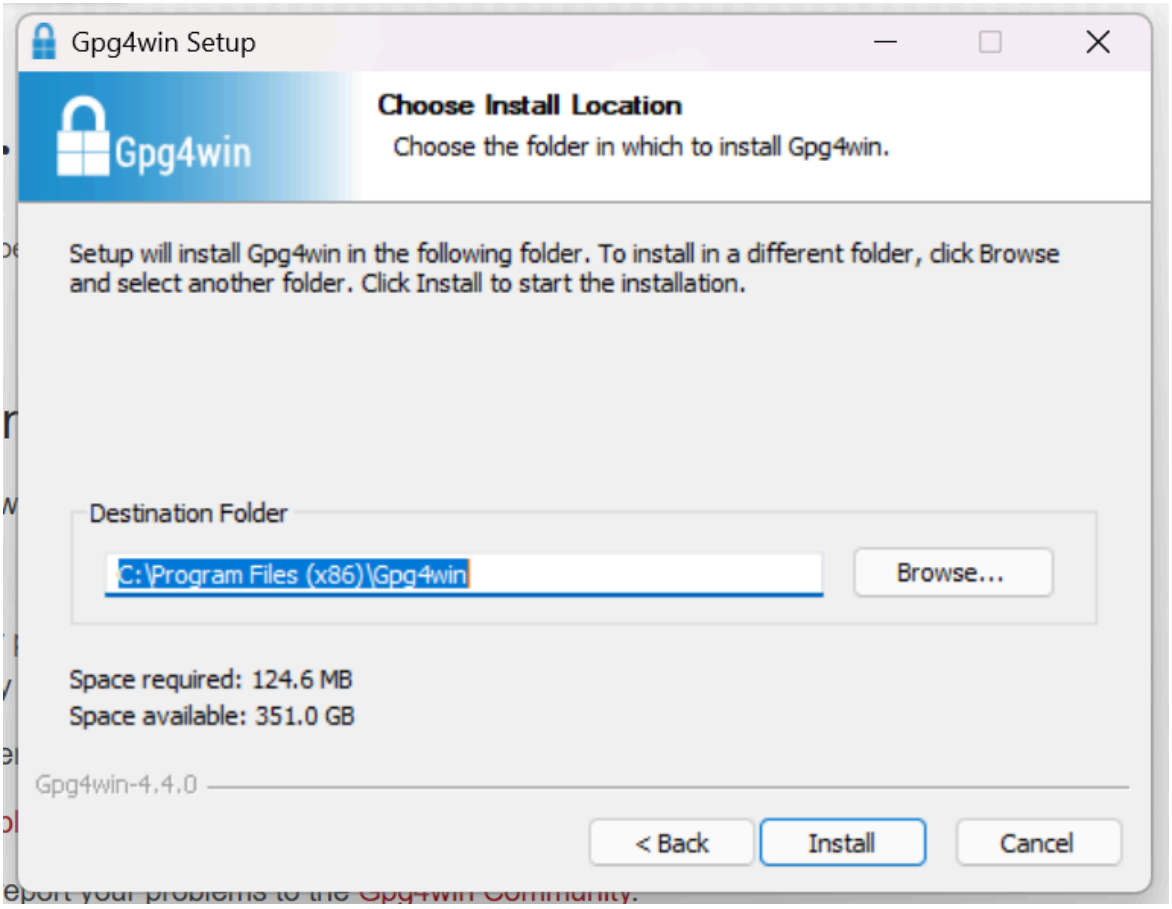
1.1. Windows Environment

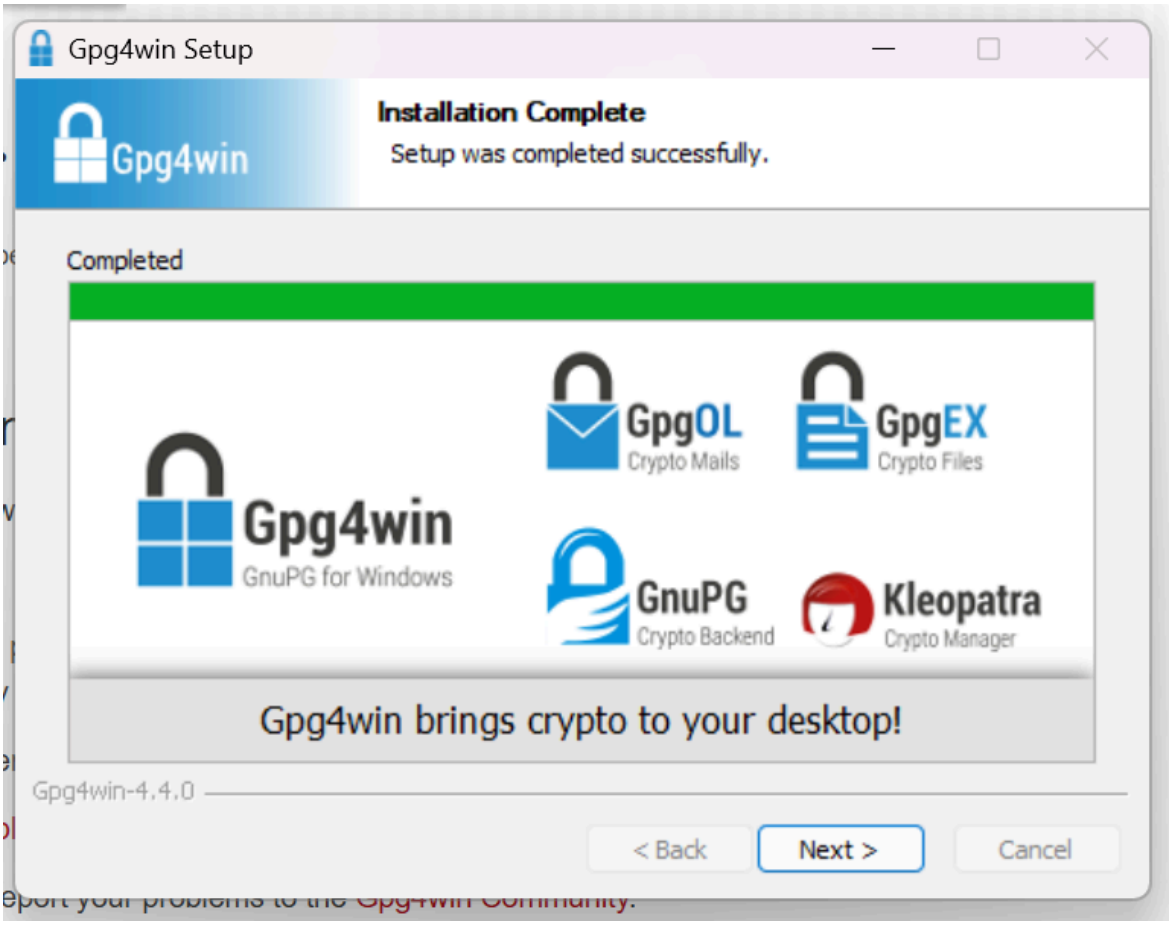
1. **Download and Install Gpg4win**
   We visited the official Gpg4win website and downloaded the latest installer (`.exe`). During installation, we chose the default components (GnuPG, Kleopatra, etc.).
   - *Screenshot 1*: Gpg4win setup wizard dialog.




Gpg4win Setup
Gpg4win
GnuPG for Windows
Welcome to the installation of Gpg4win
Gpg4win is an installer package for Windows for EMail and file encryption using the core component GnuPG for Windows. Both relevant cryptography standards are supported, OpenPGP and S/MIME. Gpg4win and the software included with Gpg4win is Free Software.
Click Next to continue.
This is Gpg4win version 4.4.0
Release date 2024-11-27
Next >
Cancel


Gpg4win Setup
Gpg4win
Choose Components
Choose which features of Gpg4win you want to install.
Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.
Select components to install:
GnuPG
Okular (GnuPG Edition)
Kleopatra
GpgOL
GpgEX
Browser integration
Description
Position your mouse over a component to see its description.
Space required: 124.6 MB
Gpg4win-4.4.0
< Back
Next >
Cancel


Gpg4win Setup
Gpg4win
Choose Install Location
Choose the folder in which to install Gpg4win.
Setup will install Gpg4win in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.
Destination Folder
C:\Program Files (x86)\Gpg4win
Browse...
Space required: 124.6 MB
Space available: 351.0 GB
Gpg4win-4.4.0
< Back
Install
Cancel


Gpg4win Setup
Gpg4win
Installation Complete
Setup was completed successfully.
Completed
Gpg4win
GnuPG for Windows
GpgOL
Crypto Mails
GpgEX
Crypto Files
GnuPG
Crypto Backend
Kleopatra
Crypto Manager
Gpg4win brings crypto to your desktop!
Gpg4win-4.4.0
< Back
Next >
Cancel

2. **Verify Installation**
   After installation, we opened the Command Prompt and typed:
   ```
   gpg --version
   ```

   A response showing the installed GnuPG version confirmed the setup.

   - *Screenshot 2*: Output displaying GnuPG version in Command Prompt.

```
C:\Users\Rudra>gpg --version
gpg (GnuPG) 2.4.7
libgcrypt 1.11.0
Copyright (C) 2024 g10 Code GmbH
License GNU GPL-3.0-or-later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Home: C:\Users\Rudra\AppData\Roaming\gnupg
Supported algorithms:
Pubkey: RSA, ELG, DSA, ECDH, ECDSA, EDDSA
Cipher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH,
        CAMELLIA128, CAMELLIA192, CAMELLIA256
Hash: SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compression: Uncompressed, ZIP, ZLIB, BZIP2
```

1.2. macOS Environment

1. **Homebrew Installation (if needed)**
   We ensured Homebrew was installed by running `brew --version`. If Homebrew was not present, we installed it from https://brew.sh/.
2. **Install GnuPG**
   Using the Terminal app:
   ```
   brew install gnupg
   ```
   - *Screenshot 3*: Terminal showing the installation process completion.
3. **Verification**
   Once installation completed, we confirmed success by running:
   ```
   gpg --version
   ```
   - *Screenshot 4*: Terminal output confirming GnuPG version.

1.3. Linux Environment

Most Linux distributions come with GnuPG preinstalled. However, we verified (and installed if necessary) using the distribution's package manager:

- **Debian/Ubuntu**:
  ```
  sudo apt-get update
  sudo apt-get install gnupg
  ```
- **Fedora/CentOS/RHEL**:
  ```
  sudo dnf install gnupg
  ```
- **Arch Linux**:
  ```
  sudo pacman -S gnupg
  ```

After installation, we typed:

```
gpg --version
```

to confirm GnuPG was available and properly installed.

- *Screenshot 5*: Terminal on Linux verifying GPG installation.

---

2. Generating a New Key Pair

Once GnuPG was set up on each OS, we proceeded to **generate a key pair**. Our intent was to create a unique public/private key pair for encryption, decryption, signing, and verification throughout the project.

1. **Key Generation Command**
   We opened the respective terminal (Command Prompt/ on Windows, Terminal on

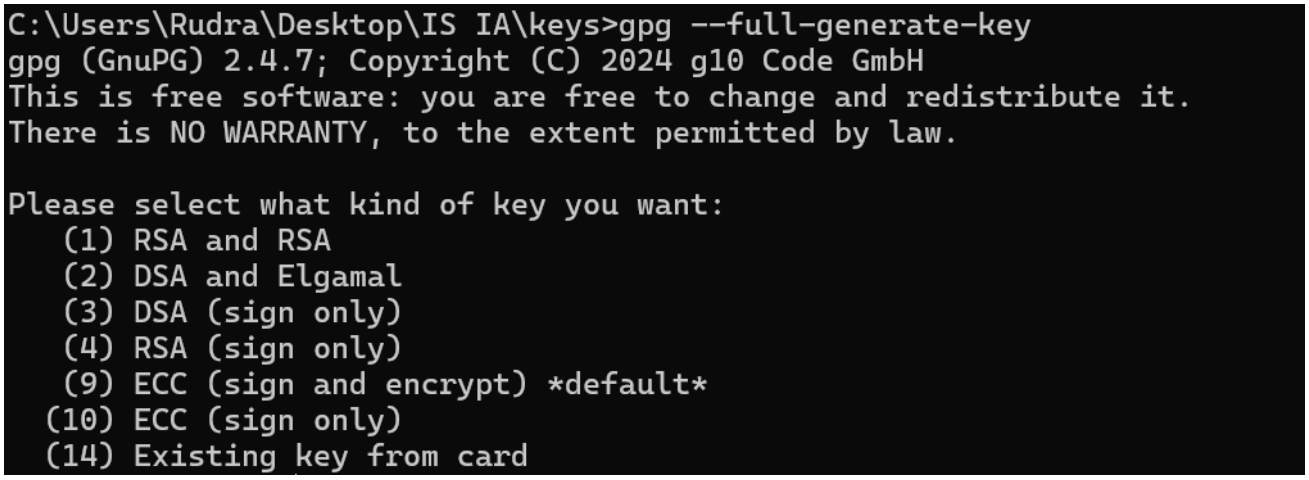
macOS/Linux) and ran:

```
gpg --full-generate-key
```

2. **Selecting Key Type**
   We chose the recommended default option "(1) RSA and RSA," providing a robust general-purpose approach.
3. **Key Length and Expiration**
   - We opted for 4096-bit RSA keys to maximize security.
   - We set our keys to expire after one year, aligning with recommended key rotation practices.
4. **User Identity**
   We were prompted for our real name, email address, and an optional comment. For clarity in our demonstration, we used a format like

   ```
   Real Name: Alice Example
   Email Address: alice@example.com
   Comment: Key for Security IA Project
   ```

5. **Passphrase Protection**
   We selected a strong passphrase to protect our private key. We ensured not to reveal this passphrase in any screenshots or project documents.
6. **Confirmation**
   GnuPG generated the keys after gathering sufficient random data (entropy).
   - *Screenshot 6*: Terminal prompts and final output displaying key generation success.



```
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (3072) 4096
Requested keysize is 4096 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 1y
Key expires at 02/09/26 13:49:02 India Standard Time
Is this correct? (y/N) y
```

```
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: directory 'C:\\Users\\Rudra\\AppData\\Roaming\\gnupg\\openpgp-revocs.d' created
gpg: revocation certificate stored as 'C:\\Users\\Rudra\\AppData\\Roaming\\gnupg\\openpgp-revocs.d\\E52BA48FCD95A77DA106
4C65A8A58FF953069A85.rev'
public and secret key created and signed.

pub   rsa4096 2025-02-09 [SC] [expires: 2026-02-09]
      E52BA48FCD95A77DA1064C65A8A58FF953069A85
uid                      Alice Example (Key for Security IA Project) <alice@example.com>
sub   rsa4096 2025-02-09 [E] [expires: 2026-02-09]
```

We could then verify that our new key was created:

```
gpg --list-keys
```

- *Screenshot 7*: Terminal output listing the newly created public key.

```
C:\Users\Rudra\Desktop\IS IA\keys>gpg --list-keys
[keyboxd]
---------
pub   rsa4096 2025-02-09 [SC] [expires: 2026-02-09]
      E52BA48FCD95A77DA1064C65A8A58FF953069A85
uid           [ultimate] Alice Example (Key for Security IA Project) <alice@example.com>
sub   rsa4096 2025-02-09 [E] [expires: 2026-02-09]
```

---

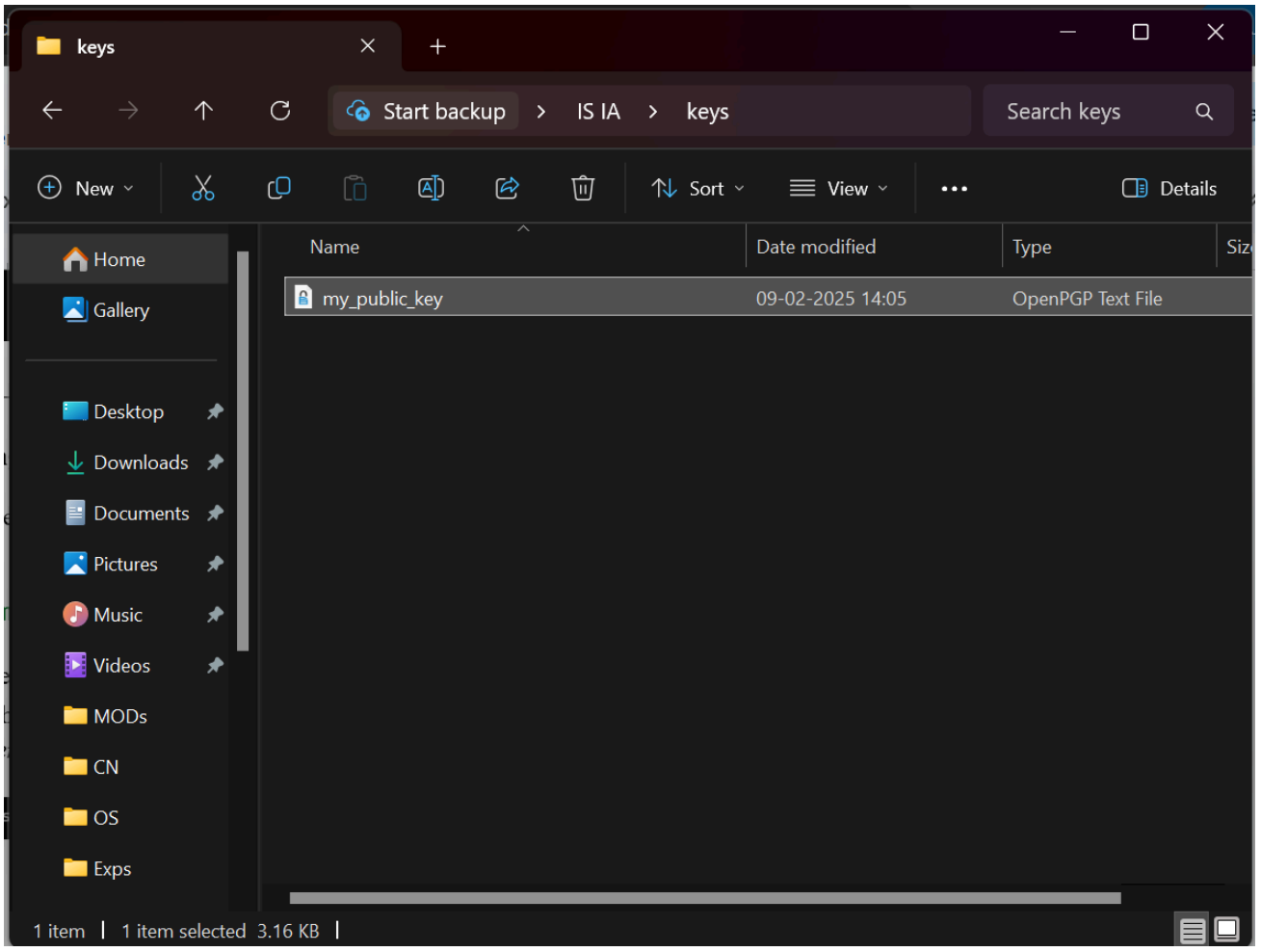
3. Exporting Our Public Key

To allow others to encrypt data for us (or verify our signatures), we needed to **export our public key**:

```
gpg --armor --export "alice@example.com" > my_public_key.asc
```

- We replaced `"alice@example.com"` with the actual email used during key generation.
- We shared the resulting `my_public_key.asc` file with our team.
- *Screenshot 8*: Terminal showing the successful export of our public key.

```
C:\Users\Rudra\Desktop\IS IA\keys>gpg --armor --export "alice@example.com" > my_public_key.asc
```



Note: We emphasized in our documentation that **the private key must never be shared**.

---

4. Importing a Teammate's Public Key

Each team member repeated the export process and provided their `.asc` file. We then imported these keys into our own keyrings:

```
gpg --import teammate_public_key.asc
```

- *Screenshot 9*: Output listing the newly imported key in our keyring.

```
C:\Users\Rudra\Desktop\IS IA\keys>gpg --import bob_public_key.asc
gpg: key D691FCBB8E0EF5E1: "Bob Example (Key for Security IA Project) <bob@example.com>" not changed
gpg: Total number processed: 1
gpg:              unchanged: 1
```

At this point, we had each other's public keys, enabling **secure file exchange**.

---

5. Encrypting Files

Our next step was to demonstrate **confidentiality** by encrypting a file using a teammate's public key:

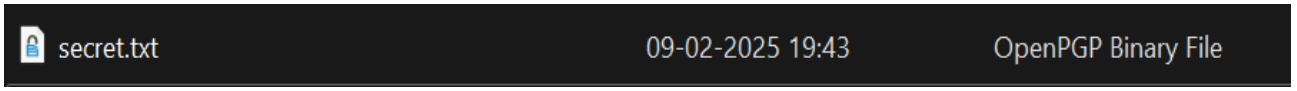
1. We created a sample text file, `secret.txt`, containing:
   `This is a confidential message for the IA project demonstration.`
2. We ran:
   `gpg --encrypt --recipient teammate@example.com secret.txt`
3. GnuPG produced `secret.txt.gpg`, which was fully encrypted.
   - *Screenshot 10*: Terminal command and resulting encrypted file.

```
C:\Users\Rudra\Desktop\IS IA\scripts>gpg --encrypt --recipient bob@example.com secret.txt
gpg: DC3A04EFA5229A78: There is no assurance this key belongs to the named user

sub  rsa4096/DC3A04EFA5229A78 2025-02-09 Bob Example (Key for Security IA Project) <bob@example.co
m>
 Primary key fingerprint: DEF5 F6F7 14E9 05E3 42D5  A096 D691 FCBB 8E0E F5E1
      Subkey fingerprint: A4CC BCCD 900C 3F5A CD9C  F455 DC3A 04EF A522 9A78

It is NOT certain that the key belongs to the person named
in the user ID.  If you *really* know what you are doing,
you may answer the next question with yes.

Use this key anyway? (y/N) y
```



We then shared `secret.txt.gpg` with our teammate via email or a secure channel. Because it was encrypted with the teammate's public key, **only** that teammate could decrypt it using their **private** key.

---

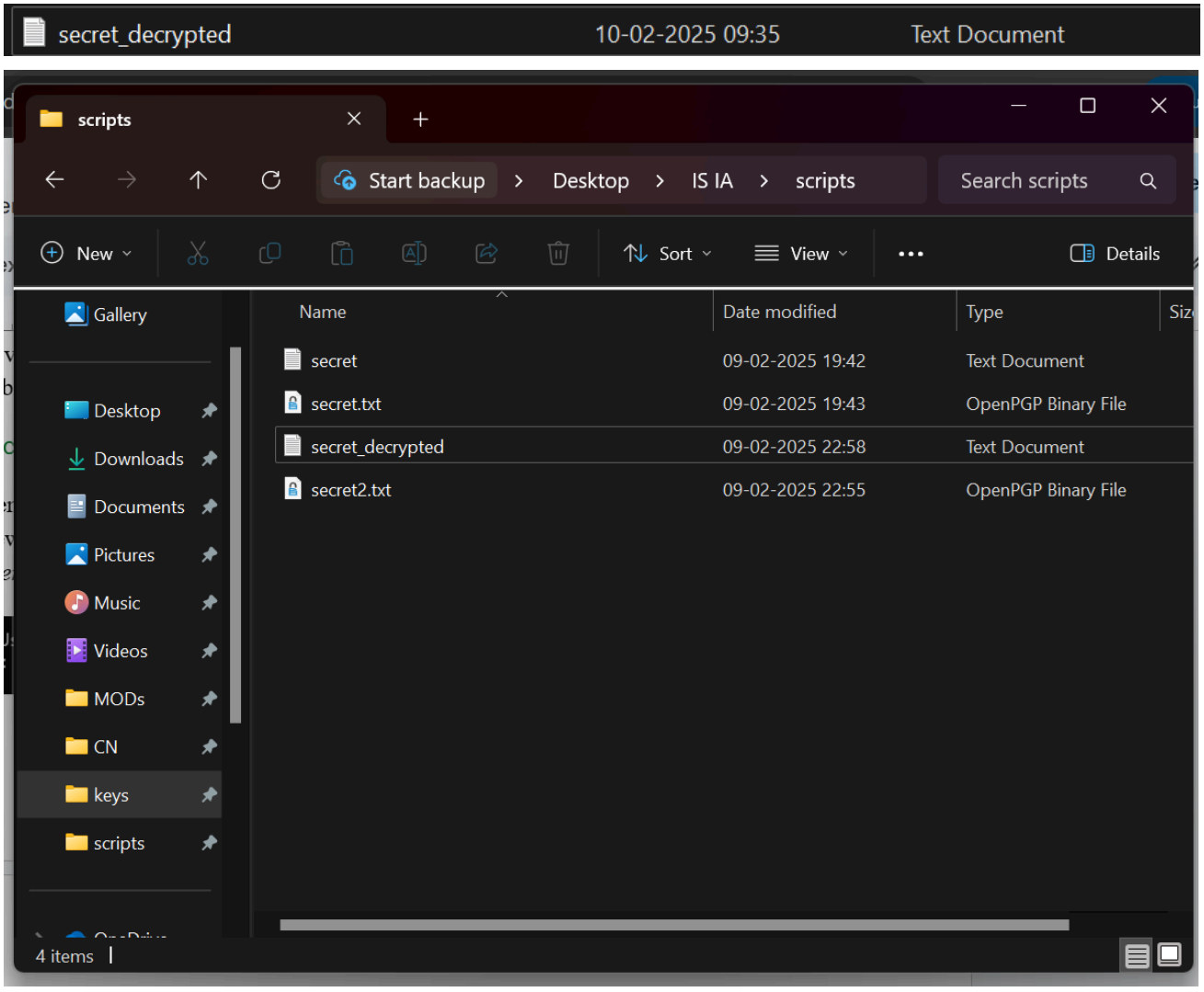
6. Decrypting Files

On the receiving end, if we received an encrypted file (e.g., `secret.txt.gpg`) that was encrypted with **our** public key, we decrypted it with our private key:

```
gpg --decrypt secret.txt.gpg > secret_decrypted.txt
```

- We entered our **private key passphrase** when prompted.
- A new file, `secret_decrypted.txt`, appeared in plaintext form.
- *Screenshot 11*: Terminal showing the decrypted content output.

```
C:\Users\Rudra\Desktop\IS IA\scripts>gpg --decrypt secret2.txt.gpg > secret_decrypted.txt
gpg: encrypted with rsa4096 key, ID 4F7711F641FDF599, created 2025-02-09
      "Alice Example (Key for Security IA Project) <alice@example.com>"
```



This step illustrated how GnuPG ensures **only** the intended recipient can read the encrypted message.

---

7. Digital Signatures for Authentication

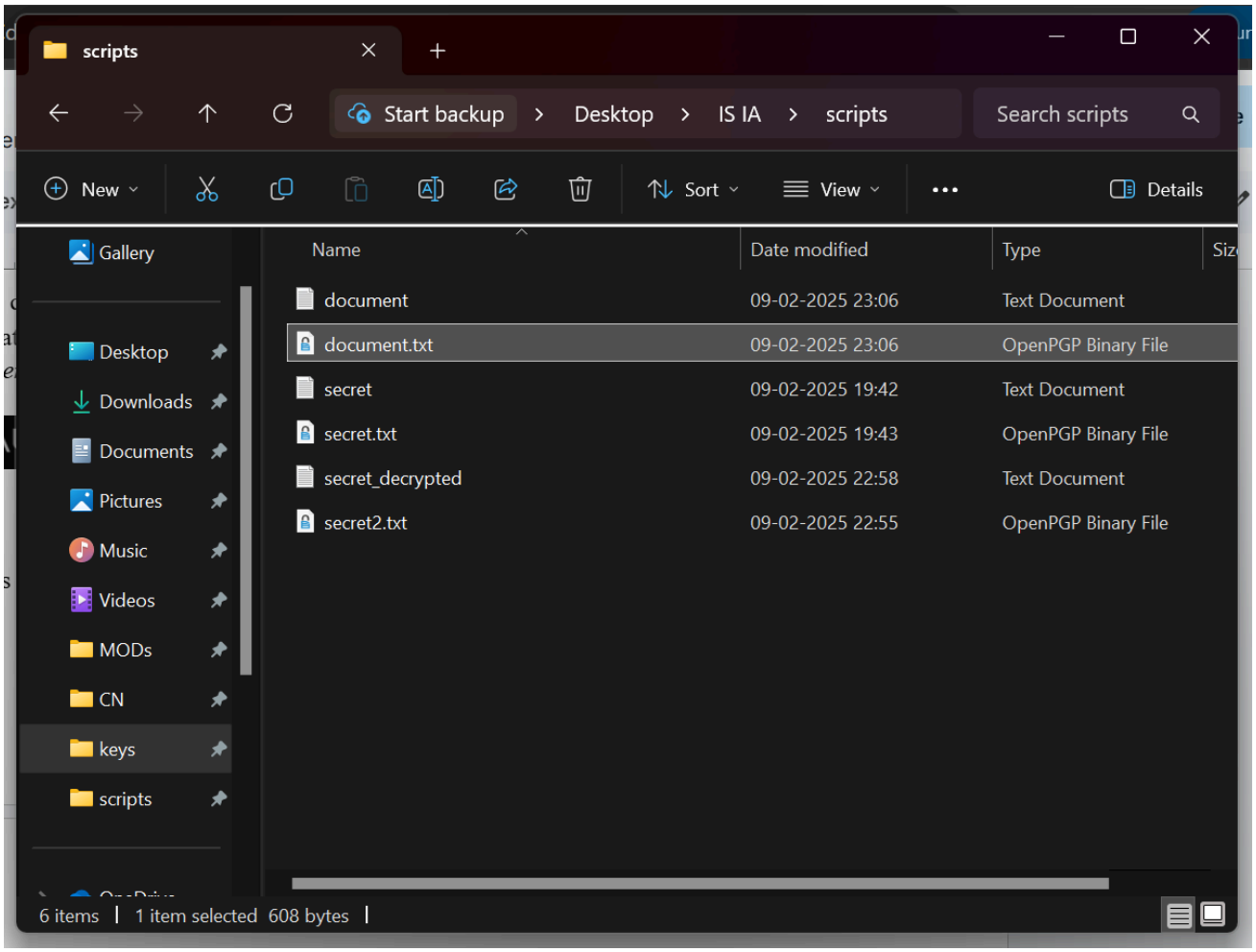
We wanted to demonstrate **signing** to prove file authenticity and detect tampering. We used two methods:

7.1. Inline Signing

```
gpg --sign document.txt
```

- This created a `document.txt.gpg` file containing both the original data and the signature.
- *Screenshot 12*: Terminal output of signing process.

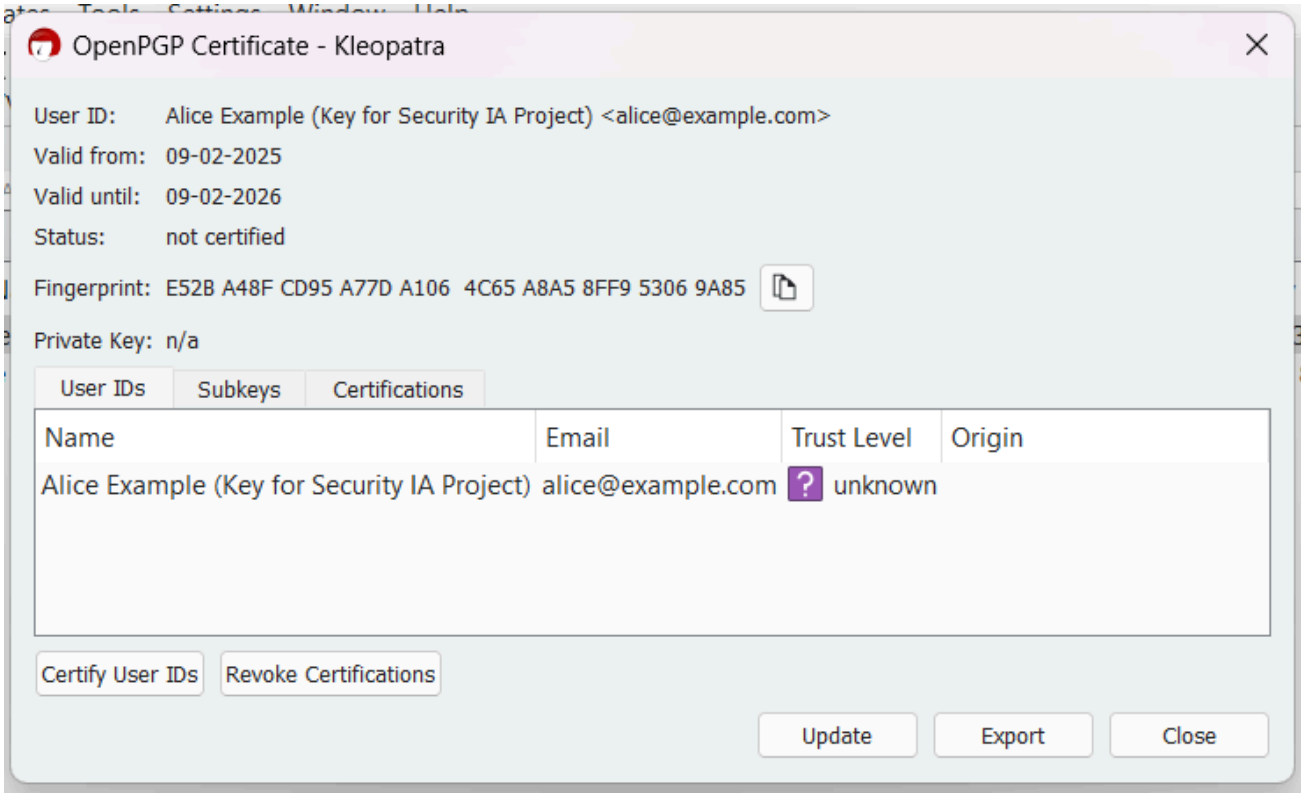
```
C:\Users\Rudra\Desktop\IS IA\scripts>gpg --sign document.txt
```



When others received `document.txt.gpg`, they could run:

```
gpg --decrypt document.txt.gpg
```

```
C:\Users\Sachin Shinde\Downloads>gpg --decrypt document.txt.gpg
gpg: Signature made 02/09/25 23:06:52 India Standard Time
gpg:                using RSA key E52BA48FCD95A77DA1064C65A8A58FF953069A85
gpg: Good signature from "Alice Example (Key for Security IA Project) <alice@example.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: E52B A48F CD95 A77D A106  4C65 A8A5 8FF9 5306 9A85
```



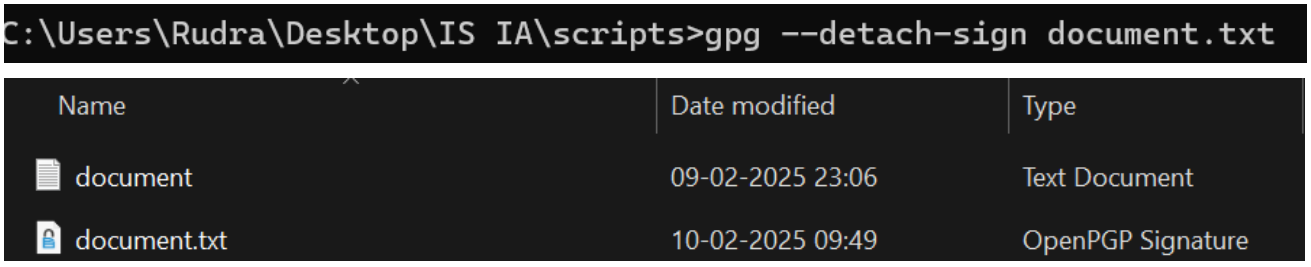
to retrieve the file and automatically verify our signature.

7.2. Detached Signing

We also practiced **detached signatures**:

```
gpg --detach-sign document.txt
```

- This created `document.txt.sig`.
- Recipients verified the signature with:
  `gpg --verify document.txt.sig document.txt`
- If the file was unaltered and our public key was trusted, GnuPG displayed a "Good signature" message.
- *Screenshot 13*: Output of a successful detached signature verification.



```
C:\Users\Sachin Shinde\Downloads>gpg --verify document.txt.sig document.txt
gpg: Signature made 02/10/25 09:49:53 India Standard Time
gpg:                using RSA key E52BA48FCD95A77DA1064C65A8A58FF953069A85
gpg: Good signature from "Alice Example (Key for Security IA Project) <alice@example.com>" [unknown]
```

---

8. Verification of Signatures

Verification proved crucial for ensuring no unauthorized modifications occurred:

- **Embedded**: `gpg --decrypt <signed_file>`
- **Detached**: `gpg --verify <signature> <original_file>`

In both cases, GnuPG either confirmed a **valid** signature or warned about a mismatch.

- *Screenshot 14*: Terminal showing the "Good signature from…" message.

```
C:\Users\Rudra\Desktop\IS IA\scripts>gpg --decrypt document.txt.sig
gpg: assuming signed data in 'document.txt'
gpg: Signature made 02/10/25 09:49:53 India Standard Time
gpg:                using RSA key E52BA48FCD95A77DA1064C65A8A58FF953069A85
gpg: Good signature from "Alice Example (Key for Security IA Project) <alice@example.com>" [ultima
te]
```

---

9. Key Revocation or Expiration (Optional)

As a best practice, we tested **revoking** or **expiring** a key. In a real scenario, this is used if a private key is compromised or if it's time to rotate keys. We generated a revocation certificate:

```
gpg --gen-revoke "alice@example.com" > revoke_alice.asc
```

If the key needed to be revoked, we would import the certificate:

```
gpg --import revoke_alice.asc
```

- *Screenshot 15*: Demonstration of revocation certificate creation.

```
C:\Users\Rudra\Desktop\IS IA\scripts>gpg --gen-revoke "alice@example.com" > revoke_alice.asc

sec  rsa4096/A8A58FF953069A85 2025-02-09 Alice Example (Key for Security IA Project) <alice@exampl
e.com>

Create a revocation certificate for this key? (y/N) y
Please select the reason for the revocation:
  0 = No reason specified
  1 = Key has been compromised
  2 = Key is superseded
  3 = Key is no longer used
  Q = Cancel
(Probably you want to select 1 here)
Your decision? 0
Enter an optional description; end it with an empty line:
> This is to test revocation
>
Reason for revocation: No reason specified
This is to test revocation
Is this okay? (y/N) y
ASCII armored output forced.
```

This ensures others know that the key should no longer be trusted.

---

10. Organizing Our GitHub Repository

Finally, we consolidated our work in a **GitHub Classroom** repository to facilitate submission and demonstration:

> **Project Structure**
> We created a folder structure like:
> https://github.com/aahanshetye/GnuPG-IA-Project
> ```
> ├── README.md
> ├── keys
> │   ├── alice_public_key.asc
> │   └── bob_public_key.asc
> ├── scripts
> │   ├── encrypt_file.sh   (Linux/macOS)
> │   └── encrypt_file.ps1  (Windows)
> ├── screenshots
> │
> └── report.pdf
> ```

Conclusion

Through these implementation steps, we successfully demonstrated the **core functionality of GnuPG** across different operating systems:

- **Key Generation & Management**: Created strong RSA keys (4096 bits), protected by secure passphrases.
- **Encryption & Decryption**: Exchanged confidential files within our team using each other's public keys.
- **Digital Signing & Verification**: Proved message authenticity and protected against tampering.
- **Revocation & Expiration**: Showed how to revoke or expire keys in case of compromise or key rotation.

By organizing all materials in GitHub, we streamlined our submission process and facilitated transparent tracking of our progress. This hands-on project solidified our understanding of **public-key cryptography**, ensuring we can integrate secure workflows into future endeavors and practical cybersecurity situations.

References and Additional Resources

1. Official GnuPG Website
2. GNU Privacy Handbook
3. OpenPGP Standard (RFC 4880)
4. Email Self-Defense Guide by FSF