Lab 3.1 – Encrypting, Hashing, and Signing Files with GPG

1. Lab Overview

1.1 Lab Description

In this lab, you'll practice using **GNU Privacy Guard (GPG)** to secure files with cryptographic techniques. GPG is a widely used open-source tool for **encryption, hashing, and digital signing**.

You'll learn how to:

- Encrypt files to protect confidentiality.
- Generate and check hashes to ensure file integrity.
- Sign files digitally to prove authenticity and prevent tampering.

These skills are directly applied in secure file sharing, backups, and even software distribution.

1.2 Learning Objectives

After completing this lab, you will be able to:

- Install and configure GPG on your system.
- Generate and manage public/private key pairs.
- Encrypt files for yourself and others.
- Use hashing to verify file integrity.
- Digitally sign files and validate signatures.
- Apply a full workflow of **Encrypt → Hash → Sign → Verify**.

1.3 Prerequisites

Before starting this lab, you should have:

- A computer running **Linux, macOS, or Windows**.
- Terminal or command-line access.

- Basic familiarity with using shell commands.
- (Windows only) The **Gpg4win** installer.
- (Optional) A text editor for creating test files.

1.4 Estimated Completion Time

Activity	**Estimated Time**
Installing and setting up GPG	45 minutes
Encrypting and decrypting files	45 minutes
Hashing and verifying file integrity	30 minutes
Signing and verifying file authenticity	45 minutes
Challenge: Encrypt → Hash → Sign workflow	60 minutes
Reflection and documentation	15 minutes
Total Estimated Time	**~4 hours**

Checkpoint
Before moving on, make sure you:

- Understand this lab will take about **4 hours**.
- Have a computer with command-line access.
- Are ready to install GPG (or confirm it's already installed).

2. Getting Started

2.1 What is GPG?

GNU Privacy Guard (GPG) is a free, open-source implementation of the **OpenPGP standard**. It provides cryptographic tools for:

- **Encryption** → Making files unreadable without the correct private key.
- **Hashing** → Generating a digital fingerprint to detect changes.
- **Signing** → Attaching a digital signature that proves authorship and authenticity.

GPG is widely used by developers, system administrators, and security professionals — from protecting personal emails to verifying the authenticity of software packages.

2.2 Why Use GPG for Encryption, Hashing, and Signing?

GPG is powerful because it combines **confidentiality, integrity, and authenticity** in one tool:

- **Confidentiality** → Encrypt files so only the intended recipient can read them.
- **Integrity** → Detect accidental corruption or malicious tampering with hashes.
- **Authenticity** → Verify the sender's identity using digital signatures.

Real-world examples:

- Software downloads are often accompanied by **hashes and signatures**.
- Teams use GPG to securely exchange sensitive files.
- Backups can be encrypted to prevent exposure if stolen.

2.3 Lab Setup Checklist

Before starting, make sure you have:

GPG installed

- Linux: Preinstalled on many systems; otherwise:
  `sudo apt install gnupg`

- macOS:
  `brew install gnupg`
- Windows: Download and install **Gpg4win** from the official site.

Test files ready

- Create a simple text file (e.g., secret.txt) for encryption, signing, and hashing.

Key pair available

- Generate one if needed:
  `gpg --full-generate-key`

Terminal/command line access

- All commands in this lab will be executed from the terminal.

Checkpoint

Before moving on:

- Do you have GPG installed on your system?
- Did you generate a key pair with gpg --full-generate-key?
- Do you have at least one text file ready for testing?

3. Installing and Setting Up GPG

3.1 Installing GPG

GPG is free and available across all major platforms:

- **Linux**
  - Often preinstalled. If missing, install with:

    ```
    sudo apt update
    sudo apt install gnupg
    ```

- **macOS**
  - Install via Homebrew:

    ```
    brew install gnupg
    ```

- **Windows**
  - Download **Gpg4win** from: https://gpg4win.org
  - Run the installer and follow the setup wizard.

Tip: Confirm installation with:

```
gpg --version
```

3.2 Generating Your First Key Pair

GPG uses a **public/private key pair**:

- The **public key** is shared with others so they can encrypt files for you.
- The **private key** is kept secret and used to decrypt and sign files.

To generate a key pair:

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

Steps:

1. Choose **RSA and RSA** (default).
2. Choose key size → **4096 bits** recommended.
3. Set expiration → **2 years** (can renew later).

4. Enter your name and email (acts as your key ID).
5. Set a strong passphrase.

3.3 Exporting and Managing Keys

Once you've generated keys, you can manage them:

- **List your keys**
  `gpg --list-keys`
- **Export your public key (to share)**
  `gpg --armor --export your@email.com > publickey.asc`
- **Export your private key (backup only – keep safe!)**
  `gpg --armor --export-secret-keys your@email.com > privatekey.asc`

Never share your private key. Only your **public key** should be distributed.

3.4 Recap: GPG Setup Essentials

Task	Command/Action
Install GPG	sudo apt install gnupg (Linux) brew install gnupg (macOS) Gpg4win (Windows)
Generate key pair	`gpg --full-generate-key`
List keys	`gpg --list-keys`
Export public key	`gpg --armor --export you@example.com > publickey.asc`
Export private key	`gpg --armor --export-secret-keys \ you@example.com > privatekey.asc`

Checkpoint
Before moving on:

- Did you install GPG successfully?
- Did you generate a key pair with gpg --full-generate-key?
- Can you list your keys and export your public key?

4. Encrypting a File

4.1 Encrypting for a Recipient

To securely send a file, you need the **recipient's public key**.
Once you've imported it into GPG, run:

```
gpg -e -r recipient@example.com secret.txt
```

- -e → Encrypt
- -r → Recipient email (key ID)
- secret.txt → File to encrypt

This produces an encrypted file named secret.txt.gpg.

Only the recipient with their private key can decrypt it.

4.2 Encrypting for Yourself

If you want to protect your own files, just encrypt them to **your own email address**:

```
gpg -e -r you@example.com notes.txt
```

This is useful for **backups** or sensitive documents you want only yourself to read.

4.3 Decrypting a File

To decrypt an encrypted file, run:

```
gpg -d secret.txt.gpg
```

- If the file was encrypted for you, GPG will ask for your **private key passphrase**.
- You can output the plaintext into a file:
  `gpg -d secret.txt.gpg > secret_decrypted.txt`

Always test encryption by decrypting your own files to ensure your keys and passphrase are working correctly.

4.4 Recap: File Encryption

Task	Command
Encrypt for recipient	`gpg -e -r recipient@example.com file.txt`
Encrypt for yourself	`gpg -e -r you@example.com file.txt`
Decrypt a file	`gpg -d file.txt.gpg`

Checkpoint

Before moving on:

- Did you successfully encrypt a test file?
- Did you decrypt the file back to its original form?
- Do you understand the role of **public vs private keys** in encryption?

5. Hashing and Verifying Files

5.1 Generating a SHA-256 Hash

A **hash function** produces a unique digital fingerprint of a file. Even the smallest change in the file creates a completely different hash.

To generate a SHA-256 hash:

```
sha256sum important.zip
```

Example output:

```
d2d2c3f3b4a5e6c7d8f9a0b1c2d3e4f56789abcdeffedcba9876543210fedcba  important.zip
```

Hashes are commonly published alongside software downloads to let users verify files.

5.2 Verifying File Integrity

Later, you can recompute the hash and compare:

```
sha256sum important.zip
```

- If the hash is the same → the file is unchanged.
- If it's different → the file was modified or corrupted.

5.3 Why Hashing Matters

Hashing is used in many real-world scenarios:

- **Software integrity** → Verify downloads aren't tampered with.
- **Backups** → Ensure stored files remain unchanged.
- **Digital forensics** → Detect whether evidence has been altered.
- **Blockchain & security protocols** → Hashes underpin digital signatures and proof-of-work.

5.4 Recap: File Hashing

Task	Command
Generate SHA-256 hash	`sha256sum filename`
Verify hash	Compare outputs at different times

Checkpoint

Before moving on:

- Did you generate a SHA-256 hash for one of your files?
- Did you verify the hash stayed consistent when the file was unchanged?
- Do you understand how hashes detect tampering or corruption?

6. Digitally Signing Files

6.1 Creating an Attached Signature

An **attached signature** bundles the signature with the file itself.

To sign a file:

```
gpg --sign report.txt
```

- This creates a new file report.txt.gpg.
- It contains both the original file and the signature.

Use attached signatures when you want to share the file and its proof of authenticity together.

6.2 Creating a Detached Signature

Sometimes you don't want to alter the file itself — instead, you can create a **detached signature**.

```
gpg --detach-sign report.txt
```

- This generates report.txt.sig (or .asc).
- The signature file is separate from the original file.

Detached signatures are commonly used in software distribution (e.g., .tar.gz file with a .sig for verification).

6.3 Verifying a Signature

To verify that a file's signature is valid:

- For an attached signature:

  ```
  gpg --verify report.txt.gpg
  ```

- For a detached signature:

  ```
  gpg --verify report.txt.sig report.txt
  ```

If the file has been altered or the wrong public key is used, GPG will warn that the signature is invalid.

6.4 Recap: File Signing and Authenticity

Task	Command
Create attached signature	`gpg --sign file.txt`
Create detached signature	`gpg --detach-sign file.txt`
Verify attached signature	`gpg --verify file.txt.gpg`
Verify detached signature	`gpg --verify file.txt.sig file.txt`

Checkpoint
Before moving on:

- Did you sign a file with an attached signature?
- Did you create a detached signature as well?
- Did you verify both signatures successfully?

7. Lab Challenge

Now it's your turn to apply all three cryptographic techniques together: **encryption, hashing, and signing**.

7.1 Task: Encrypt → Hash → Sign a File

1. **Create a test file**

   ```
   echo "This is my secret lab file." > labfile.txt
   ```

2. **Encrypt it using your public key**

   ```
   # Produces labfile.txt.gpg:
   gpg -e -r you@example.com labfile.txt
   ```

3. **Generate a hash of the original file**

   ```
   sha256sum labfile.txt > labfile.txt.sha256
   ```

4. **Sign the file with your private key**

   ```
   # Produces labfile.txt.sig.
   gpg --detach-sign labfile.txt
   ```

5. **Verify the signature and hash**

   ```
   gpg --verify labfile.txt.sig labfile.txt
   sha256sum -c labfile.txt.sha256
   ```

7.2 Step-by-Step Instructions

Step	Command	Expected Result
Create file	`echo "text" > labfile.txt`	File created
Encrypt file	`gpg -e -r you@example.com labfile.txt`	labfile.txt.gpg generated
Hash file	`sha256sum labfile.txt > labfile.txt.sha256`	SHA-256 checksum saved

Step	Command	Expected Result
Sign file	`gpg --detach-sign labfile.txt`	labfile.txt.sig generated
Verify files	`gpg --verify labfile.txt.sig labfile.txt`	Signature valid message
Verify hash	`sha256sum -c labfile.txt.sha256`	Integrity check passes

7.3 Reflection Questions

Write your answers in your lab notebook or submission:

1. What's the difference between encryption and signing?
2. Why might someone publish a file's **hash** as well as a **signature**?
3. Which step protects **confidentiality**?
4. Which step protects **integrity**?
5. Which step protects **authenticity**?
6. How would you explain this workflow to a non-technical colleague?

Challenge Complete!
You've successfully applied the full **Encrypt → Hash → Sign → Verify** cycle. This is exactly how secure file distribution is done in the real world.

8. Wrap-Up and What's Next

8.1 Key Takeaways

In this lab, you practiced real cryptographic techniques using **GPG** and related tools. You:

Installed and set up GPG on your system.
Generated a public/private key pair.
Encrypted files for yourself or a recipient.
Verified file integrity with **SHA-256 hashes**.
Digitally signed files (attached and detached).
Verified authenticity and integrity of signed files.
Completed a full **Encrypt → Hash → Sign → Verify** workflow.

These are the **same techniques used in industry** for software distribution, backups, secure communication, and digital forensics.

8.2 Skills You've Gained

By completing this lab, you can now:

- Explain the difference between **encryption, hashing, and signing**.
- Protect file confidentiality with encryption.
- Detect tampering or corruption with hashes.
- Prove authorship and authenticity with digital signatures.
- Apply combined cryptographic protections in a practical workflow.

These skills form the foundation for understanding how secure communication protocols and trust systems work.

8.3 Next Lab: HTTPS and Certificates in Practice

Next, you'll see how these concepts extend beyond files into the **web environment**.

In **Lab 3.2: HTTPS and Certificates in Practice**, you will:

- Inspect HTTPS connections in a browser.
- Understand how **certificates** establish trust online.
- Explore the **Certificate Authority (CA)** system.
- See how hashing and signing protect websites, just like they did your files.

Next Steps

- Save your .gpg, .sig, and .sha256 files as lab artifacts.
- Document your reflection answers.
- Review your commands so you can repeat them confidently.
- Get ready for **Lab 3.2 – HTTPS and Certificates in Practice**.

You've now gone from **personal file crypto** to preparing for **web-scale encryption and trust models**.

9. Appendix

9.1 GPG Command Reference

Task	Command
Generate new key pair	gpg --full-generate-key
List keys	gpg --list-keys
Export public key	gpg --armor --export you@example.com > publickey.asc
Export private key (backup only)	gpg --armor --export-secret-keys you@example.com > privatekey.asc
Encrypt file for recipient	gpg -e -r recipient@example.com file.txt
Encrypt file for yourself	gpg -e -r you@example.com file.txt
Decrypt file	gpg -d file.txt.gpg
Sign file (attached)	gpg --sign file.txt
Sign file (detached)	gpg --detach-sign file.txt
Verify attached signature	gpg --verify file.txt.gpg
Verify detached signature	gpg --verify file.txt.sig file.txt

9.2 Common Hashing Commands

Task	Command
Generate SHA-256 hash	sha256sum filename
Verify hash (against file)	sha256sum -c filename.sha256

9.3 Troubleshooting Tips

Problem	Possible Cause	Solution
GPG not found	Not installed	Install via package manager or Gpg4win
Key generation fails	Weak entropy (Linux)	Move mouse, type, or run background processes to generate randomness
"No public key" error when encrypting	Recipient's public key not imported	Import their key with gpg --import key.asc
Wrong passphrase on decrypt	Incorrect private key or password	Retry with correct passphrase
Hash mismatch	File modified or corrupted	Re-download original file, compare again

9.4 Further Resources

- **GnuPG Official Documentation**: https://gnupg.org/documentation/
- **Gpg4win (Windows Installer)**: https://gpg4win.org
- **SHA256SUM Tutorial (Linux/Unix)**: https://linux.die.net/man/1/sha256sum
- **Practical Cryptography with GPG (Book/Guide)**: https://www.debian.org/doc/manuals/securing-debian-manual/ch-crypto.en.html#gpg