# Encryption/Hashing utilities

## GNU Privacy Guard(AKA GnuPG or GPG)

---

**We can encrypt a file using GnuPG (GPG) using the following command:**

```
gpg --symmetric --cipher-algo CIPHER message.txt
```

**The encrypted file will be saved as** `message.txt.gpg`. Replace CIPHER for you encryption alogtithm for example AES-256-CBC

**The default output is in the binary OpenPGP format; however, if you prefer to create an ASCII armoured output, which can be opened in any text editor, you should add the option** `--armor`**. For example,** `gpg --armor --symmetric --cipher-algo CIPHER message.txt`.

---

**You can decrypt using the following command:**

```
gpg --output original_message.txt --decrypt message.gpg
```

---

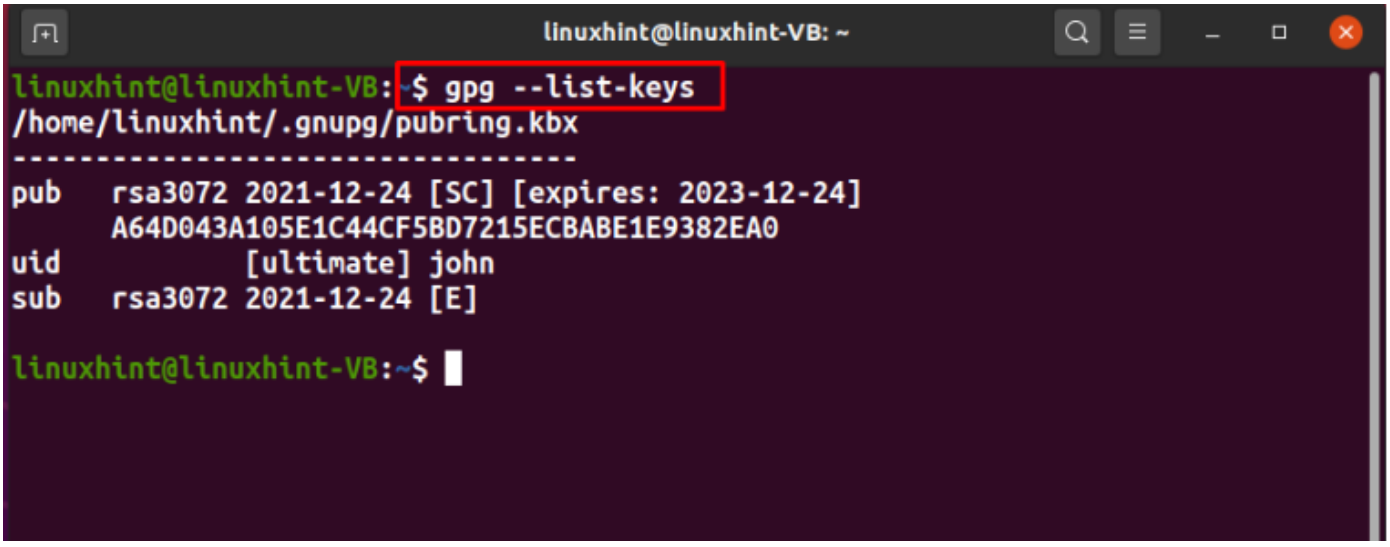
List available gpg keys:

**gpg --list-keys**

---

To Export a public key:

first note the uid of the user that you want to export his Public key, for example here the uid is john:



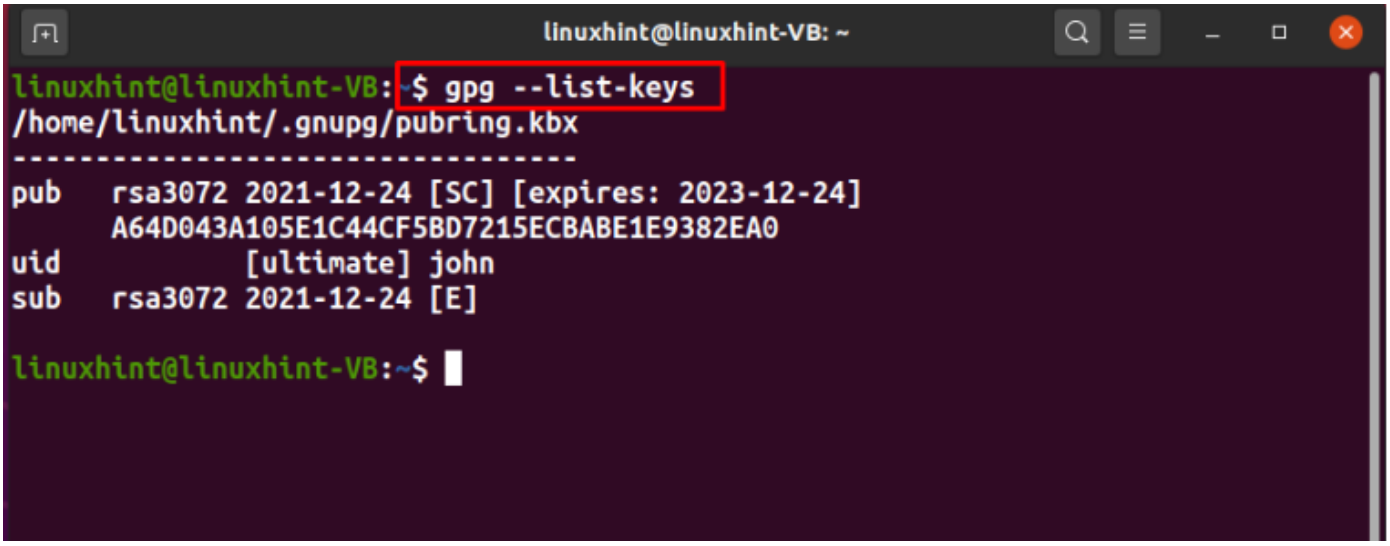

then we type: NOTE the -a followed by the uid

**gpg --export -a john > public.key**

---

To Export a private key:

first note the uid of the user that you want to export his Private key, for example here the uid is john:




then we type: NOTE the -a followed by the uid

**gpg --export-secret-key -a john > private key**

the above command will ask for a passphrase then will generate the private key.

---

To import a Public key:

**gpg --import public-key-name.key**

---

To import a Private key:

**gpg --import private-key-name.key**

NOTE you will be asked for a passphrase to import the private key.

# OpenSSL Project

## 1) Encrypt & Decrypt files

---

A)

**We can encrypt a file using OpenSSL using the following command:**

```
openssl aes-256-cbc -e -in message.txt -out encrypted_message
```

---

B)

**We can decrypt the resulting file using the following command:**

```
openssl aes-256-cbc -d -in encrypted_message -out original_message.txt
```

---

C)

**To make the encryption more secure and resilient against brute-force attacks, we can add `-pbkdf2` to use the Password-Based Key Derivation Function 2 (PBKDF2); moreover, we can specify the number of iterations on the password to derive the encryption key using `-iter NUMBER`. To iterate 10,000 times, the previous command would become:**

```
openssl aes-256-cbc -pbkdf2 -iter 10000 -e -in message.txt -out encrypted_message
```

**Consequently, the decryption command becomes:**

```
openssl aes-256-cbc -pbkdf2 -iter 10000 -d -in encrypted_message -out original_message.txt
```

---

# 2) RSA

---

A) Generate **RSA** private key:

**openssl genrsa -out private-key.pem 2048**

Generate public key and Specify the private key you generated above as input for the public key:

**openssl rsa -in private-key.pem -pubout -out public-key.pem**

---

B) To see real RSA variables use --text --noout

**The values of *p*, *q*, *N*, *e*, and *d* are `prime1`, `prime2`, `modulus`, `publicExponent`, and `privateExponent`, respectively.**

**openssl rsa -in private-key.pem -text -noout**

---

C)

**If we already have the recipient's public key, we can encrypt it with the command `openssl pkeyutl -encrypt -in plaintext.txt -out ciphertext -inkey public-key.pem -pubin`**

**The recipient can decrypt it using the command `openssl pkeyutl -decrypt -in ciphertext -inkey private-key.pem -out decrypted.txt`**

---

# 3) Certificat Signing request

---

**You can use `openssl` to generate a certificate signing request using the command `openssl req -new -nodes -newkey rsa:4096 -keyout key.pem -out cert.csr`. We used the following options:**

- `req -new` create a new certificate signing request
- `-nodes` save private key without a passphrase
- `-newkey` generate a new private key
- `rsa:4096` generate an RSA key of size 4096 bits
- `-keyout` specify where to save the key
- `-out` save the certificate signing request

---

**You can use `openssl` to generate a self-signed certificate.**

**openssl req -x509 -newkey -nodes rsa:4096 -keyout key.pem -out cert.pem -sha256 -days 365**

**The `-x509` indicates that we want to generate a self-signed certificate instead of a certificate request.**

---

To view your certificate details you can use:

**openssl x509 -in cert.pem -text**

# Hashing and HEX utilities

---

**HMAC**

**Hash-based message authentication code (HMAC) is a message authentication code (MAC) that uses a cryptographic key in addition to a hash function.**

**To calculate the HMAC on a Linux system, you can use any of the available tools such as `hmac256` (or `sha224hmac`, `sha256hmac`, `sha384hmac`, and `sha512hmac`, where the secret key is added after the option `--key`).**

A

**n example of calculating the HMAC using `hmac256` and `sha256hmac` with two different keys**:
NOTE that hmac256 works but sha256hmac didn't work but they do the same thing basically.

main

**@main$ hmac256 s!Kr37 message.txt**

**3ec65b7e80c5bf2e623e52e0528f1c6a74f605b10616621ba1c22a89fb244e65  message.txt**

**main@main$ hmac256 1234 message.txt**

**4b6a2783631180fca6128592e3d17fb5bff6b0e563ad8f1c6afc1050869e440f  message.txt**

**main@main$ sha256hmac message.txt --key s!Kr37**

**3ec65b7e80c5bf2e623e52e0528f1c6a74f605b10616621ba1c22a89fb244e65  message.txt**

**main@main$ sha256hmac message.txt --key 1234**

**4b6a2783631180fca6128592e3d17fb5bff6b0e563ad8f1c6afc1050869e440f  message.txt**

---

Dump the HEX of a file:

**hexdump text1.txt -C**

---

Get the Sha256 hash of a file:

**sha256sum** text1.txt

---