- TP Cryptographie -
Confidentialité, Intégrité et Authenticité des Données

Rudy Gabert - Rémy Barras - Jérémy Rincker - Marie Michel

Sommaire

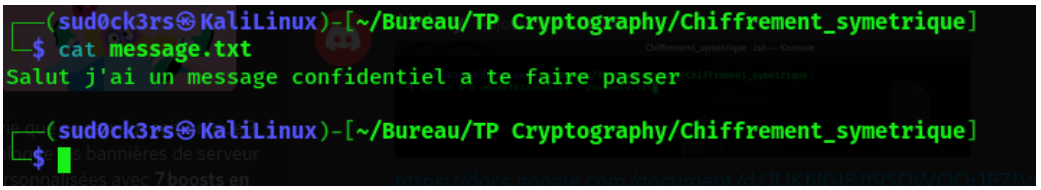
1. Chiffrement symétrique

1.1 AES

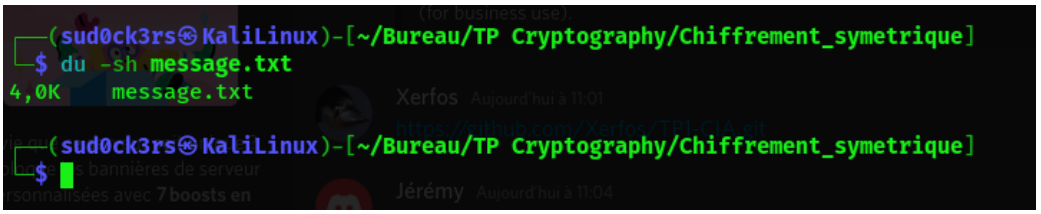
a. Chiffrement

Fichier léger :

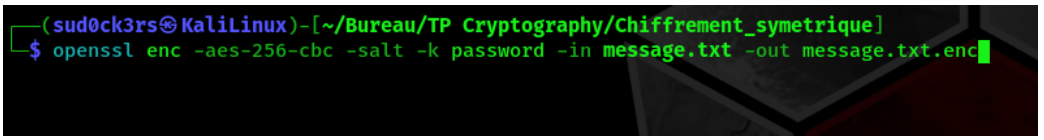
Message court à chiffrer :



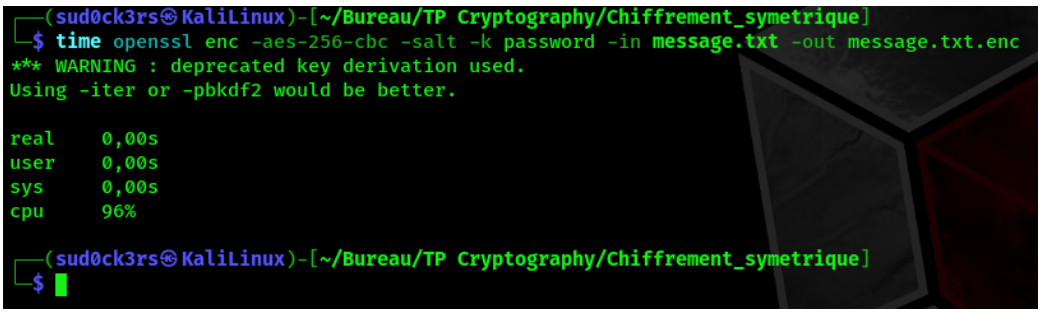
On peut voir avec la commande **du** la taille du fichier, qui est ici très léger :



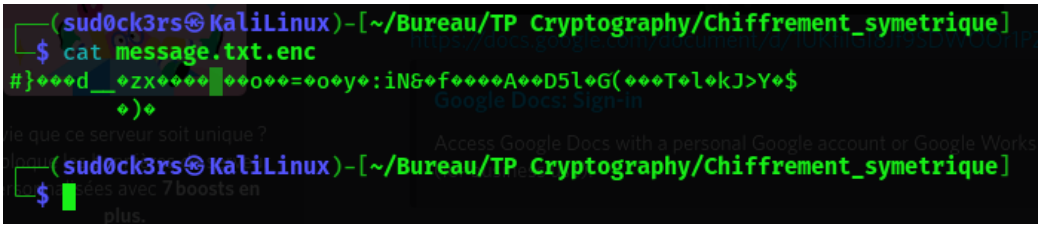
Chiffrement du fichier avec aes-256-cbc avec **openssl** :



On peut voir avec la commande **time** que l'encryption est instantanée (0s) à ce niveau de précision:

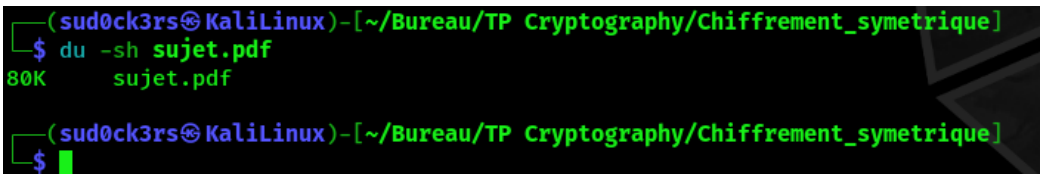


Le fichier est maintenant chiffré, voici son contenu :

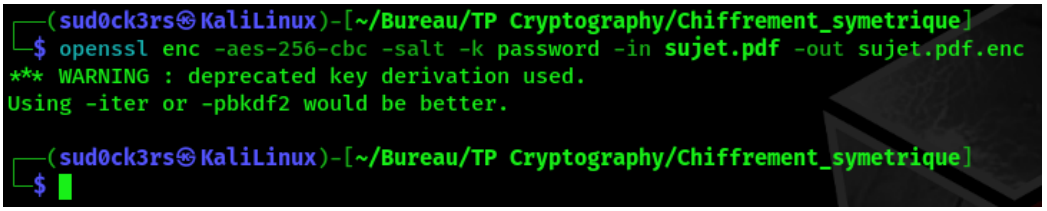


Fichier de taille moyenne type .PDF

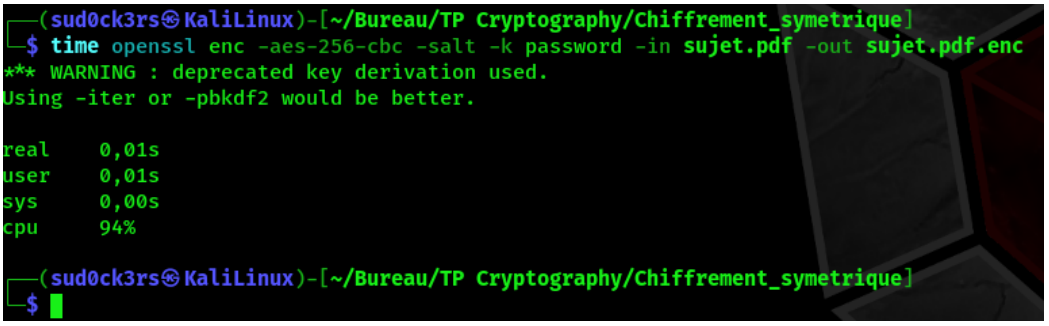
Même opération sur le sujet du TP :



Chiffrement du fichier en aes-256 avec **openssl** :

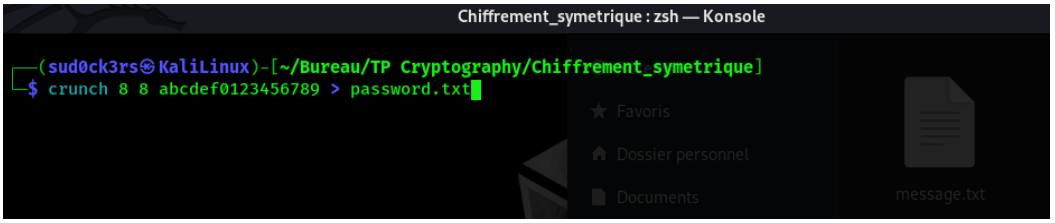


Commande **time**, le chiffrement est également presque instantané : (0.01s)



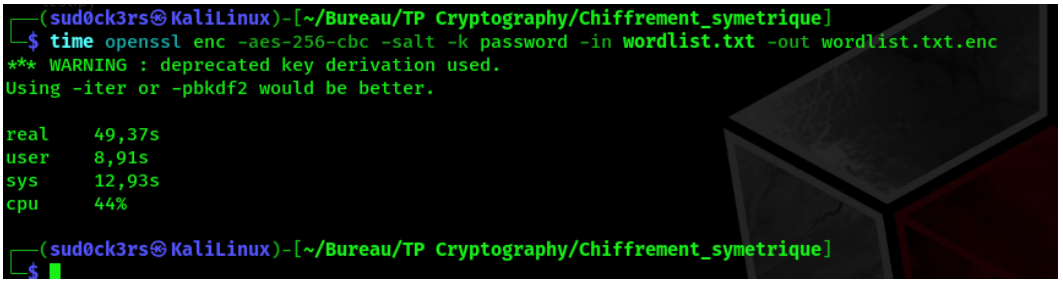
Fichier lourd

Même manipulation, mais avec un fichier lourd de plusieurs GigaBits, créé à partir de la commande **crunch** :



Nous avons généré une wordlist de 5.5 GB. Ce fichier nous permettra de mieu voir la différence de temps de chiffrement et de déchiffrement en symétrique.

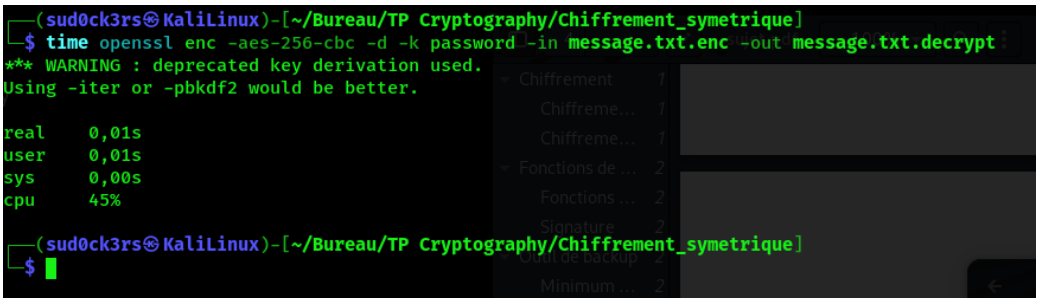
On observe que dans ce cas, le chiffrement est plus long, mais à échelle humaine il reste très court : **49,37s**



b. Déchiffrement

Fichier léger :

Déchiffrement du fichier message.txt.enc :



On observe que le déchiffrement d'un fichier si léger est presque instantané.

Fichier lourd :

On observe que pour le déchiffrement, le temps est légèrement plus long mais toujours acceptable : **56,80s**

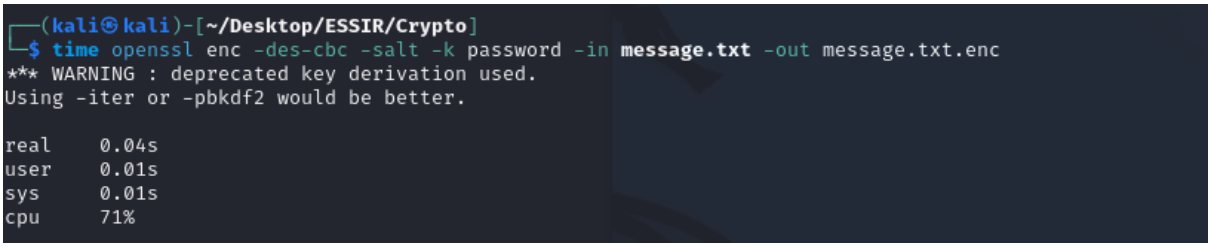
Tableau comparatif :

Taille fichier	Durée chiffrement (secondes)	Durée déchiffrement (secondes)
4 ko	0,00s	0,00s
80 ko	0,01s	X
5,5 G	49,37s	56,80s

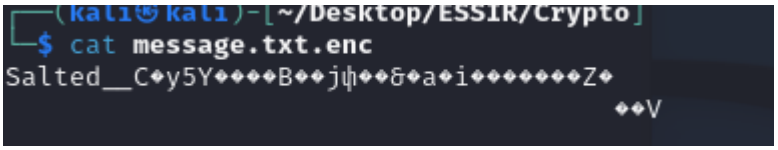
1.2. Chiffrement DES

a. Fichier léger

Nous avons créé un fichier léger appelé “message.txt”, dont nous pouvoir la taille avec la commande du



Nous utilisons maintenant l’outil openssl pour chiffrer le fichier “message.txt” à l’aide de des-cbc. On peut voir que le chiffrement est presque instantané grâce à la commande time. Voici le contenu du fichier “message.txt”, qui est à présent chiffré.



b. Fichier lourd

Nous allons maintenant répéter l’opération avec un fichier plus lourd. Nous avons utilisé un iso d’ubuntu. Nous pouvons voir sa taille à l’aide de la commande du

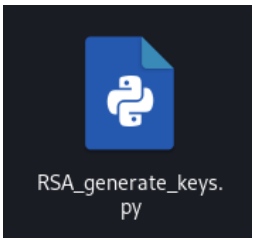
Puis nous utilisons l’outil openssl pour chiffrer l’iso d’Ubuntu à l’aide de des-cbc. On peut voir que le chiffrement prend beaucoup plus de temps, du fait de la taille conséquente de l’iso.

=> Nous avons donc remarqué que le chiffrement aes est non seulement plus rapide que le chiffrement des mais en plus, il est davantage sécurisé. Cela est dû aux formules mathématiques employées.

2. Chiffrement asymétrique

1.1 RSA

Pour effectuer le chiffrement asymétrique nous avons créé deux scripts en Python. Le premier permet de générer une paire de clé RSA privée / publique et le deuxième permet de chiffrer ou de déchiffrer un fichier cela selon les paramètre que l'on passe en arguments.

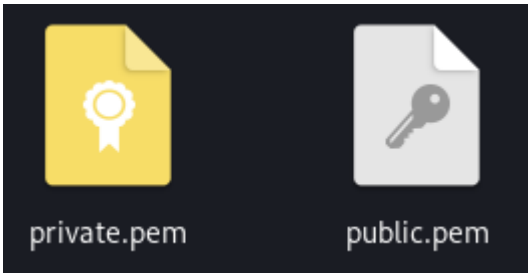


Pour commencer nous allons utiliser le script RSA_generate_keys.py pour générer une paire de clé.

Nous avons juste à exécuter ce script avec la commande suivante: python RSA_generate_keys.py

```
┌──(sud0ck3rs㉿KaliLinux)-[~/Bureau/TP Cryptography/Chiffrement_Asymetrique]
└─$ python RSA_generate_keys.py
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEA9YLt5SBC+njaCVlMAu7ndXrsgGj4akul1OtfWKRMBtRF4zLz
o6J6map9kYM1RBfbSytKNq5i6h69Uj6soAoR2Mghskg3I0ZLHaWZVxKXqmzEuRSK
n7Q0IllMRzrPTdzsL7zDMxL5eGfR7HwdAHTRy4WiPGGbW2fHhTG4dYlbA5d3dII/
fRduxufIe8HO9cHyIxcdgCqnLzSQW488CTpt+JWRSnuR3VT3V/z0ZjriqBsoifAj
P6U67JAc+yfbOKbxICr3z3ZbKtpRvx8iHMEfW13QKZmhcK9HWSoumYF42ZZ6uMdN
4Qk7CBKEAoLpswuckLvmab+FcARqdWMtN3OtnwIDAQABAoIBAF0bWEjqoTwIFPgy
q0b5f0HoakSQipUVN0C2OqSNOl/kpNRzAD7wYXphU9mZXtwCCCJLuoteOxj6q2XD
awgOiZIi49SO2opAIECQCsUgwe/Z7Akcpy3F03erUeg5f2DSBahc99Iowf9NsCj1
MLJ5mw6KEWrAwYA6rUnv/zpj6uZDDttgMhIP7Qd2obwWiL0f9XneIAwICEfAYNHO
zvqH17K1LuPKFF1Yt6lPGH+tSWW5rK3f0bIrXNjbdVFRdnELQR5Rt6I2jq+2eCT3
dH42+Ynd7MqKmbYvL6cJboiqKKveClZk3WFCScsotsjdKgb1FJbpUIaWnGKFB/ze
qxWG7cECgYEA+HwimyhvT0U3Yrrin+lDhrPP54T2cNdTmOLe5/UYGuQHoQT0QmTe
```

Nous pouvons voir que le programme affiche la clé privée dans le terminal ainsi que la clé publique un peu plus en dessous.



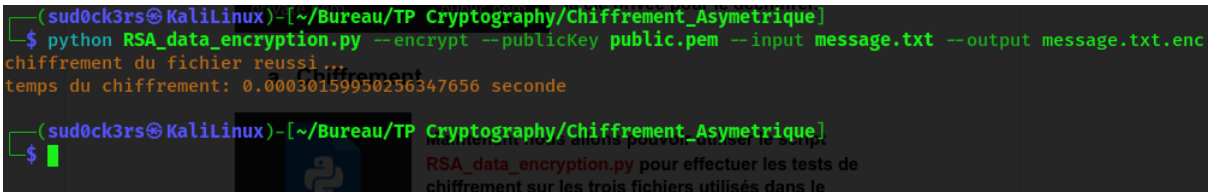
Enfin nos deux fichiers (clé privée / publique) ont été créés avec succès.

Nous pourrons alors utiliser la clé publique pour chiffrer notre fichier et la la clé privée pour le déchiffrer.

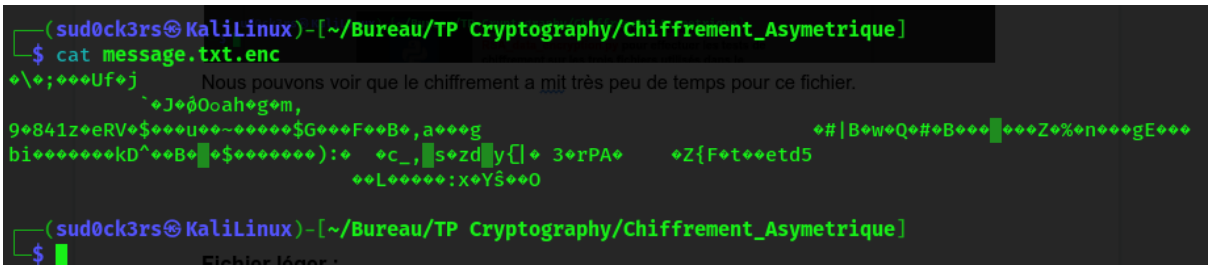
a. Chiffrement

Maintenant nous allons pouvoir utiliser le script RSA_data_encryption.py pour effectuer les tests de chiffrement sur les trois fichiers utilisés dans le chiffrement symétrique.

Chiffrement du fichier message.txt avec RSA



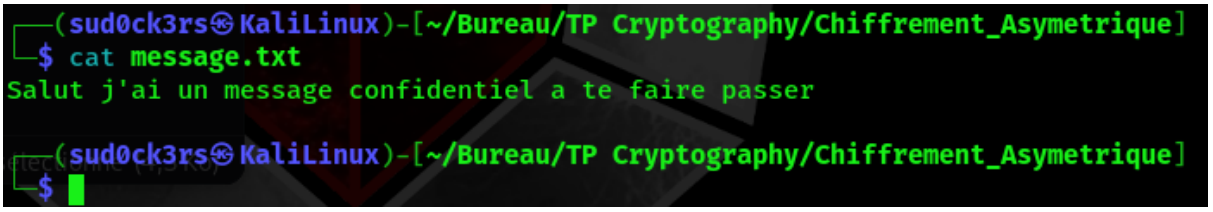
Nous pouvons voir que le chiffrement a mis très peu de temps pour ce fichier.



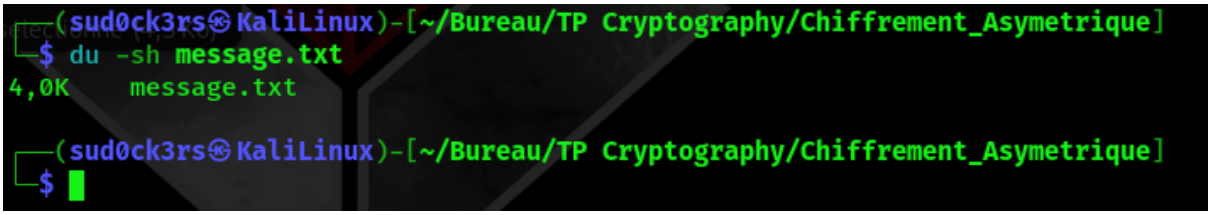
Nous pouvons voir le contenu du fichier chiffré.

Fichier léger :

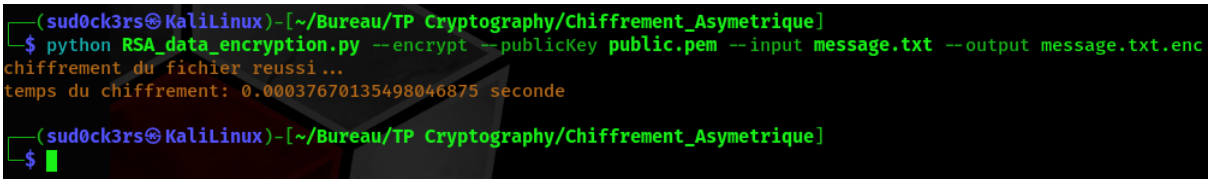
Message court à chiffrer:



On peut voir avec la commande du la taille du fichier, qui est très léger:



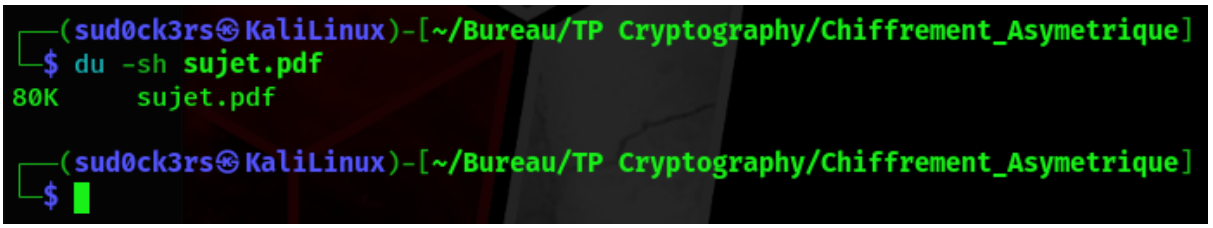
Chiffrement du fichier avec RSA via le **script python**.



On peut voir que le script a mis 0 secondes et très peu de millisecondes pour chiffrer ce petit fichier. (grâce au script, nous gagnons en précision pour mesurer le temps de chiffrement)
Le fichier est maintenant chiffré, voici son contenu

Fichier de taille moyenne type .PDF

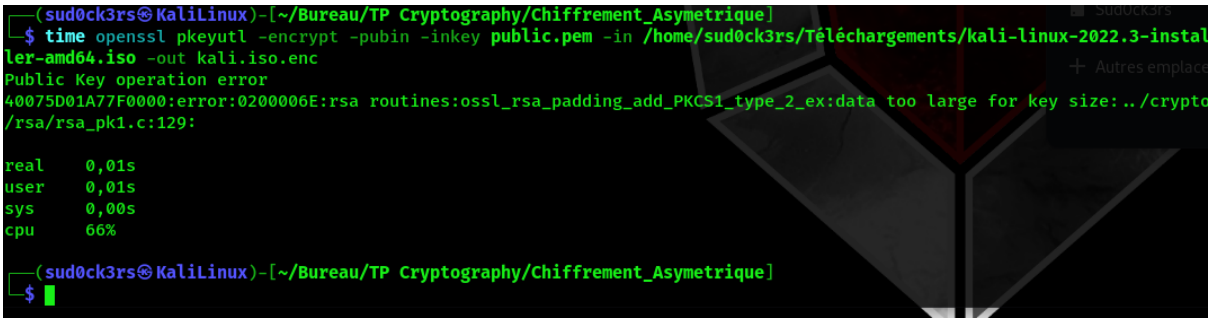
Même opération sur le sujet du TP :



Chiffrement du fichier en RSA via le **script python** :

Le chiffrement prend légèrement plus de temps mais cela reste très rapide.

Chiffrement du iso avec RSA en utilisant openssl



Nous pouvons faire une remarque très importante sur le chiffrement de ce fichier qui fait 3 GB. Ce fichier est plutôt lourd et nous observons un message d'erreur "data too large for key size" ce qui pour un fichier de veut dire que la donnée est trop grande pour la clé. Pas étonnant 3 GB.

b. Déchiffrement

Afin de retrouver l'origine du message nous allons déchiffrer ce fichier grâce à la clé privée.

Le déchiffrement reste très rapide car nous manipulons de petites données

```
┌──(sud0ck3rs㉿KaliLinux)-[~/Bureau/TP Cryptography/Chiffrement_Asymetrique]
└─$ cat message.txt.decrypt
j'ai un messsage confidentiem a te faire passer

┌──(sud0ck3rs㉿KaliLinux)-[~/Bureau/TP Cryptography/Chiffrement_Asymetrique]
└─$
```

enfin nous retrouvons bien le message de base.

c. Conclusion

Le chiffrement RSA n'est pas adapté pour chiffrer de très grandes données (fichier 3GB par exemple) mais plutôt des petites comme une clef AES lors de la connexion via le protocol SSL ou TLS.

Lors de nos tests avec le chiffrement AES nous avons réussi à chiffrer une wordlists de 5GB bien plus lourd que notre fichier iso et en quelque second. AES sera bien plus adapté et plus rapide pour de nombreux échanges entre un client et serveur.

3. Hachage

3.1 SHA256

Fichier lourd (iso Win10), on remarque que cela est rapide **17,11s**

```
daag-jeremy@jere:~/Téléchargements$ time cat Win10_21H2_French_x64.iso | openssl dgst -sha256
SHA2-256(stdin)= 2cc9731ee278666a632bdf5944105fc5f215f59ced98d75aeccf8185bd5bca3a

real    0m17,114s
user    0m14,399s
sys     0m6,651s
daag-jeremy@jere:~/Téléchargements$ du -sh Win10_21H2_French_x64.iso
5,4G    Win10_21H2_French_x64.iso
```

3.2 MD5

Même fichier, avec cet algorithme nous somme à **11,15s**

```
daag-jeremy@jere:~/Téléchargements$ time cat Win10_21H2_French_x64.iso | openssl dgst -MD5
MD5(stdin)= 74f15a7aaa171b91be317f1adf1cf815

real    0m11,158s
user    0m8,672s
sys     0m6,436s
daag-jeremy@jere:~/Téléchargements$ du -sh Win10_21H2_French_x64.iso
5,4G    Win10_21H2_French_x64.iso
```

3.3 SHA1

Ici, **10,63s**

```
daag-jeremy@jere:~/Téléchargements$ time cat Win10_21H2_French_x64.iso | openssl dgst -sha1
SHA1(stdin)= 37e5b758dea5256932f164d36dc0f7e88a747825

real    0m10,639s
user    0m6,991s
sys     0m6,363s
daag-jeremy@jere:~/Téléchargements$ du -sh Win10_21H2_French_x64.iso
5,4G    Win10_21H2_French_x64.iso
```

2.4 Tableau comparatif

Taille de fichier	SHA256	MD5	SHA1
5,4G	17,11s	11,5s	10,63s

On remarque donc que le temps diffère selon l'algorithme de hashage (plus long si plus puissant), néanmoins la différence n'étant pas impactante à l'utilisation on préfèrera utiliser SHA256 car plus fiable.
En effet les hashs md5 et sha1 sont connus pour être contournables facilement par brute force, il y a même des outils en ligne avec d'immenses banques de hashs pour les contourner.

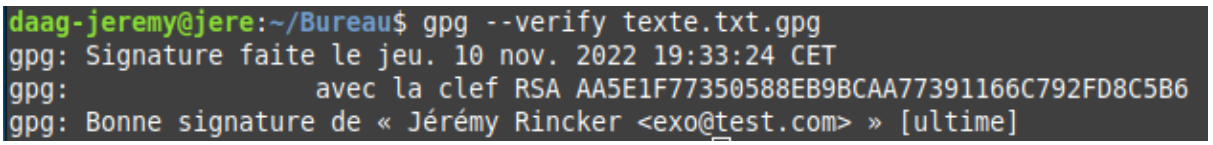
Fichier modifié/corrompu :

On peut observer qu'avec une seule modification d'un caractère (rajout d'un "4" au début du texte, l'empreinte hash est complètement différente , le hash assure donc une intégrité de la donnée.

```
daag-jeremy@jere:~/Bureau$ cat texte.txt
4Lorem ipsum dolor sit amet. Est nobis accusamus aut sint corporis ut ipsa iste. Ut enim voluptatem a autem eius ut ra
tione corporis ea totam autem sit distinctio voluptatem cum fugiat autem ut natus iste. Qui iste assumenda ut cupidita
te sunt et perferendis autem aut autem asperiores sit numquam repellat. Eos sequi fuga eos tempore voluptatibus nam do
loribus tempora qui deserunt rerum.
daag-jeremy@jere:~/Bureau$ time cat texte.txt | openssl dgst -sha256
SHA2-256(stdin)= 91143011d7bad522f3177fe0c4d63dd23afe355fcbc948f5c22042f8e3ad520d

real    0m0,006s
user    0m0,008s
sys     0m0,001s
```

On a signé le fichier texte.txt, avec les commandes :
> gpg --gen-key
> gpg --sign texte.txt
Exemple d'une vérification d'authenticité et d'intégrité avec une clé publique :



Ici nous avons créé un hash du texte.txt et l'avons chiffré avec une clé privée RSA.
L'outil gpg nous permet d'assurer de manière fiable la traçabilité des données.
L'utilisateur devra déchiffrer le hash avec la clé publique pour le comparer avec le hash du fichier en clair, ce qui garantit l'intégrité.