

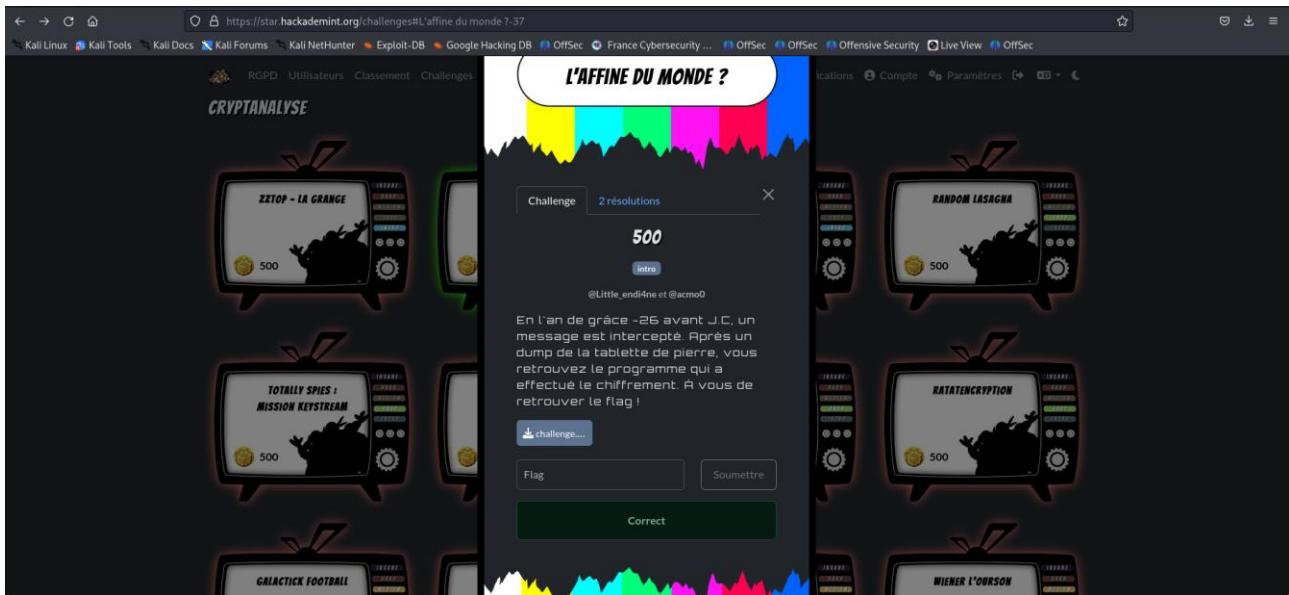
# Intro...Affine du monde

## 1. Résumé du challenge

Le challenge « Intro...Affine du monde ? » fournit le programme de chiffrement et un chiffré RSA. La faiblesse provient de la génération de la clé : q est choisi comme le premier nombre premier après p ( $q = \text{next\_prime}(p)$ ), ce qui rend p et q extrêmement proches. On factorise alors N rapidement (méthode de Fermat), puis on déchiffre pour retrouver le flag.

### Write-up – synthèse

#### Capture – énoncé (référence)



## 2. Notions utiles

Les éléments suivants suffisent à résoudre le challenge :

- RSA : chiffrement asymétrique avec  $N = p \cdot q$ , exposant public e, chiffré  $c = m^e \bmod N$ .
- Faiblesse :  $q = \text{next\_prime}(p) \Rightarrow p$  et  $q$  très proches  $\Rightarrow$  factorisation rapide par différence de carrés (Fermat).
- Arithmétique modulaire :  $\varphi(N) = (p-1)(q-1)$  et calcul de l'inverse modulaire  $d = \text{invmod}(e, \varphi(N))$ . (inverse modulaire de e modulo  $\varphi(N)$ ).
- Encodage : conversion int $\leftrightarrow$ bytes (big-endian) pour récupérer une chaîne UTF-8/ASCII (le flag).
- Validation a posteriori : re-chiffrement du clair m et vérification  $m^e \bmod N = c$ .

## 3. Collecte : récupérer N, e, c

Le challenge ne fournit pas toujours un fichier « output » dédié : les paramètres RSA sont simplement affichés par le script sous trois lignes ( $N$ ,  $e$ ,  $c$ ). Dans le fil de discussion, les fichiers `params.txt` et `output` correspondent à cette sortie ; `params.err` indique seulement une exécution interrompue (SIGINT/SIGTERM) après impression des paramètres.

### 3.1 Générer localement les paramètres

Deux variantes du générateur existent dans l'archive :

- `chall\_sage\_only.py` : version Sage (référence), reproductible dans un environnement Sage.
- `chall.py` : variante Python ; selon la version, elle peut contenir des restes de développement ou des dépendances (ex. PyCryptodome).

Commandes de génération (produit un `params.txt` exploitable par le solveur) :

```
sage -python chall_sage_only.py > params.txt  
# ou  
python3 chall.py > params.txt
```

### 3.2 Format attendu

Le fichier de paramètres doit contenir, en décimal :

```
N = <entier décimal>  
e = <entier décimal>  
c = <entier décimal>
```

## 4. Analyse de la faiblesse

Le point clé du challenge est la construction suivante :  $q$  est choisi comme le prochain nombre premier après  $p$  ( $q = \text{next\_prime}(p)$ ). Ainsi, l'écart  $q-p$  est faible, ce qui implique que  $p$  et  $q$  sont proches de  $\sqrt{N}$ . Cette propriété rend la factorisation de  $N$  très rapide via la méthode de Fermat (différence de carrés).

### 4.1 Rappel : méthode de Fermat (différence de carrés)

On cherche  $a$  et  $b$  tels que :

```
N = a^2 - b^2 = (a-b)(a+b)  
a = ceil(sqrt(N))  
b^2 = a^2 - N
```

Dès que  $b^2$  est un carré parfait, on obtient directement  $p = a-b$  et  $q = a+b$ . Quand  $p \approx q$ , la solution apparaît immédiatement (souvent dès  $i=0$ ).

## 5. Principales étapes de résolution

1. Lire  $N$ ,  $e$ ,  $c$  depuis `params.txt` (ou depuis la sortie du service).
2. Poser  $a = \text{ceil}(\sqrt{N})$ .
3. Calculer  $b^2 = a^2 - N$  et tester si  $b^2$  est un carré parfait.
4. En déduire  $p = a - b$  et  $q = a + b$ , puis vérifier  $p \cdot q = N$ .
5. Calculer  $\varphi(N) = (p-1)(q-1)$ .

6. Calculer la clé privée  $d = \text{invmod}(e, \varphi(N))$ .
7. Déchiffrer  $m = c^d \bmod N$ .
8. Convertir  $m$  (entier) en bytes (big-endian), puis décoder UTF-8/ASCII.
9. Afficher le flag.
10. Valider a posteriori en re-chiffrant : vérifier que  $m^e \bmod N = c$ .

## 6. Solveur et mode verbose

Le solveur fourni (`solver\_intro\_verbose\_stdout.py`) implémente exactement les étapes ci-dessus.

Dans le fil de discussion, le mode verbose a d'abord été conçu pour écrire sur stderr (pratique pour garder le flag seul sur stdout). À la mise à jour, il a été ajusté pour écrire sur stdout : les logs apparaissent, et le flag est imprimé en dernière ligne.

Exemples :

```
python3 solver_intro_verbose_stdout.py --params params.txt
python3 solver_intro_verbose_stdout.py --params params.txt -v
# en verbose, extraire uniquement le flag :
python3 solver_intro_verbose_stdout.py --params params.txt -v | tail -n 1
```

## 7. Résultat et preuve a posteriori

Sur les paramètres fournis dans `params.txt`, le message clair obtenu est :

Star{Th3\_eSC@Pe\_w4S\_\$0\_clo5e...}

Justification (preuve) : le solveur re-chiffre le clair  $m$  avec  $(N, e)$  et vérifie que le résultat est identique à  $c$  ( $c$ 'est une preuve constructive que le déchiffrement est correct).

## 8. Pièges et erreurs fréquentes

- Tronquer  $N/e/c$  (copie avec « ... ») : toute factorisation/déchiffrement devient impossible.
- Chercher une attaque RSA généraliste (Pollard Rho/ECM) alors que Fermat suffit ( $p$  et  $q$  trop proches).
- Oublier la conversion int→bytes (big-endian) ou utiliser une longueur incorrecte, ce qui corrompt le décodage.
- Ne pas faire la vérification a posteriori  $\text{pow}(m, e, N) == c$  (risque de faux positif).

## Annexe A – Fichiers à archiver

Pour un archivage propre (reproductibilité), conserver au minimum :

- chall.py
- chall\_sage\_only.py
- Capture d'écran 2025-09-04 185322.png

- params.txt (ou output)
- params.err (optionnel)
- solver\_intro\_verbose\_stdout.py

Preuve de lancements en environnement :

```

kali@kali:~/mnt/Share/Starhackademint2025/Cryptanalyse/Intro
$ python3 solver_intro.py
Star{Th3_eSC@Pe_w45_$.clo5e...

```

```

total 362
drwxr-x--- 1 root vboxsf 4096 Jan 4 18:30 .
drwxr-x--- 1 root vboxsf 12288 Jan 4 11:32 ..
-rw-rwx--- 1 root vboxsf 309527 Sep 4 12:53 Capture d'écran 2025-09-04 185322.png
-rw-rwx--- 1 root vboxsf 1024 Sep 4 12:53 challenge_by_chall.py
-rw-rwx--- 1 root vboxsf 526 Sep 4 15:11 chall_sage_only.py
-rw-rwx--- 1 root vboxsf 1641 Sep 4 15:19 exploit_dalton_fermat.py
-rw-rwx--- 1 root vboxsf 1253 Sep 4 13:40 output
-rw-rwx--- 1 root vboxsf 1253 Sep 4 13:40 params_err
-rw-rwx--- 1 root vboxsf 1253 Sep 4 13:40 params_out
-rw-rwx--- 1 root vboxsf 1131 Jan 4 12:18 solver_intro.py
-rw-rwx--- 1 root vboxsf 9712 Jan 4 12:40 solver_intro_v2.py
-rw-rwx--- 1 root vboxsf 9292 Jan 4 18:29 solver_intro_verbose_stdout.py

[kali@kali]:~/mnt/Share/Starhackademint2025/Cryptanalyse/Intro
$ python3 solver_intro.py
Star{Th3_eSC@Pe_w45_$.clo5e...

[kali@kali]:~/mnt/Share/Starhackademint2025/Cryptanalyse/Intro
$ python3 solver_intro_verbose_stdout.py --params params.txt -v | tail -n 1
Star{Th3_eSC@Pe_w45_$.clo5e...

[kali@kali]:~/mnt/Share/Starhackademint2025/Cryptanalyse/Intro
$ python3 solver_intro_verbose_stdout.py --params params.txt -v

[*] reading params from: params.txt
[*] N bits = 2048
[*] e = 65537
[*] c bits = 2046
[*] factoring N with Fermat (expects p and q close)
[fermat] N = 0x8f24bdcb1b999..5957d060e02d (bits=2048)
[fermat] N = 13442542607139301882687864264193989873084611277643153854540012996839424511009797134543235157887377316043198803348202818839902079504849370523371603176797845881268326113856412488450468838581283440328667542063971300745
[fermat] p = 0xbff69c07b390..44f2e3f31a51 (bits=1024)
[fermat] q = 0xbff69c07b390..44f2e3f231d (bits=1024)
[fermat] q-p = 2252
[*] phi = (p-1)(q-1) = (p-1)(q-1)
[*] phi bits = 2048
[*] 3/5 computing d = inverse(e) mod phi(N)
[*] d bits = 2045
[*] 4/5) decrypting ... x^d mod N
[*] 4/5) 0x73727b54..35652e1e2e7d (bits=255)
[*] plaintext length (bytes) = 32
[*] plaintext (hex) head/tail = 5374617727b5468335f6553434050655f .. 773453f244f5f636c6f35652e2e2e7d
[*] decoded flag candidate = 'Star{Th3_eSC@Pe_w45_$.clo5e ...}'
[*] validation ok
[*] validation ok
Star{Th3_eSC@Pe_w45_$.clo5e...

```