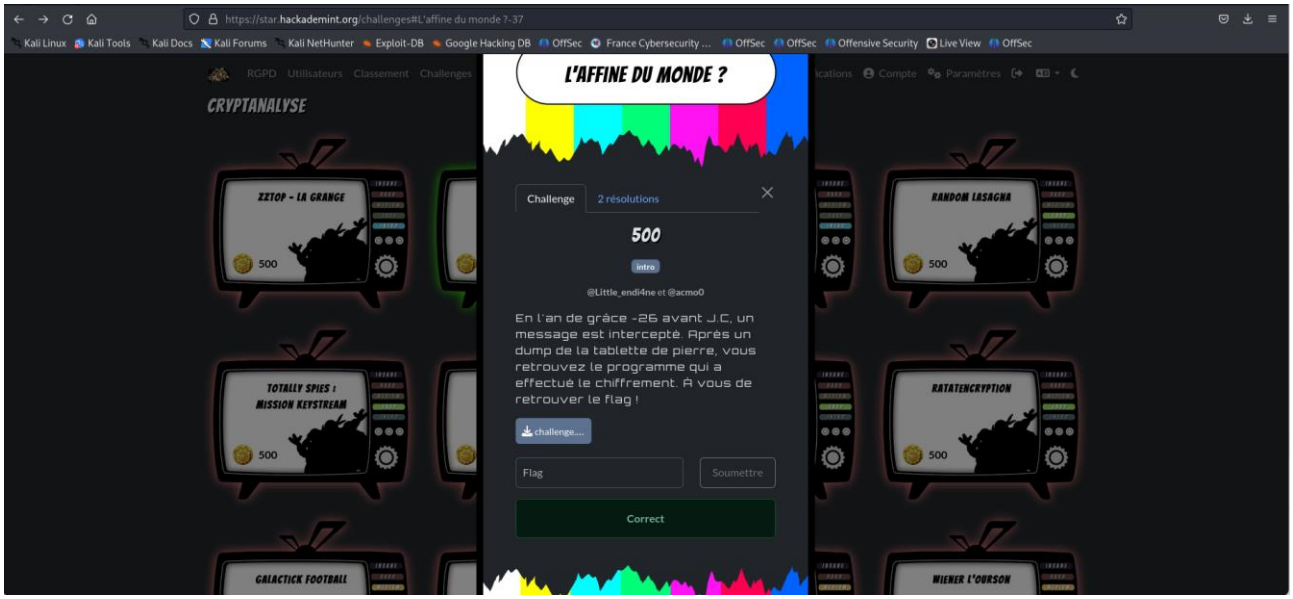
Intro…Affine du monde

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 = 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·q, exposant public e, chiffré c = m^e mod N.
- Faiblesse : q = next_prime(p) ⇒ p et q très proches ⇒ factorisation rapide par différence de carrés (Fermat).
- Arithmétique modulaire : φ(N) = (p−1)(q−1) et calcul de l'inverse modulaire d = invmod(e, φ(N)). (inverse modulaire de e modulo φ(N)).
- Encodage : conversion int↔bytes (big-endian) pour récupérer une chaîne UTF-8/ASCII (le flag).
- Validation a posteriori : re-chiffrement du clair m et vérification m^e mod 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 = 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≈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 = ceil(sqrt(N)).

3. Calculer b2 = a^2 − N et tester si b2 est un carré parfait.

4. En déduire p = a − b et q = a + b, puis vérifier p·q = N.

5. Calculer φ(N) = (p−1)(q−1).

6. Calculer la clé privée d = invmod(e, φ(N)).

7. Déchiffrer m = c^d mod 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 mod 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_$O_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 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 :