

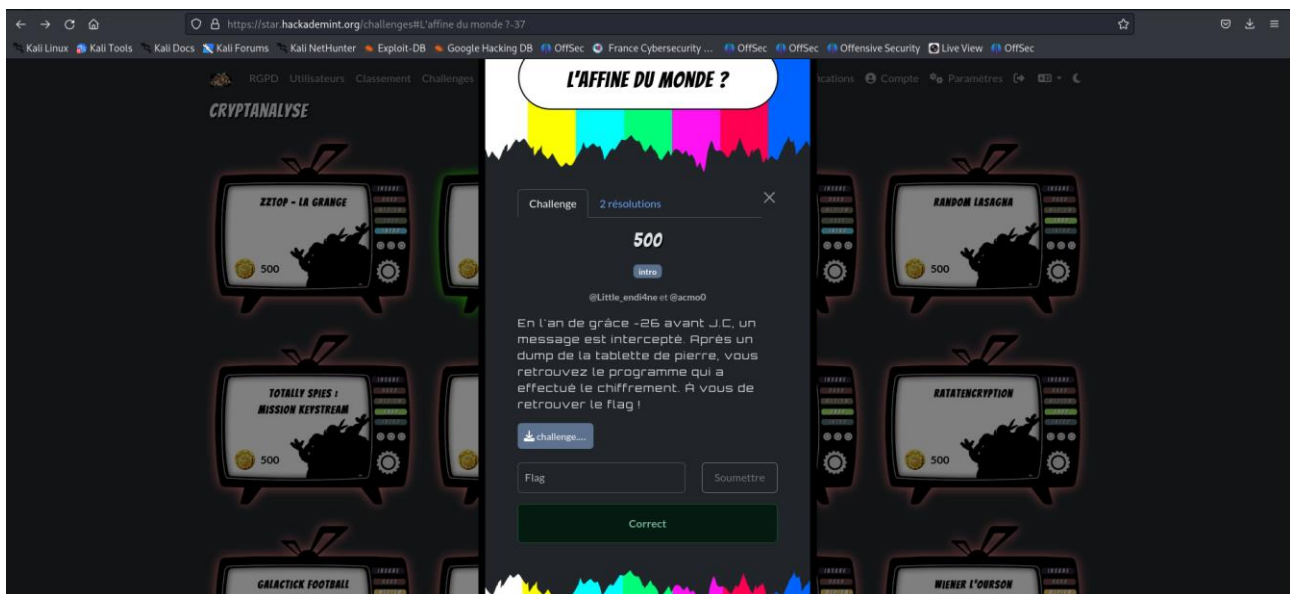
# Intro...Affine du monde

## 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  $\text{int} \leftrightarrow \text{bytes}$  (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 :

$N = \text{<entier décimal>}$

$e = \text{<entier décimal>}$

$c = \text{<entier décimal>}$

## 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 :

$$N = a^2 - b^2 = (a-b)(a+b)$$

$$a = \text{ceil}(\sqrt{N})$$

$$b^2 = a^2 - N$$

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\_\$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).

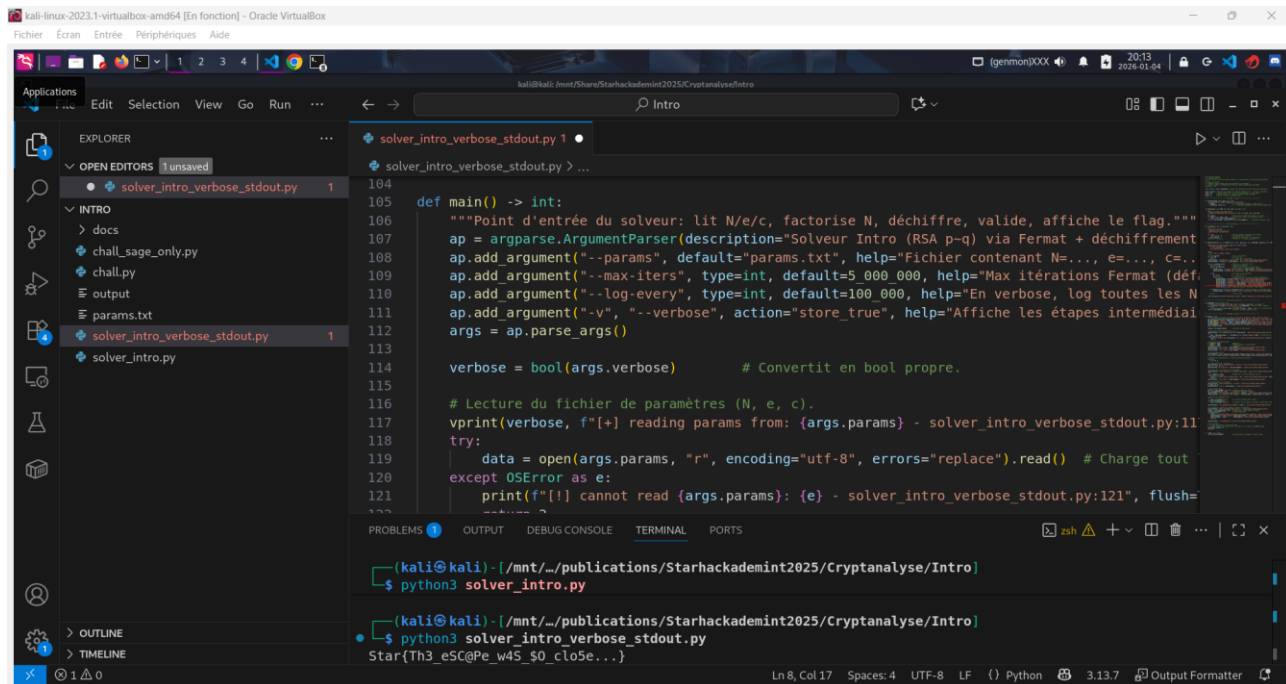
## Annexe A – Fichiers à archiver

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 :

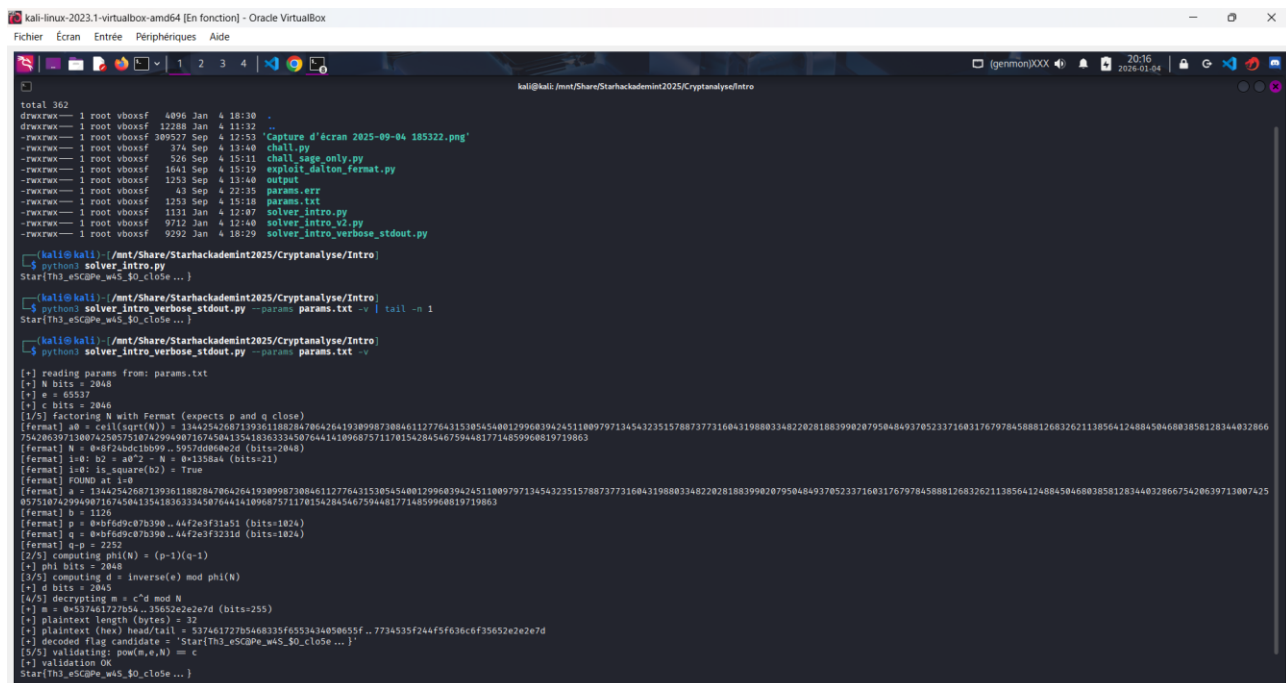


The screenshot shows a Kali Linux virtual machine environment. The main window is a code editor displaying a Python script named `solver_intro_verbose_stdout.py`. The script is a command-line tool for solving RSA problems using Fermat's method. It includes a `main()` function that parses command-line arguments, reads parameters from a file, and performs the solving process. The terminal window at the bottom shows the command `python3 solver_intro.py` being executed, which results in a `Star{Th3_eSc@Pe_w45_$0_c105e...}` output.

```
def main() -> int:
    """Point d'entrée du solveur: lit N/e/c, factorise N, déchiffre, valide, affiche le flag."""
    ap = argparse.ArgumentParser(description="Solveur Intro (RSA p-q) via Fermat + déchiffrement")
    ap.add_argument("--params", default="params.txt", help="Fichier contenant N=..., e=..., c=...")
    ap.add_argument("--max-iters", type=int, default=5_000_000, help="Max itérations Fermat (déf. 5M)")
    ap.add_argument("--log-every", type=int, default=100_000, help="En verbose, log toutes les N")
    ap.add_argument("-v", "--verbose", action="store_true", help="Affiche les étapes intermédiaires")
    args = ap.parse_args()

    verbose = bool(args.verbose) # Convertit en bool propre.

    # Lecture du fichier de paramètres (N, e, c).
    vprint(verbose, f"[+] reading params from: {args.params} - solver_intro_verbose_stdout.py:11")
    try:
        data = open(args.params, "r", encoding="utf-8", errors="replace").read() # Charge tout
    except OSError as e:
        print(f"[!] cannot read {args.params}: {e} - solver_intro_verbose_stdout.py:121", flush=True)
        return 1
```



The screenshot shows a Kali Linux virtual machine environment. The terminal window displays the output of the `solver_intro.py` script. It shows the script reading parameters from `params.txt`, factoring `N` using Fermat's method, and successfully finding the flag `Star{Th3_eSc@Pe_w45_$0_c105e...}`.

```
[+] reading params from: params.txt
[+] N bits = 2048
[+] e = 65537
[+] c bits = 2048
[+] factoring N with Fermat (expects p and q close)
[+] a0 = 13442542687139361188204786426419389987308461127764315305454001299683942451100979713454323515788737731604319880334822028188399020795048493705233716031767978458881268326211385641248845046803858128344032866754286397130074258751074299490716745041354183633345076441410968757117015428454675944817714859960819719863
[+] N = 0x8f24dc13b09...9937d08b0e2d (bits=2048)
[+] i0: b2 = a0^2 - N = 0x138aa (bits=21)
[+] i0: is_square(b2) = True
[+] FOUND at i=0
[+] a = 13442542687139361188204786426419389987308461127764315305454001299683942451100979713454323515788737731604319880334822028188399020795048493705233716031767978458881268326211385641248845046803858128344032866754286397130074258751074299490716745041354183633345076441410968757117015428454675944817714859960819719863
[+] b = 1126
[+] p = 0xbfd9c07b390...44f2e3f31a51 (bits=1024)
[+] q = 0xbfd9c07b390...44f2e3f3231d (bits=1024)
[+] q-p = 2252
[+] computing phi(N) = (p-1)(q-1)
[+] phi bits = 2048
[+] computing d = inverse(e) mod phi(N)
[+] d bits = 2045
[+] decrypting m = c^d mod N
[+] m = 0x2746127b54...3862e2e2e7d (bits=255)
[+] plaintext length (bytes) = 32
[+] plaintext (hex) head/tail = 537461727b546833f6553434050655f...7734535f244f5f636cf35652e2e2e7d
[+] decoded flag candidate = 'Star{Th3_eSc@Pe_w45_$0_c105e...}'
[+] validating: pow(m,e,N) == c
[+] validation OK
Star{Th3_eSc@Pe_w45_$0_c105e...}
```