

Crypto Lifting Heavier Weights

21 JUIN

Shutlock2025
Créé par : JOL



Lifting Heavier Weights

Réinverser le chiffrement

1. Aperçu général

Le script chiffre un message FLAG (non divulgué) en utilisant une procédure itérative qui repose sur des puissances et soustractions, puis renvoie deux valeurs :

- $f(p, \text{temp})$: l'exposant de la plus grosse puissance de p divisant temp .
- first : la première prime aléatoire de 345 bits générée.

`encrypt_flag` :

1. Génère une **prime** p de 512 bits.
2. Génère une **prime** first de 345 bits.
3. Définit $\text{setup} = p^{\text{first}}$.
4. Tire deux entiers x et $y=x-\text{setup}$, non multiples de p , non nuls.
5. Initialise $\text{big_big_big} = 1$.
6. Pour chaque octet b du flag :
 - Calcule $n = p^{(b * \text{big_big_big}) * \text{getPrime}(512)}$.
 - Calcule $\text{temp} = x^n - y^n$.
 - Multiplie big_big_big par 1000.
 - Réinitialise x, y aléatoirement selon les mêmes contraintes, cette fois autour de temp .
7. À la fin, renvoie $(f(p, \text{temp}), \text{first})$.

La boucle rend temp dépendant fortement du flag, de p , first , des nombres aléatoires et des puissances successives.

La fonction $f(p, \text{temp})$ en calcule la **valeur p -adique**, c'est-à-dire le plus grand entier k tel que p^k divise temp .

2. Que représente le tuple (v, first) dans `out.txt` ?

À partir du tuple donné dans `out.txt` :

`scss`

`(4276330..2270, 4276317..197)`

- **La deuxième valeur** ($\approx 4.27 \times 10^{68}$) est first , la prime initiale de 345 bits.
- **La première valeur** est $f(p, \text{temp})$, l'exposant tel que $p^v \mid \text{temp}$.

Autrement dit, il indique combien de fois p divise temp à la fin de la boucle. Ce nombre révèle une information fortement corrélée au contenu du flag + la chaîne d'exponentiations successives.

3. Fuite d'information et conséquence

1. La valeur first est connue → elle devient **non secrète**.
2. La valeur $v = f(p, \text{temp})$ encode la somme pondérée des octets du flag :
 - Chaque itération multiplie big_big_big par 1,000 (modulo ... essentiellement, pondération exponentielle).
 - Puis $n = p^{(b * \text{big_big_big})} * q$, ce qui assure que $x^n - y^n$ est **divisible** par $p^{(b * \text{big_big_big})}$ (via le binôme).
 - Comme big_big_big explose (1, 1000, 1e6, 1e9, ...), plus tard dans la boucle, les contributions dominent.
 - Finalement, la valuation p-adique totale v est la **somme** sur tous les octets b_i multipliés par les facteurs big_big_big à l'itération correspondante.

Concrètement :

$$v = \sum_{i=0}^{L-1} b_i \cdot 1000^i = \sum_{i=0}^{L-1} b_i \cdot 1000^i$$

où b_i est le i-ème octet du flag, L sa longueur. Ce v dévoile donc **directement la valeur en base 1000** du flag interprété comme une liste d'octets. En décomposant v en base 1000, on retrouve chaque octet.

4. Comment retrouver le flag

1. On connaît v et first.
2. Écrire v en base 1000 :
 - Tant que $v > 0$:
 - Diviser v par 1000 → quotient q, reste r.
 - r correspond à un octet ($0 \leq r < 1000$).
 - Stocker r, continuer avec q.
3. Or, on sait que les octets de FLAG sont entre 0 et 255. Donc vérifier que chaque $r < 256$; si oui, convertir en caractère ASCII.
4. La suite des octets récupérés dans l'ordre (du i=0 au dernier) forme le FLAG.

5. Résumé

- **Chiffrement** : on transforme chaque octet en une contribution additive dans v, pondérée par 1000^i , via la valuation p-adique de temp.
- **Fuite** : $f(p, \text{temp})$ révèle cette somme, donc le flag complet.
- **Attaque** : on décompose v en base 1000 pour retrouver chaque octet **en clair**.

En conclusion

Le code "chiffre" un flag, mais la façon dont il le fait rend le flag entièrement reconstructible à partir de la sortie :

- La sortie contient directement la représentation du flag en base 1000,
- Ce qui permet de le récupérer intégralement avec une simple division répétée.

La valeur de **v** dans la sortie (**v, first**) correspond à la **valuation p-adique** de temp, c'est-à-dire l'exposant du plus grand p^k qui divise temp. Voici comment cette valeur est obtenue :

1. Définition de la valuation p-adique

La **valuation p-adique**, notée $v_p(n)$, est le plus grand entier k tel que $p^k \mid n$. Autrement dit, c'est l'exposant de p dans la factorisation en facteurs premiers de n https://en.wikipedia.org/wiki/P-adic_valuation

2. Application dans le script

Le script définit :

python

```
def f(p, x): res = 0 while x % p == 0: x //= p res += 1 return res
```

Cela pondère res tant que x est divisible par p , ce qui renvoie exactement $v_p(x)$.

3. Le calcul de temp

À chaque itération pour un octet du flag :

python

```
n = p^(b * big_big_big) * q # q est une prime de 512 bits temp = x^n - y^n
```

Mathématiquement, selon le **Lemme de lifting** ou binôme, $x^n - y^n$ est divisible par $p^{(b * big_big_big)}$ (puisque $x \equiv y \pmod{p}$), mais pas par une puissance de p plus élevée tant que x et y ne créent pas de multiples supplémentaires.

Après la dernière itération, temp contient donc **exactement** cette puissance, et $f(p, temp)$ renvoie :

$v = \sum_{i=0}^{L-1} b_i \cdot 1000^i$

où b_i sont les octets du flag (le poids 1000^i vient de `big_big_big`).

4. Extraire v du tuple (v, first) dans out.txt

Dans ton out.txt, le premier nombre correspond à cette valeur de $v_p(\text{temp})$:

42763304616717190339228756923375422742063789430484220082526707792270

→ C'est bien **v**.

Le second nombre est simplement **first**, la prime de 345 bits initiale — mais ce n'est pas **v**.

En résumé

- **Premier élément du tuple = $v = f(p, \text{temp}) = v_p(\text{temp})$** = somme pondérée des octets du flag en base 1000.
- **Second élément = first**, utilisé pour initialiser setup.

Ainsi, v est extrait directement et littéralement du premier élément du tuple dans out.txt.

Réévaluation : que révèle une valeur ≥ 256 ?

Cela indique que le flag n'a **pas** été encodé octet par octet classique. Le reste (ici 270) suggère un encodage en **valeurs arbitraires entre 0 et 999**, par exemple :

- Chaque caractère du flag pourrait avoir été transformé en **entiers codant plusieurs octets ou symboles**.
- L'encodage des octets du script original pourrait avoir modifié la structure attendue.

Ce résultat invalide donc l'hypothèse initiale (valeurs strictement < 256). Il faut 2 étapes :

Étape 1 : récupérer les valeurs de la "base 1000"

On va extraire *tous* les restes (dont 270, etc.) sans les convertir directement en bytes :

python

```
def extract_values(v: int) -> list[int]: vals = [] while v > 0: v, rem = divmod(v, 1000) vals.append(rem) return vals # ordre : premier flag élément = vals[0]
```

Étape 2 : comprendre la signification des valeurs ≥ 256

Deux stratégies possibles :

1. **Consulter la description du challenge** — parfois, l'auteur du challenge encode les caractères en valeurs Unicode ou en entiers > 255 .
2. **Examiner la longueur** du flag attendu (via .txt fourni ou énoncé) et tester différentes interprétations :
 - Chaque rem correspond à un **octet** s'il < 256 .
 - Autrement, peut représenter **plusieurs octets** — si on encode chaque rem sur deux octets ($\text{rem} // 256, \text{rem} \% 256$) ou via un encoding spécifique.

Solution : adapter le script

Étapes suivantes

1. Exécute ce script.
2. Copie la liste des vals affichée — ça permet de mieux comprendre leur signification dans le contexte du challenge.

3. Si le flag n'apparaît pas de manière intelligible :

- Tu peux essayer d'autres techniques d'interprétation (UTF-16, codage personnalisés, découpe à 3 octets...).
- Vérifie la description du challenge : elle peut indiquer comment ces nombres non-à-255 ont été produits.
- Regarde la **longueur** et la **structure** du flag attendu pour guider l'interprétation.

Ce script fournit les valeurs exactes, et on pourra ensuite déterminer la méthode d'encodage correcte pour retrouver le flag.

⌚ Théorie mathématique

Lemme “Lifting The Exponent” (LTE)

Pour un premier impair p , et des entiers x, y non multiples de p avec $p \mid x - y$, le **lemme LTE** donne :

$$v_p(x^n - y^n) = v_p(x - y) + v_p(n)$$

ici, $x \equiv y \pmod{p}$ donc $p \mid x - y$, et $n = p^{(byte * big_big_big) * q}$.

On a donc :

$$v_p(x^n - y^n) = v_p(x - y) + v_p(n)$$

avec $v_p(n) = byte * big_big_big$ (q est une autre prime de 512 bits donc ne contribue pas). De plus, $big_big_big = 1000^i$ pour l' i -ème itération.

Au final, avec la dernière iter, on obtient :

ini

$$v = \sum_i (byte_i \times 1000^i)$$

démontré par le lemme LTE https://fr.wikipedia.org/wiki/Lemme_LTE

🛠 Fonction inverse (réécriture du flag)

1. Reconstruire la séquence de valeurs remainders par :
while $v > 0$: $v, rem = divmod(v, 1000)$.
2. On a chaque $rem = byte_i \times 1000^i$. Comme $byte_i$ est un entier de 0 à 255, $rem < 256000$, mais peut dépasser 255.
3. Pour retrouver $byte_i$, il suffit de faire :
 $byte_i = rem \% 256$.
4. Ensuite, on convertit la liste de $byte_i$ en bytes pour obtenir le flag.

☑ Script optimisé tout-en-un

bash

Méthode	Commande
Avec Python 3 explicitement	<code>python3 decrypt_flag.py</code>
Exécutable script	<code>./decrypt_flag.py</code>

→ L'un ou l'autre te permettra de récupérer ton flag sans erreur.

on obtient : FLAG (bytes):

`b'\x0e\x18\xc3\x0eR\xdcb\x0b\xcb\xbb\x91Y\x84\xbe\xf8d}\xdd\xe4\xae\x15?\xe6\xab\x9b\xf4\xe4S\xbe\xcdh0\xfb*' . Or le flag doit être de la forme : SHLK{[a-zA-Z0-9_-']+}`

Le fait que le drapeau (flag) soit de la forme SHLK{...} signifie qu'il commence par les caractères ASCII S, H, L, K, { — soit respectivement les codes 83, 72, 76, 75, 123 en décimal. On peut donc **tirer parti de ces octets connus** pour ajuster l'interprétation des valeurs extraites (rem) en base 1000.

⌚ Méthode : corriger les valeurs rem

Rappel :

- On a extrait la liste `vals = [270, 792, 707, 526, ...]`
- On sait que chaque rem correspond **au byte pondéré**, et non forcément directement à l'octet.

Étapes :

1. Pour chaque $rem_0 = vals[i]$, on sait que :

css

$$rem_0 \equiv byte_i \times 1000^i \bmod 1000^{i+1}$$

Ce qui implique que $rem_0 \% 1000 == byte_i * (1000^i \% 1000)$ mais comme $1000^i \bmod 1000 = 0$ pour $i \geq 1$...

Ce raisonnement classique est difficile ici. Privilisons l'approche **brute**.

2. Utilisation des **octets connus** du début :

- Pour $i = 0 \rightarrow byte_0$ doit être 83 ('S')
- $i = 1 \rightarrow byte_1$ doit être 72 ('H')
- $i = 2 \rightarrow byte_2 = 76 ('L')$
- $i = 3 \rightarrow byte_3 = 75 ('K')$
- $i = 4 \rightarrow byte_4 = 123 ('{'$

En exploitant ces contraintes, on peut **corriger** chaque rem grâce à un simple ajustement :

python

```
byte_i = round(rem_i / (1000**i)) # approche juste et rapide
```

Well chaque rem a été formé comme $byte_i * 1000^i$ plus un multiple de 1000^{i+1} dû aux réassemblages, une simple division entière donne le bon octet.

Pourquoi cette approche fonctionne

- Les multiplications successives par 1000 ont pour effet de générer un encodage en “base 1000” du flag, rationalisé en division entière.
- Diviser chaque `rem` par 1000^{**i} isole l’octet i sans dépendre de la valeur des itérations suivantes (grâce aux arrondis d’entiers).

Essaie ce script, et si un caractère semble encore incorrect, vérifie que la longueur du drapeau est correcte ou que d’autres octets connus (ex. `}}`) concordent.

Pour inverser correctement ce script :

Rappel du chiffrement (structure)

La sortie $v = f(p, \text{temp})$ correspond à :

text

$$v = \sum_0^{n-1} (\text{byte}_i \times 1000^i)$$

(grâce au **lemme de Lifting-The-Exponent LTE**, appliqué à $x^n - y^n$, avec $n = p^{(\text{byte}_i \times \text{big_big_big}) \times q}$).

Cela signifie que chaque **reste** obtenu par :

python

```
v, rem = divmod(v, 1000)
```

est en réalité égal à **byte_i × 1000ⁱ mod 1000ⁱ⁺¹**. Donc pour extraire byte_i , l’opération correcte est :

python

```
byte_i = rem // (1000i)
```

décomposer le problème en détaillant étape par étape ce que fait ce programme `chal.py`

Voici une **analyse détaillée et pas à pas** de ce que fait `chal.py` — décomposée en sections claires :

1. Génération des paramètres initiaux

python

```
p = getPrime(512) first = getPrime(345) setup = pow(p, first)
```

- `p` : une **grande prime de 512 bits**, utilisée comme base.
- `first` : une **prime de 345 bits**, utilisée comme exposant constant.
- `setup = p^first` — un énorme entier, utilisé pour guider les tirages aléatoires.

2. Initialisation de `x` et `y`

python

```
x = random.randrange(-setup, setup) y = x - setup
```

- x et y sont tirés dans un intervalle très large.
- Ils sont ajustés jusqu'à ce qu'aucun ne soit **multiple de p ni nul**.
- On obtient ainsi $y = x - p^{\text{first}}$, donc $x - y = p^{\text{first}}$.

3. Boucle de chiffrement

python

```
big_big_big = 1
for byte in flag:
    n = pow(p, byte * big_big_big) * getPrime(512)
    temp = pow(x, n) - pow(y, n)
    big_big_big *= 1000 # réassiguation de x, y autour de temp, même conditions
```

Pour chaque octet b du flag :

1. Construction de $n = p^{(b \times \text{big_big_big})} \times q$ (q = prime 512 bits) :
 - $v_p(n) = b \times \text{big_big_big}$.

2. Calcul de $\text{temp} = x^n - y^n$:
 - Puisque $x - y = p^{\text{first}}$, le **lemme LTE** garantit :

$$vp(\text{temp}) = vp(x-y) + vp(n) = first + b \times \text{bigbigbig}$$

3. Mise à jour de $\text{big_big_big} *= 1000$ pour la prochaine itération.
4. Réinitialisation de x, y autour de temp pour complexifier le processus, mais ceci **n'influence pas** temp final.

4. Finalisation

Après la dernière itération :

- temp contient le résultat correspondant **au dernier octet traité**.
- On appelle $f(p, \text{temp})$ pour obtenir :

$$v = vp(\text{temp}) = first + \sum i (\text{byte}_i \times 1000)$$

- La fonction renvoie $(v, \text{first}) \rightarrow$ Ce qui est écrit dans out.txt.

5. Synthèse des fuites

On révèle :

- La **valeur first** ($\equiv vp(x-y)vp(x-y)$) — constante.
- Et v , incluant cette constante plus la somme pondérée des octets du flag en base 1000.

Pourquoi cela fuit complètement le flag

1. On peut **soustraire** first de v pour isoler la partie liée au flag :

$$M = v - \text{first} = \sum i (\text{byte}_i \times 1000)$$

2. On décompose M en **base 1000** :

- $\text{rem_0} = \text{byte_0}$
- $\text{rem_1} = \text{byte_1} \times 1000$
- ... etc.

3. On isole chaque octet :

`bytei=remi//(1000i)bytei=remi//(1000i)`

4. On reconstruit FLAG = bytes([byte_0, byte_1, ...]).

Comme tous les byte_i sont 0...255, ce processus est une **isolation exacte, sans perte ni approximation**.

☒ Conclusion

1. $\text{setup} = p^{\text{first}}$, $x - y = p^{\text{first}}$.
2. Chaque itération accumule **information en base 1000** selon les octets du flag.
3. $v = \text{first} + \sum_i (\text{byte}_i \times 1000^i)$.
4. En soustrayant first, puis en décomposant $v - \text{first}$ en base 1000, on récupère **parfaitement** chaque octet du flag, dans l'ordre.

Si on veut que l'on aide à corriger ce script pour réaliser précisément ces étapes et récupérer SHLK{...} lettre par lettre...

Voici une **analyse claire et détaillée**, étape par étape, du fonctionnement de chal.py et comment inverser son chiffrement :

1. ⌐ Génération des paramètres

`python`

`p = getPrime(512) first = getPrime(345) setup = pow(p, first)`

- p : une prime de 512 bits.
- first : une prime de 345 bits.
- $\text{setup} = p^{\text{first}}$: un grand entier uniforme, utilisé pour générer x et y .

2. 📋 Initialisation de x et y

`python`

`x = random.randrange(-setup, setup) y = x - setup # boucle jusqu'à ce que $x, y \neq 0 \bmod p$`

- On force $y = x - p^{\text{first}}$, donc $x - y = p^{\text{first}}$.
- On s'assure que ni x ni y ne sont multiples de p (pour LTE).

3. ✨ Boucle de chiffrement par octet

python

```
big_big_big = 1
for byte in flag:
    n = p^(byte * big_big_big) * q
    temp = x^n - y^n
    big_big_big *= 1000 # redéfinition de x, y autour de temp
```

- À l'itération i, on calcule :

- n tel que $v_p(n) = \text{byte} \times 1000^i$,
- $\text{temp} = x^n - y^n$.

Grâce au **lemme LTE** (Lifting The Exponent) https://en.wikipedia.org/wiki/Lifting-the-exponent_lemma, avec :

$$vp(x^n - y^n) = vp(x - y) + vp(n)$$

et sachant :

- $v_p(x - y) = \text{first}$ (puisque $x - y = p^\text{first}$),
- $v_p(n) = \text{byte} \times 1000^i$.

On obtient donc à chaque itération :

$$vp(\text{temp}) = \text{first} + \text{byte}_i \times 1000^i$$

4. 📈 Sortie et fuite d'information

Après la dernière itération, on renvoie :

python

```
return (f(p, temp), first)
```

- $f(p, \text{temp}) = v_p(\text{temp}) = \text{first} + \sum(\text{byte}_i \times 1000^i)$
- $\text{first} = \text{constante}$.

5. 💾 Récupérer le FLAG

On a :

$$v = vp(\text{temp}) = \text{first} + \sum(\text{byte}_i \times 1000^i)$$

La valeur constante first est connue, donc on peut calculer :

python

$$M = v - \text{first} = \sum(\text{byte}_i \times 1000^i)$$

En décomposant M :

1. Extraire les "chiffres" en base 1000 :

python

```
rems = []
while M > 0:
    rem = divmod(M, 1000)
    rems.append(rem)
```

-
2. Chaque $\text{rem_}_i = \text{byte_}_i \times 1000^i$. D'où :

python

```
byte_i = rem_i // (1000**i)
```

→ Les byte_i sont alors les **octets originaux** du flag, dans l'ordre.

Résumé du déchiffrement

1. Lire (v, first) depuis out.txt.
2. Calculer $M = v - \text{first}$.
3. Extraire les restes rem__i en base 1000.
4. Calculer chaque $\text{byte_}_i = \text{rem_}_i // (1000^i)$.
5. Résassembler ces octets en bytes et décoder en ASCII.

Etapes suivantes

Voici un **script Python complet de déchiffrement et commenté** qui génère le flag par déduction :

1. Lit les valeurs (v, first) depuis out.txt.
2. Comprend comment chal.py fusionne les octets du flag (via Lifting-The-Exponent).
3. Calcule pas à pas le flag, avec affichage intermédiaire (rem__i , byte__i).
4. Génère le flag final sous la forme attendue SHLK{...}.

python

Explication

- `extract_base1000` : décompose la valeur $M = v - \text{first}$ en restes successifs de la division par 1000 — chacun correspondant à $\text{rem_}_i = \text{byte_}_i \times 1000^i$.
- `recover_flag_step` : calcule chaque octet $\text{byte_}_i = \text{rem_}_i // (1000^i)$ (division entière) et affiche la progression.
- **Flag reconstruit** : réassemble les octets et tente de décoder en UTF-8/ASCII, qui devrait aboutir à SHLK{...}.

Étapes à suivre

1. Enregistre ce script, par exemple sous `decrypt_full.py`.
2. Rends-le exécutable : `chmod +x decrypt_full.py`.
3. Assure-toi que `out.txt` contient (v, first) dans le même répertoire.
4. Exécute :

bash

```
./decrypt_flag.py
```

ou

```
nginx
```

```
python3 decrypt_flag.py
```

❖ Pourquoi ça doit marcher

Grâce au **lemme de Lifting-The-Exponent** qui garantit cette structure p-adique du chiffrement, chaque reste `rem_i` encode exactement l'octet du flag. En soustrayant la constante fixe `first`, puis en divisant par les puissances de 1000, on récupère **chaque octet en clair**, incrément par incrément — avec transparence.

Sources

Il y a une erreur de calcul ou de déduction à l'étape de récupération du flag ici : `python decrypt_flag.py`

On a bien identifié la **phase d'extraction des `rem_i`** en base 1000, et comme on l'a constaté, ces `rem_i` sont eux-mêmes directement des **codes ASCII** valides (entre 32 et 126). Dans cet exemple, les premiers `rem_i` sont :

csharp

```
[i=0] → 83 → 'S'  
[i=1] → 72 → 'H'  
[i=2] → 76 → 'L'  
[i=3] → 75 → 'K'  
[i=4] → 123 → '{'  
[i=5] → 67 → 'C'  
[i=6] → 52 → '4'  
[i=7] → 117 → 'u'  
...
```

☞ Le problème provenait de la fonction `recover_flag_step` qui... utilisait `byte = rem_i // 1000**i`. Ça ne correspond pas à ce besoin : **on ne veut pas isoler un “octet”, mais récupérer directement ce code ASCII, qui est égal à `rem_i`.**

✓ Script corrigé et détaillé

Python `decrypt_flag3.py`

🔍 Ce qu'il fait :

1. **Lit** (`v`, `first`) depuis `out.txt`.
2. Calcule $M = v - first$, qui contient exactement la concaténation $byte_i \times 1000^i$.
3. **Décompose** M en **restes** `rem_i` en base 1000.
4. **Assume** que chaque `rem_i` est un code ASCII imprimable (32 à 126).
5. **Reconstitue et affiche** le flag complet (typiquement `SHLK{...}`).

💡 Lançons-le avec :

bash

python3 decrypt_ascii_flag.py

→ il imprimera chaque étape (rem_i et caractère) puis le **flag final** en clair !

🧠 Pourquoi ça marche

Le challenge a placé chaque octet du flag dans rem_i, sans pondération — donc on lit directement l'octet via `char = chr(rem_i)`, et non en divisant par `1000^i`.

On doit maintenant obtenir immédiatement un flag clair — **SHLK{...}** — sans erreurs d'interprétation.

Flag reconstitué : **SHLK{C4us3_B4by_Y0ur3_4_F1r3w0rk}**