

Crypto Lifting Weights

21 JUIN

Shutlock2025
Créé par : JOL



Lifting Weights solution

Décomposer le problème étape par étape sur ce que fait ce script : chal.py

Voici une décomposition pas à pas du code de **chal.py** (issu de cet énoncé) :

Un message a été chiffré par un algorithme lourd à déchiffrer. Ma mission : déchiffrer ce colosse sans attendre des années.→ Résultat attendu : un flag de la forme SHLK{...}, en conformité avec cet énoncé.

☞ Pour la résolution, sont utilisés :

1. Le script de l'énoncé **chal.py**
2. Le **token chiffré** (ou équivalent hexa ou base64) ?

Avec ces éléments, il peut se cartographier puis s'exploiter ce “colosse chiffré à déchiffrer”💡 :

- Analyser précisément l'algorithme utilisé (comme on a vu pour la version DEMO),
- Rechercher des faiblesses (padding oracle, collisions MD5, faiblesse du IV, etc.),
- Proposer ou automatiser une attaque pour retrouver le flag SHLK{...} rapidement.

```
> Lifting weights > 📁 chal.py > ...
from Crypto.Util.number import getPrime
import random

FLAG = b'????????????????????????????????????'

def f(x):
    res = 0
    while x % 2 == 0 :
        x //=
        res += 1
    return res

def encrypt_flag(flag):
    encrypted = []
    for byte in flag:
        r = getPrime(512)
        n = pow(2, r * byte)
        x = random.randrange(-n, n, 2) + 1
        y = x - n
        t = pow(x, n) - pow(y, n)

        encrypted.append((r, f(t)))

    return encrypted

with open('out.txt', 'w') as f:
    f.write(str(encrypt_flag(FLAG)) + "\n")
```

Voici comment contourner ce chiffrement « colossal » :

🔍 Comprendre l'algorithme

Pour chaque octet b du flag :

1. On génère un grand premier r (512 bits).
2. On calcule $n = 2^{r * b}$.
3. On choisit x aléatoire dans $[-n, n]$ impair +1, puis $y = x - n$.
4. On calcule $t = x^n - y^n$.
5. La sortie est $(r, f(t))$ où $f(t)$ compte le nombre de divisions par 2 (exposant de 2 dans la décomposition de t).

🧠 Analyse de la fonction $f(t)$

- x et $y = x - n$.
- Puisque $n = 2^{\lfloor r * b \rfloor}$, on peut estimer $t = (x - y)^n = n * (...)$.
- Cela indique que la valuation en 2, $v_2(t)$, est au moins $r * b$ plus celle de l'expression entre parenthèses.
- En fait, pour x impair et n puissance de deux, on montre qu'**exactement** $f(t) = r * b$ (la partie entre parenthèses reste impaire).

📋 Reconstruction du flag

Étant donné chaque (r, f) , on fait simplement :

- $b = f // r$
- r et $f = f(t) = r * b$ (précisément).
- Comme r est premier et large, on s'attend à une division entière sans reste.
- b est l'octet original du flag.

☑ Exemple de code Python pour récupérer le flag

```
encrypted = [...] # liste de tuples (r, f) depuis out.txt  
flag_bytes = bytes([f // r for (r, f) in encrypted])  
print(flag_bytes)
```

Cela affiche le flag, typiquement sous la forme SHLK{...}.

❖ Conclusion

- L'algorithme fuit l'octet chiffré par la valuation en 2 (v_2) de $t = x^n - y^n$.
- Comme n est exactement $2^{\lfloor r * b \rfloor}$, cette information suffit à reconstruire b .

- Avec quelques lignes, on récupère chaque octet — le chiffrement est **entièrement cassé**.

Il y a ce qu'il faut pour extraire chaque octet du flag avec les données d'entrée du fichier (out.txt) :

Pour chaque couple (r, f) , la relation

$$f = f(t) = \nu_2(x^n - y^n) = r \cdot b$$

est exacte, grâce au **lemme du relèvement des exposants** (LTE) appliqué à $p = 2$:

$$\nu_2(x^n - y^n) = \nu_2(x - y) + \nu_2(n),$$

- et ici $n = 2^{rb}$ et $x - y = n$, ainsi, on montre que les autres facteurs n'ajoutent rien à la valuation.
Donc simplement :

$$b = \frac{f}{r}.$$

Étapes pour retrouver le flag

1. Lire out.txt pour récupérer la liste des couples (r, f) .
2. Pour chaque (r, f) , calculer l'octet $b = f // r$.
3. Vérifier que $0 \leq b \leq 255$ (sinon quelque chose cloche).
4. Convertir chaque b en caractère ASCII.
5. Assembler les caractères pour obtenir le flag (format SHLK{...}).

Exemple de code Python

```
# on suppose que 'encrypted' contient la liste lue depuis out.txt
```

```
encrypted = [(r, f), (r2, f2), ...] # nos données importées du fichier
```

Pourquoi ça casse en un instant

- Le chiffrement est trop informatif : la valuation 2-adique révèle directement r^*b .
- r est un grand nombre premier, donc la division f/r donne directement b .
- On n'a pas besoin d'inverser des exposants, de factoriser, ni d'essayer d'énormes nombres : juste une division entière et un cast en ASCII.

Conclusion

Il n'y a aucun besoin d'attaques complexes ou de temps astronomique (**chiffrement trivial réversible**).

Voici le script complet en Python pour extraire le flag à partir de la donnée out.txt, en utilisant directement les paires (r, f) :

```
# lecture de la liste depuis out.txt

with open('out.txt') as fd:
    encrypted = eval(fd.read().strip())
    flag_bytes = []
for i, (r, f) in enumerate(encrypted):
    if f % r != 0:
        raise ValueError(f"Tuple {i} : l'incongruence : f={f} n'est pas divisible par r={r}")
    b = f // r
    if not (0 <= b < 256):
        raise ValueError(f"Tuple {i} : octet invalide b={b} hors de [0;255]")
    flag_bytes.append(b)
flag = bytes(flag_bytes)
print(flag) # typiquement : b'SHLK{...}'
```

Pourquoi ça doit fonctionner (ou est « censé » fonctionner)

- https://en.wikipedia.org/wiki/Lifting-the-exponent_lemma

Soient

$$n = 2^{r \cdot b} \quad \text{et} \quad t = x^n - y^n, \quad \text{où} \quad x - y = n.$$

En appliquant le **lemme du relèvement des exposants** (LTE) pour $p = 2$ et sachant que $4 \mid (x - y)$, on obtient :

$$\nu_2(x^n - y^n) = \nu_2(x - y) + \nu_2(n).$$

Ici :

$$\nu_2(x - y) = \nu_2(n) = r \cdot b.$$

Donc,

$$\nu_2(t) = r \cdot b,$$

ce qui signifie que

$$f = \nu_2(t) = r \cdot b$$

et alors

$$b = \frac{f}{r}.$$

Exécution

Ce script peut se lancer **immédiatement**, sans attendre ni utiliser de mathématiques avancées : puisque **ce chiffrement est totalement réversible** en un temps négligeable. Résultat attendu : b'SHLK{...}'.

Étape 1 : préparer le script

1. Copier le script Python dans un fichier, par exemple : extract_flag.py.

-
2. **S'assurer** que le fichier out.txt est dans le **même dossier** que le script.

🔧 Étape 2 : ouvrir un terminal

- **Windows** : lancer PowerShell ou l'invite de commandes (cmd).
- **Linux / macOS** : ouvrir le Terminal (ex. GNOME Terminal, iTerm).

📁 Étape 3 : se déplacer en dossier

Utiliser la commande cd pour te rendre dans le dossier contenant ton script et le fichier out.txt.

▶ Étape 4 : exécuter le script

Selon la version de Python :

- Si c'est Python 3 installé comme python, taper :

```
python extract_flag.py
```

- Si c'est python3, faire :

```
python3 extract_flag.py
```

⌚ Le script lit out.txt, calcule les octets et affiche b'SHLK{...}'

🛠 Optionnel : rendre le script exécutable (Linux/macOS)

Si on veut lancer directement ./extract_flag.py :

1. Ajouter la première ligne au script : python `#!/usr/bin/env python3`
2. Rendre exécutable : chmod +x extract_flag.py
3. Puis lancer : ./extract_flag.py

☑ En explication

Le format d'extraction du flag : pour confirmer que **b = f / r** fonctionne, cela peut s'appuyer sur le **lemme du relèvement des exposants (LTE)**, qui établit précisément que :

En une ligne:

$$\nu_2(x^n - y^n) = \nu_2(x - y) + \nu_2(n) \quad \text{quand} \quad 4 \mid (x - y) \quad \text{et} \quad x, y \text{ impairs.}$$

Dans notre cas :

1. **Définition** : $x - y = n = 2^{rb}$, calcul de $\nu_2(x - y)$.
2. **Validation** : pareil pour n .
3. **LTE** : donne un total de $rb + rb$.
4. **Factorisation** : on décompose $x^n - y^n$.
5. **Valeur de chaque facteur** : chaque $x^{2^i} + y^{2^i}$ contribue 1 à la valuation.
6. **Synthèse** : on retire rb pour obtenir le résultat final rb .
7. **Conclusion** : $f(t) = rb$.

Ainsi, $\nu_2(x^n - y^n) = rb$.

En math :

1. Définitions :
- $n = 2^k b$, avec b impair, donc par définition :
 $k = \nu_2(n)$.
2. Conditions sur x et y :
 x, y impairs et $4 \mid (x - y)$ impliquent que
 $x - y = 2^r b'$ avec b' impair, donc
 $\nu_2(x - y) = r$.
3. Application du lemme LTE (pour $p = 2$ et $4 \mid (x - y)$) :

$$\nu_2(x^n - y^n) = \nu_2(x^{2^k b} - y^{2^k b}) = \nu_2(x^{2^k} - y^{2^k}).$$

4. Factorisation :

$$x^{2^k} - y^{2^k} = (x - y) \times \prod_{i=0}^{k-1} (x^{2^i} + y^{2^i}).$$

5. Étude des facteurs :

Pour tout $0 \leq i \leq k - 1$, puisque x et y sont impairs :

$$x^{2^i} \equiv y^{2^i} \equiv 1 \pmod{4} \implies x^{2^i} + y^{2^i} \equiv 2 \pmod{4} \implies \nu_2(x^{2^i} + y^{2^i}) = 1.$$

$$b = \left\lfloor \frac{f}{r} \right\rfloor.$$

Donc la formule est a priori correcte (Cf. https://fr.wikipedia.org/wiki/Lemme_LTE).

⌚ Démarrage depuis zéro (étapes claires)

```
encrypted = [
    (1009579732637943509200..., 1675902356178986225...),
    (1193815495880735934..., 1719094314068259746...),
    # ... tous les couples de out.txt ... en remplaçant la lecture possible du fichier par la variable littérale
    # contenant la ligne complète (pour éviter les confusions eval ou formats)
```

```

]
flag_bytes = []
for i, (r, f) in enumerate(encrypted):
    if f % r != 0:
        raise ValueError(f"Tuple {i}: f non multiple de r")
    b = f // r
    if not (0 <= b < 256):
        raise ValueError(f"Tuple {i}: octet invalide b={b}")
    flag_bytes.append(b)
print(bytes(flag_bytes))

```

Depuis un terminal : python3 extract_flag.py : devrait afficher un résultat du type flag b'SHLK{...}'

Points à surveiller

- **Reproduire tous les couples (r, f) exactement**, sans supprimer de valeurs ni reformater.
- Utiliser Python 3 et pas de modifications d'encodage.
- S'il faut parser out.txt, s'assurer que le contenu est bien entre crochets [...] sur une seule ligne.

En résumé

1. **Validation** : LTE assure que $f = r \cdot b$.

$$b = \left\lfloor \frac{f}{r} \right\rfloor.$$
2. **Extraction** : donne chaque char.
3. **Exécution** : un script Python est utile à afficher le flag.

Concernant le **lemme du relèvement des exposants (LTE)** :

Un détail du LTE (pour $p = 2$) : si n est pair et $4 \mid (x - y)$, alors

$$\nu_2(x^n - y^n) = \nu_2(x - y) + \nu_2(x + y) + \nu_2(n) - 1.$$

Appliquons-la dans ton cas :

- On a $x - y = n = 2^{rb}$ donc $\nu_2(x - y) = rb$.
- x et y sont impairs, donc $x + y$ est pair mais **non divisible par 4**, donc $\nu_2(x + y) = 1$.
- De même $n = 2^{rb}$, d'où $\nu_2(n) = rb$.

On remplace dans la formule :

$$f = \nu_2(x^n - y^n) = rb + 1 + rb - 1 = 2rb.$$

Donc dans ce cas précis, la formule donne $\nu_2(t) = 2rb$.

Cela explique les valeurs très élevées qui sont obtenues.

Une fois calculé, b est vraisemblablement un entier ayant une signification – par exemple :

$$b = \frac{f}{r}.$$

?

- un octet (valeur entre 0 et 255),
- un code ASCII,
- un index dans un tableau.

Avec des valeurs d'indices successives pour divers (r, b) , on peut :

1. Extraire des octets (via $b = \left(\frac{f}{r}\right) \bmod 256$ ou $b \equiv \frac{f}{r} \pmod{256}$).
2. Convertir chaque octet en caractère (ASCII, hexa, base64, etc.).

❖ Application :

Si on connaît r , f , alors l'octet original b est donné par $b = \left\lfloor \frac{f}{2r} \right\rfloor$ et non $b = \left\lfloor \frac{f}{r} \right\rfloor$

✓ Résolution

```
Lifting weights > ⚡ extract_flag.py > ...
# lecture depuis out.txt :
with open('out.txt') as fd:
    encrypted = eval(fd.read().strip())

flag_bytes = []
for i, (r, f) in enumerate(encrypted):
    if f % (2 * r) != 0:
        raise ValueError(f"Tuple {i} : f={f} non multiple de 2*r={2*r}")
    b = f // (2 * r)
    if not (0 <= b < 256):
        raise ValueError(f"Tuple {i} : octet invalide b={b}")
    flag_bytes.append(b)

flag = bytes(flag_bytes)
print(flag)
```

❖ Pourquoi l'issue devient bonne après correction (parité) :

Avec : $f = 2rb$, cette formule corrige le facteur d'erreur possible.

⌚ Relancer simplement du script et cela donne le flag attendu.

Désormais, il y a un flag lisible, avec le format $b'SHLK{\dots}'$.

==== Flag complet ====

b'SHLK{W0rk_1t_h4rd3r_m4k3_1t_b3tt3r_D0_1t_f4st3r_m4k3s_us_str0ng3r}'