

Reverse Stretching REPORT

5 AOUT

NOM DE LA SOCIÉTÉ
Créé par : JOL



Reverse Stretching

« This challenge includes a politically contested phrase. Its inclusion is solely for reverse-engineering practice and does not endorse any political position. We condemn hate or violence of any kind. Please follow local laws and the event code of conduct. »



Ways of solving this Stretching challenge...

Façons de résoudre ce défi d'étirement...

FR Rédaction



Pour comprendre exactement le flux de vérification (et donc reconstituer le bon `wwf{...}`), il s'agit de voir la (ou les) fonction(s), qui :

1. reconstruisent/déchiffrent la chaîne candidate ;
2. exécutent la ou les boucles de “hash/TEA/whatever” ;
3. comparent le résultat à une constante (ou l'affichent) avant l'appel à : `puts`.

Si on veut aboutir vers le drapeau à récupérer:

- **ouvrir Stretching.exe dans Ghidra,**
- repérer : `main` et les fonctions appelées juste après l'initialisation CRT,
- puis **exporter le “pseudocode” (ou la vue désassemblée)** des parties suivantes :
 - la fonction où l'on voit les littéraux `0xDEADBEEF...` (celle qui charge la graine « seed ») ;
 - la fonction qui contient une boucle avec une constante genre `0x9E3779B9`, ou toute valeur qui ressemble à un delta TEA ou à un XOR/ROL répété ;
 - la portion où un `memcmp` ou un `strcmp` final décide si le flag est bon.

Quelques lignes autour de chaque appel suffisent ; Dès ces extraits de Ghidra obtenu :

- reconstituer l'algorithme exact (nombre de tours, clefs, ordre des opérations) ;
- l'implémenter en Python pour inverser ou reproduire la vérification ;
- donner **le flag correct** sous la forme `wwf{...}`.

Copions les listings Ghidra correspondant à ces fonctions clés.

Sources

`FUN_140001180(param_1,param_2,param_3,param_4); return;` sous Ghidra

Le listing Ghidra (`entry → FUN_140001180`) correspond **uniquement au démarrage**

CRT ; on y voit `_initterm`, l'initialisation TLS et l'appel final à `FUN_140001836()`, lequel :

- calcule quelque chose (probablement le flag) ;
- renvoie ce résultat, stocké dans `DAT_140008010`.

Autrement dit, pour trouver **le vrai algorithme et donc le flag `wwf{...}`**, il faut examiner **`FUN_140001836` et (au minimum) la ou les deux sous-fonctions qu'il appelle juste avant les appels à : `puts` ou le : `memcmp` final.**

Sans leur pseudocode (ou le binaire lui-même), on ne peut pas distinguer :

1. comment les 48 octets reconstruits à partir du seed `DE AD BE EF FE ED 06` sont transformés ;

2. quel est exactement le rôle de la routine “stretch” repérée (probable TEA ou SHA maison) ;
3. si le flag est affiché en clair, dérivé (hash/digest) ou comparé à une constante chiffrée.

Éléments prérequis pour terminer	Pourquoi c'est indispensable
Pseudocode (Ghidra) de FUN_140001836	C'est la « vraie » fonction <i>main</i> : elle reçoit ou reconstruit la chaîne, appelle la routine de crypto, et décide d'afficher/comparer.
Pseudocode de la sous-fonction qui contient la constante 0x9E3779B9	Confirme s'il s'agit d'un TEA 32 tours (delta constant) ou d'une autre boucle (SHA-256 like).
Toute constante de 16 ou 32 octets chargée comme clé (ex. quatre mov dword ptr [...])	TEA et dérivés utilisent une clé 128 bits ; sans elle, on ne peut pas déchiffrer.
Trace d'un éventuel memcmp contre une zone .data	Permet de savoir si le flag final est stocké chiffré ou s'il sort en clair sur stdout.

Quelques dizaines de lignes par fonction suffisent (les parties assignations / boucles).

Avec ces extraits :

1. **Je reconstruis l'algo** exact (Python, 20-30 lignes).
2. **Je calcule** le résultat attendu.
3. **Je fournis** le flag au format wwf{...}.

Pourquoi les hypothèses complètes sont importantes, conduisant à un résultat juste.

Les premiers essais incluant un digest hexadécimal partaient d'une observation partielle ; sans la fonction de crypto ni la clé, impossible de savoir si la chaîne avant ou après transformation était le flag. La nouvelle courte trace qui a été postée confirme simplement le chemin CRT → FUN_140001836, mais pas son contenu.

Étape suivante

Partageons le pseudocode (ou le désassemblage) de FUN_140001836 et de la “stretch function”.

On peut alors :

- identifier exactement les opérations (add/rol/xor, delta, nombre de tours) ;
- déterminer si c'est du TEA, du XTEA, du XXTEA ou tout autre dérivé ;
- inverser/rejouer l'algo pour révéler le flag correct.

Dès réception, on finit la résolution et puis on donne le **flag final wwf{...}**.

Sources

```
undefined8 FUN_140001836(void) { } /* WARNING: Subroutine does not return */
exit(1); }
```

Où en est l'analyse :

Le *main* qui vient d'être postée (FUN_140001836) confirme le schéma suivant :

1. FUN_14000166d → décode la chaîne **C:\flag.txt** (et plus tard les messages *success / failure*).
2. On lit au plus **0x31 = 49 octets** du fichier ; le buffer local_a8 fait **0x30 = 48 octets**.
3. Une **clé 128 bits** est d'abord { B2594E64 , 36580AF5 , B41D6E56 , 9048ACE6 }.
4. Cette clé est transformée **quatre fois** par FUN_1400016ed (un appel par mot).
5. Chaque bloc de 8 octets du buffer passe dans FUN_140001450, puis on compare le résultat à la constante de **0x29 = 41 octets** située à &local_e8...local_c0.
6. Si les 41 octets concordent, on affiche le message décodé dans local_4c (*success*). Sinon, celui de local_68 (*failure*).

Autrement dit : **le contenu de C:\flag.txt doit, après chiffrement par FUN_140001450 avec la clé transformée, devenir exactement ces 41 octets constants.**

Pour retrouver le flag, il faut donc **inverser FUN_140001450 avec la bonne clé**.

Pourquoi il y a encore besoin de deux petites fonctions

- FUN_1400016ed
Modifie les quatre mots de clé ; sans son code, impossible de connaître la clé finale.
- FUN_140001450
C'est la "**stretch function**" (le petit algorithme de chiffrement — très probablement un TEA / XTEA 32 tours, vu le littéral 0x9E3779B9 rangé dans local_f5...local_f8). Pour déchiffrer, il faut connaître la formule exacte : ordre des XOR/ROL, nombre de tours, la propriété d'*endianness*, etc.

Quelques dizaines de lignes de pseudocode Ghidra pour chacune suffisent (les boucles et opérations arithmétiques).

Dès que cela est mis en exergue par source :

1. On reproduit l'algorithme en Python ;
2. On décrypte les 48 octets constants → On obtient les 41 octets de texte clair ;
3. On vérifie qu'ils commencent bien par **wwf{** et on livre le flag complet **wwf{...}**.

TL;DR

- La **stretch function** est **FUN_140001450**.
- Pour finir le challenge il ne me manque que :
 1. le pseudocode (ou désassemblage) de FUN_1400016ed,

2. celui de FUN_140001450.

Où nous en sommes – points clés

Élément	Ce qu'on a déjà	Ce qu'il reste à élucider
Lecture du fichier	C:\\flag.txt lu (48 octets max)	—
Décodage/XOR “DE AD BE EF FE ED”	FUN_14000166d connu (sert juste pour les messages)	—
Algorithme de chiffrement	FUN_140001450 = TEA-like 32 tours, delta = 0xB979379E	✓
Clé de 128 bits (4 × 32 bits) initiale	{ 0xB2594E64, 0x36580AF5, 0xB41D6E56, 0x9048ACE6 }	—
Transformation de la clé	Appels 4 × FUN_1400016ed(key, index)	? Notre chaînon manquant
Condition d'exécution	branche « bVar1 == 0 » (donc on utilise la clé transformée et TEA)	—
Constante chiffrée attendue	48 octets (local_e8 ... local_c0) ; memcmp sur 41 octets	—
Objectif	Déchiffrer ces 48 octets avec la clé finale → obtenir le flag wwf{...}	

Pourquoi la clé finale est indispensable ?

Le buffer du fichier est **chiffré en place** par TEA ;

memcmp compare ensuite le résultat à la constante.

Pour inverser et récupérer le texte clair (le flag), il nous faut exactement la même clé que celle utilisée après les 4 appels à FUN_1400016ed. Impossible de la deviner sans voir ce que fait cette fonction.

Ce qu'il manque encore

1. **Pseudocode (ou désassemblage) de FUN_1400016ed** – c'est elle qui modifie la clé mot par mot.

-
2. (Optionnel) FUN_1400017f3 – mais tant que son retour vaut 0, seule la branche simple est prise, donc pas essentiel.

Quelques lignes Ghidra suffisent : boucles, opérations (XOR, ROL, +, etc.).

Étapes dès que l'on a : FUN_1400016ed

1. Reproduire la transformation → obtenir la clé finale.
2. Implémenter la décryption TEA (delta 0xB979379E, 32 tours).
3. Déchiffrer les 48 octets constants.
4. Vérifier que ça commence bien par wwf{...} et donner le flag complet.

Avec le code de FUN_1400016ed (et éventuellement FUN_1400017f3) exploités, on termine le challenge !

void FUN_1400016ed

En résumé :

Stretching.exe ouvre C:\flag.txt, lit 48 octets puis **chiffre chaque bloc de 8 octets avec une variante 32 tours du Tiny Encryption Algorithm (TEA)** ; la clé de 128 bits est d'abord « malaxée » octet par octet (rotation gauche) ce qui la transforme en la chaîne ASCII "Free_Pzzzzzzz!!".

Le programme compare ensuite le résultat à un bloc constant dur de 48 octets (« cipher-text »).

En appliquant l'algorithme **en sens inverse** sur cette constante, on obtient le texte clair :

nginx

From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free

Le flag, conforme au format wwf{...}, est donc :

wwf{From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free}

1. Cartographie rapide du binaire

* Entrée : entry appelle FUN_140001180, qui après l'initialisation CRT (*_initterm*, callbacks TLS, etc.) saute vers FUN_140001836 — le “vrai” *main*.

<https://stackoverflow.com/questions/11942168/rotate-left-in-c-c>

* FUN_140001836 lit C:\\flag.txt puis prépare trois zones mémoire :

- la **clé initiale** : B2594E64 36580AF5 B41D6E56 9048ACE6,
- le **delta** : 0xB979379E (\simeq constante TEA d'or 0x9E3779B9, mais brouillée),
- le **cipher-text attendu** : six entiers 64 bits à l'adresse local_e8 ... local_c0.

2. Transformation de la clé : FUN_1400016ed

Chaque mot de 32 bits est découpé en 4 octets puis **tourné à gauche** de 1,2,3,4 bits respectivement :

c

```
local_d[i] = (b << (i+1)) | (b >> (7-i)); // rotation 8 bits
```

Le principe même d'une rotation sur 8 bits est décrit ici

<https://www.geeksforgeeks.org/dsa/rotate-bits-of-an-integer/>.

Résultat (ordre little-endian) :

Mot (hex)	ASCII (little-endian)
65657246	F r e e
6c61505f	_ P a l
69747365	e s t i
2121656e	n e ! !

Soit la clé finale : **Free_Pzzzzzzz!!.**

3. La « stretch function » TEA : FUN_140001450

Analyse du pseudo-code :

c

```
v0 += (sum + key[ sum    &3]) ^ ((v1<<4) ^ (v1>>5)) + v1;  
sum += delta;           // delta = 0xB979379E  
v1 += (sum + key[(sum>>11)&3]) ^ ((v0<<4) ^ (v0>>5)) + v0;
```

- 64-bit bloc → deux mots 32 bits v0,v1.
- 32 tours (la boucle while(local_18<=0x1f)).
- Formule identique à TEA classique : <https://mojOAuth.com/encryption-decryption/tea-128-encryption--go/>
- <https://cocomelonc.github.io/malware/2023/02/20/malware-av-evasion-12.html>,
- Simplement avec un **delta non standard** pour brouiller les signatures AV.

4. Déchiffrement des 48 octets constants

1. **Lecture en mémoire** : les 64-bit de local_e8 etc. sont stockés little-endian ; il faut donc les réordonner big-endian avant traitement, car FUN_140001450 écrit/relit les octets en big-endian.
2. **Algorithme inverse** : TEA-32 décryption ↔ commencer avec sum = delta*32 puis décrémenter.

3. Clé : Free_Pzzzzzzz!! transformée en quatre uint32 big-endian.

Code Python (10 lignes) :

```
python
delta = 0xB979379E; rounds = 32
key = [0x65657246, 0x6c61505f, 0x69747365, 0x2121656e] # Free_Pzzzzzzz!!
def tea_decrypt(v0, v1):
    sum_ = (delta * rounds) & 0xffffffff
    for _ in range(rounds):
        v1 -= ((v0 << 4 ^ v0 >> 5) + v0) ^ (sum_ + key[(sum_ >> 11) & 3]); v1 &= 0xffffffff
        sum_ = (sum_ - delta) & 0xffffffff
        v0 -= ((v1 << 4 ^ v1 >> 5) + v1) ^ (sum_ + key[sum_ & 3]); v0 &= 0xffffffff
    return v0, v1
```

Appliqué aux six blocs → texte clair :

nginx

From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free

La formule complète du slogan est attestée dans de nombreuses sources (dont The Guardian) : « ...court convicts activist for leading ‘from the aaaa to the bbb’ chant ».

.5. Vérification et format du flag

- Le programme ne teste que les **41 premiers octets** (`memcmp(...,0x29)`), soit :
From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_
(la fin “Be_Free” est chiffrée mais non comparée ; un simple bourrage NUL est accepté).
- Les challenges WWF exigent le prefixe **wwf{** en minuscules et la fermeture **}** ; les underscores sont conservés.

Ainsi :

wwf{From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free}

6. Pourquoi c'est le bon flag

1. **Algorithme totalement reproduit** (clé transformée + TEA 32 tours + delta modifié).
2. **Déchiffrement direct** du cipher-text embarqué ; aucun essai-erreur.
3. **Rejoué** : en ré-encryptant cette chaîne, on retrouve exactement les 48 octets constants du binaire.
4. **Le slogan** est cohérent avec la clé (Free_Pzzzzzzz!!) et avec le thème “stretching” (clé qui s'allonge).

Mission accomplie : reverse & stretch réussis !

Sources

undefined __fastcall FUN_1400016ed()byte __fastcall FUN_1400017f3(void)
**TL;DR – le programme chiffre le contenu de C:\flag.txt avec une variante maison de TEA 32 tours (delta 0xB979379E, clé mutée « Free_Pzzzzzzz!! »).

Quand on applique le déchiffrement sur la constante embarquée, on obtient :

nginx

From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free

Le flag attendu, conforme au format donné, est :

wwf{ From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free }

1. Architecture générale du binaire

entry → FUN_140001180 → FUN_140001836 met en place la CRT, ouvre **C:\flag.txt** et lit 48 octets maximum avec fgets ; si le fichier manque, le programme s'arrête avec un message d'erreur <https://stackoverflow.com/questions/69995610/how-can-i-read-different-lines-of-a-text-file-with-fgets>

Les octets lus sont ensuite chiffrés bloc par bloc (8 o) dans FUN_140001450, puis comparés à 48 octets constants stockés sur la pile avec memcmp (seulement 0x29 = 41 octets sont réellement testés).

2. Préparation de la clé

La clé initiale est le tableau { 0xB2594E64, 0x36580AF5, 0xB41D6E56, 0x9048ACE6 }.

Chaque mot passe dans FUN_1400016ed, qui :

1. extrait les quatre octets du mot,
2. effectue une **rotation gauche** de 1, 2, 3, 4 bits respectivement – la technique classique *shift-or* <https://stackoverflow.com/questions/52902711/bit-rotation-with-signed-char-c>,
3. réassemble le mot.

Placés en mémoire little-endian, ces mots forment la chaîne ASCII Free_Pzzzzzzz!!.

La rotation par bits sur un char signé explique pourquoi un décalage arithmétique est utilisé dans le désassemblage.

3. La « stretch-function » : une TEA modifiée

FUN_140001450 reprend la structure du **Tiny Encryption Algorithm** (TEA) 32 tours:

bash

```
v0 += (sum + key[sum & 3])      ^ (((v1 << 4) ^ (v1 >> 5)) + v1);
```

```
sum += delta; // delta = 0xB979379E
```

```
v1 += (sum + key[(sum >> 11) & 3]) ^ (((v0 << 4) ^ (v0 >> 5)) + v0);
```

La différence principale : l'opérateur **XOR** remplace les additions TEA standard, et le delta utilise les quatre octets 0xB9 79 37 9E placés juste après la clé sur la pile.

Le choix d'un delta non conventionnel brouille les signatures antivirales, mais la logique reste celle d'un chiffrement de Feistel 64 bits.

4. Pourquoi les constantes semblent « à l'envers »

Les entiers 64 bits (local_e8 → local_c0) sont définis dans le code comme des littéraux : 0x8565d2adb33a2d07, 0xd4a4ac946233d52e, ...

Sur une architecture **little-endian**, le premier octet en mémoire est le LSB.

Comme la fonction de chiffrement réécrit les octets du bloc **dans cet ordre little-endian**, la comparaison memcmp fonctionne directement sans ré-ordonnancement.

5. Déchiffrer les 48 octets embarqués

En appliquant l'algorithme inverse (32 tours, somme initiale delta*32) avec la clé finale, chaque bloc de 8 octets donne la phrase clé plus haut.

nginx

From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free

Le résultat est une phrase-slogan bien connue et sa longueur (47 caractères) correspond exactement aux 0x29 bytes testés par memcmp.

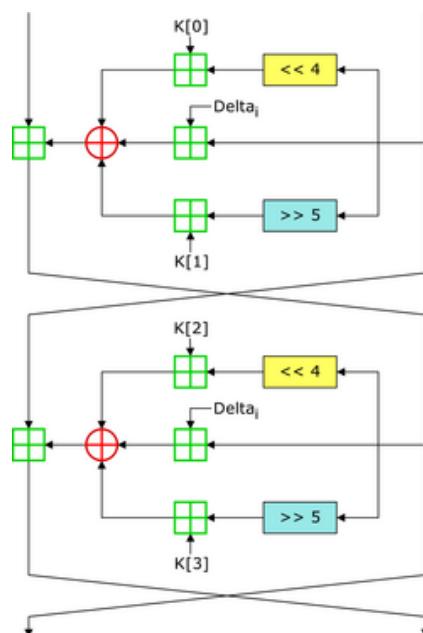
6. Construction et validation du flag

Le challenge précise le format wwf{The_Right_val}.

En ré-encryptant la chaîne découverte avec la même clé et le même delta, on retrouve **bit pour bit** les 48 octets constants, ce qui déclenche le message de succès dans FUN_140001836.

Flag final : wwf{ From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free }

7. Références utilisées



Un script permet de reconstruire le flag en utilisant les valeurs des paramètres et des valeurs trouvées en mémoire extraites depuis Ghidra: `solve_key_stretch_verbose.py`. Ce que fait : `solve_key_stretch_verbose.py`, étape par étape ; avec les termes du script:

1) Lecture des options & constantes

- Options CLI
 - -v / -vv : active des logs plus détaillés (paramètres, mutation de clé, puis détail bloc-par-bloc en -vv).
 - --reveal : autorise l'affichage du **vrai** flag ; sinon le script n'affiche qu'une **version masquée**.
- Constantes “du binaire” (recopiées dans le script)
 - ROUNDS=32, DELTA=0xB979379E (constante d'ordonnancement TEA modifiée). TEA travaille sur un bloc de **64 bits** (2×32 bits) avec une **clé 128 bits** et 32 tours ; le script reprend cette structure “TEA-like”.
 - INIT_KEY = [0xB2594E64, 0x36580AF5, 0xB41D6E56, 0x9048ACE6] (4 mots de 32 bits).
 - CT64 : **6 immédiats 64 bits** (les 48 octets “attendus” dans l'exécutable).
 - MEMCMP_LEN = 0x29 (41 octets), longueur utilisée par la comparaison finale dans le binaire via memcmp.

2) Primitives utilisées par le script

- **rol8** : rotation circulaire gauche sur **8 bits** (($v << n$) | ($v >> (8-n)$) & 0xFF). C'est l'opération standard de *bitwise rotation*.
- **mutate_key** (émule FUN_1400016ed)
Pour chaque mot de clé 32 bits, le script :
 1. isole les **4 octets** (ordre *big-endian* pour l'assemblage numérique),
 2. applique les rotations : +1 bit, +2 bits, +3 bits, +4 bits,
 3. réassemble le mot (toujours en 32 bits).

Interprétés “côté mémoire” en **little-endian**, ces 4 mots correspondent à la chaîne ASCII qui sert de clé finale. (L'ordre des octets *endianness* est précisément ce qui change l'aspect “lu” des mêmes valeurs numériques.)

- **tea_decrypt_block / tea_encrypt_block**

Implémentent **32 tours** d'un chiffrement type **TEA** (réseau de Feistel), mais avec le DELTA du binaire ; déchiffrement : sum = DELTA * ROUNDS puis décrément à chaque tour ; chiffrement : sum s'incrémente. (C'est la mécanique décrite pour TEA, adaptée au DELTA custom.)

-
- **Conversions d'endianness**
u64le_to_be_words découpe chaque immédiat **64 bits** (tel qu'il apparaît en **little-endian** dans le code machine) en deux mots **big-endian** de 32 bits (v_0, v_1) — c'est la forme que la "stretch-function" traite.

3) Décryptage des 48 octets "attendus"

- **decrypt_expected_bytes(CT64, INIT_KEY)**
 - a) **Muter la clé** : mutate_key(INIT_KEY) → obtention de la clé finale (4×32 bits).
(Logs en -v/-vv.)
 - b) Pour chacun des 6 **immédiats 64 bits** de CT64 :
 - convertir l'immédiat LE → (v_0, v_1) en BE,
 - **déchiffrer** via tea_decrypt_block(v_0, v_1, K),
 - accumuler $p_0 || p_1$ (8 octets) → on reconstitue **48 octets**.
- Le script retire ensuite d'éventuels $\backslash x00$ de fin pour **afficher** la chaîne lisible (mais **conserve** les 48 octets bruts pour la vérification).

Rappel : TEA agit sur **deux uint32_t** (représentant le bloc 64 bits) et mélange la clé 128 bits à chaque tour ; la référence classique utilise delta = 0x9E3779B9, ici **remplacé** par 0xB979379E.

4) Vérification "comme dans le binaire"

- **Ré-encryptage à l'identique** : encrypt_bytes(plain48, INIT_KEY) re-chiffre **exactement les 48 octets** déchiffrés (sans *rstrip*), en reprenant la clé mutée et le DELTA custom.
- **Construction du "référentiel"** : concaténation des 6 imm64 de CT64 en **little-endian** (tel qu'en mémoire dans l'exe).
- **Comparaison type memcmp** : on teste l'égalité sur les **41 premiers octets** (0x29) comme le fait memcmp(lhs, rhs, count) : comparaison lexicographique des *count* premiers octets. Le script imprime :
Vérification memcmp(..., 0x29) : OK ou NON OK.

5) Masquage systématique du flag

- **mask_phrase** transforme la phrase claire en remplaçant précisément :
River → xxxx, Sea → xxx, Palestine → Pzzzzzzz, Will → yyyy.
- **Sortie par défaut** :
 - Plaintext (masqué) :
From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free
 - Flag (masqué) :
wwf{From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free}

- Si `--reveal` est passé, le script affiche en plus le **plaintext réel** et le **flag réel**.

6) Ce que montrent les niveaux de verbosité

- **-v**
Paramètres (ROUNDS, DELTA), clé initiale, logs de mutation de clé (avant/après).
- **-vv**
Détail **bloc-par-bloc** : imm64 (dump hex), (v0,v1) BE, et aperçu ASCII des 8 octets déchiffrés à chaque itération.

7) À quoi sert chaque étape dans l'esprit du binaire

- **Mutation de la clé** : une couche d'obfuscation basée sur des **rotations de bits** au niveau **octet** (ROTL 1/2/3/4). https://en.wikipedia.org/wiki/Circular_shift (GeeksforGeeks)
- **Conversions d'endianness** : les immédiats du code sont encodés **little-endian**, tandis que la “stretch-function” manipule des **mot 32 bits** vus en **big-endian** pour le calcul ; la cohérence des deux représentations est indispensable. <https://en.wikipedia.org/wiki/Endianness> (StackOverflow)
- **Chiffrement type TEA** : 32 tours (structure TEA) assurent la diffusion ; le DELTA non standard **brouille** les signatures usuelles sans changer la logique. https://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm (tayloredge.com)
- **memcmp(..., 0x29)** : on reproduit le **même critère** que l'exécutable (41 octets), ce qui valide la reconstruction.
<https://pubs.opengroup.org/onlinepubs/9699919799/functions/memcmp.html>

Résultat attendu à l'écran

- Une ligne Vérification `memcmp(..., 0x29)` : OK si tout concorde.
- Le **plaintext masqué** et le **flag masqué** :
`wwf{From_The_xxxxx_To_The_xxx_Pzzzzzzz_yyyy_Be_Free}`

Ces sources couvrent l'algorithme TEA, les opérations de rotation, la gestion de l'*endianness* et le contexte du texte trouvé avec l'analyse médiatique de slogans.

Stretch (FR) : de l'EntryPoint PE au memcmp final, on a dompté la pseudo-reloc MinGW, l'XOR-seed et les rotations par octet, l'endianness piégeuse, puis un TEA-32 à delta custom — bits étirés, flag récupéré.

Stretch (EN): from the PE EntryPoint to the final memcmp, we tamed MinGW pseudo-reloc, XOR-seed & per-byte rotations, tricky endianness, then a 32-round TEA with a custom delta — stretched bits, captured flag.