CongoCTF 2026 – ANSSI & GTC
Guide de Participation et Préparation

Équipe d'organisation

Novembre 2026

Table des matières

1 Déroulement de la compétition **2**
1.1 Phase 1 : Préqualification en ligne . . . 2
1.2 Phase 2 : Compétition Finale en présentiel . . . 2

2 Outils à installer **2**
2.1 Outils pour le Reverse Engineering . . . 2
2.2 Outils pour la Cryptographie . . . 2

3 Règlement de la compétition **3**

A Annexe : Épreuves Types et Write-ups détaillés **4**
A.1 Cryptographie . . . 4
A.1.1 C1. ROT13 : "Rotten Message" (Facile) . . . 4
A.1.2 C2. XOR à clé répétée (Moyen) . . . 4
A.1.3 C3. RSA : Exposant public faible $e = 3$ (Moyen) . . . 4
A.1.4 C4. AES-ECB : Attaque Byte-at-a-Time (Difficile) . . . 5
A.1.5 C5. RSA : Attaque du module commun (Difficile) . . . 5
A.1.6 C6. ECDSA : Nonce Reuse (Extrêmement Difficile) . . . 5
A.1.7 C7. Attaque Padding Oracle AES-CBC (Difficile) . . . 6
A.1.8 C8. Attaque Hash Length Extension (Difficile) . . . 6
A.1.9 C9. Diffie-Hellman : Logarithme Discret Faible (Difficile) . . . 6
A.1.10 C10. Chiffre de Vigenère - Indice de coïncidence (Facile) . . . 6
A.2 Reverse Engineering . . . 7
A.2.1 R1. Strings basiques sur ELF (Facile) . . . 7
A.2.2 R2. Binaire Packé (UPX) + Anti-Debug (Moyen) . . . 7
A.2.3 R3. Décompilation Python .pyc (Moyen) . . . 7
A.2.4 R4. Patching de branchement via Hexedit (Moyen) . . . 7
A.2.5 R5. Analyse d'application .NET (Facile) . . . 8
A.2.6 R6. Analyse APK Android (Moyen) . . . 8
A.2.7 R7. Résolution de contraintes avec Z3 (Difficile) . . . 8
A.2.8 R8. Restauration de binaire Go strippé (Difficile) . . . 8
A.2.9 R9. Hooking dynamique avec Frida (Extrêmement Difficile) . . . 8
A.2.10 R10. Machine Virtuelle Imbriquée (Extrêmement Difficile) . . . 9

ctf.anssi.cg

1 Déroulement de la compétition

Le CongoCTF 2026 a pour objectif de détecter et valoriser les talents en cybersécurité en République du Congo. La compétition est axée à **100% sur la Cryptographie et le Reverse Engineering**. Elle se déroule en deux phases distinctes :

1.1 Phase 1 : Préqualification en ligne

- **Date et Format** : 14 novembre 2026. Questionnaire en ligne (QCM et mini-challenges).
- **Durée** : 120 minutes (chrono non-pausable, de 10h à 12h heure de Brazzaville).
- **Départage** : Un challenge "Bonus Expert" facultatif est prévu pour départager les ex-aequo.
- **Objectif** : Sélectionner les 40 meilleures équipes pour la phase finale.

1.2 Phase 2 : Compétition Finale en présentiel

- **Dates et Lieu** : 8 et 9 décembre 2026, en présentiel à Brazzaville (locaux de l'ANSSI).
- **Format** : CTF type *Jeopardy* sur 2 jours (environ 14h de compétition effective).
- **Infrastructure** : Les challenges sont hébergés sur une infrastructure isolée accessible via VPN (WireGuard). Le scoreboard en temps réel restera public jusqu'à 16h00 le deuxième jour (gel des scores).

2 Outils à installer

Pour être performant, vous devez préparer votre environnement de travail (une distribution Linux comme Kali, Parrot ou Ubuntu est recommandée).

2.1 Outils pour le Reverse Engineering

- **Désassembleurs et Décompilateurs** : Ghidra (recommandé, gratuit et puissant), IDA Free, radare2 (`r2` / `Cutter`), dnSpy (pour .NET), `jadx` (pour Android).
- **Débogueurs** : gdb (avec plugins GEF ou Pwndbg pour Linux), x64dbg (pour Windows).
- **Analyse dynamique et Instrumentation** : `ltrace`, `strace`, `Frida`, `unicorn-engine`.
- **Manipulation de binaires** : upx (unpacking), `hexedit`, `objdump`, `strings`.

2.2 Outils pour la Cryptographie

- **Langage** : Python 3 avec les librairies `pycryptodome`, `ecdsa`, `pwntools`, `z3-solver`.
- **Outils mathématiques** : SageMath (indispensable pour l'algèbre matricielle, les réseaux euclidiens et les courbes elliptiques).
- **Utilitaires divers** : CyberChef (couteau suisse du décodage), `RsaCtfTool` (attaques RSA automatisées), `hashpump` (attaques d'extension de signature).

ctf.anssi.cg

3 Règlement de la compétition

Pour garantir l'équité, le règlement suivant s'applique strictement :

1. **Composition des équipes** : De 2 à 4 membres pour les préqualifications, et de 1 à 4 joueurs physiquement présents pour la finale.
2. **Travail indépendant** : La collaboration entre équipes est strictement interdite. Aucune aide extérieure ne doit être sollicitée (forums, réseaux sociaux).
3. **Format des réponses (Flags)** : La preuve de résolution d'un challenge prend la forme d'une chaîne de caractères : `CONGO{...}`. Chaque équipe dispose de 10 tentatives maximum par challenge.
4. **First-blood** : La première équipe à résoudre un challenge final obtient un bonus de points de $+20\%$.
5. **Rapports (Write-ups) obligatoires** : Les 10 premières équipes doivent soumettre un rapport détaillé expliquant leur méthodologie de résolution dans les 72 heures. Tout flag non justifié ou justifié par un texte généré aléatoirement par une IA (sans compréhension humaine) sera annulé.
6. **Sanctions** : Toute attaque visant l'infrastructure du CTF (DDoS, scan agressif, tentative de compromission du scoreboard) entraînera une disqualification immédiate et définitive.

ctf.anssi.cg

A Annexe : Épreuves Types et Write-ups détaillés

Cette annexe contient 20 épreuves types (10 en Cryptographie, 10 en Reverse) avec des solutions très détaillées afin de vous préparer aux techniques attendues.

A.1 Cryptographie

A.1.1 C1. ROT13 : "Rotten Message" (Facile)

Énoncé : Fichier `cipher.txt` : PBATB{ebg13_vf_rnfl}. Indice : « Jules ».
Méthodologie : L'indice "Jules" et le format pointent vers le chiffrement de César. Le décalage classique en CTF est de 13 (ROT13). On utilise la commande `tr` sous Linux pour translater l'alphabet.

```
$ cat cipher.txt | tr 'a-zA-Z' 'n-za-mN-ZA-M'
```

Flag : `CONGO{rot13_is_easy}`

A.1.2 C2. XOR à clé répétée (Moyen)

Énoncé : `cipher.bin` chiffré par XOR avec une clé de 5 octets.
Méthodologie : En regroupant les octets par intervalle de 5, on isole l'effet de chaque octet de la clé. On applique une analyse de fréquence sur l'alphabet anglais (*e, t, a, o...*) pour chaque colonne.

```
data = open("cipher.bin", "rb").read()
key = bytearray()
for i in range(5):
    col = data[i::5]
    # Test des 256 octets possibles
    best_k = max(range(256), key=lambda k: sum(chr(b^k) in " etaoinsh" for b in
        col))
    key.append(best_k)
plain = bytes(b ^ key[i % 5] for i, b in enumerate(data))
```

Flag : `CONGO{x0r_1s_n0t_s3cur3}`

A.1.3 C3. RSA : Exposant public faible $e = 3$ (Moyen)

Énoncé : On fournit $(n, e = 3, c)$. Le message est court et n'a pas de padding (bourrage).
Méthodologie : Si le message m est suffisamment petit, $m^3 < n$. L'opération modulo n'a donc pas "bouclé". Il suffit de calculer la racine cubique entière du texte chiffré c.

```
import gmpy2
from Crypto.Util.number import long_to_bytes
# c = ...
m, exact = gmpy2.iroot(c, 3)
if exact:
    print(long_to_bytes(m))
```

Flag : `CONGO{r54_n0_p4dd1ng_1s_b4d}`

A.1.4 C4. AES-ECB : Attaque Byte-at-a-Time (Difficile)

Énoncé : Un oracle chiffre `AES-ECB(votre_input + flag)`.
Méthodologie : Le mode ECB chiffre des blocs de 16 octets indépendamment. Si on envoie 15 octets de padding (`A*15`), le 16ème octet du premier bloc sera la première lettre du flag. On enregistre ce bloc chiffré. Ensuite, on envoie `A*15 + char` pour tous les caractères possibles jusqu'à trouver un bloc chiffré correspondant.

```python
import string
flag = ""
for i in range(30): # longueur max du flag
    pad = b"A" * (15 - (i % 16))
    target = oracle(pad)[:len(pad) + len(flag) + 1]
    for c in string.printable:
        if oracle(pad + flag.encode() + c.encode())[:len(target)] == target:
            flag += c
            break
```

Flag : `CONGO{ecb_p3ngu1n_b10ck}`

A.1.5 C5. RSA : Attaque du module commun (Difficile)

Énoncé : Un même message a été chiffré deux fois pour la même personne avec deux exposants différents $(e_1 = 17, e_2 = 65537)$, mais avec le même module n. On a C_1 et C_2.
Méthodologie : Puisque $PGCD(e_1, e_2) = 1$, le théorème de Bachet-Bézout stipule qu'il existe u et v tels que $e_1 \cdot u + e_2 \cdot v = 1$. Le déchiffrement devient : $C_1^u \cdot C_2^v \equiv (m^{e_1})^u \cdot (m^{e_2})^v \equiv m^{e_1 \cdot u + e_2 \cdot v} \equiv m^1 \pmod{n}$.

```python
def egcd(a, b):
    if a == 0: return (b, 0, 1)
    g, y, x = egcd(b % a, a)
    return (g, x - (b // a) * y, y)

g, u, v = egcd(e1, e2)
# Attention si u ou v est negatif, calculer l'inverse modulaire du Chiffre
if u < 0:
    C1 = pow(C1, -1, n)
    u = -u
if v < 0:
    C2 = pow(C2, -1, n)
    v = -v
m = (pow(C1, u, n) * pow(C2, v, n)) % n
```

Flag : `CONGO{n3v3r_sh4r3_n}`

A.1.6 C6. ECDSA : Nonce Reuse (Extrêmement Difficile)

Énoncé : 2 signatures (r, s_1) et (r, s_2) partagent le même r (donc le même nonce k). Retrouvez la clé privée d.
Méthodologie : $s_1 \cdot k \equiv z_1 + r \cdot d \pmod{n}$ et $s_2 \cdot k \equiv z_2 + r \cdot d \pmod{n}$. En soustrayant : $k \equiv (z_1 - z_2) \cdot (s_1 - s_2)^{-1} \pmod{n}$. Puis on déduit la clé privée $d \equiv r^{-1} \cdot (s_1 \cdot k - z_1) \pmod{n}$. (On utilise `pow(x, -1, n)` pour l'inverse modulaire).
Flag : `CONGO{n0nc3_r3u53_1s_f4t4l}`

A.1.7 C7. Attaque Padding Oracle AES-CBC (Difficile)

Énoncé : Un serveur web déchiffre des cookies AES-CBC. Il renvoie 200 OK si le padding (PKCS7) est correct, et 500 s'il est invalide.
Méthodologie : En modifiant le vecteur d'initialisation (IV) byte par byte, on observe les erreurs 500. Lorsqu'on obtient 200, on sait que l'octet modifié a produit un déchiffrement finissant par `0x01`. Cela nous permet de déduire le texte clair (Plaintext XOR IV). L'outil `padbuster` automatise cette attaque complexe.

```
$ padbuster http://target/ check_cookie=... [BlockSize] -cookies
    check_cookie=...
```

Flag : `CONGO{p4dd1ng_0r4c13_c4b4c}`

A.1.8 C8. Attaque Hash Length Extension (Difficile)

Énoncé : Le serveur vérifie `SHA1(SECRET + message)`. Vous avez le hash de `"data"`. Ajoutez `"admin=1"` au message sans connaître le secret.
Méthodologie : La construction de Merkle-Damgård (utilisée par MD5, SHA1, SHA256) permet de reprendre l'état interne d'un hachage pour y ajouter des données. On utilise l'outil `hashpump`.

```
$ hashpump -s <hash_original> -d "data" -a "admin=1" -k <longueur_secrete>
```

Flag : `CONGO{l3ngth_3xt3ns10n}`

A.1.9 C9. Diffie-Hellman : Logarithme Discret Faible (Difficile)

Énoncé : Le serveur utilise Diffie-Hellman, mais avec un nombre premier p de 256 bits, ce qui est très faible. On intercepte $A = g^a \pmod p$. Trouver a.
Méthodologie : Un entier de 256 bits peut être "cassé" en quelques minutes avec SageMath via l'algorithme Pohlig-Hellman ou Baby-step Giant-step.

```
# Dans SageMath
p = 10000000... # 256 bits
F = GF(p)
g = F(2)
A = F(1234567...)
a = discrete_log(A, g)
print("Secret =", a)
```

Flag : `CONGO{d1scr3t3_l0g_p0wn3d}`

A.1.10 C10. Chiffre de Vigenère - Indice de coïncidence (Facile)

Énoncé : Un très long texte chiffré sans la clé. L'indice de coïncidence révèle une clé de longueur 6.
Méthodologie : Une fois la longueur de clé déterminée (via le test de Kasiski), on divise le texte en 6 colonnes. Chaque colonne devient un code de César simple qu'on résout par analyse de fréquence. Le site web dCode.fr propose des solveurs de Vigenère très performants pour ce type de challenge.
Flag : `CONGO{v1g3n3r3_k4s1sk1}`

A.2 Reverse Engineering

A.2.1 R1. Strings basiques sur ELF (Facile)

Énoncé : Binaire `check_pass`, demande un mot de passe.
Méthodologie : Les développeurs oublient souvent de "stripper" (nettoyer) les binaires. Les variables et mots de passe en clair sont visibles.

```
$ file check_pass
# ELF 64-bit... not stripped
$ strings check_pass | grep "CONGO{"
```

Flag : `CONGO{str1ngs_4r3_y0ur_fr13nd}`

A.2.2 R2. Binaire Packé (UPX) + Anti-Debug (Moyen)

Énoncé : Exécutable Linux lourd, non lisible, se ferme si on lance GDB.
Méthodologie : 1. Le fichier est compressé. On utilise l'outil officiel : `upx -d binaire`. 2. Le binaire utilise l'appel système `ptrace(PTRACE_TRACEME)` pour détecter GDB. 3. On crée un faux `ptrace` en C et on le précharge :

```
// hook.c
long ptrace(int request, int pid, void *addr, void *data) { return 0; }
```

```
$ gcc -shared -fPIC hook.c -o hook.so
$ LD_PRELOAD=./hook.so ./binaire
```

Flag : `CONGO{upx_unpack_pwn3d}`

A.2.3 R3. Décompilation Python .pyc (Moyen)

Énoncé : Fichier `logic.pyc` (bytecode Python compilé).
Méthodologie : Python compile les `.py` en `.pyc`. On utilise l'outil `uncompyle6` (ou `decompyle3` pour Python 3.7-3.8, ou le module `dis`) pour récupérer le code source d'origine.

```
$ uncompyle6 logic.pyc > logic.py
$ cat logic.py
# On analyse ensuite l'algorithme pour forger le flag attendu.
```

Flag : `CONGO{pyth0n_byt3c0d3_r3v}`

A.2.4 R4. Patching de branchement via Hexedit (Moyen)

Énoncé : Un binaire vérifie une licence. Si elle est fausse, il quitte. La fonction de génération de licence est introuvable (asymétrique).
Méthodologie : Au lieu de deviner la clé, on modifie le binaire. Dans un désassembleur (Ghidra/IDA), on identifie l'instruction conditionnelle (ex: `JZ` - Jump if Zero, opcode `74`) qui saute vers le message "Échec". Avec `hexedit`, on remplace cet opcode `74` par `75` (`JNZ` - Jump if Not Zero) ou par des `NOP` (opcode `90`). Le programme validera alors n'importe quelle entrée.
Flag : `CONGO{p4tch1ng_1nstruction5}`

A.2.5 R5. Analyse d'application .NET (Facile)

Énoncé : Fichier `EnterpriseManager.exe`. Outil C# Windows.
Méthodologie : Le framework .NET (C#/VB.NET) compile en code intermédiaire (MSIL). Cet IL se décompile presque parfaitement en code source d'origine. Il suffit d'ouvrir l'exécutable avec l'outil open-source dnSpy (ou `ILSpy`). La logique de validation (souvent une classe `LicenseChecker`) s'affiche en clair.
Flag : `CONGO{d0tn3t_r3fl3ct10n}`

A.2.6 R6. Analyse APK Android (Moyen)

Énoncé : Application Android `vault.apk`. Le flag est caché.
Méthodologie : Un APK est un fichier ZIP contenant du bytecode Dalvik (`classes.dex`). On utilise l'outil `jadx-gui` pour ouvrir directement l'APK. Jadx va décompiler le bytecode en code Java lisible. On inspecte le `AndroidManifest.xml` pour trouver la `MainActivity`, puis on navigue dans le code Java pour trouver un tableau d'octets chiffré en AES et la clé hardcodée.
Flag : `CONGO{4ndr01d_j4dx_r3v}`

A.2.7 R7. Résolution de contraintes avec Z3 (Difficile)

Énoncé : Binaire contenant 15 équations mathématiques non linéaires complexes vérifiant les caractères du mot de passe.
Méthodologie : Inverser manuellement ce genre de code est un cauchemar. On extrait les équations et on les confie à un "SMT Solver" mathématique : **Z3** (développé par Microsoft).

```python
from z3 import *
solver = Solver()
# flag de 8 caracteres
flag = [BitVec(f"c_{i}", 8) for i in range(8)]
solver.add(flag[0] + flag[1] * 2 == 150)
solver.add(flag[2] ^ flag[3] == 42)
# ... ajout des autres contraintes
if solver.check() == sat:
    m = solver.model()
    print("".join(chr(m[flag[i]].as_long()) for i in range(8)))
```

Flag : `CONGO{z3_th30r3m_s0lv3r}`

A.2.8 R8. Restauration de binaire Go strippé (Difficile)

Énoncé : Exécutable massif (5Mo) compilé en Golang (Go), les noms des fonctions ont été supprimés (stripped).
Méthodologie : Les binaires Go stockent toujours dynamiquement la structure des types et des fonctions (la `pclntab`) pour gérer les panics et la garbage collection. On utilise un outil spécialisé comme `GoReSym` ou `Redress` pour parser ces métadonnées et renommer automatiquement des milliers de fonctions dans Ghidra/IDA. L'analyse devient instantanément triviale.
Flag : `CONGO{g0l4ng_pclnt4b_m4g1c}`

A.2.9 R9. Hooking dynamique avec Frida (Extrêmement Difficile)

Énoncé : Binaire très lourdement obfusqué calculant le flag en mémoire.
Méthodologie : L'analyse statique est impossible. On repère avec Ghidra la fonction `decrypt_chunk`

qui prend un pointeur vers le flag déchiffré, mais l'efface juste après. On écrit un script JavaScript `Frida` pour intercepter l'appel, lire le pointeur et dumper la mémoire à la volée pendant l'exécution.

```
Interceptor.attach(ptr("0x00401234"), {
    onLeave: function (retval) {
        // Le registre RAX contient le pointeur vers la string finale
        console.log(Memory.readUtf8String(this.context.rax));
    }
});
```

```
$ frida -l hook.js -f ./binaire
```

Flag : `CONGO{fr1d4_dyn4m1c_h00k}`

A.2.10 R10. Machine Virtuelle Imbriquée (Extrêmement Difficile)

Énoncé : Le programme implémente sa propre architecture processeur avec 200 opcodes inconnus et un bytecode chiffré.
Méthodologie : Il faut écrire un script Python qui lit la table des opcodes (depuis `.rodata`) pour créer un mini-désassembleur. Ensuite, on utilise `unicorn-engine` (framework d'émulation CPU) pour tracer chaque registre de la VM pendant son fonctionnement et repérer à quel moment l'entrée utilisateur est vérifiée. (Reportez-vous à la section complète "VM from Hell" du document original de conception pour un script exhaustif).
Flag : `CONGO{vms_w1th1n_vms_n3v3r_3nd}`

ctf.anssi.cg