HyperPacker – FCSC 2022 – 0xE0L

Introduction :

En se connectant au challenge via le template fourni et en récupérant les premiers binaires packés, on se rend compte de la chose suivante :

- Certains programmes sont très vite unpackés, même si l'on sent que l'unpacking met parfois 3 ou 4 secondes, comme si une tâche précise du programme prenait beaucoup de temps.
- D'autres programmes packés mettent un temps conséquent à être unpackés (genre 30-40s), et d'autres semblent mettre un temps infini (pas unpackés même après 10-20 minutes) …

Sans vouloir spoiler, certains programmes mettraient effectivement un temps infini (ou plusieurs années peut-être) avant de pouvoir être correctement unpackés. On va expliquer après pourquoi.

Problématique vu qu'on a que 60s pour unpacker chaque programme ! Il faudra donc trouver un moyen de les extraire rapidement.

Concernant les packers, ils font tous la même taille (302 592 octets) et sont très similaires :

- Le `.text` de chaque packer (le code) est exactement le même partout, je l'ai vérifié en testant avec le plugin Diaphora d'IDA Pro (qui permet de comparer 2 binaires).
- La seule chose qui change entre chaque packer est le `.data`, donc probablement juste le programme « packé / à unpacker » qui est renfermé à l'intérieur.

Les programmes unpackés sont également tous similaires, il s'agit juste d'un programme qui effectue un MessageBox() affichant le secret à renvoyer. Un simple "strings" sur le programme unpacké peut permettre de récupérer le secret, cette étape-là n'est vraiment pas compliquée.

Comme indiqué, l'étape délicate sera d'unpacker rapidement chaque programme qui va nous être envoyé et de comprendre pourquoi certains mettent autant de temps avant de s'unpacker.

I - Reverse du programme :

En reversant un packer, on constate assez rapidement qu'il semble utiliser les instructions x86 AES-NI pour déchiffrer le programme packé. Reste à savoir de quel mode d'AES il s'agit et quelle est la clé (voire l'IV) !

Au final, je me rends compte des choses suivantes :

- Il s'agit d'AES-CBC (avec clé de 128-bits, on va y revenir).
- L'IV est hardcodé dans le programme, à l'adresse 0x140096A80.
- La clé n'est pas hardcodée dans le programme, elle est donc plus ou moins « bruteforcée » par le programme pendant son exécution ! C'est visiblement cette étape-là qui met beaucoup de temps.

Voici la vue d'ensemble de la fonction principale du packer :

```
  __m128i AesKey; // [rsp+118h] [rbp-18h] BYREF
  LPVOID lpAddress; // [rsp+128h] [rbp-8h]

  addrAesKey = &AesKey;
  v2 = 16i64;
  do
  {
    addrAesKey->m128i_i8[0] = 0;
    addrAesKey = (__m128i *)((char *)addrAesKey + 1);// tout ceci memset juste l'AesKey à 0
    --v2;
  }
  while ( v2 );
  tempBuffPackedData = (__m128i *)VirtualAlloc(0i64, 0x48A80ui64, 0x3000u, 4u);
  if ( tempBuffPackedData )
  {
    lpAddress = tempBuffPackedData;
    qmemcpy(tempBuffPackedData, locPackedProg14004E000, 0x48A80ui64);// --> recopie les données packées (dans .data: 0x14004E000) dans le buffer alloué par VirtualAlloc
LABEL_5:
    if ( isDecryptionKey_1400971C8(locPackedProg14004E000, &AesKey) )// quand ce IF est pris : ça y est, la bonne clé de déchiffrement a été trouvée !!!
    {
      aesDecryptLoop_140097132((const __m128i *)lpAddress, locPackedProg14004E000, &AesKey, 18600i64);// il se sert ensuite du buffer alloué dans VirtualAlloc pour déchiffrer les données...
                                                // et les recopier (déchiffrées) dans .data: 0x14004E000 !
      LODWORD(v4) = VirtualFree(lpAddress, 0i64, 0x8000u);// maintenant il n'a plus besoin de ce buffer, donc il le supprime
      if ( v4 )
        return 1i64;                            // return "success", la bonne clé a été trouvée et le programme packé correctement déchiffré !
    }
    else
    {
      qmemcpy(locPackedProg14004E000, lpAddress, sizeof(__m128i));
      while ( stillKeysToBeGenerated_1400973B3((__int64)&AesKey) )// take a longgg time with slow packers?
      {
        if ( !generateKeyCandidate_140097358(&AesKey, addrInitializationVector_140096A80, 16) )// cette fonction boucle en incrémentant la clé petit à petit et en en sortant des candidats valides !
                                                // si un bon candidat est généré, le "goto LABEL_5" est pris !
                                                // (NOTE : c'est elle qui fait prendre un temps MONSTRUEUX à l'exécution du programme.
                                                // Il va falloir qu'on la recrée nous-même mais de manière bien plus optimisée pour déchiffrer le programme packé en quelques secondes
                                                // Quand les clés sont générées sur un keyspace de 2 ou 3 octets le packer va assez vite, mais quand c'est 8 octets ça prendre un temps exponentiellement long
          goto LABEL_5;
      }
    }
  }
```

Et la fonction de déchiffrement :

```
void __fastcall aesDecryptLoop_140097132(const __m128i *srcDataToDecrypt, _OWORD *dstWhereDecryptedWritten, const __m128i *AesKey, __int64 nbData)
{
  __int64 v4; // rcx
  _BYTE *v5; // r9
  const __m128i *currentIV; // r8
  const __m128i *blockToDecrypt; // [rsp+20h] [rbp-30h]
  __int64 v10; // [rsp+28h] [rbp-28h]
  __int64 v12; // [rsp+38h] [rbp-18h]
  char isFirstBlock; // [rsp+40h] [rbp-10h]

  isFirstBlock = 1;
  derivateAesRoundKeys_14009707B(AesKey);       // génère les rounds keys AES (laissées dans les registres XMM*) à partir de la clé passée en argument
  while ( nbData )
  {
    aesDecryptCore_14009702F(srcDataToDecrypt, dstWhereDecryptedWritten);// déchiffre un bloc en utilisant les rounds keys présentes dans les registres XMM*
    v4 = 0i64;
    v5 = (_BYTE *)v10;
    if ( isFirstBlock == 1 )
    {
      currentIV = (const __m128i *)addrInitializationVector_140096A80;// lors du déchiffrement du tout premier block, l'IV est celui hardcodé dans le programme
      isFirstBlock = 0;
    }
    else
    {
      currentIV = blockToDecrypt - 1;           // ensuite, conformément à la norme AES-CBC (voir les schémas en ligne), pour le déchiffrement des blocks suivants l'IV utilisé est le ciphertext du block N-1 !
    }
    do
    {
      *v5++ ^= currentIV->m128i_i8[0];          // XORe la donnée en sortie d'AES avec l'IV ==> donne le block final déchiffré (cf schémas AES-CBC) !
      currentIV = (const __m128i *)((char *)currentIV + 1);
      ++v4;
    }
    while ( v4 != 16 );
    srcDataToDecrypt = blockToDecrypt + 1;
    dstWhereDecryptedWritten = (_OWORD *)(v10 + 16);
    nbData = v12 - 1;
  }
```

Et le cœur de la fonction de déchiffrement, ou on voit quelques-unes des instructions AES-NI utilisées :

```
.text:0000000140097020
.text:0000000140097020 ; =============== S U B R O U T I N E =======================================
.text:0000000140097020
.text:0000000140097020 ; déchiffre le block passé en 1er arg en utilisant les round keys présentes dans les registr
.text:0000000140097020 ; Attributes: bp-based frame
.text:0000000140097020
.text:0000000140097020 ; void __fastcall aesDecryptCore_14009702F(const __m128i *srcEncrypted, _OWORD *dstDecrypted
.text:0000000140097020 aesDecryptCore_14009702F proc near      ; CODE XREF: aesDecryptLoop_140097132+3E↓p
.text:0000000140097020                 push    rbp
.text:0000000140097021                 mov     rbp, rsp
.text:0000000140097024                 movdqu  xmm0, xmmword ptr [rcx]
.text:0000000140097028                 pxor    xmm0, xmm15
.text:000000014009702D                 aesdec  xmm0, xmm14
.text:0000000140097033                 aesdec  xmm0, xmm13
.text:0000000140097039                 aesdec  xmm0, xmm12
.text:000000014009703F                 aesdec  xmm0, xmm11
.text:0000000140097045                 aesdec  xmm0, xmm10
.text:000000014009704B                 aesdec  xmm0, xmm9
.text:0000000140097051                 aesdec  xmm0, xmm8
.text:0000000140097057                 aesdec  xmm0, xmm7
.text:000000014009705D                 aesdec  xmm0, xmm6
.text:0000000140097063                 aesdeclast xmm0, xmm5   ; aesdeclast: 1st OP = dataToDecrypt, 2nd OP = (last
.text:0000000140097069                 movdqu  xmmword ptr [rdx], xmm0 ; write AES decrypted result back to dstDecr
.text:000000014009706D                 leave
.text:000000014009706E                 retn
.text:000000014009706E aesDecryptCore_14009702F endp
.text:000000014009706E
.text:000000014009706F
.text:000000014009706F ; =============== S U B R O U T I N E =======================================
```

```
// déchiffre le block passé en 1er arg en utilisant les round keys présentes dans les registres XMM14, XMM13, XMM12, etc.
void __fastcall aesDecryptCore_14009702F(const __m128i *srcEncrypted, _OWORD *dstDecrypted)
{
  __m128i v12; // xmm15

  _XMM0 = _mm_xor_si128(_mm_loadu_si128(srcEncrypted), v12);
  __asm
  {
    aesdec  xmm0, xmm14
    aesdec  xmm0, xmm13
    aesdec  xmm0, xmm12
    aesdec  xmm0, xmm11
    aesdec  xmm0, xmm10
    aesdec  xmm0, xmm9
    aesdec  xmm0, xmm8
    aesdec  xmm0, xmm7
    aesdec  xmm0, xmm6
    aesdeclast xmm0, xmm5; aesdeclast: 1st OP = dataToDecrypt, 2nd OP = operand
  }
  *dstDecrypted = _XMM0;                        // PRIMORDIAL : à ce stade, le block déchiffré est contenu dans XMM0
                                                // (ça tombe bien, les registres XMM font la taille d'un block en AES soit 128 bits
                                                // ==> le bloc déchiffré est donc recopié dans la destination dstDecrypted !
}
```

Quant aux données du programme packé, elles sont contenues à l'adresse 0x14004E000. Elles sont déchiffrées en AES-CBC avec l'IV hardcodé dans le programme + une clé bruteforcée par le programme, puis quand les données ont été correctement déchiffrées, le packer fait un "`jmp RAX`" sur l'OEP (Original Entry Point) du programme !

Finalement : avant je disais que la seule chose qui changeait entre chaque programme est le .data qui contient le programme packé / à unpacker… Certes, mais du coup le .data contient aussi l'IV à partir duquel la clé de déchiffrement est bruteforcée/générée !

Et c'est d'ailleurs cet IV le nerf de la guerre, puisqu'en reversant le programme plus en détail, on se rend compte que **la clé AES est bruteforcée / dérivée à partir de l'IV hardcodé dans le programme** !

Voici comment le packer fonctionne pour générer la clé :

1) La fonction `generateKeyCandidate_140097358()` génère, à partir de l'IV qui lui est passé en argument, des « candidats » de clés AES potentielles qui pourraient déchiffrer le programme. Nous allons voir comment elle fonctionne juste après, juste retenir que ça va retourner plusieurs clés AES possibles pour effectuer le bruteforce.

2) La clé est ensuite passée à `isDecryptionKey_1400971C8()` qui déchiffre le contenu du programme packé avec la clé AES passée en argument et copie le résultat dans un buffer temporaire. Elle vérifie ensuite si le buffer déchiffré commence avec les magic bytes "MZ" des exécutables MZ-PE : si c'est le cas c'est la bonne clé de déchiffrement donc la fonction retourne True ! Sinon, ce n'est pas la bonne clé donc la fonction retourne False.

3) Si `isDecryptionKey_1400971C8()` a returné False, nous n'avons pas trouvé la bonne clé donc on retourne à l'étape 1. Si `isDecryptionKey_1400971C8()` a returné True, on a trouvé la bonne clé ! Dans ce cas, le programme packé situé en 0x14004E000 est déchiffré à nouveau avec la bonne clé, puis la fonction principale retourne et peu de temps après le packer va sauter sur l'OEP (Original Entry Point) du programme unpacké pour l'exécuter !

Voici à quoi ressemble la fonction de génération de « candidats de clés » potentielles (`generateKeyCandidate_140097358()`) :

```
// cette fonction dérivée une clé AES possible à partir de l'IV
// pour ça, incrémente la clé (initialisée à 0) octet par octet, et considère qu'une clé est un bon candidat pour être la clé de déchiffrement si...
// ... addrAesKey[i]  AND  NOT(ptrToIV[i]) == 0
__int64 __fastcall generateKeyCandidate_140097358(_BYTE *addrAesKey, _BYTE *addrInitializationVector, char val16)
{
  _BYTE *ptrToIVModified; // r15
  _BYTE *ptrToIV; // [rsp+18h] [rbp+18h]

  ptrToIV = addrInitializationVector;
  ptrToIVModified = addrInitializationVector;
  while ( 1 )
  {
    LOBYTE(ptrToIVModified) = val16 + (_BYTE)addrInitializationVector;// si addrIV = 0xdead0000, ptrToIvModified = 0xdead0010
    if ( ptrToIV == ptrToIVModified )           // break si on a parcouru les 16 octets de l'IV en gros
      break;
    if ( (*addrAesKey & (unsigned __int8)~*ptrToIV) != 0 )// fait cette comparaison octet par octet (de l'indice 0 à 16) entre l'IV et la clé
      return 1i64;                              // si correspond, return "True" --> la clé fait partie de celles possibles / est une bonne candidate !
    ++ptrToIV;
    ++addrAesKey;
  }
  return 0i64;                                  // retourne 0 quand tout l'IV a été parcouru, donc que toutes les clés dérivées possibles ont été générées
                                                // ==> bref, retourne 0 quand cette fonction a fini son travail !
```

Notons que cette fonction fonctionne en duo avec `stillKeysToBeGenerated_1400973B3()`, mais cette dernière ne représente qu'une petite partie du travail effectué. Résumons en fait le fonctionnement de ces deux fonctions ici :

- La clé AES (memset à 0 dans la fonction principale) est parcourue, octet par octet. Un octet est incrémenté (pour passer de 0x8 à 0x9 par exemple) puis passe à travers `generateKeyCandidate_140097358()` qui vérifie si elle est un « bon candidat » ou non. Si elle est un bon candidat, elle est passée à `isDecryptionKey_1400971C8()` comme on l'avait expliqué auparavant. Sinon, elle continue d'être incrémentée jusqu'à ce qu'elle corresponde à un « bon candidat » de clé.

- Dès qu'un octet de la clé (d'indice 0 par exemple – pour rappel taille clé et IV = 16 octets ici) en cours d'incrémentation et de vérification passe à sa valeur maximale 0xFF, il est réinitialisé à la valeur 0x0 puis on passe à l'octet suivant. Bref, en fait ça permet tout simplement de bruteforcer les clés possibles de 0x000000[…]00 à 0xFFFFFF[…]FF par exemple !
- Mais qu'est un « bon candidat » de clé ?
  C'est une clé pour laquelle **l'octet en cours d'incrémentation / vérification va répondre favorablement à cette condition :** `byteKey AND (NOT(byteIV)) == 0` **!**

Ce qu'on comprend, c'est que si le programme met autant de temps à bruteforcer la bonne clé à trouver, c'est qu'il génère les clés possibles des valeurs 0 à 256^16 (vu que c'est une clé de 16 octets), ce qui fait 3.4E+38 possibilités !

Pour certains programmes où la clé est située dans les premiers octets qui sont bruteforcés, ça va assez vite (3 à 4 secondes), mais dès que l'on parle d'une clé plus grosse, ça va mettre un temps fou qui pourrait durer potentiellement plusieurs années !

Notre but va être de **réimplémenter ce processus de bruteforcing en l'optimisant** ! Pour cela, plutôt que de tester toutes les clés possibles sur 256^16, on peut effectuer ce bruteforce octet par octet :

- Pour l'octet d'indice [0] de l'IV, on génère des octets de clé de 0 à 255 et on regarde lesquels répondent à la condition `byteKey AND (NOT(byteIV)) == 0`. Par exemple, on va obtenir que les octets 0x0 et 0x48 répondent à cette condition.
- On refait la même chose avec l'indice [1], on a cette fois les valeurs 0x8 et 0x32 par exemple.
- Et cetera, on fait ceci sur les 16 octets de l'IV.

Au bout du compte, pour reprendre l'exemple ci-dessus en se concentrant juste sur les 2 premiers octets de l'IV (et en oubliant les 14 autres juste pour l'exemple), voyons les possibilités de clés :

- 0x0008
- 0x0032
- 0x4808
- 0x4832

On combine en fait les différentes valeurs possibles pour chaque octet de la clé !

Bref, pour donner un autre exemple avec seulement les 3 premiers octets de la clé : si on a 3 octets de clé qui correspondent pour l'indice [0], 4 pour l'indice [1], 5 pour l'indice [2], au bout du compte on va avoir 3*4*5 = 60 clés possibles ! Une fois nos 60 clés possibles trouvées, on a plus qu'à toutes les utiliser pour déchiffrer le programme, jusqu'à ce que le déchiffré commence avec les magic bytes "MZ" des exécutables MZ-PE → ça voudra dire qu'on a trouvé la bonne clé !

Ce processus permet donc de réduire drastiquement le nombre « d'essayage » de clé effectué. Au lieu d'effectuer un bruteforcing sur 256^16 clés, on va **effectuer du bruteforcing sur quelques milliers** voire parfois quelques millions (pas plus) de clés, ce qui peut être **suffisant pour unpacker correctement le programme en moins d'1 minute.**

Autre chose que je remarque, non pas en reversant mais en débuggant dynamiquement plusieurs packers qui s'unpackent rapidement : **quand un octet de l'IV est égal à 0, l'octet de la clé correspondant sera forcément égal à 0 lui aussi** ! Intéressant car cette observation va nous permettre d'éliminer encore plus de possibilités de clés et donc d'accélérer notre méthode !

Je ne sais pas si c'est quelque-chose qu'on était censé observer par l'expérience ou si j'ai juste mal reversé le binaire, mais en tout cas cette observation va payer pour la suite. Spoiler, on va voir d'ailleurs qu'une seconde observation manque et que celle que nous venons de faire peut être « généralisée », mais j'y reviendrai un peu après !

Maintenant, au travail ! Nous allons réimplémenter cette méthode en Python.

II - PoCs de résolution :

1) Optimisations de base :

J'implémente tout ce que je viens d'expliquer ici dans un premier script Python, `3-PoC.py`.

En gros, ce PoC 1) lit l'IV hardcodé dans le packer ainsi que les données packées, 2) le dérive selon ce que nous avons expliqué pour former plusieurs possibilités de clés, 3) teste chaque possibilité de clé en déchiffrant les données packées et en regardant si elles donnent un MZ-PE valide → si c'est le cas on a trouvé la bonne clé et correctement unpacké le programme !

Voici ce à quoi ressemble la fonction qui génère toutes les possibilités de clés :

```
def generatePossibleKeys(IV):
    # For each byte of IV, we bruteforce corresponding possibilities of byte for the key
    possibilities = []
    n = 0
    for i in range(0, 16):
        possibilities.append([])

        # OBSERVATION: if a byte of IV == 0, corresponding byte of key will be always == 0 too!
        if IV[i] == 0:
            possibilities[n].append(0x0)

        # Byte generation process
        else:
            for bKey in range(0,255):
                bIVnot = int(bitwise_not(bin(IV[i])), 2)
                if bKey & bIVnot == 0:
                    possibilities[n].append(bKey)
        n+=1

    # Generating all possibilities of keys using itertools.product which greatly does the job for us!
    possibleKeys = [] # returned as byte array
    for possKey in itertools.product(*possibilities):
        keyHex = ""
        for nb in possKey:
            keyHex += "{:02x}".format(nb)
        possibleKeys.append(bytes.fromhex(keyHex))

    print("[*] Number of possible keys:", len(possibleKeys))
    return possibleKeys
```

J'ai commencé par implémenter ce PoC en déchiffrant l'intégralité des données packées, soit – dans le fichier exécutable du packer – les données contenues de l'offset 0x400 à 0x48e70 (l'IV de 16 octets étant lui contenu à l'offset 0x48E80 du fichier).

En testant sur plusieurs programmes, la plupart sont unpackés dans la seconde quand il n'y a que quelques milliers de possibilités, mais quand il commence à y avoir 3 ou 4 millions de possibilités, ça prend du temps ! Pour gagner encore plus de temps sur cette méthode, j'ai décidé de n'unpacker qu'une partie du programme (= une partie où le secret est contenu évidemment), soit de l'offset 0x400 à 0x3000 (au lieu de 0x48E80) dans le packer. Cela fait en effet gagner un peu de temps.

J'ai jusqu'ici deux optimisations, pour résumer :

- J'avais remarqué que quand un octet d'IV est égal à 0, l'octet de clé correspondant sera égal à 0 aussi (c'est implémenté dans la fonction en screenshot ci-dessus).
- Je ne déchiffre qu'une partie du programme packé (où l'IV est contenu) pour que le déchiffrement, et donc le bruteforcing des clés, aille plus vite.

Je me rends compte que pour 90% des programmes reçus, j'arrive à les dépacker dans la minute. Mais malheureusement, il reste 10% d'irréductibles gaulois qui se dépackent en plus de temps et je n'arrive pas à récupérer le flag !

Il va donc falloir trouver d'autres optimisations.

2) Optimisation supplémentaire :

Une idée me vient à l'esprit : je peux probablement réduire encore plus le nombre de blocs AES à déchiffrer. Pour l'instant, comme indiqué, je déchiffre les blocs de 0x400 à 0x3000 (0x400 étant le début des données chiffrées, et 0x3000 une valeur plus ou moins arbitraire).

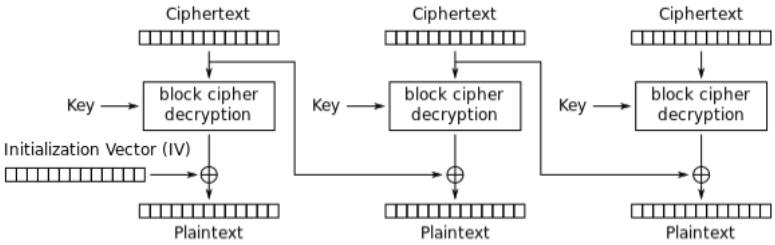
Plutôt que de m'embêter à faire ça, **je dois pouvoir juste déchiffrer les 3 ou 4 blocs où est contenu le secret qui nous intéresse** !

Je décide donc d'ouvrir plusieurs exemples de programmes unpackés, et je remarque qu'ils sont encore plus similaires entre eux que je ne le pensais :

```
00002200 54 61 6b 65 20 79 6f 75 72 20 73 65 63 72 65 74 Take your secret
00002210 20 21 00 00 00 00 00 00 53 45 43 52 45 54 20 69  !......SECRET i
00002220 73 20 3a 20 46 39 47 4a 4e 38 52 54 57 4b 35 4d s : F9GJN8RTWK5M
00002230 45 57 34 50 4f 39 36 4f 4b 4e 32 45 54 35 53 55 EW4PO96OKN2ET5SU
00002240 4e 4f 42 35 00 00 00 00 00 00 00 00 00 00 00 00 NOB5............
```

Le secret est toujours contenu, comme on le voit ci-dessus, de l'offset 0x2200 à 0x2240 dans le programme unpacké ! Dans le programme packé, vu qu'on avait dit que les données packées/chiffrées commencent à partir de l'offset 0x400, **les blocs à déchiffrer qui nous intéressent sont donc ceux de 0x2600 à 0x2650** ! On va donc juste chercher à déchiffrer ceux-là à chaque fois, ça va nous faire gagner un temps fou !

Resortons nos vieux cours et revoyons le fonctionnement d'AES-CBC :



Cipher Block Chaining (CBC) mode decryption

On voit que si on veut commencer le déchiffrement non pas au bloc 0, mais au bloc 10 par exemple, il ne faudra pas prendre comme IV celui qui est hardcodé dans le programme (il nous servira toujours pour générer les clés possibles bien sûr !) mais le ciphertext du bloc N-1, soit le ciphertext du bloc 9 !

Je modifie donc mon programme dans `3-PoC-opti.py` qui va rajouter cette optimisation. Il déchiffre les 5 blocs AES de **0x2600 à 0x2650** (en prenant comme **IV le bloc N-1 à l'offset 0x25F0 !**) et ne vérifie plus que le déchiffrement s'est bien effectué en regardant si le déchiffré commence par "MZ\x90" mais tout simplement en regardant s'il commence par les 4 octets "Take" (de la phrase "Take your secret") ! Prendre 2 ou 3 octets pour effectuer la vérification est un peu light par ailleurs, ça risque de créer des collisions / faux positifs quand on va tester plusieurs millions de clés.

En faisant ceci, **j'arrive effectivement à gagner du temps, mais désormais pour 2-3% des programmes à unpacker cela met malheureusement plus d'une minute** ! Cela arrive quand il y a quelque-chose comme 16 millions de clés à tester en moins d'une minute…

Malgré tout, **en relançant mon script qui dialogue avec le serveur du FCSC m'envoyant les binaires à unpacker, avec un peu de réussite j'arrive au bout du 8ème ou 9ème essai à ne tomber sur aucun binaire ultra lent et à obtenir le flag** ! Incroyable !

Je me rends compte à ce moment-là qu'il y a 30 packers à unpacker (chacun en moins d'une minute) pour parvenir jusqu'au flag, il m'a donc fallu un peu de chance pour que sur les 30 binaires m'étant envoyés à chaque essai, je finisse par ne tomber sur aucun qui prenne énormément de temps !

Mais tout ceci indique que ma méthode n'est pas la meilleure et la plus fiable, j'ai donc voulu aller plus loin pour qu'elle soit propre, rapide et fiable à 100%. Si nous avons optimisé au maximum le déchiffrement et que cela ne semble pas suffire, c'est logiquement **qu'on peut réduire encore plus le nombre de possibilités de clés à bruteforcer** !

3) Optimisation ultime :

Je décide donc de regarder tous les couples IV – clé bruteforcées en ma possession, en espérant remarquer quelque-chose qui me ferait gagner encore plus de temps. Et finalement, la lumière finit par venir : **la valeur d'un octet d'indice i de la clé, est forcément inférieur ou égal à la valeur de l'octet d'indice i de l'IV hardcodé dans le programme** !

Donc par exemple, si un octet de l'IV est égal à 200, un octet de clé peut être égal à 32, 120 ou 200, mais ne peut PAS être égal à 236 (et ce MÊME si la valeur 236 satisfait la condition `byteKey AND (NOT(byteIV)) == 0`).

Cette observation est donc une **généralisation de l'observation que j'avais faite au début, en disant que quand un octet d'IV est égal à 0 l'octet de clé correspondant est forcément égal à 0** !
Dommage que je n'ai pas fait cette généralisation plus tôt !

Je reporte donc cette optimisation dans **`3-PoC-ultimate.py`**, voici le bête patch qu'il y a à faire :

```
def generatePossibleKeys(IV):
    # For each byte of IV, we bruteforce corresponding possibilities of byte for the key
    possibilities = []
    n = 0
    for i in range(0, 16):
        possibilities.append([])

        # OBSERVATION: if a byte of IV == 0, corresponding byte of key will be always == 0 too!
        if IV[i] == 0:
            possibilities[n].append(0x0)

        # Byte generation process
        else:
            for bKey in range(0,255):  ->  for bKey in range(0, IV[i]+1):
                bIVnot = int(bitwise_not(bin(IV[i])), 2)
                if bKey & bIVnot == 0:
                    possibilities[n].append(bKey)
        n+=1
```

Cela étant fait, miracle, j'arrive à **déchiffrer quasi instantanément TOUS les programmes** (y compris ceux à 16 millions de possibilités de clés "avant le patch" qui prenaient plus d'une minute à être bruteforcés) **car j'ai drastiquement réduit le nombre de clés possibles** !

L'optimisation sur le déchiffrement AES était inutile (bien que formatrice, ça m'a permis de réviser AES-CBC), plutôt que de chercher à gagner du temps sur le déchiffrement j'aurais dû trouver un moyen de réduire le nombre de possibilités de clés ! Cette ultime observation est donc la clé du succès.

III - Script final :

Je finis par produire un script `solve.py` qui reprend celui donné par le challmaker, et j'y incorpore toutes les optimisations que j'ai décrites jusqu'ici.

Je l'exécute et miracle, en une 20aine de secondes à tout casser, il arrive à **unpacker le contenu des 30 packers qui nous sont envoyés** :

```
b'Well done!\n'
b'Here is your binary:\n'
[+] Binary written at /tmp/hyperpacker27.exe
b'Please input the secret within 60 seconds:\n'
IV: b'\x00\x00\x00\x04\x8d\x11\xa4\x001\x10T\x00\x00\x00\x00\x00'
[*] Number of possible keys: 131072
[+] Successfully decrypted the packed MZ-PE!
[+] Correct key was b'\x00\x00\x00\x04\x8d\x11\xa4\x00\x10\x00\x04\x00\x00\x00\x00\x00'
[+] Unpacked binary extracted to /tmp/unpacked27.exe
[\o/] Found secret: DC49CJ7J7IMOH87SP1C1FIIH0ZELAHLO
b'Well done!\n'
b'Here is your binary:\n'
[+] Binary written at /tmp/hyperpacker28.exe
b'Please input the secret within 60 seconds:\n'
IV: b'\x00\x00\x08\x04@`\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
[*] Number of possible keys: 32
[+] Successfully decrypted the packed MZ-PE!
[+] Correct key was b'\x00\x00\x08\x04@@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
[+] Unpacked binary extracted to /tmp/unpacked28.exe
[\o/] Found secret: KH2YDE5OL3YTXYFEJ0O40FE4TBLSUT1I
b'Well done!\n'
b'Here is your binary:\n'
[+] Binary written at /tmp/hyperpacker29.exe
b'Please input the secret within 60 seconds:\n'
IV: b'\x00\x00\x00\x00\x00\x00\x00\x00\x00 \x00r\xb8c\x10\x00'
[*] Number of possible keys: 16384
[+] Successfully decrypted the packed MZ-PE!
[+] Correct key was b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00`\x18\x02\x00\x00'
[+] Unpacked binary extracted to /tmp/unpacked29.exe
[\o/] Found secret: AGJCMQK13FBXX4NLMXRA4S4QQ6PH7FF5
b'Well done!\n'
b'Here is your binary:\n'
[+] Binary written at /tmp/hyperpacker30.exe
b'Please input the secret within 60 seconds:\n'
IV: b'\x00\x00\x00\x00\x00\xa5\x00\x01\x00\x00\xaa\xa2-\x00\x00\x00'
[*] Number of possible keys: 65536
[+] Successfully decrypted the packed MZ-PE!
[+] Correct key was b'\x00\x00\x00\x00\x00\xa0\x00\x00\x00\x00"\xa2!\x00\x00\x00'
[+] Unpacked binary extracted to /tmp/unpacked30.exe
[\o/] Found secret: 87T665NDNKF3DLTITNW4YN2G1BPT4J2Y
b'Well done!\n'
Flag is: b'Congratulations! Here is your flag: FCSC{2b60aef3d241d3c37c30373a9a4446017a6d5b6761ae5492e3f824e280cb8ceb}\n'
[*] Closed connection to challenges.france-cybersecurity-challenge.fr port 2202
[redacted]:~/Bureau/FCSC_REV$
```

Cette méthode est donc la plus rapide et la plus fiable possible.

Flag : **FCSC{2b60aef3d241d3c37c30373a9a4446017a6d5b6761ae5492e3f824e280cb8ceb}**