Reverse Engineering - Rapport CrackMe

yocvito

Octobre 2022

Contents

1 Preparation

1.1 CFF Explorer

Tout d'abord, on analyse les en-têtes et autres informations utiles du binaire avec CFF Explorer. On voit premièrement que c'est un binaire PE 64 bits, le code a été écrit en C++ (version 8.0). Dans la section `Nt Headers` → `Optional Header`, on peut trouver l'adresse de base du programme `0x140000000` ainsi que par exemple les charactéristiques du binaire comme certaines protections.

En allant maintenant dans la section `Import Directories`, on peut voir les différentes dll importés par notre binaire et on peut éventuellement analyser les fonctions importés depuis chacune d'entre elles par notre binaire. Peut être que cela pourrait éventuellement nous donner des indices sur le type d'opération que fait le programme. Par exemple, on voit des fonctions pour intéragir avec des zones mémoires, récupérer et afficher du texte mais rien ne semble nous indiquer l'utilisation de sockets.

1.2 Run du programme

Afin d'avoir une première idée de ce que fait le programme, on le lance (idéalement dans un environnement sandboxé). On remarque que le programme nous demande un code ("regcode:"), on tape une chaine aléatoire et il nous renvoit un message d'erreur ("unregistered").

2 Première analyse rapide - IDA

Tout les offsets présentés en fonction de la base du binaire `Exam_CSI.exe` sont relatifs à l'adresse de base de celui ci dans IDA (`0x140010000`).

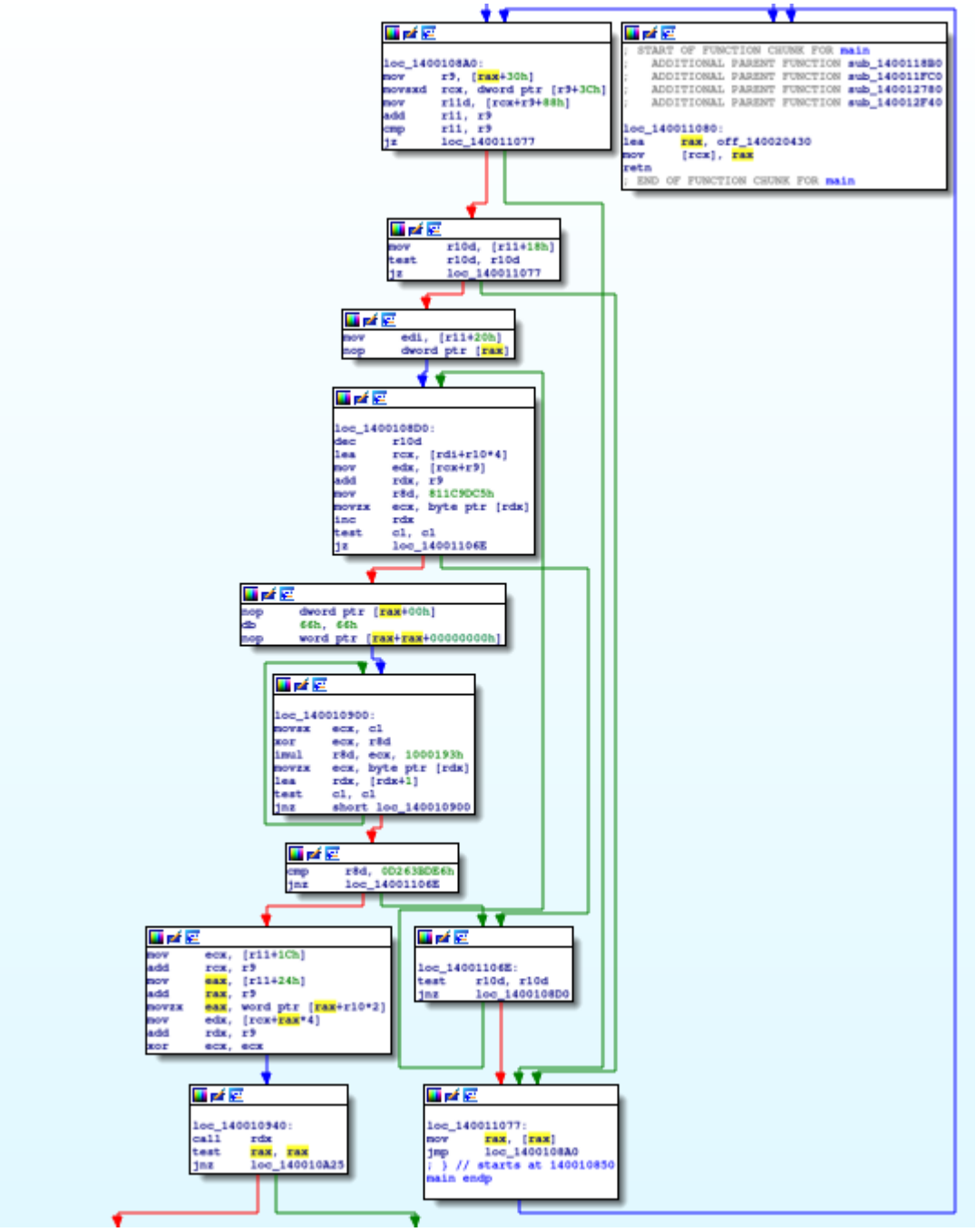
2.1 Deblayage

On va premièrement effectuer une analyse statique du binaire, afin de regarder l'architecture globale du programme, essayer de comprendre comment il fonctionne, expliciter des variables locales/fonctions (leurs donner un nom, les commenter) et éventuellement essayer de retrouver des secrets si ils ne sont pas durement obfusqués.

On ouvre donc le binaire dans IDA qui nous amene directement au main. En faisant `CTRL + E` et en cliquant sur le point d'entrée (`start`), on peut lire le code et remonter jusqu'au main. On voit que la fonction qui appelle le main (`__scrt_common_main_seh`) et en fait une fonction d'initialisation lié au système d'exploitation Windows.

En étant positionné sur le main, on va maintenant faire une première passe sur le code de cette fonction pour essayer de dégrossir le travail pour une analyse statique plus approfondie qui va suivre. On va donc essayer de repérer les boucles, conditions qui amènent à des branches différentes dans le flot d'exécution du programme, et les appels de fonctions.

Tout d'abord, on remarque du junk code en haut à droite du graphe de la fonction main. En effet, ce code n'est pas accédé depuis la fonction main et termine juste celle ci en modifiant des registres et de la mémoire. On garde tout de même cela en tête, car le programme pourrait éventuellement sauté sur ces instructions depuis une autre fonction (ce qui amènerait à un comportement tout autre).
On continue et on remarque une première boucle qui commence à l'offset `Exam_CSI.exe+0x8a0`.



En descendant un peu plus bas dans le code, on peut observer la condition de sortie, à savoir que le registre `r8d` (partie 32 bits de poid faible pour le registre r8) soit différent de `0x0D263BDE6`. En recherchant cette valeur sur google, rien ne ressort ce qui signifie que ce n'est pas une valeur signante (tel quel). En revanche, on remarque qu'une des valeurs dans la boucle (`0x1000193`) est, elle, sig-

nante et semble être en rapport avec des fonctions de hashage. Néanmoins, rien ne nous permet de rapidement comprendre ce que fait exactement cette boucle, on y reviendra donc plus tard si nécessaire.

On continue à la sortie de la boucle, un certain nombre de registre est modifié en fonction de valeurs dépendantes de la boucle précédante, puis l'on met `rcx` (le premier argument pour la convention d'appel `__fastcall`) à 0 avec un xor. On appel ensuite la fonction dont l'adresse est contenue dans le registre rdx. On ne sait pas pour l'instant ce que contient rdx mais il pourrait éventuellement être construit par la boucle précédente. En fonction du retour de la fonction, on continue dans un cas vers un long enchainements de blocs d'instructions, et dans l'autre cas on exécute un certains nombre d'instructions ressemblant au pattern de la boucle que l'on vient de voir (appellant cette fois la fonction contenue dans rax), puis on arrive à la fin du programme. On peut donc spéculer sur le fait que ce soit une fonction d'initialisation qui est nécessaire au bon fonctionnement du programme.
Dans le cas de succès, on appelle une nouvelle fonction `sub_140013690` en lui passant le retour de la fonction précédente en argument. Comme précédemment, cette fonction nous renvoit à la fin du programme si elle échoue (de façon encore plus directe que la fonction précédente). En regardant rapidement son code, on voit qu'il y'a de multiples boucles et appels de fonction ce ne nous permet pas de savoir ce qu'elle fait directement.

A la suite de ces 2 checks on initialise 2 variables locales (dans la stack), une avec une valeur de 16 octets et une autre dont l'on met le premier octet à zero (`@@ret-0x30`), surement une chaine, on peut donc renommer cette variable pour signifier cela.
Ensuite, on voit que le programme effectue tout un tas d'opération XOR pour initialiser des variables dans la pile. On peut directement remarquer que certains des caractères initialisés se suivent ce qui peut faire penser que l'on construit une chaine, ou un tableau de maniere plus générale. On peut récupérer la clé utilisé pour le XOR, qui est d'ailleurs stocké dans la pile (`@@ret-0x80`), et calculer la valeur finale (ce qui pourrait être utile si il déchiffre du texte). En l'occurence, on remarque surtout que les valeurs de base avant de passer dans le XOR sont dans la table ASCII et en appuyant sur `r` on peut voir petit à petit apparaître le message **"regcode:"**, qui correspond au prompt du code d'enregistrement vu précédemment lors du run.

```
mov     dl, 0BEh        ; xor key
mov     [rsp+178h+v80_key_xor_msg], dl
xor     eax, eax
mov     [rsp+178h+var_7C], rax
mov     [rsp+178h+var_74], rax
mov     [rsp+178h+var_6C], rax
mov     [rsp+178h+var_64], rax
movzx   eax, dl
xor     al, 72h ; 'r'   ; 0xBE ^ 0x72 'r' = 0xCC
mov     [rsp+178h+var_50], al
movzx   ecx, dl
xor     cl, 65h ; 'e'   ; 0xBE ^ 0x65 'e' = 0x
mov     [rsp+178h+var_4F], cl
movzx   eax, dl
xor     al, 67h ; 'g'   ; 0xBE ^ 0x67 'g' = 0x
mov     [rsp+178h+var_4E], al
movzx   eax, dl
xor     al, 63h ; 'c'   ; 0xBE ^ 0x63 'c' = 0x
mov     [rsp+178h+var_4D], al
movzx   eax, dl
xor     al, 6Fh ; 'o'   ; 0xBE ^ 0x6F 'o' = 0x
mov     [rsp+178h+var_4C], al
movzx   eax, dl
xor     al, 64h ; 'd'   ; 0xBE ^ 0x64 'd' = 0x
mov     [rsp+178h+var_4B], al
mov     [rsp+178h+var_4A], cl ; 'e'
xor     dl, ':'
mov     [rsp+178h+var_49], dl ; 0xBE ^ 0x3A ':' = 0x
xor     eax, eax
mov     [rsp+178h+var_48], al
```

On vient donc de chiffrer ce message qui est destiné à nous être affiché, en toute logique la clé de chiffrement doit être réutilisé par la suite. En analysant les occurences de la variable locale où le programme stock cette clé, on voit qu'elle est utilisé pour déchiffrer le contenue d'une chaine renvoyé par la fonction `sub_140010560` juste avant de passer cette même chaine en paramètre à la fonction `puts`. On en déduit que le code entre les opération XOR et la fonction `puts` peut potentiellement servir a obfusquer encore plus cette chaine avant de la récupérer pour l'affichage.

Après l'appel à `puts`, on voit qu'on appelle la fonction `sub_140011430` en lui passant en argument le stream standard de sortie c++ et un pointer vers `@@ret-0x30` (ce qu'on pensait être une string). Vu que l'on a mis cette chaine à 0 et que l'on vient d'afficher un message de prompt, on peut imaginer que cette fonction récupère notre entrée. En regardant rapidement son code, on voit l'appel aux méthodes `getc` ou `nextc` ce qui confirme ce que l'on pensait. On peut donc renommer cette fonction mais il faut garder en tête que cette fonction pourrait aussi modifier notre entrée ou effectuer d'autres actions sur le programme.

Finalement, on appel la fonction `sub_1400106d0` en lui passant en paramètre notre input. Le code de cette fonction semble relativement simple, à savoir qu'il semble éventuellement allouer de la mémoire, copier notre input, puis parcourir celle ci dans une boucle en appliquant des opérations crypto dessus. On voit qu'à la fin, on met une variable globale dans la zone de donnée à 1 (`cs:qword_140030AE0`) si le registre ebx est égale à une certaine valeur `0x0A50B8B0`. On peut donc spéculer que cette fonction qui

analyse notre input et la compare à une certaine valeur doit surement être le fonction de vérification. Juste après l'appel à cette fonction dans le main, on voit que la même variable globale modifié dans la fonction est comparé dans le main est donne lieu à 2 branches d'exécution différentes du programme.

En regardant les 2 blocs de code suivants, on remarque le pattern précédent avec le xor de chaines en clair, des opérations crypto et appels de fonctions, puis la dé-obfuscation du message avant son affichage.
On voit qu'après avoir récupéré le message dé-obfusqué, on appel la fonction `sub_140011630` en lui passant en paramètre `std::cout` (le descripteur de fichier c++ représentant stdout) ainsi que le message. En récupérant les messages clairs pour les 2 branches, on peut détecter la branche de succès (variable globale de check à 1) où l'on voit le message "registered", où celle d'echec avec le message qu'on a vu auparavant lors de l'exécution : "unregistered".

2.2 Récapitulatif

On peut maintenant faire un synthèse de nos découvertes afin de voir quelle est l'architecture globale du programme (surtout du main) et quels sont les principaux points d'intérêts à reverse.
Nous avons donc pu tout d'abord voir une phase **d'initialisation** où l'on appelle 2 fonctions. Ensuite, vient la partie **d'obfuscation/de-obfuscation du message** "regcode", et son affichage. Juste en suivant vient la **récupération de notre input** et la **vérification** de celle ci. Enfin, on affiche un message pour alerter l'utilisateur sur le résultat du test sur notre input.

Premièrement, la partie la plus intéressante est clairement la vérification de l'input car trouver une faille dans l'algo de verification nous permettrait de passer le test de manière arbitraire. Ensuite, la phase d'initialisation peut potentiellement effectuer des actions très intéressantes pour nous (connection à un serveur distant, lecture de fichiers, activation de backdoors, ...). Il sera aussi nécessaire de vérifier si l'étape de récupération de notre entrée ne modifie pas celle ci à la volée. Enfin, on pourra si on en a le temps documenter en détail les méthodes d'obfuscation et anti-debug à l'oeuvre dans ce binaire.

3 Analyse approfondie - IDA / x64dbg

3.1 Récupération de pointeurs de fonction

Nous allons maintenant analyser la première boucle dans le programme main afin de comprendre comment elle construit le pointeur contenue dans `rdx`. On lance donc le binaire dans x64dbg.

Regardons tout d'abord le premier bloc du main, on remarque que l'on charge une adresse dans le registre rax, depuis lequel on récupère ensuite une autre adresse. En regardant de plus près, on voit que cette première adresse `gs:[60]` correspond à l'adresse du PEB (Process Environment Block) qui contient des informations sur notre processus. Il contient d'ailleurs un octet qui définit si le processus est en train de se faire débugger, ce qui peut être intéressant à modifier si on lance le binaire avec x64dbg. Le binaire récupère donc l'adresse de son PEB, puis récupère ensuite l'adresse du module "ntdll", qui semble importer des fonctions de librairies et bas niveau de windows. On remarque nottament cela en voyant que l'adresse récupéré dans le PEB pointe vers une section du module ntdll (plus précisemment vers la section `.data`)

On passe ensuite dans le début de la boucle, on récupère tout d'abord dans `r9` un pointeur vers la base de notre binaire, puis on ajoute un offset pour récupérer une autre adresse dans `r11` qu'on compare à r9. On voit que ce test n'es vrai que lors du premier passage dans la boucle. Par ailleurs, r11 semble pointer dans la section `.rodata` de ntdll, cela pourrait être un tableau ou structure de donnée statique.
Ensuite, on récupère une valeur depuis la structure pointé par r11 dans `r10d` et on la compare à 0. Dans le cas où elle n'est pas égale à 0, on continue vers le corps de la boucle en initialisant le registre edi juste avant (depuis la structure pointée par r11). On voit par ailleurs que la valeur récupéré dans r10d est une taille ou un nombre d'élément, puisqu'il est décrémenter directement au début de la boucle puis utiliser pour déréférencer un pointeur depuis r9.
Pour ce qui est du corps de la boucle, le binaire récupère un pointeur vers le symbole courant indexé par r10d depuis r9 (le module courant). Après avoir récupéré cette adresse de chaine de caractère, on initialise r8d avec la valeur `0x811C9DC5` puis on parcourt la chaine en effectuant un xor du caractère courant avec r8d, avant de multiplier le résultat par `0x1000193` et remettre le résultat dans r8d. On obtient à la fin une valeur dans r8d en rapport de notre chaine que l'on compare à une certaine valeur fixe. Il semblerait que la sous-boucle (parcourt de chaine) servirait à hasher la chaine. En se positionnant en sortie de boucle, on voit que le registre r9 pointe vers le module "kernel32". Le binaire est donc en train de parcourir tout les symboles de chaque module et s'arrête quand il a trouvé celui qui correspond au bon hash qu'il a stocké en dur.
En l'occurence, on voit que la fonction qu'il essaye de retrouver est `GetModuleHandleW(NULL)` (l'adresse de la fonction est retrouvé depuis r9 et r11 en utilisant l'indice r10d).
On voit rapidement que cette fonction permet de récupérer un handle vers un module (HMODULE), ou vers le binaire quand NULL est passé en paramètre. Ce handle peut servrir à intéragir avec l'OS et le binaire via l'appel de différentes fonctions.

On garde en tête ce pattern de hashage car il pourrait être amené à réapparaitre dans le binaire.

En recherchant les immediates sur internet, on trouve une fonction de hashage correspondant à la boucle qu'on vient de reverse.

```
1  // dans notre cas la fonction est surement inline
2  inline uint32_t fnv32hash(const std::string& str) {
3      uint32_t hash = 0x811c9dc5;
4      const uint32_t primeVal = 0x1000193;
5
6      for(int i = 0; i < str.size(); i++) {
7          uint8_t byte = str[i];
8          hash ^= byte;
9          hash *= primeVal;
10     }
11
12     return hash;
13 }
```

3.2 Fonction utilisant le pointeur de HMODULE

Juste après l'appel à `GetModuleHandleW()`, on appelle une deuxième fonction `sub_140013690` qui amène vers la fin du programme en cas d'echec.

On continue juste l'exécution du programme dans x64dbg et on entre dans cette fonction. On voit directement que cette fonction est consistante, on va donc d'abord faire un passage avec IDA.

On observe que la fonction contient de nombreuses boucles et en regardant plus attentivement, on remarque le pattern précédent de récupération d'adresse de fonction. Cela va nous simplifier la tache, on regarde juste les appels de fonction, et les arguments qui semblent être passés en paramètre afin de se faire une idée de l'architecture de la fonction. On peut d'ailleurs même coder un petit script python qui nous renvoit le nom de fonction à partir du hash.

```
1  #!/bin/python3
2
3  import sys
4
5  def get_hashes(dll):
6      import pefile
7      pe = pefile.PE(dll)
8      hashes = {}
9      for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
10         if exp.name != None:
11             hashes[fnvhash(exp.name.decode('utf-8'))] = exp.name.decode('utf-8')
12     return hashes
13
14 def fnvhash(data):
15     h = 0x811c9dc5
16     for c in data:
17         h = h ^ ord(c)
18         h = (h * 0x01000193) & 0xffffffff
19     return h
20
21 def get_func_hashes():
22     dlls = [
23         "kernel32.dll",
24         "ntdll.dll",
25         "msvcp140.dll",
26         "vcruntime140.dll"
27     ]
28     hashes = {}
29     for dll in dlls:
30         hashes[dll.split('.')[0]] = get_hashes(dll)
31     return hashes
32
33 def main(argc, argv):
34     if argc < 2:
35         print("Usage: %s <hash>" % argv[0])
36         exit(1)
37
38     hash = int(argv[1], 16)
39
40     hashes = get_func_hashes()
41
42     for module, funcs in hashes.items():
```

```
43          if hash in funcs:
44              print("%s:%s" % (module, funcs[hash]))
45              exit(0)
46
47
48  if __name__ == "__main__":
49      main(len(sys.argv), sys.argv)
```

On va donc pouvoir utiliser ce programme pour reverse statiquement cette fonction. Par exemple, la première fonction qui est appellé à un hash égale à `0x0F3C25A15`. On voit que la fonction appellé est RtlImageNtHeader.

```
1 $ python3 ../get_func_hashes.py 0F3C25A15
2 ntdll:RtlImageNtHeader
```

On va maintenant construire un pseudo code source simplifié pour comprendre ce que fait globalement cette fonction. On retrouve la fonction dont l'adresse est construire avec une boucle, regarde quels sont ses arguments, et vers où on branche en fonction du retour de la fonction.

```
1  // full static analysis reconstruction of sub_140013690
2  bool
3  init_binary(HMODULE *handle)
4  {
5      IMAGE_NT_HEADERS *headers_bin, *headers_bin_2;
6      SYSTEM_INFO sys_info;
7
8      headers_bin = ntdl::RtImageNtHeader(handle);
9      if (headers)
10     {
11         kernel32::GetSystemInfo(&sys_info);
12
13         void *mem = kernel32::VirtualAlloc(headers_bin->OptionalHeader,
14                                            headers_bin->OptionalHeader->ImageBase,
15                                            MEM_COMMIT | MEM_RESERVE,
16                                            PAGE_EXECUTE_READWRITE);
17
18         sub_140013E10(headers_bin, mem); // array copy ?
19         // bad things happens in this func
20         sub_140013EB0(mem); // retrieve section + crypto primitive ?
21
22         headers_bin_2 = ntdll::RtlImageNtHeader(handle);
23
24         if (headers_bin_2->OptionalHeader)
25         {
26             if (call_more_funcs(handle)) // sub_1400141A0
27             {
28                 kernel32::VirtualFree(mem, 0, MEM_RELEASE);
29             }
30         }
```

```
31      }
32  }
33
34  void *
35  call_more_funcs(HMODULE *handle)
36  {
37      MEMORY_BASIC_INFORMATION mem_info;
38      int nbytes = kernel32::VirtualQuery(0, &mem_info, 0x30);
39      if (nbytes > 0)
40      {
41          return kernel32::VirtualProtect(); // need dynamic analysis to retrieve args
42      }
43      return 0;
44  }
45
46  void
47  sub_140013EB0(void *mem)
48  {
49      IMAGE_NT_HEADER *headers;
50
51      headers = ntdll::RtlImageNtHeader(mem);
52
53      // map section + crypto operations
54      // and more ...
55  }
```

Je me suis arrêté au premier niveau d'appel de fonction, car par exemple la fonction `sub_140013EB0` était trop complexe et prenait trop de temps à reverse.
En revanche, j'avais cru remarquer pendant mes sessions de debug que lorsque je mettais des breakpoints dans le main, à partir d'un moment lors de l'exécution, les instructions suivants mes breakpoints était complètement modifiés ce qui entrainait un crash du programme. En positionnant un breakpoint juste avant la fonction que l'on vient de reverse dans le main (`sub_140013690`) et un juste à l'instruction d'après. On voit qu'a la sortie de la fonction, les iunstructions suivants le deuxième breakpoint sont modifiés.

En prenant en compte cette remarque et en regardant le pseudo code C++ que l'on a construit, on peut se dire que cette fonction map en mémoire le binaire (il se map lui même), pour qu'ensuite `sub_140013EB0` s'occupe de repérer et neutraliser des éléments à perturber (comme nos breakpoints). On pourra éventuellement analyser le fonctionnement de cette dernière fonction qui s'occupe de l'anti-debug.

3.3 Récupération de l'input

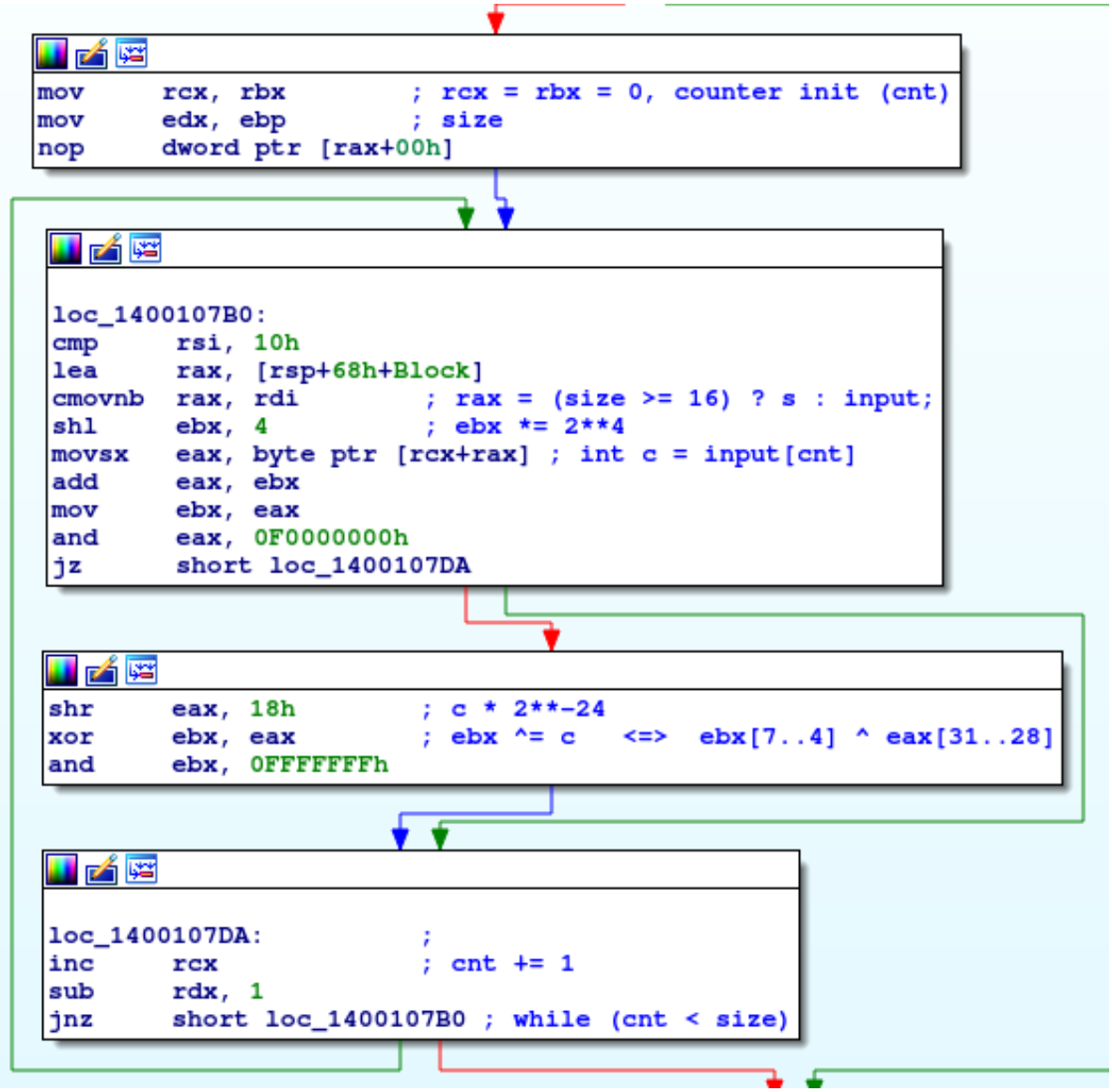
Il convient maintenant de vérifier si notre entrée n'est pas modifié par la fonction qui la récupère (`sub_140011430`). Pour cela, on met un breakpoint à l'instruction suivant l'appel. On saisit la chaine 'AAAAAA' et on voit qu'après l'exécution de la fonction, la zone mémoire dans la stack contenant notre input n'a pas été modifié.

3.4 Vérification de l'input

On procède maintenant à l'analyse de la fonction `sub_1400106d0`. Celle ci prend en paramètre un pointeur vers une chaine de caractère c++, ou une structure de donnée représentant une chaine. En regardant plus en détail l'implémentation des strings en c++, on voit qu'on peut retrouver différents layout mémoire. On voit que pour le layout `long`, la chaine de caractère est stockée dans la heap et le structure de donnée `string` contient le pointeur vers cette zone mémoire, la taille et la capacité max (8+8+8=24 octets). Dans le cas du layout `short`, la structure de donnée `string` contient directement les données et un octet de la structure de donnée sert à stocker la taille. On voit aussi dans l'implémentation que l'ordre des champs de ces structures de données peuvent être inversés. Cela va permettre de comprendre pourquoi l'on accède par la suite à des champs depuis le pointeur de chaine passé en paramètre.

Pour ce qui est de l'architecture de la fonction, on a une première phase d'initialisation où l'on vérifie la taille de notre chaine (instruction `Exam_CSI.exe+0x705`), puis on effectue une copie de notre chaine de différentes façons en fonction de la taille (cela est surement lié à des optimisations).
On voit ensuite la boucle qui parcourt notre chaine et crée un entier sur 4 octets qui est utilisé dans la derniere partie pour la comparer à une valeur en dur avant de définir la valeur globale de retour.
Ce sont surtout ces 2 dernières parties qui nous intéressent (la première n'ayant pas de réel impact sur déroulement de la boucle).

Regardons le code de cette boucle.

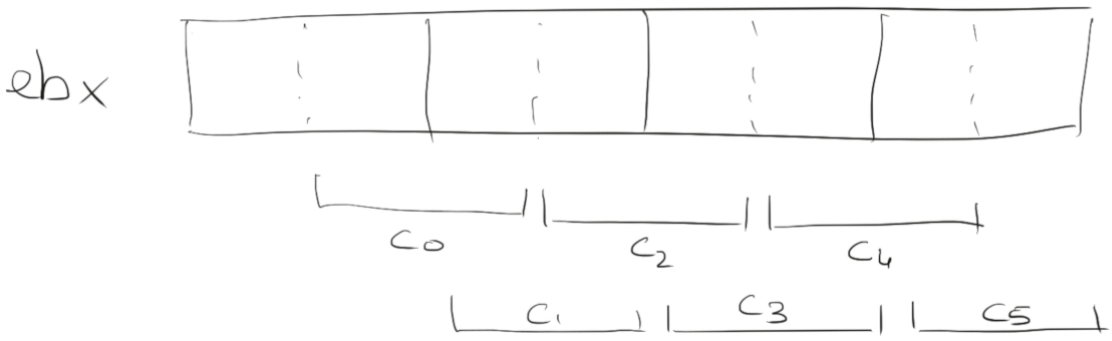


On voit que l'on initialise un index `cnt` à 0, et `ebx` a été initialisé à 0 précedemment (c'est le mdp courant qu'on construit). Lors du corps de la boucle, on décale `ebx` de 4 bits puis on additionne ebx et le caractère courant pointé par `ptr_input[cnt]`. On met ce résultat dans le registre `ebx` et si le block de 4 bits de poids fort (à gauche) de celui ci est différent de 0, on effectue une action supplémentaire. Plus précisemment, on récupère ce bloc de 4 bits que l'on vient xor au 2nd bloc de 4 bits de poids faible (en partant de la droite) du nouveau `ebx` avant de mettre le bloc de poid fort à 0. On passe ensuite au tour suivant.

On peut traduire cette fonction en code C pour nous faciliter la compréhension.

```
1  int
2  check(char *input)
3  {
4      int pass = 0;
5      for (int i=0; i<strlen(input); i++)
6      {
7          pass <<= 4;
8          int c = input[i] + pass;
9          pass = c;
10         c &= 0xF0000000;
11         if (c != 0)
12         {
13             c >>= 24;
14             pass ^= c;
15             pass &= 0x0FFFFFFF;
16         }
17     }
18
19     if (pass == 0x0A50B8B0)
20         return 1;
21     return 0;
22 }
```

A première vue, si l'on entre une série de au plus 6 caractères ASCII on ne passe pas dans la condition `if (c != 0)`, comme on peut le voir sur le schéma suivant le dernier bloc de 4 bits est forcément à 0.



En revanche, la partie gauche de C0 serait Ah = 10d = 1010b qui n'est pas un caractère ASCII valide (le bit de signe est à 1), ce qui peut nous indiquer que la chaine valide fait au moins 7 caractères. Si on définit Left(C) et Right(C) comme les demi octets de poids fort et faible d'un caractère courant, alors on remarque aussi que pour les caractètres autre que le premier et le dernier, on additionne Left(Ci) avec Right(Ci-1), et Right(Ci) avec Left(Ci+1). Pour le dernier caractère, Right(Cn-1) est laissé intacte dans ebx, de plus au vue de la valeur final à atteindre, Right(Cn-1) = 0 ce qui implique

que Cn-1 doit être pris dans un ensemble bien délimité de caractères ASCII valides, à savoir

$$Left(C_{n-1}) \in \{0111, 0110, 0011, 0101, 0100, 0010, 0001\} \rightarrow C_{n-1} \in \{'p','`','0',' P',' @',' '\}$$

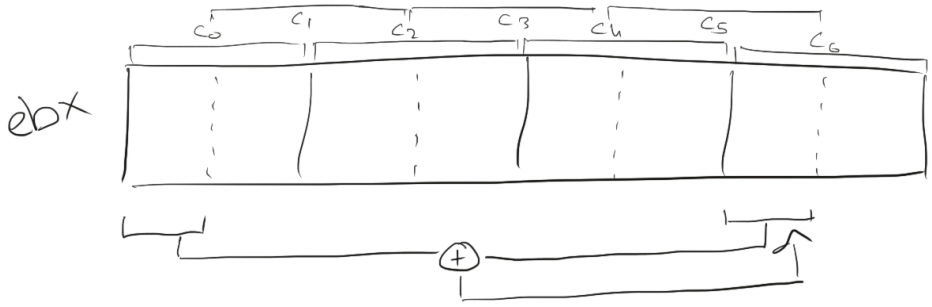
(0x10 ne sera surement pas accepté si la fonction qui lit l'entrée interprete ou s'arrete sur les caractères spéciaux)

On voit bien que l'on peut grâce à ces additions construires des valeurs qu'on ne pourrait pas représenter avec des caractères ASCII directement. Par exemple dans la configuration du schéma précédent, imaginons qu'on veuille avoir la valeur 0x99 dans le 2eme octet en partant de la gauche. On voit que si l'on met Right(C1) à 0, alors C2 ne peut pas être un caractère ASCII et permettre de valider la valeur requise. En revanche, Right(C1) n'a pas de contrainte comparé à Left(C2), on peut donc se débrouiller pour passer un octet ressemblant presque à la valeur finale, et dont l'addition avec le bloc précédent va donner la bonne valeur.
Si l'on essaye de formaliser ce schéma pour les caractères du milieu, on a :

$$Left(C_i) = \text{pattern}, Right(C_i) = \text{ajustement bits } Left(C_{i+1})$$

On a le même schéma pour Cn-1 = C6 à la différence que Right(C6) = 0.

On remarque néanmoins qu'on doit au moins passer une fois dans l'étape du xor lors du 7ème tour de boucle. Cela amène à l'étape suivante :



Pour C0, on a :

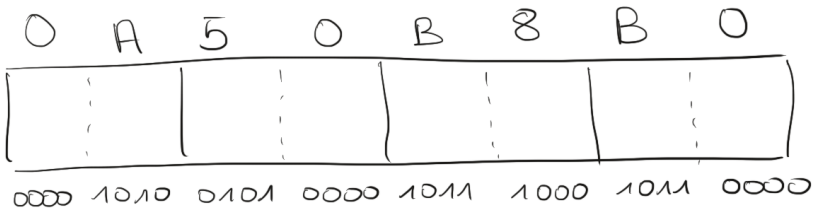
$$Left(C_0) = \text{ajustement bits } Left(C_6), Right(C_0) = \text{ajustement bits } Left(C_1)$$

On peut donc créer des caractères qui en repectant ce schéma vont permettre de construire la valeur `0x0A50B8B0`. Pour récapituler, on doit satisfaire les équations suivante (on peut en fait même utiliser les retenues des opérations suivantes, par exemple pour créer la valeur 0 en position 4).

$$A = Right(C_0) + Left(C_1)$$
$$5 = Right(C_1) + Left(C_2)$$
$$0 = Right(C_2) + Left(C_3)$$
$$B = Right(C_3) + Left(C_4)$$
$$8 = Right(C_4) + Left(C_5)$$

$$B = (Right(C_5) + Left(C_6)) \oplus Left(C_0)$$
$$0 = Right(C_6)$$

Plus particulièrement, il faut que chaque caractère construit soit un ASCII valide. On trouve ainsi le code **"D‘LH4Hp"** qui permet de valider le challenge (voir croquis page suivante).



$m = (c_0, c_1, \cdots, c_6)$ $c_6 \in \{$' ', 'P', '@', '`', ' ', '0'$\}$

$c_0 = 0100\ 0100$ / A

$c_1 = 0110\ 0000$ / 5

$c_2 = 0100\ 1100$ / 0

$c_3 = 0100\ 1000$ / B

$c_4 = 0011\ 0100$ / 8

$c_5 = 0100\ 1000$ / B

$c_6 = 0111\ 0000$

3.5 Obfuscation des messages

Etant donné que les messages obfusqués étaient présents en clair dans le binaire, j'ai préféré me concentrer sur d'autres points.

3.6 Anti-Debug