Analyse du binaire « fatrat.exe »

Bonjour !

Dans cette analyse du jour, nous allons faire une analyse statique ainsi que du reverse-engineering sur un binaire malveillant généré via TheFatRat (module Slow But Powerfull, qui est un vieux module).

Cette analyse sera particulièrement intéressante, car TheFatRat avait la réputation de créer des payloads « fully indetectables », nous allons donc y voir la réalité en 2026.

Pour cette analyse je serai armé de divers outils de threat intelligence comme VirusTotal et Hybrid Analysis, ainsi que de mon Kali Linux, de ses outils pré-intégrés et de Ghidra.

Partie 1 : Les présentations

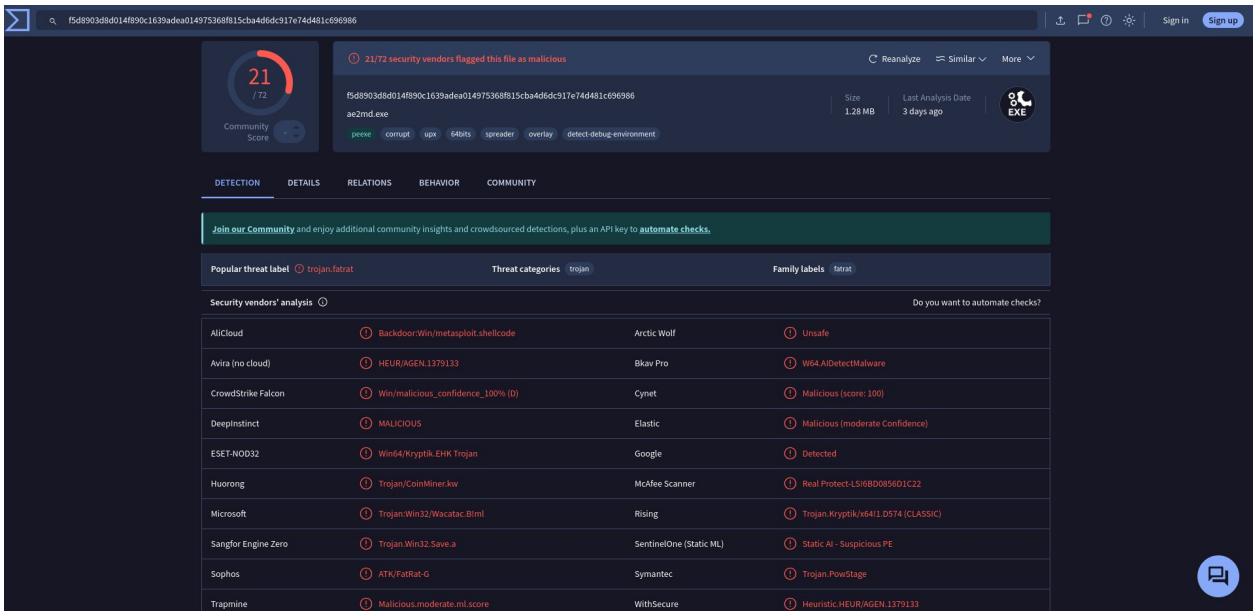
Pour commencer les présentations avec ce programme, nous allons déjà récupérer sa signature numérique :

```
┌──(kali㉿kali)-[~/malwares]
└─$ sha256sum fatrat.exe
f5d8903d8d014f890c1639adea014975368f815cba4d6dc917e74d481c696986  fatrat.exe
```

Puis nous allons confirmer le type de fichier :

```
┌──(kali㉿kali)-[~/malwares]
└─$ file fatrat.exe
fatrat.exe: PE32+ executable for MS Windows 5.02 (console), x86-64, 3 sections
```

Maintenant que son hash est récupéré, nous allons faire un tour sur VirusTotal et chercher par son hash, car j'ai déjà analysé le binaire il y a 3 jours et je suis curieux de savoir si quelque chose à bougé :

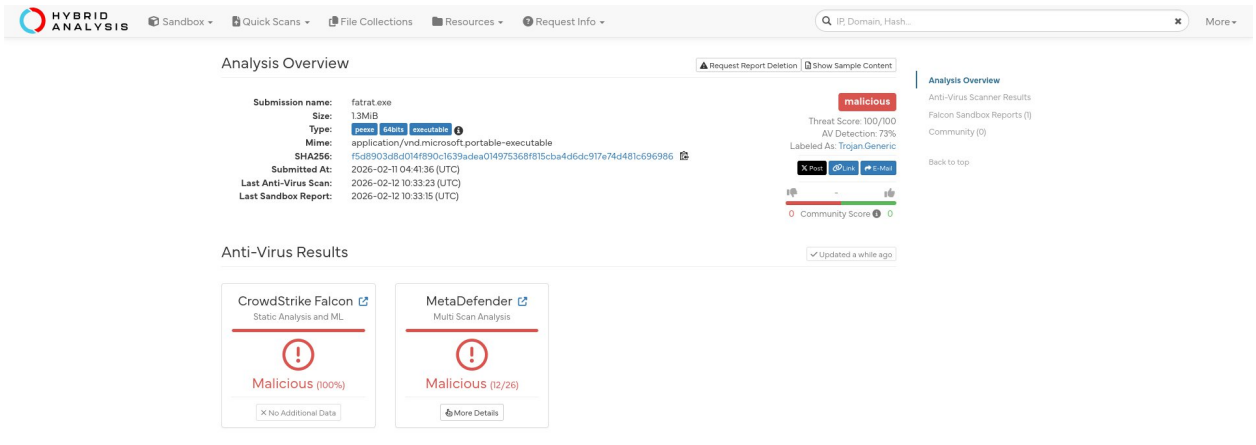


Url du report : https://www.virustotal.com/gui/file/f5d8903d8d014f890c1639adea014975368f815cba4d6dc917e74d481c696986/detection

Nous voyons dans cette capture d'écran, que le binaire est détecté seulement par 21 antivirus sur 72, ce qui est un mauvais score pour un payload connu et rôdé, et surtout avec les moyens actuels.

Cela indique que les détections de notre époque sont encore largement améliorables.

Afin de pousser le test avec des outils automatisés plus loin, j'ai testé le binaire sur HybridAnalysis, et voici les résultats :



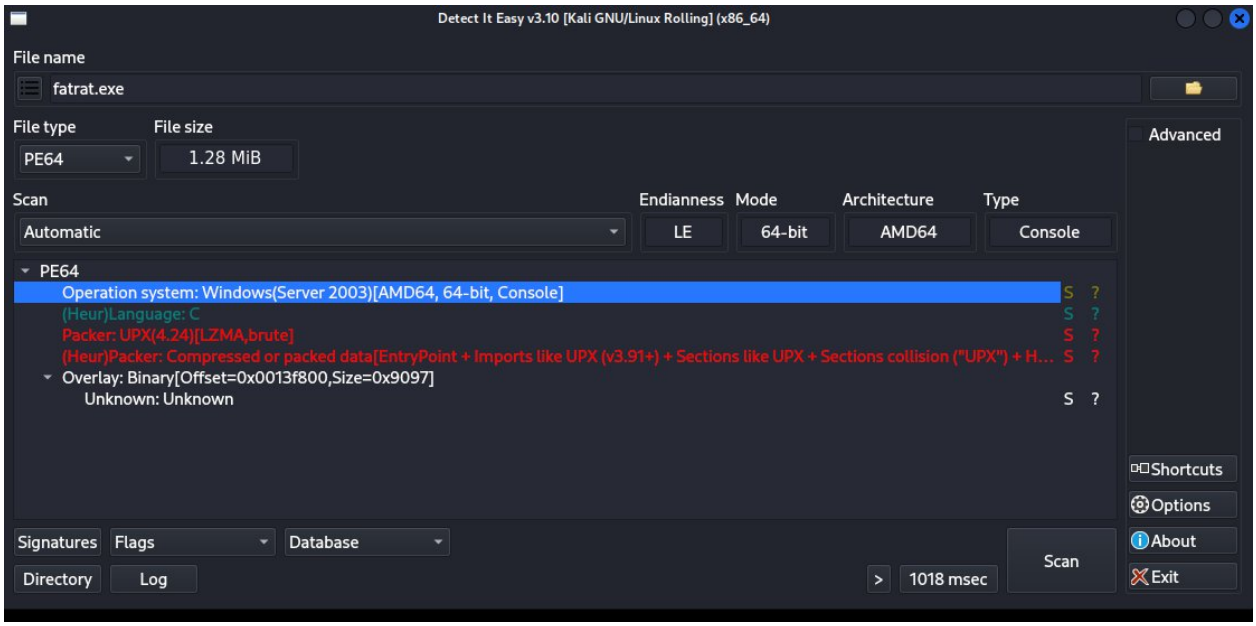
Url du report : https://hybrid-analysis.com/sample/f5d8903d8d014f890c1639adea014975368f815cba4d6dc917e74d481c696986

Le résultat de l'analyse dynamique proposée par HybridAnalysis est déjà bien plus rassurant que l'analyse performée avec VirusTotal !

Je rajouterai à cela que quand j'ai tenté d'exécuter le binaire sur ma machine test, il à été bloqué instantanément par MDE, mais MDE n'arrivait pas à supprimer le fichier, il était bloqué à chaque tentative de lancement, mais MDE ne pouvais pas modifier son état sur le disque, je rajouterai aussi que le téléchargement depuis le web (serveur local de mon kali) est passé sans problèmes, ce qui veut dire que le payload bypass les protections de navigateurs, et l'analyse statique classique.

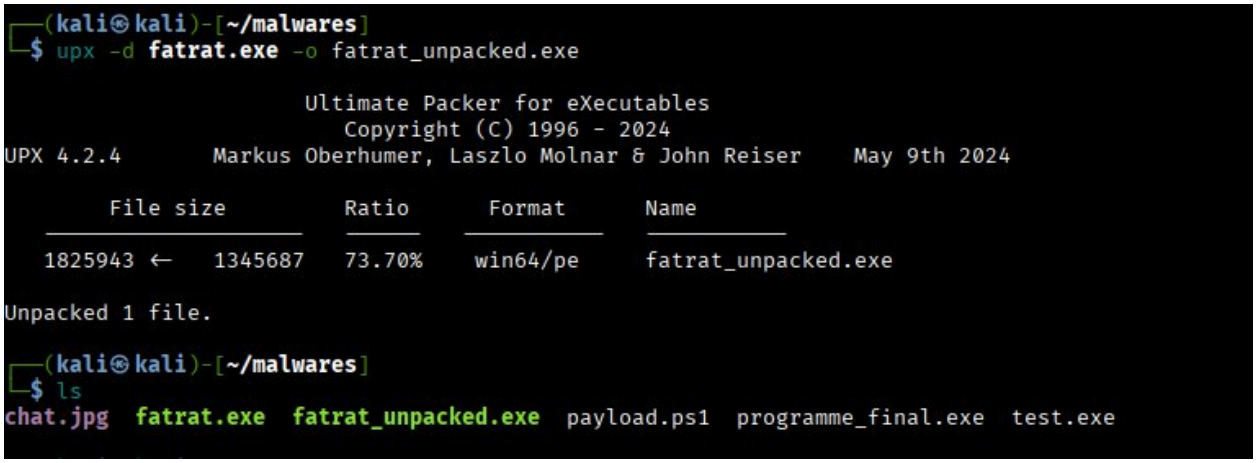
Partie 2 : Analyse statique

Pour commencer notre analyse statique, nous allons détecter le packer utilisé pour ce payload, nous allons le faire via la commande suivante : « die fatrat.exe »



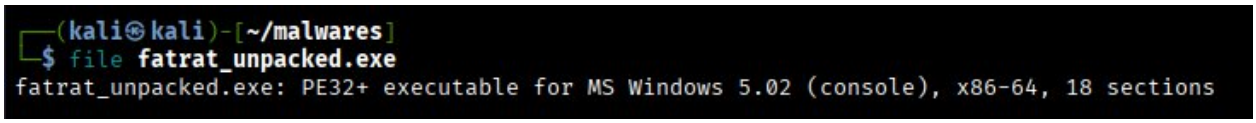
Le binaire étant packé avec UPX, nous ne pouvons pas faire d'analyse statique dans ces conditions, il faut donc dé-packer le binaire.

Maintenant, j'unpack le malware

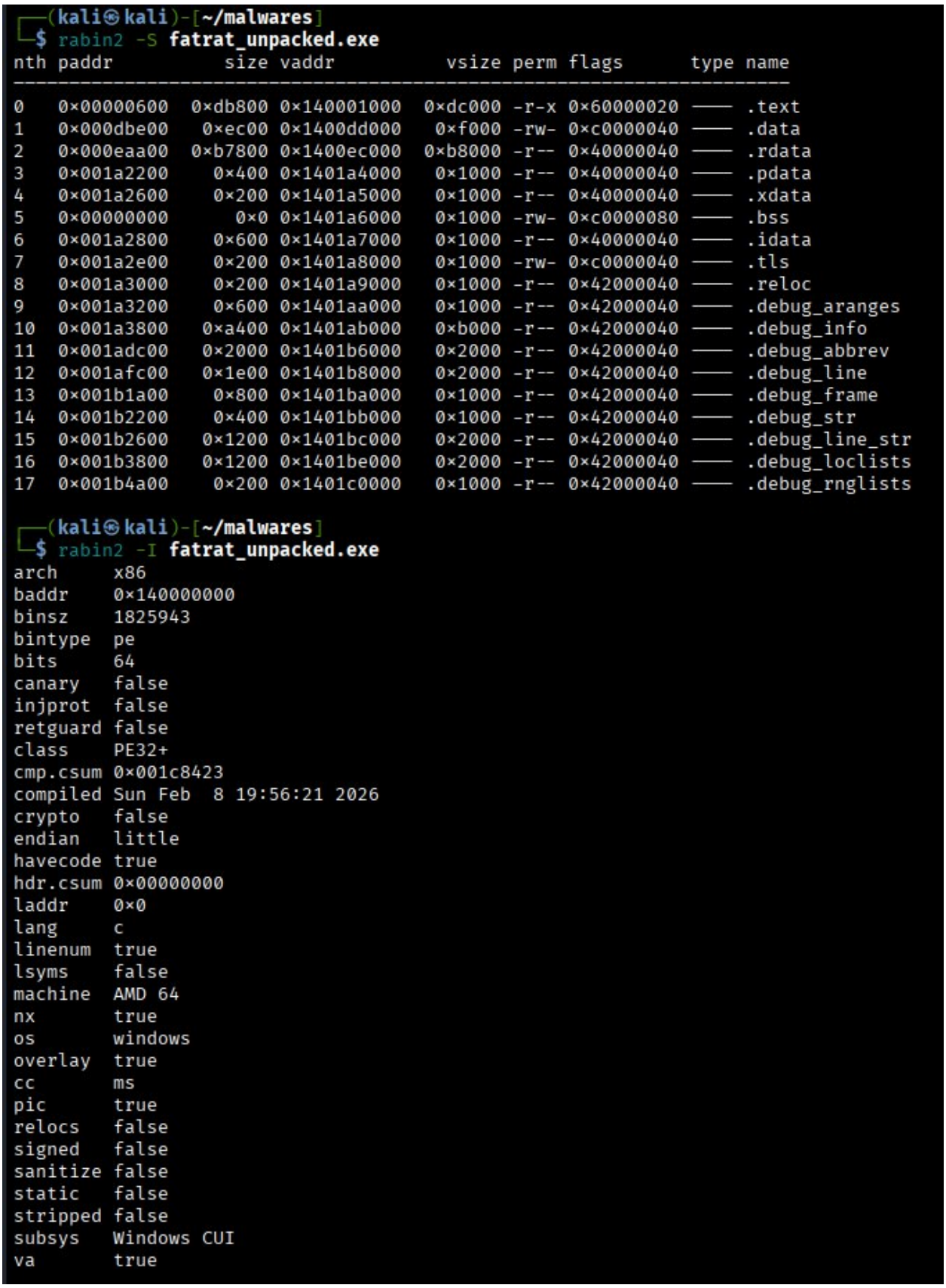


La sortie de la commande « ls » nous prouve que le binaire est unpacké sous le nom de « fatrat_unpacked.exe ».

Il est temps de prendre de nouvelles informations sur le fichier unpacké !



Cette sortie nous prouve que le fichier à été unpacké avec succès.



```
┌──(kali㉿kali)-[~/malwares]
└─$ rabin2 -S fatrat_unpacked.exe
nth paddr          size vaddr          vsize perm flags      type name
-------------------------------------------------------------------------
0   0×00000600  0×db800 0×140001000  0×dc000 -r-x 0×60000020 ——— .text
1   0×000dbe00   0×ec00 0×1400dd000   0×f000 -rw- 0×c0000040 ——— .data
2   0×000eaa00  0×b7800 0×1400ec000  0×b8000 -r-- 0×40000040 ——— .rdata
3   0×001a2200    0×400 0×1401a4000   0×1000 -r-- 0×40000040 ——— .pdata
4   0×001a2600    0×200 0×1401a5000   0×1000 -r-- 0×40000040 ——— .xdata
5   0×00000000      0×0 0×1401a6000   0×1000 -rw- 0×c0000080 ——— .bss
6   0×001a2800    0×600 0×1401a7000   0×1000 -r-- 0×40000040 ——— .idata
7   0×001a2e00    0×200 0×1401a8000   0×1000 -rw- 0×c0000040 ——— .tls
8   0×001a3000    0×200 0×1401a9000   0×1000 -r-- 0×42000040 ——— .reloc
9   0×001a3200    0×600 0×1401aa000   0×1000 -r-- 0×42000040 ——— .debug_aranges
10  0×001a3800   0×a400 0×1401ab000   0×b000 -r-- 0×42000040 ——— .debug_info
11  0×001adc00   0×2000 0×1401b6000   0×2000 -r-- 0×42000040 ——— .debug_abbrev
12  0×001afc00   0×1e00 0×1401b8000   0×2000 -r-- 0×42000040 ——— .debug_line
13  0×001b1a00    0×800 0×1401ba000   0×1000 -r-- 0×42000040 ——— .debug_frame
14  0×001b2200    0×400 0×1401bb000   0×1000 -r-- 0×42000040 ——— .debug_str
15  0×001b2600   0×1200 0×1401bc000   0×2000 -r-- 0×42000040 ——— .debug_line_str
16  0×001b3800   0×1200 0×1401be000   0×2000 -r-- 0×42000040 ——— .debug_loclists
17  0×001b4a00    0×200 0×1401c0000   0×1000 -r-- 0×42000040 ——— .debug_rnglists

┌──(kali㉿kali)-[~/malwares]
└─$ rabin2 -I fatrat_unpacked.exe
arch     x86
baddr    0×140000000
binsz    1825943
bintype  pe
bits     64
canary   false
injprot  false
retguard false
class    PE32+
cmp.csum 0×001c8423
compiled Sun Feb  8 19:56:21 2026
crypto   false
endian   little
havecode true
hdr.csum 0×00000000
laddr    0×0
lang     c
linenum  true
lsyms    false
machine  AMD 64
nx       true
os       windows
overlay  true
cc       ms
pic      true
relocs   false
signed   false
sanitize false
static   false
stripped false
subsys   Windows CUI
va       true
```

L'analyse structurelle met en évidence un exécutable Windows 64 bits généré automatiquement, non signé et non optimisé pour l'évasion. L'absence de stripping et la présence de sections de debug traduisent un manque de sophistication du payload. La présence d'un overlay suggère l'intégration de données embarquées, possiblement liées à la charge malveillante principale.

Passons à l'analyse des imports du binaire :

```
┌──(kali㉿kali)-[~/malwares]
└─$ rabin2 -i fatrat_unpacked.exe
nth vaddr       bind type lib          name
-----------------------------------------
1   0x1401a7158 NONE FUNC KERNEL32.DLL DeleteCriticalSection
2   0x1401a7160 NONE FUNC KERNEL32.DLL EnterCriticalSection
3   0x1401a7168 NONE FUNC KERNEL32.DLL GetLastError
4   0x1401a7170 NONE FUNC KERNEL32.DLL InitializeCriticalSection
5   0x1401a7178 NONE FUNC KERNEL32.DLL LeaveCriticalSection
6   0x1401a7180 NONE FUNC KERNEL32.DLL SetUnhandledExceptionFilter
7   0x1401a7188 NONE FUNC KERNEL32.DLL Sleep
8   0x1401a7190 NONE FUNC KERNEL32.DLL TlsGetValue
9   0x1401a7198 NONE FUNC KERNEL32.DLL VirtualProtect
10  0x1401a71a0 NONE FUNC KERNEL32.DLL VirtualQuery
1   0x1401a71b0 NONE FUNC msvcrt.dll   __C_specific_handler
2   0x1401a71b8 NONE FUNC msvcrt.dll   __getmainargs
3   0x1401a71c0 NONE FUNC msvcrt.dll   __initenv
4   0x1401a71c8 NONE FUNC msvcrt.dll   __iob_func
5   0x1401a71d0 NONE FUNC msvcrt.dll   __set_app_type
6   0x1401a71d8 NONE FUNC msvcrt.dll   __setusermatherr
7   0x1401a71e0 NONE FUNC msvcrt.dll   _amsg_exit
8   0x1401a71e8 NONE FUNC msvcrt.dll   _cexit
9   0x1401a71f0 NONE FUNC msvcrt.dll   _commode
10  0x1401a71f8 NONE FUNC msvcrt.dll   _fmode
11  0x1401a7200 NONE FUNC msvcrt.dll   _initterm
12  0x1401a7208 NONE FUNC msvcrt.dll   abort
13  0x1401a7210 NONE FUNC msvcrt.dll   atexit
14  0x1401a7218 NONE FUNC msvcrt.dll   calloc
15  0x1401a7220 NONE FUNC msvcrt.dll   exit
16  0x1401a7228 NONE FUNC msvcrt.dll   fprintf
17  0x1401a7230 NONE FUNC msvcrt.dll   free
18  0x1401a7238 NONE FUNC msvcrt.dll   malloc
19  0x1401a7240 NONE FUNC msvcrt.dll   memcpy
20  0x1401a7248 NONE FUNC msvcrt.dll   signal
21  0x1401a7250 NONE FUNC msvcrt.dll   strlen
22  0x1401a7258 NONE FUNC msvcrt.dll   strncmp
23  0x1401a7260 NONE FUNC msvcrt.dll   vfprintf
```

L'analyse des imports du binaire montre que le malware utilise des fonctions de gestion mémoire et de manipulation de chaînes pour charger et exécuter un shellcode en mémoire. L'absence d'API réseau statiques indique que la communication C2 est probablement résolue dynamiquement, ce qui est typique des payloads générés par TheFatRat.

Suite à cela, nous nous attaquons à l'analyse des chaînes de caractères en lien avec le réseau :

```
┌──(kali㉿kali)-[~/malwares]
└─$ strings -n 6 fatrat_unpacked.exe > fatrat_strings.txt


┌──(kali㉿kali)-[~/malwares]
└─$ grep -Ei "http|https|tcp|ip|socket|connect|user-agent|\\.exe|AppData|Temp|Mutex" fatrat_strings.txt

IpY~k|6
J{JIp#
1tVMip
;B.Ip#
iptIn+
;2-z    ip
Ip'xw`
LastBranchToRip
LastBranchFromRip
LastExceptionToRip
LastExceptionFromRip
LastBranchToRip
LastBranchFromRip
LastExceptionToRip
LastExceptionFromRip
_IMAGE_IMPORT_DESCRIPTOR
IMAGE_IMPORT_DESCRIPTOR
PIMAGE_IMPORT_DESCRIPTOR
lDbyVhoLfIP
tFZipmgQhjBW
```

Cette analyse nous démontre que toutes les chaînes de caractère en lien avec le réseau ou des adresses IP sont obfusquées, ce qui est typique pour échapper à la détection. Cette technique indique que le binaire utilise un loader pour charger la charge utile en mémoire.

Nous allons maintenant regarder si l'overlay est présent sur le payload :

```
┌──(kali㉿kali)-[~/malwares]
└─$ rabin2 -I fatrat_unpacked.exe | grep overlay

overlay   true
```

Maintenant que nous en avons la confirmation, analysons les sections du fichier unpacké :

```
┌──(kali㉿kali)-[~/malwares]
└─$ rabin2 -S fatrat_unpacked.exe

nth paddr          size vaddr          vsize perm flags      type name
-------------------------------------------------------------------------
0   0x00000600  0xdb800 0x140001000  0xdc000 -r-x 0x60000020 ——— .text
1   0x000dbe00   0xec00 0x1400dd000   0xf000 -rw- 0xc0000040 ——— .data
2   0x000eaa00  0xb7800 0x1400ec000  0xb8000 -r-- 0x40000040 ——— .rdata
3   0x001a2200    0x400 0x1401a4000   0x1000 -r-- 0x40000040 ——— .pdata
4   0x001a2600    0x200 0x1401a5000   0x1000 -r-- 0x40000040 ——— .xdata
5   0x00000000      0x0 0x1401a6000   0x1000 -rw- 0xc0000080 ——— .bss
6   0x001a2800    0x600 0x1401a7000   0x1000 -r-- 0x40000040 ——— .idata
7   0x001a2e00    0x200 0x1401a8000   0x1000 -rw- 0xc0000040 ——— .tls
8   0x001a3000    0x200 0x1401a9000   0x1000 -r-- 0x42000040 ——— .reloc
9   0x001a3200    0x600 0x1401aa000   0x1000 -r-- 0x42000040 ——— .debug_aranges
10  0x001a3800   0xa400 0x1401ab000   0xb000 -r-- 0x42000040 ——— .debug_info
11  0x001adc00   0x2000 0x1401b6000   0x2000 -r-- 0x42000040 ——— .debug_abbrev
12  0x001afc00   0x1e00 0x1401b8000   0x2000 -r-- 0x42000040 ——— .debug_line
13  0x001b1a00    0x800 0x1401ba000   0x1000 -r-- 0x42000040 ——— .debug_frame
14  0x001b2200    0x400 0x1401bb000   0x1000 -r-- 0x42000040 ——— .debug_str
15  0x001b2600   0x1200 0x1401bc000   0x2000 -r-- 0x42000040 ——— .debug_line_str
16  0x001b3800   0x1200 0x1401be000   0x2000 -r-- 0x42000040 ——— .debug_loclists
17  0x001b4a00    0x200 0x1401c0000   0x1000 -r-- 0x42000040 ——— .debug_rnglists
```

Le binaire contient 18 sections PE, dont de nombreuses sections debug non supprimées.

Après la dernière section, on trouve un overlay d'environ 37KB, probablement utilisé pour stocker le payload encodé.

Voici comment j'ai pu calculer la taille de la dernière section du PE :

« Dernière section : .debug_rnglists

paddr = 0x001b4a00

size = 0x200

Fin officielle PE = 0x001b4a00 + 0x200 = 0x001b4c00 »

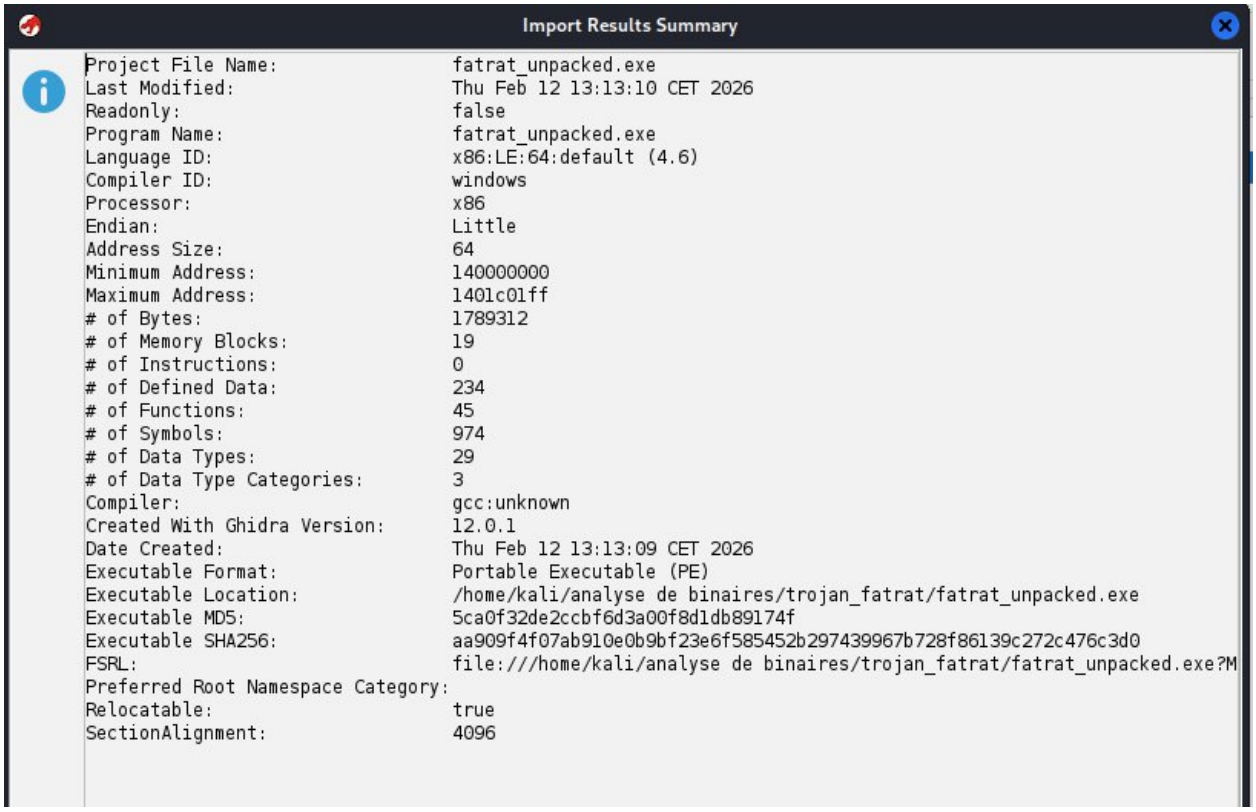
J'ai pu également obtenir la taille totale du fichier avec la commande « ls -l fatrat_unpacked.exe ».

Synthèse du calcul : 1825943 – 1788928 = 37 015 bytes ≈ 37 KB

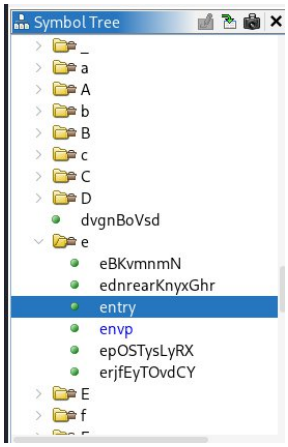
Il est maintenant temps de regarder tout cela plus en détail avec une petite séance de reverse-engineering avec Ghidra !

Partie 3 : Reverse-Engineering

Au lancement du projet dans Ghidra, le programme nous donne un peu plus d'informations sur le payload :



En faisant une recherche dans les fonctions du payload, nous voyons des noms de fonction obfusquées !



Cette obfuscation est volontaire pour nous empêcher de savoir ce que fait le programme.

Après avoir trouvé l'entrypoint, je me dirige vers la fonction « _tmainCRTStartup » car après analyse de ma part sur l'entrypoint, j'ai remarqué que le binaire à été compilé avec MinGW.

La chaîne d'exécution est la suivante :

entry point

↓

mainCRTStartup()

↓

_tmainCRTStartup()

↓

main() ou WinMain()

↓

CODE DU PROGRAMME

Plus d'explications :

_tmainCRTStartup = fonction intermédiaire du runtime

Elle prépare l'environnement avant d'appeler la vraie fonction du programme

Voici une capture d'écran de la fonction « __tmainCRTStartup » :

```
Decompile: __tmainCRTStartup - (fatrat_unpacked.exe)
 1
 2 /* WARNING: Removing unreachable block (ram,0x0001400013aa) */
 3 /* WARNING: Removing unreachable block (ram,0x0001400013b4) */
 4 /* WARNING: Unknown calling convention -- yet parameter storage is locked */
 5
 6 int __tmainCRTStartup(void)
 7
 8 {
 9   void *pvVar1;
10   _TCHAR **pp_Var2;
11   void *pvVar3;
12   int iVar4;
13   int iVar5;
14   int extraout_EAX;
15   int extraout_EAX_00;
16   void *pvVar6;
17   char ***pppcVar7;
18   int *piVar8;
19   _TCHAR **pp_Var9;
20   size_t sVar10;
21   _TCHAR *_Dst;
22   _TCHAR **pp_Var11;
23   longlong lVar12;
24   longlong lVar13;
25   longlong unaff_GS_OFFSET;
26   bool bVar14;
27   _startupinfo startinfo;
28
29                     /* Unresolved local var: void * lock_free@[???]
30                        Unresolved local var: void * fiberid@[???]
31                        Unresolved local var: BOOL nested@[???]
32                        Unresolved local var: int ret@[???] */
33                     /* Unresolved local var: ulonglong ret@[???] */
34   pvVar1 = *(void **)(*(longlong *)(unaff_GS_OFFSET + 0x30) + 8);
35   while( true ) {
36     pvVar6 = (void *)0x0;
37     LOCK();
38     bVar14 = __native_startup_lock == (void *)0x0;
39     pvVar3 = pvVar1;
40     if (!bVar14) {
41       pvVar6 = __native_startup_lock;
42       pvVar3 = __native_startup_lock;
43     }
44     __native_startup_lock = pvVar3;
45     UNLOCK();
46     if (bVar14) {
47       bVar14 = false;
48       goto LAB_14000105c;
49     }
50     if (pvVar1 == pvVar6) break;
51     Sleep(1000);
52   }
```

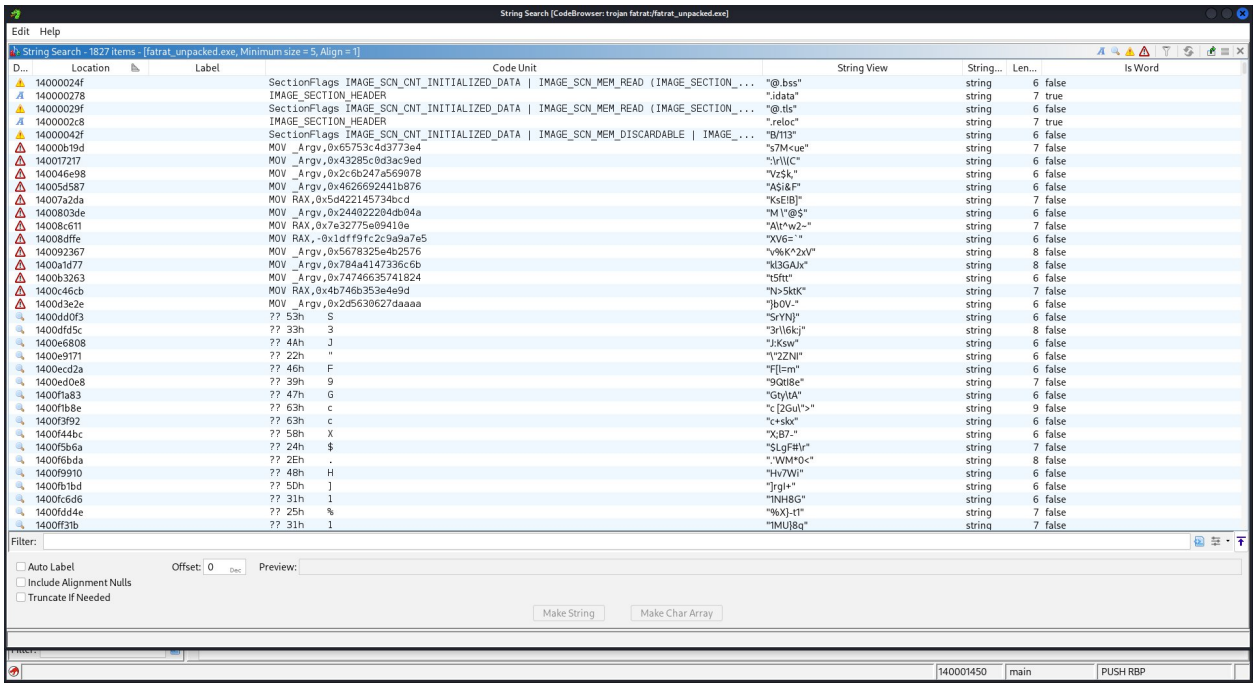
Cette fonction confirme que le malware utilise un flux standard C et qu'il n'a pas d'obfuscation à cette partie là de son code.

Après des recherches, je tombe enfin sur la fonction main, qui ne peut pas s'afficher.

Cela est en général dû à deux causes principales : fonction trop grosse ou trop obfusquée. En vue du binaire analysé, je pense que c'est pour les deux raisons.

Si je demande à Ghidra de me donner plus de détails sur cette fonction, ou de me la séparer en plus petites sections, il en trouve des centaines et fini par crash.

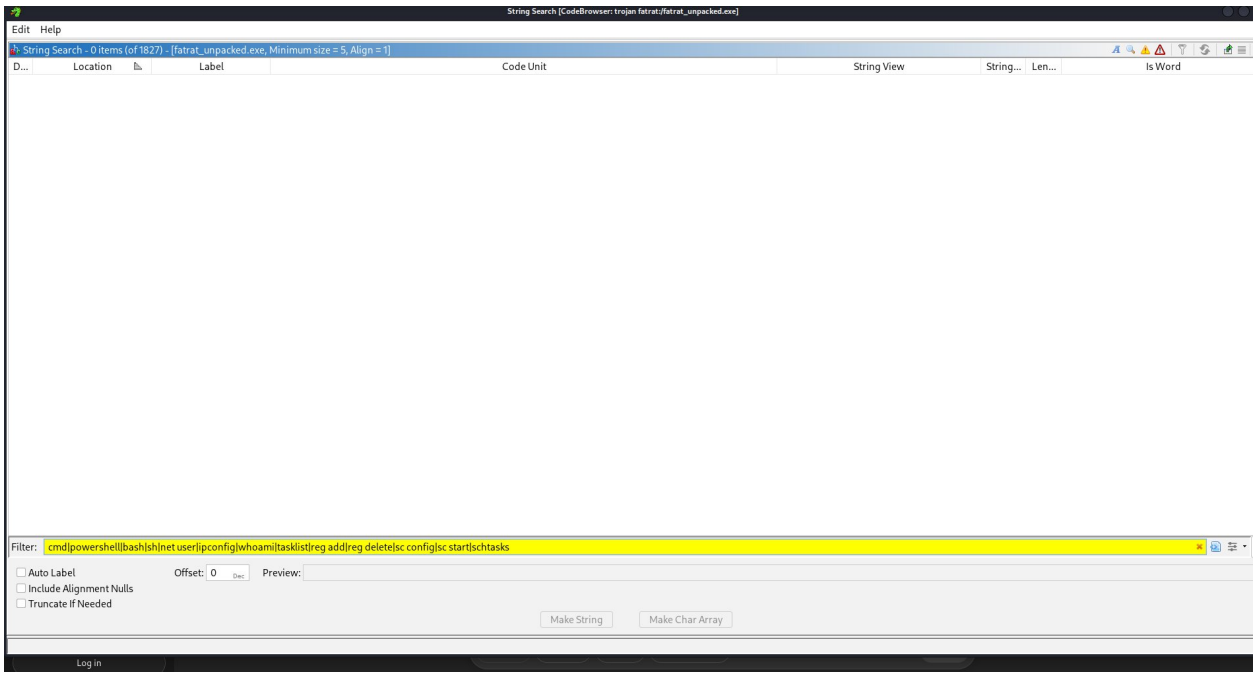
Après avoir passé un certain temps à essayer de prendre des informations sur cette fonction main (sans succès), je passe à la rechercher par strings :



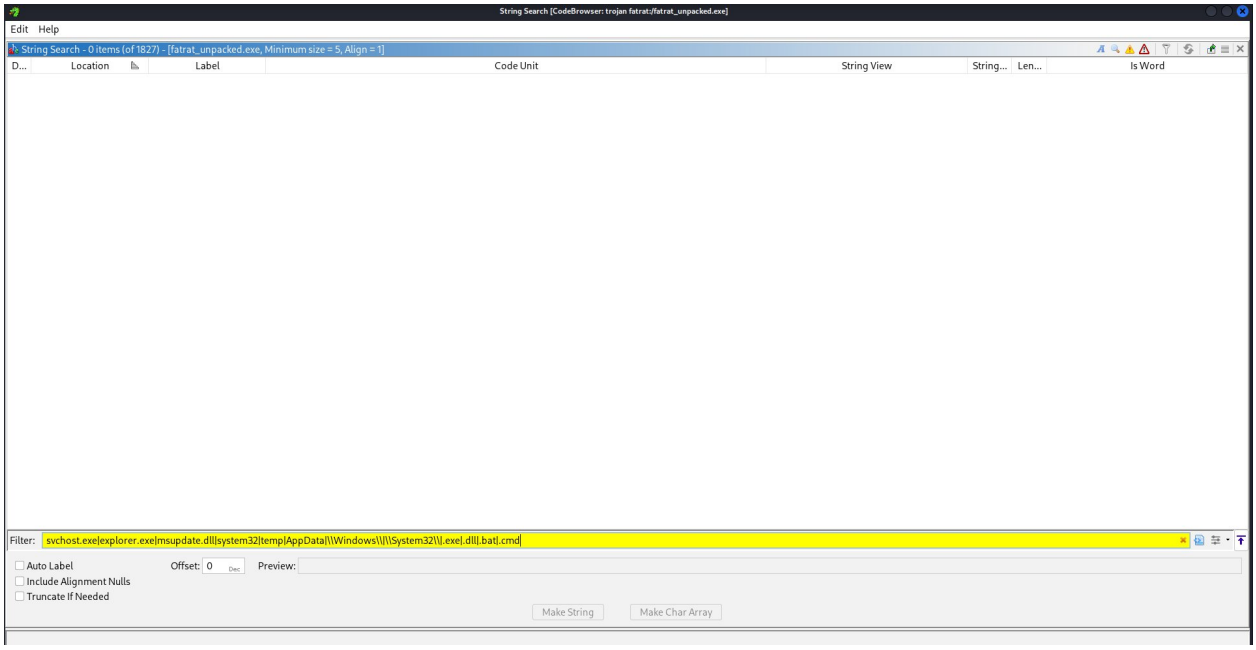
Au premier coup d'oeil je vois des preuves supplémentaire d'obfuscation, ce qui, en vue de ce qui à été démontré précédemment, renforce la malveillance du binaire, avec un ensemble de techniques pour évader la détection : packing, obfuscation, loading du code malveillant en mémoire.

Après avoir parcouru toutes les chaînes textuelles, je me suis rendu compte que toutes les chaînes de caractères que je cherche sont les seules qui sont obfusquées, le reste semble totalement légitime.

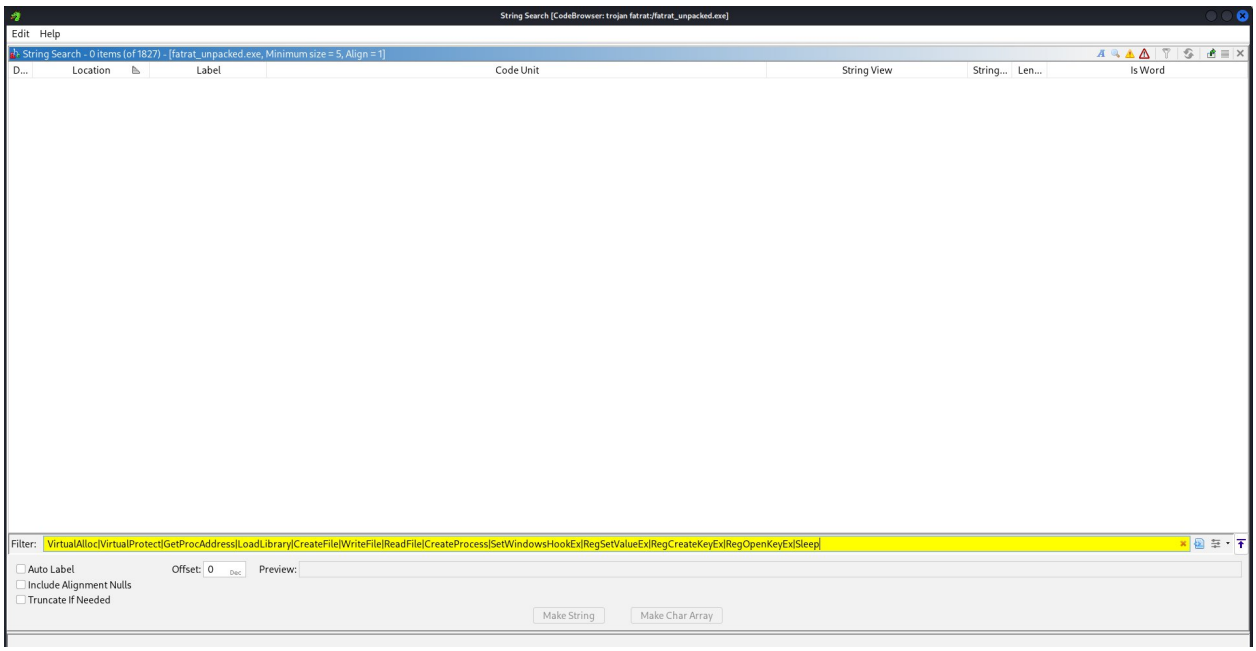
Afin d'optimiser ma recherche et d'aller plus loin, je continue sur ma lancée avec des recherches par mots clés dans les chaînes de caractères !



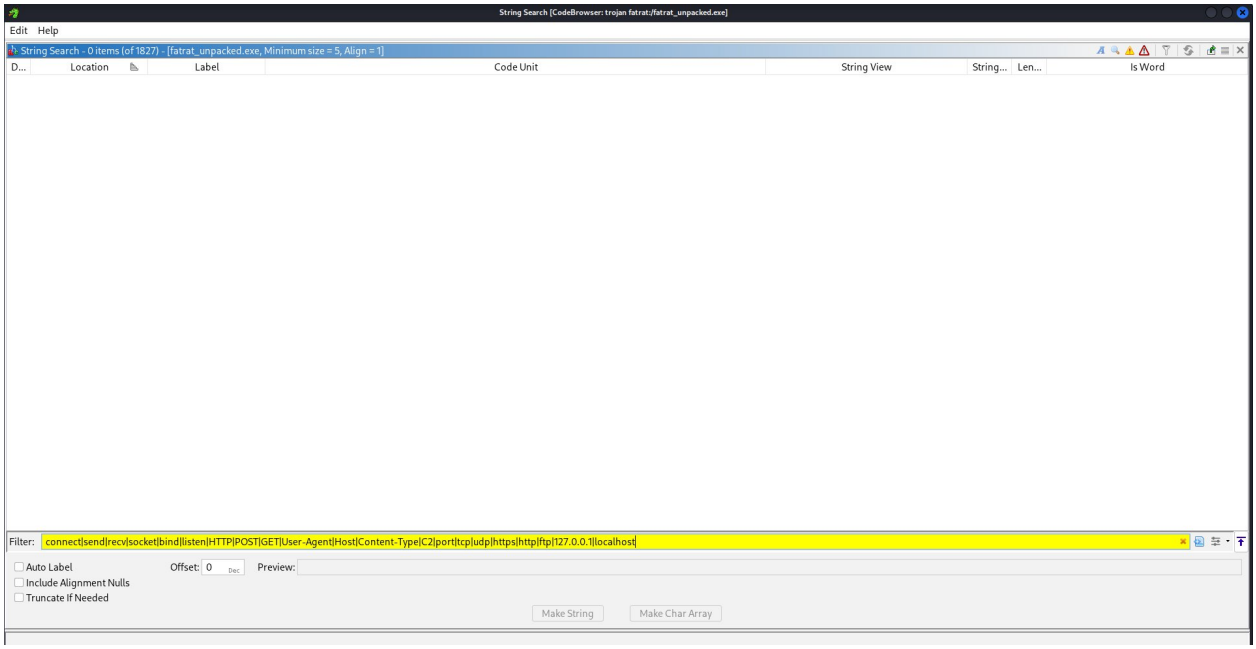
Une recherche de preuves de commandes suspects ou d'utilisation d'un terminal ne renvoie aucun résultat. Pourtant quand j'ai exécuté le binaire sur ma machine test, elle a ouvert un cmd un court instant avant de se faire stopper net par MDE, cela prouve que ce que je cherche est obfusqué.



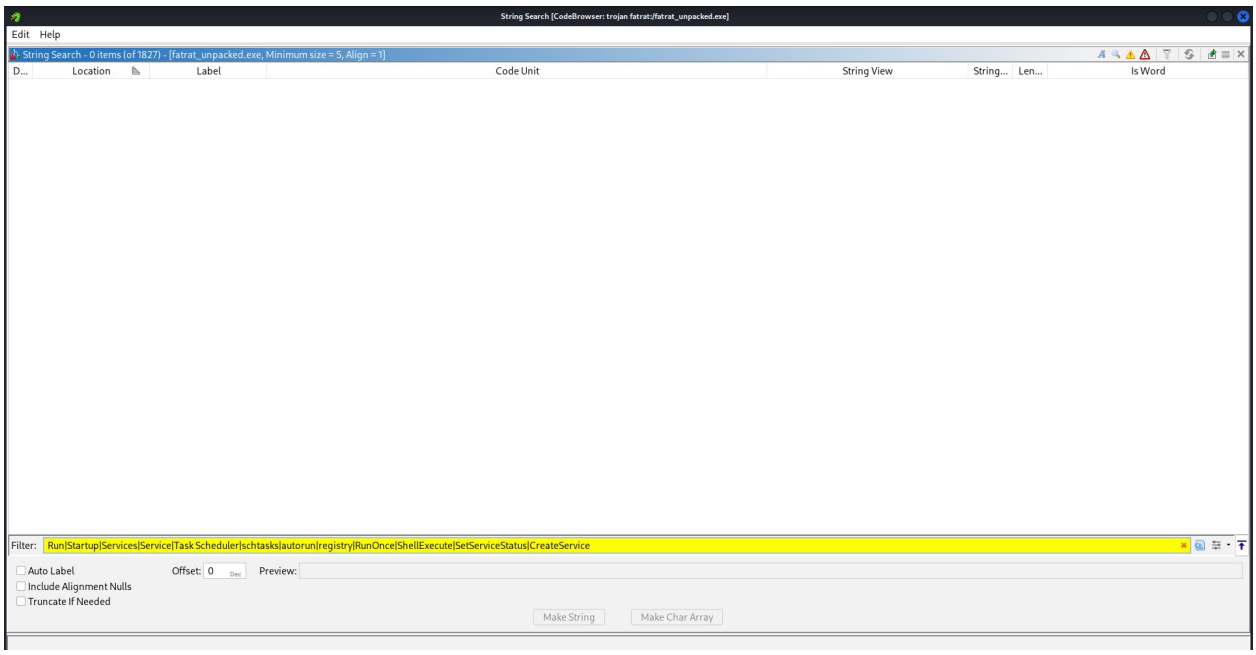
La recherche d'artefacts de persistance de stockage ne donne rien non plus.



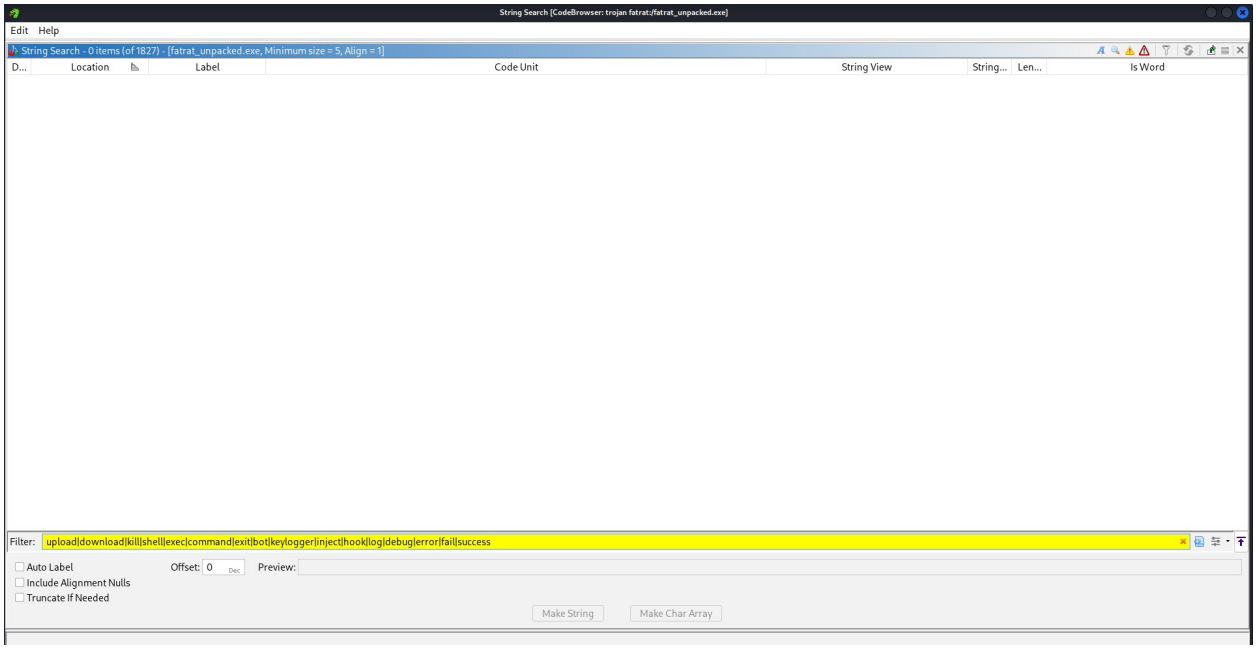
La recherche d'API Windows critiques ne renvoie rien non plus.



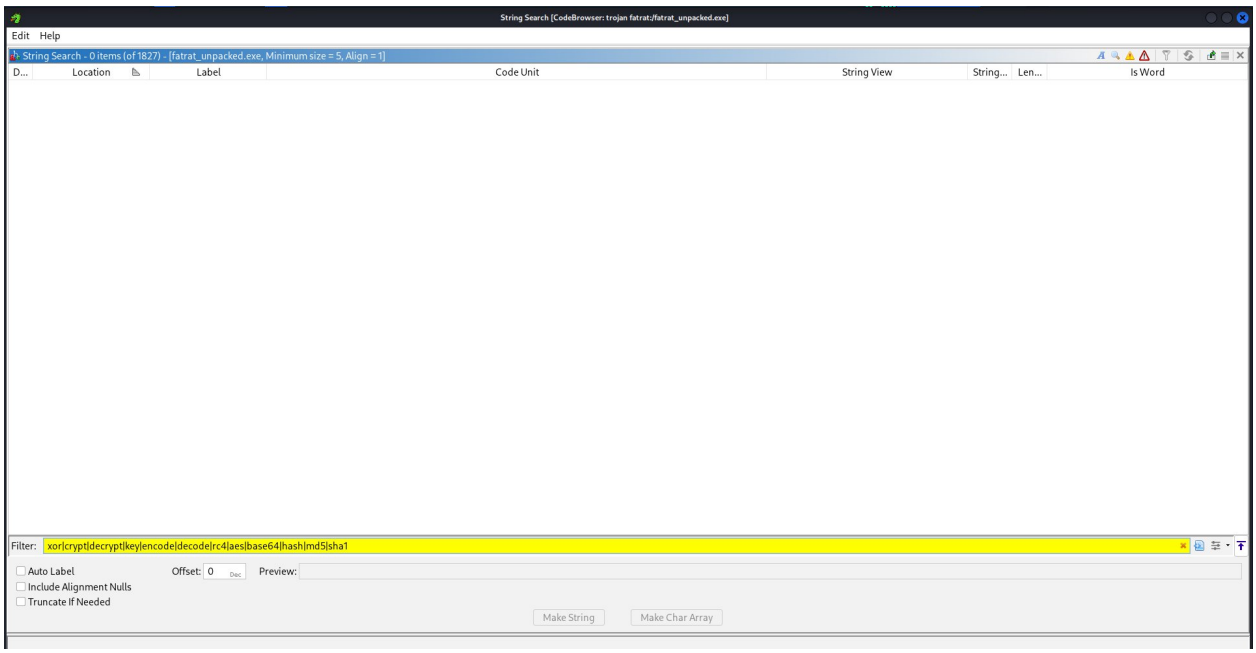
La recherche d'appels réseau ou de connexions vers un serveur de Command and Control ne renvoie rien non plus, et pourtant, pour le test, mon listener était hébergé sur ma machine Kali Linux, donc cette partie est censée exister également dans le binaire.



La recherche de persistance en lien avec des services ne renvoie rien.



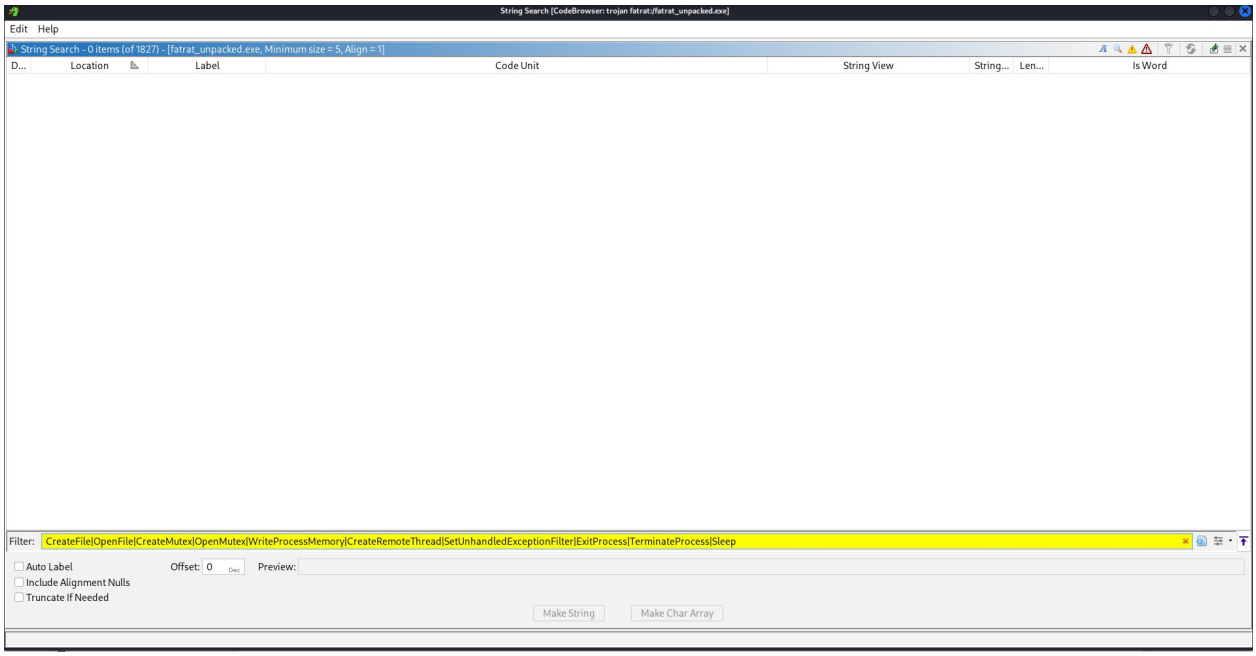
Je n'ai pas non plus de résultats quand je cherche des mots clés typiques de comportement d'un malware.



Une recherche de preuve supplémentaire d'obfuscation ne donne rien non plus, le travail à très bien été fait par le développeur du malware.

Au début de ce rapport, j'ai expliqué que le binaire ne pouvais pas être mis en quarantaine par MDE mais que le code malveillant était bloqué par ce dernier.

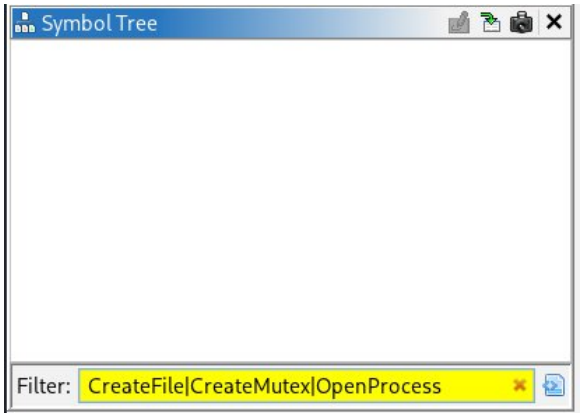
Je vais maintenant essayer de répondre à cette question, à savoir comment cela est possible.



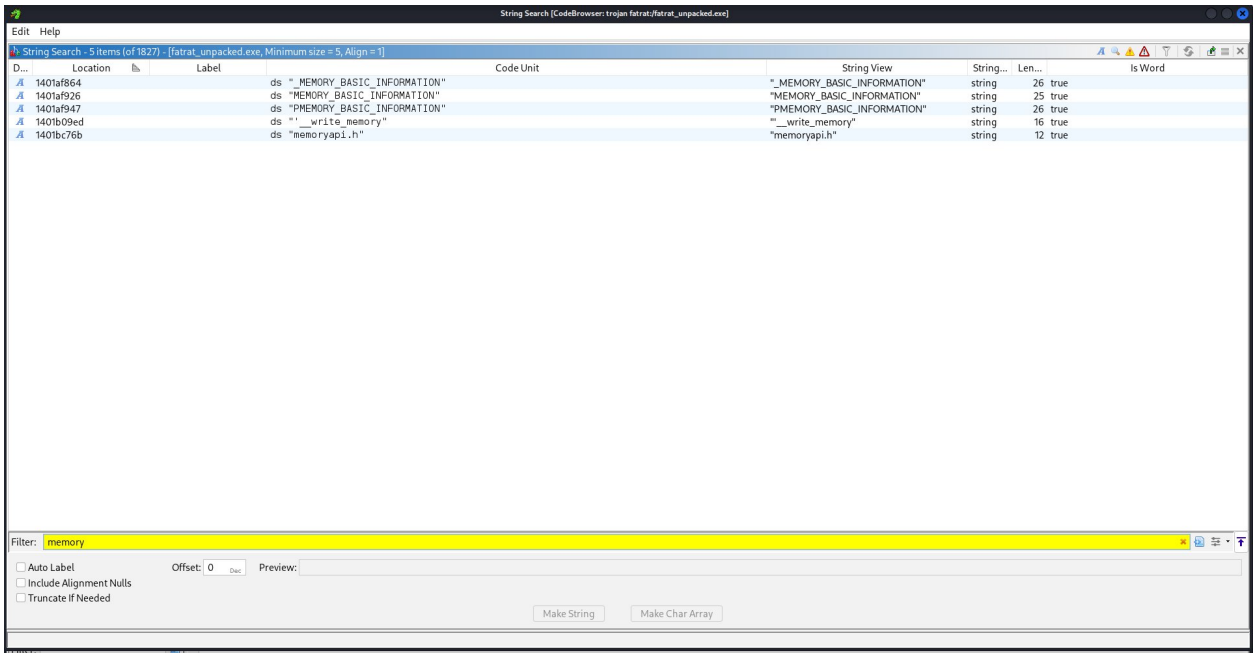
En cherchant ces informations, je ne trouve rien comme habituellement avec ce binaire.

Toutes les informations que je cherche sont décidément obfusquées.

Je fait une recherche dans le symbol tree pour aller plus loin dans ce sens et je n'ai pas de résultat :



Suite à ce résultat je passe à autre chose.



En cherchant le mot « memory » je remarque qu'une fonction du binaire demande à écrire dans la mémoire. Cela confirme ma théorie comme quoi le payload charge son code en mémoire pour bypasser toujours plus de détection !

Histoire d'aller plus loin, j'ai rédigé un script python permettant de tenter ma chance pour décoder toutes ces chaînes de caractères obfusquées !

Le script à bien fait son travail, mais malheureusement cette obfuscation est bien plus poussée que ce que je pensais.

Après ces longues heures de recherche, je décide de m'arrêter là car la fatigue se fait ressentir.

Hypothèse :

Je n'ai pas pu le confirmer à 100% en vue de toute cette obfuscation, mais, je pense que le payload s'auto protège en écriture, afin que les antivirus ne puissent pas modifier son état sur le disque, c'est un signe supplémentaire de robustesse et de persistance.

Ce malware généré par TheFatRat est encore une plaie, même en 2026, et je comprends mieux en vue de toutes les techniques découvertes lors de cette analyse, pourquoi il échappe si bien à la détection par une majorité d'antivirus.

A noter que les plus performants connaissent visiblement bien ces techniques, car il n'est pas évident de détecter ce payload.

Voici ce que l'on peut en conclure :

- Le binaire est très obfusqué, afin de rendre sa détection statique le plus inefficace possible.
- Le binaire charge son code en mémoire, afin d'avoir la meilleure évasion possible à la détection dynamique.
- Le fichier était packé pour être encore plus incompréhensible par les antivirus, et sa taille à été réduite le plus possible aussi par ce processus, ainsi, il avait encore plus de chances de passer inaperçu.

Je vous remercie pour votre lecture, et espère que vous avez apprécié plonger dans cette recherche avec moi !