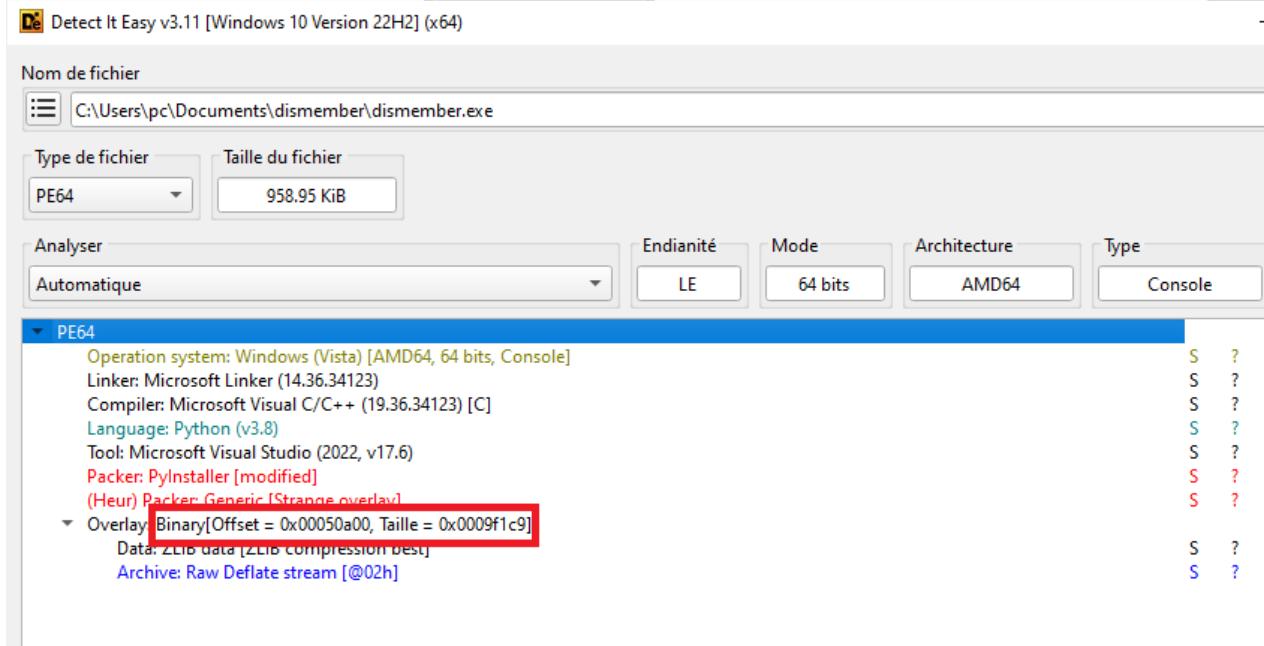


Question 1:

L'Overlay est un **bloc de données** ou un **fichier de données ajouté à la fin d'un exécutable valide, mais qui n'est pas référencé par la structure formelle de ce fichier PE.**

Pour trouver cela , on peut juste utiliser **DetectItEasy** qui trouve facilement l'adresse , ou voir la dernière section du PE headers , .reloc dans notre cas , et calculer l'offset ou il fini , c'est cela où commence l'overlay.



l'offset de l'overlay est : 0x50a00 , on peut confirmer ca en regardant les informations sur PE headers:

#	Name	VirtualSize	VirtualAddress	SizeOfRawData	PointerToRawData	PointerToRelocations	PointerToLinenumbers	NumberOfRelc
F...	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
0	.text	0002b170	00001000	0002b200	00000400	00000000	00000000	
1	.rdata	00012802	0002d000	00012a00	0002b600	00000000	00000000	
2	.data	00005408	00040000	00000e00	0003e000	00000000	00000000	
3	.pdata	000022bc	00046000	00002400	0003ee00	00000000	00000000	
4	.rsrc	0000ef8c	00049000	0000f000	00041200	00000000	00000000	
5	.reloc	00000768	00058000	00000800	00050200	00000000	00000000	

PointerToRawData est l'adresse physique, qui est 50200 , on addition la taille des données , 800 on obtient:

$$\text{Offset Fin PE} = 0x00050200 + 0x00000800 = \textbf{0x00050a00}$$

D'après vous, que contient cet overlay ?

Vu que l'exécutable est compilé avec pyinstaller, cette overlay contient le l'**archive complète** (pyinstaller Archive)nécessaire à l'exécution du programme Python.

Le bytecode python(compressé en Zlib dans notre cas) , les bibliothèques /modules, données et ressources.

étapes execution dans un language compilé comme python :

Lecture du code source , puis compilation en bytecode , et finalement execution par la PVM (python virtual machine).

Pourquoi ca ne marche pas?

```
PS C:\Users\pc\Documents\dismember> python .\pyinstxtractor.py .\dismember.exe
[+] Processing .\dismember.exe
[!] Error : Missing cookie, unsupported pyinstaller version or not a pyinstaller archive
PS C:\Users\pc\Documents\dismember>
```

en essayant d'extraire l'archive contenant le code source , on remarque que ca ne marche pas.

En analysant le code source de **PYINSXTRACTOR** , on trouve la partie qui contient cette erreur:

```
self.cookiePos = -1

if endPos < len(self.MAGIC):
    print('[!] Error : File is too short or truncated')
    return False

while True:
    startPos = endPos - searchChunkSize if endPos >= searchChunkSize else 0
    chunkSize = endPos - startPos

    if chunkSize < len(self.MAGIC):
        break

    self.fPtr.seek(startPos, os.SEEK_SET)
    data = self.fPtr.read(chunkSize)

    offs = data.rfind(self.MAGIC)

    if offs != -1:
        self.cookiePos = startPos + offs
        break

endPos = startPos + len(self.MAGIC) - 1

if startPos == 0:
    break

if self.cookiePos == -1:
    print('[!] Error : Missing cookie, unsupported pyinstaller version or not a pyinstaller archive')
    return False
```

Explication : PYINSXTRACTOR définit la valeur de CookiePos =-1 au début , puis il parcours le fichier en essayant de trouver le magic Code , s'il le trouve il remplace cookiePos par startPos+offs , sinon il print dans le terminal l'erreur que nous avons recu.

Quel est le magic number:

En analysant pyinstxtractor , on remarque dans la ligne 108 le nombre MAGIC qu'il identifie pour commencer l'extraction , s'il ne trouve pas ce nombre dans le binaire , l'extraction va échouer.

```
105     class PyInstArchive:  
106         PYINST20_COOKIE_SIZE = 24          # For pyinstaller 2.0  
107         PYINST21_COOKIE_SIZE = 24 + 64      # For pyinstaller 2.1+  
108         MAGIC = b'MEI\014\013\012\013\016'  # Magic number which identifies pyinstaller
```

MEI\014\013\012\013\016

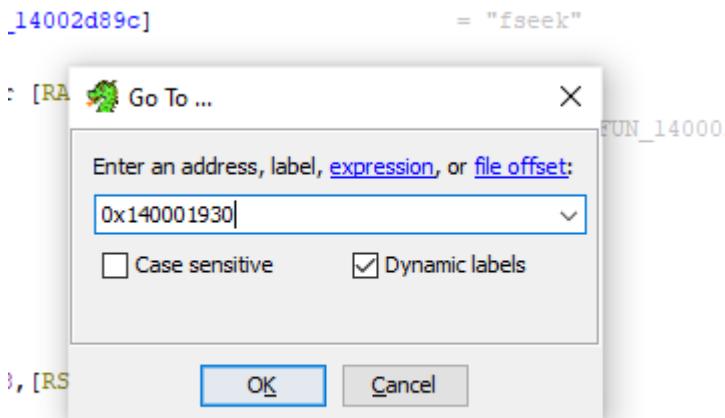
Pourquoi l'auteur du malware pourrait-il avoir décidé de modifier cette valeur?

L'auteur de malware a modifié cette valeur afin de contourner les mécanismes d'analyse statique , en extractant le source code directement avec Pyinsxtractor , il ajoute cette étape d'obfuscation.

Analyser la fonction 0x140001930 de run.exe

Décrivez le but de cette fonction

On ouvre le binaire dans ghidra , et on va vers l'adresse de cette fonction



```
37 _File = (FILE *)FUN_1400039d0(param_1,"rb");
38 if (_File != (FILE *)0x0) {
39     Magic_byte = 0xe0b0a0b0c49454d;
40     LVar7 = FUN_1400073d0(_File,&Magic_byte,(LARGE_INTEGER)0x8);
41     if (LVar7.QuadPart != 0) {
42         iVar5 = FUN_14000fc2c(_File,LVar7,0);
43         if (iVar5 < 0) {
44             puVar8 = __doserrno();
45             FUN_140002020("fseek",*puVar8,0x14002d960,param_4);
46         }
47     else {
48         pFVar14 = _File;
49         sVar9 = fread(local_78,0x58,1,_File);
50         if (sVar9 == 0) {
51             puVar8 = __doserrno();
52             FUN_140002020("fread",*puVar8,0x14002d988,pFVar14);
53         }
54     else {
55         puVar10 = (undefined *)_calloc_base(1,0x1070);
56         if (puVar10 == (undefined *)0x0) {
57             puVar8 = __doserrno();
58             FUN_140002020("calloc",*puVar8,0x14002d9a0,pFVar14);
59         }
60     }
61 }
```

Explication : *File*: Ouvre le fichier executable en mode *rb*

Lvar7: est une fonction de recherche du magicNumberSi elle réussit, LVar7 reçoit l'offset de début du Footer de l'archive.

si magic number trouvé , fseek déplace le pointeur de lecture du fichier _File directement à offset Lvar7 (début du frotter) puis il lit 88 octets de données à partir de cette adresse , ces 88 octets sont les métadonnées clés de l'archive

puis il corrige l'endianess et charge la table des matières , puis alloue de mémoire et lit cette table directement vers cette mémoire.

À la sortie de cette fonction, le *bootloader* a toutes les informations nécessaires (offsets, tailles, noms, compression) pour commencer à extraire les fichiers Python de l'Overlay et lancer l'exécution.

C'est une grande fonction qui a pour but principal de lire, valider, et charger les métadonnées de l'archive PyInstaller (PKG archive). Elle agit comme le cœur du *bootloader* en préparant les données pour l'extraction du code Python.

On retrouve le magic number (je l'ai renommé en GHIDRA)

```
f (_File != (FILE *)0x0) {
    Magic_byte = 0xe0b0a0b0c49454d;
    LVar7 = find_archive (_File, &Magic_byte, (LARGE_INTEGER)0x8);
    if (LVar7.QuadPart != 0) {
        iVar5 = FUN_14000fc2c( File,LVar7,0);
```

Le *magic number* est modifié (dé-chiffré) dynamiquement pour contourner l'analyse statique.

En ne stockant jamais la valeur de signature (MEI...) en clair, on rend plus difficile pour un analyste de trouver la clé de l'archive sans exécuter et déboguer le code pas à pas. C'est une méthode de **protection du code**.

Analyser maintenant le code de la fonction 0x140001930 de dismember.exe. Qu'est-ce qui change ?

```
File = (FILE *)FUN_1400039d0(param_1,"rb");
f (_File != (FILE *)0x0) {
magicbyte = 0xe0b0a0b0c47544d;
LVar7 = FUN_1400075d0(_file,&magicbyte,(LARGE_INTEGER)0x8);
if (LVar7.QuadPart != 0) {
```

Dans le fichier binaire dismember.exe , la valeur du magicbyte change

offset

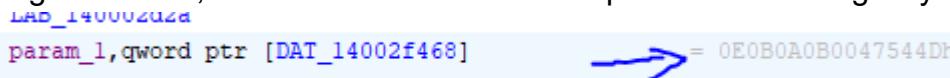
On recherche la valeur HEX dans le binaire ou bien la chaîne MTG , on obtient 2 occurrences , (la deuxième valeur est changé dynamiquement avec 0C au place de 00 , mais cela ne change pas MTG)

0002DA50	54 6B 5F 47 65 74 4E 75 6D 4D 61 69 6E 57 69 6E	Tk_GetNumMainWin
0002DA60	F 6F 77 73 00 00 00 00 4D 54 47 00 0B 0A 0B 0E	dows....MTG.....
0002DA70	00 20 00 00 25 73 25 63 00 00 00 00 00 25 2E 2A 73%s%c....%.*s
0002DA80	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0002DA90	4C 00 4F 00 41 00 44 00 45 00 52 00 3A 00 20 00	L.O.A.D.E.R.:..
0002DAA0	66 00 61 00 69 00 6C 00 65 00 64 00 20 00 74 00	f.a.i.l.e.d. .t.
000EFB60	34 00 7A 50 59 5A 2D 30 30 2E 70 79 7A 00 00 00	4.zPYZ-00.pyz...
000EFB70	F 00 4D 54 47 0C 0B 0A 0B 0E 00 09 F1 C9 00 09 EF	.MTG.....ñÉ..í
000EFB80	E1 00 00 01 90 00 00 01 34 70 79 74 68 6F 6E 33	á.....4python3
000EFB90	38 2E 64 6C 6C 00 00 00 00 00 00 00 00 00 00 00	8.dll.....

modifier dismember.exe pour qu'il fonctionne avec pyinstxtractor et extraire dismember.pyc.

Pour cela , on va utiliser HXD editor , qui est un éditeur de HEX .

On localise le magic number , On doit d'abord trouver l'emplacement du magic byte

 = 0E0B0A0B0047544Dh

d'après ghidra

on possède la valeur brut du HEX telle qu'elle est présente dans le binaire , on cherche cette valeur après l'avoir converti en format plus clair + correction de little-endian:

4d 54 47 00 0b 0a 0b 0e

```

0002DA50 54 6B 5F 47 65 74 4E 75 6D 4D 61 69 6E 57 69 6E Tk_GetNumMainWin
0002DA60 64 6F 77 73 00 00 00 00 4D 54 47 00 0B 0A 0B 0E dows....MTG.....
0002DA70 00 20 00 00 25 73 25 63 00 00 00 00 25 2E 2A 73 . . .%s%c....%.*s

```

On trouve avec succès le magicnumber !

Malheureusement après avoir changé cette valeur , le package ne veut pas s'extraire , ce qui indique que ce n'est pas la bonne valeur

en revenant vers le magic byte dans le PSEUDOCODE GHIDRA , on trouve la bonne valeur du hex qu'on doit changer.

4d 54 47 0C 0b 0a 0b 0e

on modifier la valeur en MEI

```

000EFB70 00 4D 45 49 0C 0B 0A 0B 0E 00 09 F1 C9 00 09 EF .MEI|.....ñÉ..í
000EFB80 E1 00 00 01 90 00 00 01 34 70 79 74 68 6F 6E 33 á.....4python3
000EFB90 38 2E 64 6C 6C 00 00 00 00 00 00 00 00 00 00 00 8.dll.....
000EFBA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000EFBB0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000EFBC0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

On extracte , et ca marche !

```

PS C:\Users\pc\Desktop\dismember> py .\pyinstxtractor.py .\dismember.exe
[+] Processing .\dismember.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.8
[+] Length of package: 651721 bytes
[+] Found 9 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: dismember.pyc
[!] Warning: This script is running in a different Python version than the one used to build the executable.
[!] Please run this script in Python 3.8 to prevent extraction errors during unmarshalling
[!] Skipping pyz extraction
[+] Successfully extracted pyinstaller archive: .\dismember.exe

You can now use a python decompiler on the pyc files within the extracted directory
PS C:\Users\pc\Desktop\dismember>

```

PYZ-00.pyz_extracted	23/10/2025 01:33	Dossier de fichiers	
dismember.pyc	11/11/2025 02:07	Compiled Python ...	13 Ko
pycdas.exe	09/11/2025 22:39	Application	2 837 Ko
pycdc.exe	09/11/2025 22:39	Application	3 519 Ko
pyiboot01_bootstrap.pyc	11/11/2025 02:07	Compiled Python ...	1 Ko
pyimod01_archive.pyc	11/11/2025 02:07	Compiled Python ...	3 Ko
pyimod02_importers.pyc	11/11/2025 02:07	Compiled Python ...	22 Ko
pyimod03_ctypes.pyc	11/11/2025 02:07	Compiled Python ...	4 Ko
pyimod04_pywin32.pyc	11/11/2025 02:07	Compiled Python ...	2 Ko
PYZ-00.pyz	11/11/2025 02:07	Python Zip Appli...	620 Ko
struct.pyc	11/11/2025 02:07	Compiled Python ...	1 Ko
text.txt	09/11/2025 22:52	Document texte	25 Ko

Partie 3:

le code est au format compilé dans dismember.pyc ,
on a utilisé l'outil PYCDC :

Decompyle++

A Python Byte-code Disassembler/Decompiler

Decompile++ aims to translate compiled Python byte-code back into valid and human-readable Python source code. While other projects have achieved this with varied success, Decompile++ is unique in that it seeks to support byte-code from any version of Python.

Decompile++ includes both a byte-code disassembler (pycdas) and a decompiler (pycdc).

As the name implies, Decompile++ is written in C++. If you wish to contribute, please fork us on github at <https://github.com/zrax/pycdc>

Building Decompyle++

```
result = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00'
self.RtlMoveMemory(result, memptr, len(result))
return result

if __name__ == '__main__':
    key = input('Enter your plane : ')
    drm = DRM()
    key_addr = drm.store_key(key.encode('ascii')[0:8] + b'\x00' * (8 - len(key)))
    drm.verify(key_addr)
    key = drm.load_key(key_addr)
    result = []
    for i in range(0, len(output)):
        result.append(output[i] ^ key[i % 8])
    output_decoded = bytarray(result).decode('ascii')
    if not output_decoded.startswith('\n.....'):
        print('\n!!!!!!!!!!!!!! You choose wrong... !!!!!!!!!!!!!!!\n')
    print(output_decoded)
    print('CTRL-C to exit')

    try:
        time.sleep(0.3)
    finally:
        pass
    except KeyboardInterrupt:
        sys.exit()

    continue
```



On sauvegarde tout le code avec :

```
pycdc.exe dismember.pyc > text.txt
```

```
def verify(self, key_addr):
    memptr = self.VirtualAlloc(0, len(self.shellcode), self.MEM_COMMIT, self.PAGE_READWRITE_EXECUTE)
    print(hex(memptr))
    self.RtlMoveMemory(memptr, self.shellcode, len(self.shellcode))
    self.VirtualProtect(memptr, len(self.shellcode), self.PAGE_READ_EXECUTE, 0)
    thread = self.CreateThread(0, 0, memptr, key_addr, 0, 0)
    self.WaitForSingleObject(thread, 0xFFFFFFFF)
```

fonctionement verify:

la fonction verify :

1-alloue une zone mémoire exécutable (RWX) de la taille du shellcode

2-copie le shellcode dans cette zone

3-bascule la zone en RX (c'est VirtualProtect)

4- lance un thread dont l'entrypoint est le shellcode, avec `key_addr` comme paramètre. et finalement , attend la fin du thread.

en résumé , cette fonction injecte et exécute un shellcode natif (x64) en mémoire, en lui passant l'adresse de la clé comme argument.

condition

```
result = []
for i in range(0, len(output)):
    result.append(output[i] ^ key[i % 8])
output_decoded = byt bytearray(result).decode('ascii')
if not output_decoded.startswith('\n.....'):
    print('\n!!!!!!!!!!!!!! You choose wrong... !!!!!!!!!!!!!!!\n')
print(output_decoded)
print('CTRL-C to exit')
```

La condition de succès est donc :`output_decoded` , `XOR(output, key)` **doit commencer par un saut de ligne puis sept points**.

CLE

clé[i] = output[i] XOR target[i]

Les 8 premiers octets de output (tirés du output = b'7\x12\x10\\@\\"LA...' dans le .pyc) sont, en hex :

0x37 0x12 0x10 0x5C 0x40 0x5C 0x4C 0x41

- (ces octets correspondent aux caractères 7, \x12, \x10, \, , @, \, , L, A).

en résumé , On récupère les 8 premiers octets de la variable `output` dans le binaire. Le code compare le texte déchiffré au préfixe attendu "`\n.....`" . Pour obtenir la clé qui produit ce préfixe, il suffit de faire un XOR octet-à-octet entre les 8 premiers octets de `output` et les 8 octets de "`\n.....`" . Le résultat (clé 8-octets) est =`rnrbo` (hex `3d3c3e726e72626f`).

Index (i)	Ci (Hex)	\oplus	Di (Hex)	=	Ki (Hex)	Ki (ASCII)
0	0x37	\oplus	0x0A	=	0x3D	=
1	0x12	\oplus	0x2E	=	0x3C	<
2	0x10	\oplus	0x2E	=	0x3E	>
3	0x5C	\oplus	0x2E	=	0x72	r
4	0x40	\oplus	0x2E	=	0x6E	n
5	0x5C	\oplus	0x2E	=	0x72	r
6	0x4C	\oplus	0x2E	=	0x62	b
7	0x41	\oplus	0x2E	=	0x6F	o

Partie 4

On sauvegarde le shellcode dans un fichier , on peut faire un script python pour cela:

```
py.py > ...
1 shellcode = b'H\x89L$\x08H\x81\xec\xc8\x00\x00\
2
3 with open('shell.bin', 'wb') as f:
4     f.write(shellcode)
```

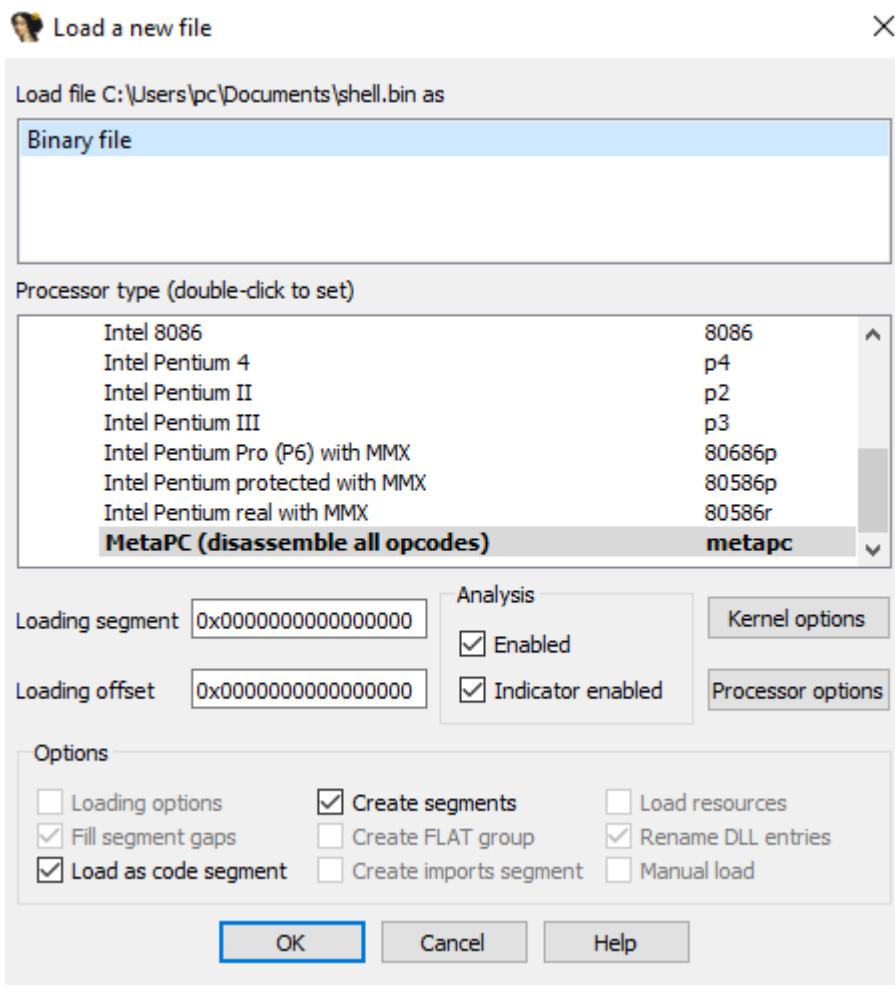
Ou bien , extraire les valeurs HEX du shellcode et créer un nouveau fichier avec HXD editor , et collant les valeurs , c'est la méthode qu'on a utilisé:



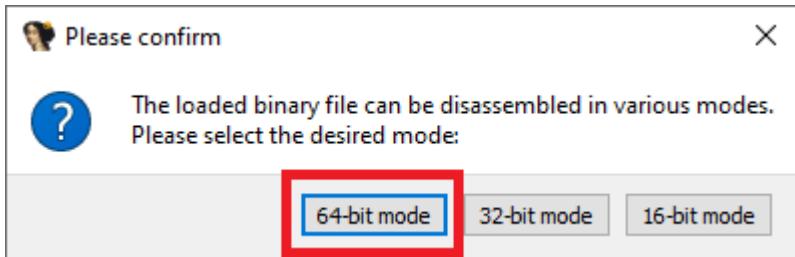
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Texte Décodé
000000000	48	89	4C	24	08	48	81	EC	C8	00	00	00	48	C7	84	24	HéL\$.H.iÈ...HÇ,,\$
000000010	A0	00	00	00	00	00	00	00	48	C7	84	24	A8	00	00	00HÇ,,\$....
000000020	00	00	00	00	48	C7	84	24	B0	00	00	00	00	00	00	00HÇ,,\$°.....
000000030	65	48	8B	04	25	60	00	00	00	48	89	44	24	48	48	8B	eH<.%`...H%D\$HH<
000000040	54	24	48	48	8B	40	18	48	89	54	24	50	48	8B	54	24	T\$HH<@.H%T\$PH< T\$
000000050	50	48	83	C0	20	48	89	54	24	58	48	8B	54	24	58	48	PHfÀ H%T\$XH< T\$XH
000000060	8B	00	48	8B	00	48	8B	00	48	83	E8	10	48	89	54	24	<.H<.H<.Hfè.H%T\$
000000070	60	48	8B	54	24	60	48	8B	40	30	48	89	54	24	28	48	`H< T\$`H< @0H%T\$ (H
000000080	8B	54	24	28	48	89	54	24	68	48	8B	54	24	68	68	63	< T\$ (H%T\$ hH< T\$ hhc
000000090	40	3C	48	8B	54	24	28	48	03	C8	48	8B	C1	48	89	54	@< H< T\$ (H.ÈH< ÁH%T
0000000A0	24	70	B8	08	00	00	00	48	6B	C0	00	48	8B	54	24	70	\$p,...HkÀ.H< T\$p
0000000B0	8B	84	01	88	00	00	00	48	8B	54	24	28	48	03	C8	48	<...^...H< T\$ (H.ÈH
0000000C0	8B	C1	48	89	54	24	30	48	8B	54	24	30	8B	40	1C	48	< ÁH%T\$OH< T\$O< @.H
0000000D0	8B	54	24	28	48	03	C8	48	8B	C1	48	89	84	24	88	00	< T\$ (H.ÈH< ÁH%,,\$^.
0000000E0	00	00	48	8B	54	24	30	8B	40	20	48	8B	54	24	28	48	..H< T\$O< @ H< T\$ (H
0000000F0	03	C8	48	8B	C1	48	89	54	24	78	48	8B	54	24	30	8B	.ÈH< ÁH%T\$ xH< T\$O<
000000100	40	24	48	8B	54	24	28	48	03	C8	48	8B	C1	48	89	84	@\$H< T\$ (H.ÈH< ÁH%,,
000000110	24	80	00	00	00	48	C7	44	24	40	00	00	00	C7	44	\$€...HÇDS@....çD	
000000120	24	24	00	00	00	00	EB	0A	8B	54	24	24	FF	C0	89	54	\$§....ë.< T\$§yÀ%T
000000130	24	24	48	8B	54	24	30	8B	40	18	39	54	24	24	0F	83	\$§H< T\$O< @.9T\$§.f
000000140	95	00	00	00	48	C7	44	24	24	48	8B	54	24	78	8B	04	*...HÇD\$SH< T\$ x< .
000000150	81	48	8B	54	24	28	48	03	C8	48	8B	C1	48	89	54	24	.H< T\$ (H.ÈH< ÁH%T\$
000000160	38	B8	01	00	00	00	48	6B	C0	00	48	8B	54	24	38	0F	8,...HkÀ.H< T\$8.
000000170	BE	04	01	83	F8	47	75	5C	5C	0B	B8	01	00	00	00	48	%..føGu\\.,...,H
000000180	6B	C0	03	48	8B	54	24	38	0F	BE	04	01	83	F8	4D	75	kÀ.H< T\$8.%..føMu
000000190	45	B8	01	00	00	00	48	6B	C0	09	48	8B	54	24	38	0F	E,...HkÀ.H< T\$8.
0000001A0	BE	04	01	83	F8	48	75	2E	48	C7	44	24	24	48	8B	8C	%..føHu.HÇD\$SH< E
0000001B0	24	80	00	00	00	0F	B7	04	41	48	8B	8C	24	88	00	00	\$€....AH< E\$^..
0000001C0	00	8B	04	81	48	8B	54	24	28	48	03	C8	48	8B	C1	48	.<..H< T\$ (H.ÈH< ÁH
0000001D0	89	54	24	40	EB	05	E9	4F	FF	FF	FF	48	83	7C	24	40	WT\$@ë.éOÿÿÿHf \$€
0000001E0	00	74	7A	33	C9	FF	54	24	40	48	89	84	24	90	00	00	.tz3ÉyT\$@H%,,\$...
0000001F0	00	B8	68	F4	02	00	48	8B	8C	24	90	00	00	00	48	03	..hô..H< E\$....H
000000200	C8	48	8B	C1	48	89	84	24	98	00	00	00	C7	44	24	20	ÈH< ÁH%,,\$~...çD\$
000000210	00	00	00	00	EB	0A	8B	54	24	20	FF	C0	89	54	24	20ë.< T\$ yÀ%T\$
000000220	83	7C	24	20	08	7D	36	48	C7	44	24	20	48	C7	44	24	f S ..}6HÇD\$ HçD\$
000000230	28	48	8B	94	24	98	00	00	00	0F	B6	0C	0A	48	8B	94	(H< "\$~....¶..H<"
000000240	24	D0	00	00	00	0F	B6	04	02	33	C1	48	C7	44	24	28	\$Ð....¶..3ÁHÇD\$ (
000000250	48	8B	94	24	D0	00	00	00	88	04	0A	EB	B9	48	81	C4	H< "\$Ð...^...ë^H.Ã
000000260	C8	00	00	00	C3												È...Ã[]

voici notre fichier shell.bin

On ouvre dans IDA:



On choisi mode 64 bits



```

seg000:0000000000000000 ; =====
seg000:0000000000000000
seg000:0000000000000000 ; Segment type: Pur
seg000:0000000000000000 seg000    seg000
seg000:0000000000000000 ass000
seg000:0000000000000000 ass000
seg000:0000000000000000
seg000:0000000000000000
seg000:0000000000000001
seg000:0000000000000002
seg000:0000000000000003
seg000:0000000000000004
seg000:0000000000000005
seg000:0000000000000006
seg000:0000000000000007
seg000:0000000000000008
seg000:0000000000000009
seg000:000000000000000A
seg000:000000000000000B
seg000:000000000000000C
seg000:000000000000000D

```

Information

You have just loaded a binary file.

IDA cannot identify the entry point automatically as there is no standard of binaries.

Please move to what you think is an entry point and press 'C' to start the autoanalysis.

Don't display this message again

OK

on va y aller a l'entry Point et clicker sur P , pour délimiter les fonctions:

avant:

```
seg000:0000000000000000  
seg000:0000000000000000  
seg000:0000000000000001  
seg000:0000000000000002  
seg000:0000000000000003  
seg000:0000000000000004  
seg000:0000000000000005  
seg000:0000000000000006  
seg000:00000000000000078  
seg000:00000000000000090  
seg000:000000000000000A8  
seg000:000000000000000C0  
seg000:000000000000000D8  
seg000:000000000000000E0
```

```
assume cs:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
db 48h ; H
db 89h
db 4Ch ; L
db 24h ; $
db 8
db 48h, 81h, 0ECH
dq 2484C748000000C8h, 0A0h, 0A82484C748h, 2484C74800000000h
dq 0B0h, 6025048B4865h, 8B48482444894800h, 4818408B48482444h
dq 24448B4850244489h, 44894820C0834850h, 485824448B485824h
dq 8B48008B48008Bh, 2444894810E88348h, 8B486024448B4860h
dq 4828244489483040h, 244489482824448Bh, 63486824448B4868h
dq 4828244C8B483C40h, 448948C18B48C803h, 4800000008B87024h
dq 70244C8B4800C06Bh, 480000008801848Bh, 48C8034828244C8Bh
dq 483024448948C18Bh, 481C408B3024448Bh, 48C8034828244C8Bh
dq 8824848948C18Bh, 8B3024448B480000h, 4828244C8B482040h
dq 44824861818C602h, 8B30244481877024h, 4828244C8B482140h
```

après:

```
seg000:0000000000000000 sub_0
seg000:0000000000000000
seg000:0000000000000000 var_A8
seg000:0000000000000000 var_A4
seg000:0000000000000000 var_A0
seg000:0000000000000000 var_98
seg000:0000000000000000 var_90
seg000:0000000000000000 var_88
seg000:0000000000000000 var_80
seg000:0000000000000000 var_78
seg000:0000000000000000 var_70
seg000:0000000000000000 var_68
seg000:0000000000000000 var_60
seg000:0000000000000000 var_58
seg000:0000000000000000 var_50
seg000:0000000000000000 var_48
seg000:0000000000000000 var_40
seg000:0000000000000000 var_38
seg000:0000000000000000 var_30
seg000:0000000000000000 var_28
seg000:0000000000000000 var_20
seg000:0000000000000000 var_18
seg000:0000000000000000 arg_0
seg000:0000000000000000
seg000:0000000000000000
seg000:0000000000000005
seg000:000000000000000C
```

```
proc near

= dword ptr -0A8h
= dword ptr -0A4h
= qword ptr -0A0h
= qword ptr -98h
= qword ptr -90h
= qword ptr -88h
= qword ptr -80h
= qword ptr -78h
= qword ptr -70h
= qword ptr -68h
= qword ptr -60h
= qword ptr -58h
= qword ptr -50h
= qword ptr -48h
= qword ptr -40h
= qword ptr -38h
= qword ptr -30h
= qword ptr -28h
= qword ptr -20h
= qword ptr -18h
= qword ptr 8

mov [rsp+arg_0], rcx
sub rsp, 0C8h
mov [rsp+0C8h+var_28], 0
```

on décompile avec IDA :

```

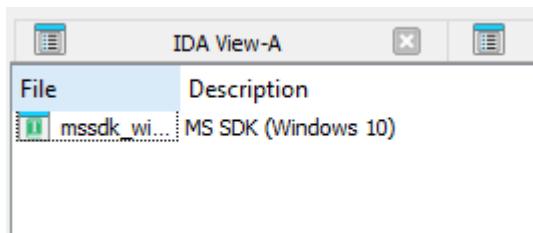
int64 __fastcall sub_0(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
{
    __int64 result; // rax
    int j; // [rsp+20h] [rbp-A8h]
    signed int i; // [rsp+24h] [rbp-A4h]
    __int64 v7; // [rsp+28h] [rbp-A0h]
    unsigned int *v8; // [rsp+30h] [rbp-98h]
    _BYTE *v9; // [rsp+38h] [rbp-90h]
    __int64 (__fastcall *v10)(__int64, __int64, __int64, _QWORD); // [rsp+40h] [rbp-88h]
    __int64 v11; // [rsp+98h] [rbp-30h]

    v7 = *(_QWORD **)(***(_QWORD ***)(*_QWORD *)(__readgsword(0x60u) + 24) + 32LL) - 16LL + 48;
    v8 = (unsigned int *)(*(unsigned int *)(*(int *)(v7 + 60) + v7 + 136) + v7);
    v10 = 0;
    for ( i = 0; ; ++i )
    {
        result = v8[6];
        if ( i >= (unsigned int)result )
            break;
        v9 = (_BYTE *)(*(unsigned int *)((v8[8] + v7 + 4LL * i) + v7));
        if ( *v9 == 71 && v9[3] == 77 && v9[9] == 72 )
        {
            result = *(unsigned int *)((v8[7] + v7 + 4LL * *(unsigned __int16 *)((v8[9] + v7 + 2LL * i))) + v7);
            v10 = (_int64 (__fastcall *)(__int64, __int64, __int64, _QWORD))result;
            break;
        }
    }
    if ( v10 )
    {
        result = v10(a1, a2, a3, 0) + 193640;
        v11 = result;
        for ( j = 0; j < 8; ++j )
        {
            *(BYTE *)(a4 + j) ^= *(BYTE *)(v11 + j);
            result = (unsigned int)(j + 1);
        }
    }
    return result;
}

```

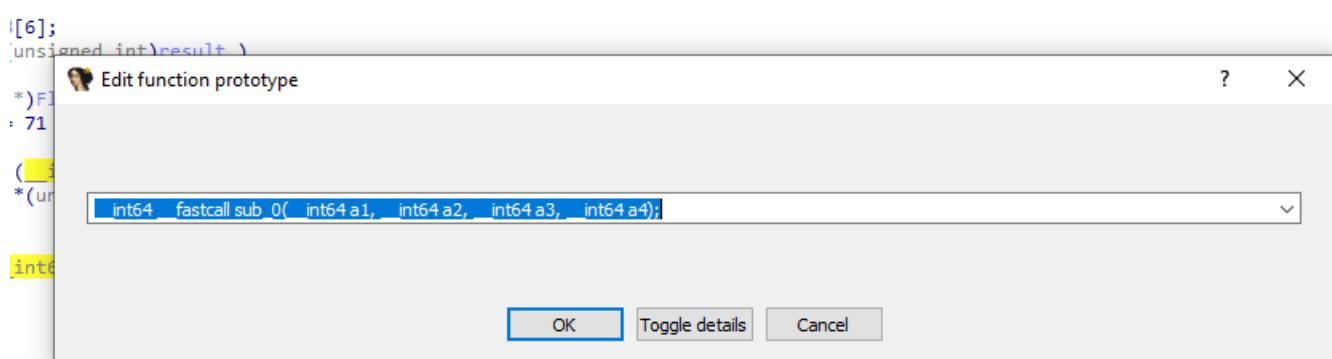
Le fichier n'étant pas au format PE, IDA ne peut pas savoir que c'est une shellcode spécifique à Windows, il faut donc charger manuellement les bibliothèques de types pour Windows.

Appuyez sur SHIFT+F11 et ajoutez la bibliothèque de type mssdk_win10



Maintenant on change les définitions des variables avec la bibliothèques que nous avons importé:

On peut faire ca en cliquant sur Y:



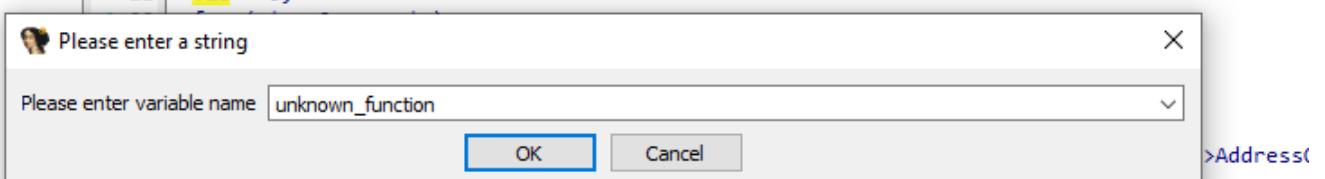
```
.0(a1, a2, a3, 0) + 193640;
.t;
!; j < 8; ++j )

signed int i; // [rsp+24h] [rbp-A4h]
struct _LIST_ENTRY *Flink; // [rsp+28h] [rbp-A0h]
unsigned int *v8; // [rsp+30h] [rbp-98h]
_BYTE *v9; // [rsp+38h] [rbp-90h]
_int64 (_fastcall *v10)(_int64, _int64, _int64, _QWORD); // [rsp+40h] [rbp-88h]
_int64 v11; // [rsp+98h] [rbp-30h]
_int64 v12; // [rsp+D0h] [rbp+8h]

v12 = v2;
Flink = NtCurrentPeb()->Ldr->InMemoryOrderModuleList.Flink->Flink[2].Flink;
v8 = (unsigned int *)((char *)Flink + *(unsigned int *)((char *)&Flink[8].Blink + SHIDWORD(Flink[3].Blink)));
v10 = 0;
for ( i = 0; ; ++i )
{
    result = v8[6];
    if ( i >= (unsigned int)v9 )
        break;
    v9 = (char *)Flink + *
    if ( *v9 == 71 && v9[3] )
    {
        result = (_int64)Flink
            + *(unsigned int *)((char *)&Flink->Flink
                + 4 * *(unsigned int *)((char *)&Flink->Flink + 2 * i + v8[9]))
```

on change les noms des fonction également:

```
8 signed int i; // [rsp+24h] [rbp-A4h]
9 PIMAGE_NT_HEADERS Flink; // [rsp+28h] [rbp-A0h]
10 PIMAGE_EXPORT_DIRECTORY v8; // [rsp+30h] [rbp-98h]
11 _BYTE *v9; // [rsp+38h] [rbp-90h]
12 _int64 (_fastcall *v10)(_int64, _int64, _int64, _QWORD); // [rsp+40h] [rbp-88h]
13 _int64 v11; // [rsp+98h] [rbp-30h]
14 _int64 v12; // [rsp+D0h] [rbp+8h]
15
16 v12 = v2;
17 Flink = (PIMAGE_NT_HEADERS)NtCurrentPeb()->Ldr->InMemoryOrderModuleList.Flink->Flink->Fl
18 v8 = (PIMAGE_EXPORT_DIRECTORY)((char *)Flink
19             + *(unsigned int *)((char *)&Flink->OptionalHeader.DataDirect
20             + (int)Flink->OptionalHeader.FileAlignment
21 v10 = 0;
```



```
29
30     {
31         result = (_int64)Flink
32             + *(unsigned int *)((char *)&Flink->Signature
33                 + 4
34                 * *(unsigned int *)((char *)&Flink->Signature + 2 * i +
+ v8->AddressOfFunctions);
```

capture écran:

```
unsigned int i; // [rsp+24h] [rbp-A4h]
PIMAGE_NT_HEADERS ntHeaders; // [rsp+28h] [rbp-A0h]
PIMAGE_EXPORT_DIRECTORY exportdirectory; // [rsp+30h] [rbp-98h]
char *v6; // [rsp+38h] [rbp-90h]
__int64 (_fastcall *unknown_function)(__int64, __int64, __int64, _QWORD); // [rsp+40h] [rbp-88h]
__int64 xorkey; // [rsp+98h] [rbp-30h]
__int64 key; // [rsp+D0h] [rbp+8h]

key = v2;
ntHeaders = (PIMAGE_NT_HEADERS)NtCurrentPeb()->Ldr->InMemoryOrderModuleList.Flink->Flink[2].Flink;
exportdirectory = (PIMAGE_EXPORT_DIRECTORY)((char *)ntHeaders
    + *(unsigned int *)((char *)&ntHeaders->OptionalHeader.DataDirectory[2].VirtualAddress
        + (int)ntHeaders->OptionalHeader.FileAlignment));
unknown_function = 0;
for ( i = 0; ; ++i )
{
    result = exportdirectory->NumberOfNames;
    if ( i >= (unsigned int)result )
        break;
    v6 = (char *)ntHeaders + *(unsigned int *)((char *)&ntHeaders->Signature + 4 * i + exportdirectory->AddressOfNames);
    if ( *v6 == 'G' && v6[3] == 'M' && v6[9] == 'H' )
    {
        result = (__int64)ntHeaders
            + *(unsigned int *)((char *)&ntHeaders->Signature
                + 4
                * *(unsigned __int16 *)((char *)&ntHeaders->Signature
                    + 2 * i
                    + exportdirectory->AddressOfNameOrdinals)
                + exportdirectory->AddressOfFunctions);
        unknown_function = (__int64) (_fastcall *)(__int64, __int64, __int64, _QWORD))result;
        break;
    }
}
if ( unknown_function )
{
    result = unknown_function(a1, v3, v1, 0) + 193640;
    xorkey = result;
    for ( j = 0; j < 8; ++j )
    {
        *(_BYTE *)(key + j) ^= *(_BYTE *)(xorkey + j);
        result = (unsigned int)(j + 1);
    }
}
return result;
}
```

Décrivez le but de la première boucle for:

La première boucle parcourt la table des fonctions exportées du module afin d'identifier l'entrée correspondant à une fonction précise.

. Elle commence à l'index 0 et vérifie un par un chaque nom de fonction dans la table d'export. Pour chaque nom, elle lit la chaîne de caractères correspondante et vérifie si elle commence par "G", suivi de "M" en 4^e position et "H" en 10^e position.
Cela correspond exactement au début du nom "GetModuleHandle". Dès que cette fonction est trouvée, le shellcode calcule son adresse réelle en utilisant les tables internes du PE (AddressOfNames, AddressOfNameOrdinals et AddressOfFunctions), puis stocke cette adresse dans unknown_function pour l'utiliser plus tard.
Une fois trouvée, la boucle s'arrête : il n'a pas besoin de continuer, car il a ce qu'il faut pour charger d'autres fonctions Windows plus tard.

question 3:

Address	Ordinal	Name	Library
KERNEL32			
000000014002D228		GetModuleHandleW	KERNEL32
000000014002D2F0		GetModuleHandleExW	KERNEL32

les fonctions qui vérifient bien cela : GetModuleHandleW ou ExW , elles se trouvent dans la DLL système kernel32.dll.

Or : Le choix se porte cependant sur **GetModuleHandleW** : l'appel ultérieur unknown_function(0) . n'utilise qu'un seul argument, alors que **GetModuleHandleExW** en requiert trois, ce qui rend son usage incompatible ici. Ainsi, unknown_function(0) correspond à **GetModuleHandleW**, qui retourne l'adresse de base du module principal — valeur utilisée ensuite pour localiser la clé statique en mémoire à l'offset 0x2F468 .

```

}
if ( unknown_function )
{
    result = unknown_function(al, v3, v1, 0) + 0x2F468;
    xorkey = result;
    for ( j = 0; j < 8; ++j )
    {

```

Est ce que cet offset (0x2F468) vous rappelle quelque chose à 0x2000 près?

Cette offset se trouve dans la partie .rdata , qui est généralement des données statiques qu'on peut pas modifier , (en lecture seule). c'est bien la clé de XOR statique.

fin

The screenshot shows the OllyDbg debugger interface. At the top, there is a 'Go To ...' dialog box with the following fields:

- Address: 0x14002F468
- Case sensitive: Unchecked
- Dynamic labels: Checked
- OK button (highlighted)
- Cancel button

Below the dialog, in the assembly view, the instruction at address 0x14002F468 is shown:

```

14002f468 4d 54 47      DAT_14002f468
          00 0b 0a
          0b 0e

```

On récupère la clé:

'4D5447000B0A0B0E'

puis on xor pour avoir la clé souhaité:

Le calcul souhaité est : C (trouvée en partie 2) et la clé statique CC (offset 0x2F468)

C+CC=key

```
1  C=bytes.fromhex('4D5447000B0A0B0E')
2  CC=b'==>rnrb0'
3  K = bytes([p ^ k for p,k in zip(C, CC)])
4  print(K)
```

On obtient la valeur :

```
SyntaxError: invalid syntax
● PS C:\Users\pc\Desktop\dismember> python -u "c:\Users\pc\Pictures\l.py"
b'phyrexia'
```

```
C:\Users\pc\Desktop\dismember\dismember>dismember.exe
Enter your plane : phyrexia
```

```
.....%oooooooooooooooooooooooooooo#-:
-+..=*#@ooooooooo#:
.=ooooooooo*:
.-ooooooooo@:
%oooooooo@:
%ooooooooo=:
@ooooooooo-
*ooooooooo+
:@ooooooooo=
:@ooooooooo%
.-ooooooooo%
.oooooooo#:
+oooooooo#:
:@oooooooo@:
:@oooooooo@=:
.@@@%*:
.@@@@+:
+@@@=:
.:@@@#:
.-@@@#
*@@#:
.@@-
.@@@:
*#:
.+. .
.....Congratulation you find a way to Phyrexia !
.....
```

CTRL-C to exit