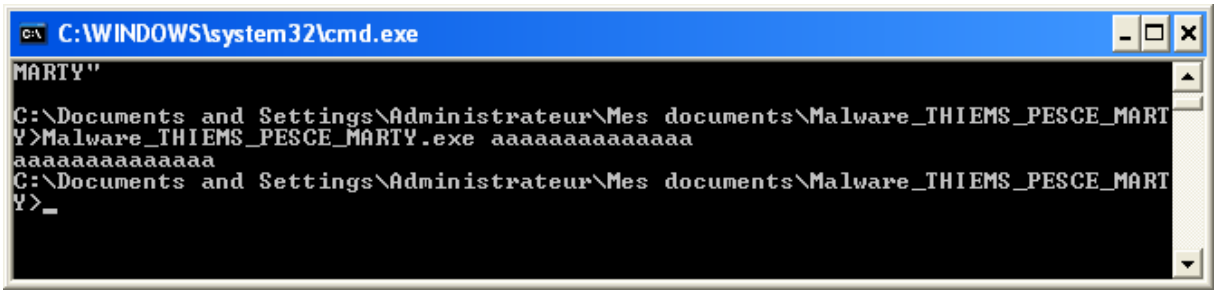
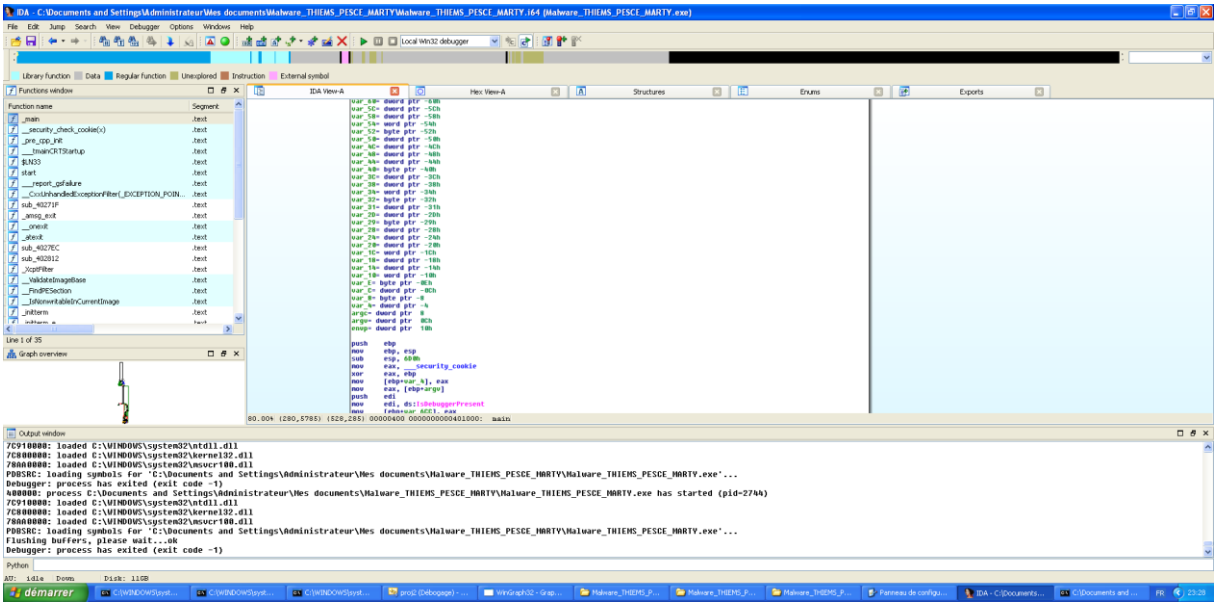
Thomas BIGEL
Christian VASAUNE
Matthieu PERRIGOT

Projet Malware : analyse du malware du groupe THIEMS_PESCE_MARTY

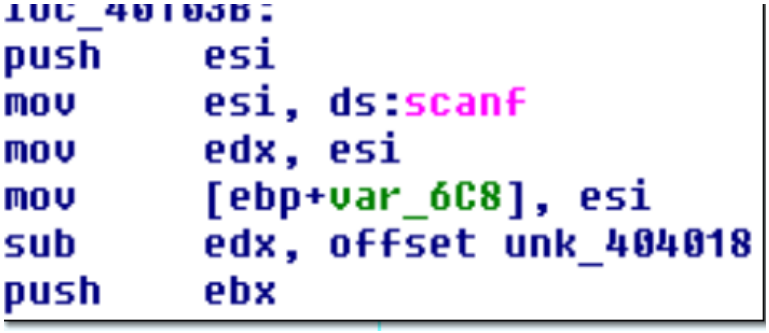
La première chose qu'on regarde est si le programme nous renvoi bien notre argument. C'est bien le cas :



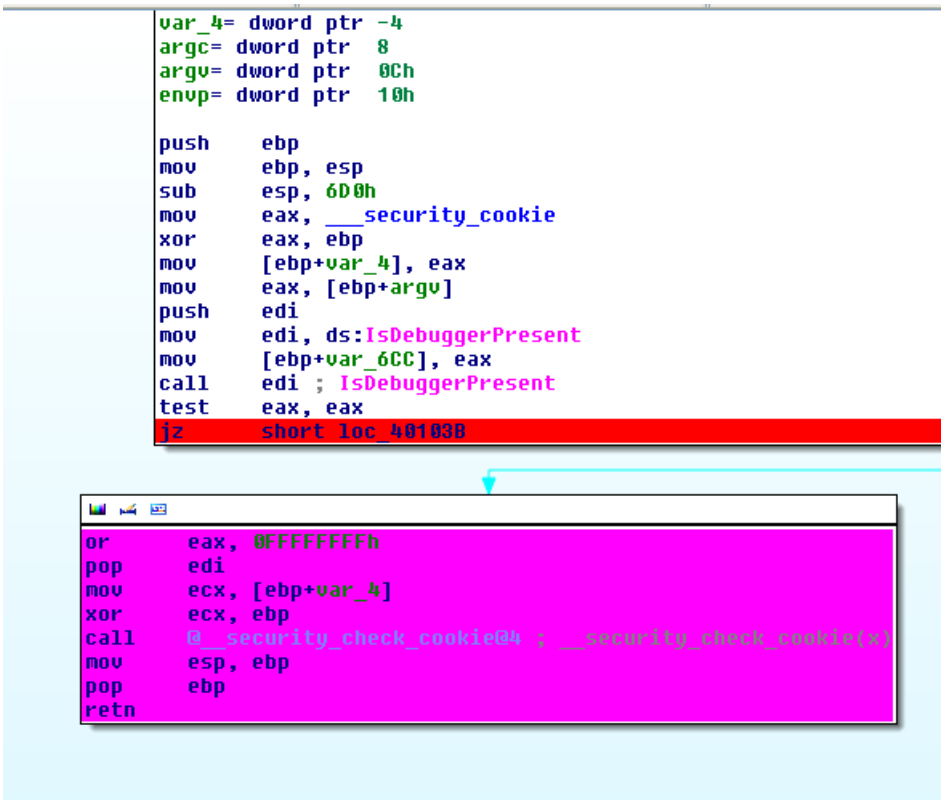
On lance donc IDA et la première chose qu'on remarque quand on regarde main est un très grand nombre de variable initialisé au début de la fonction main. La très grande majorité de ces variables contiennent une valeur négative.



On regarde donc les référence à ces variables et on voit que les VAR_ sont utilisé dans une addition avec ebp : [ebp+Var_...].



On voit aussi l'utilisation de la fonction IsDebuggerPresent qui si on dézoome un peu sur le graphe on la voit utilisé plein de fois tous le long du programme (et plus tard tous le long du chemin qui permet d'avoir le message Bravo c'est la bonne clé !).

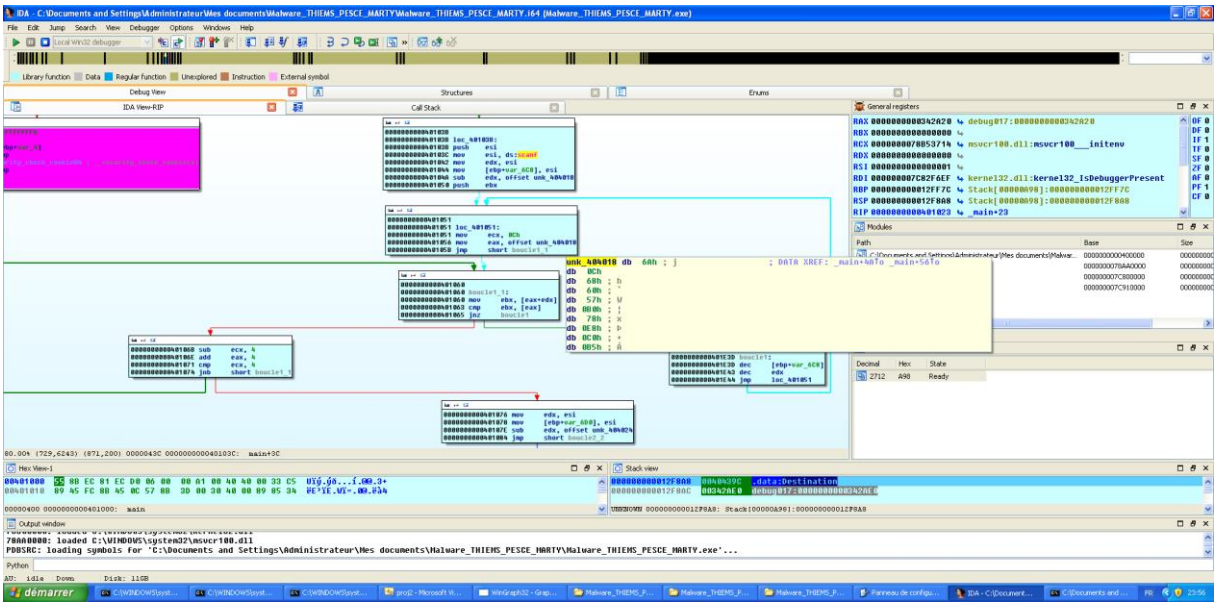


Le IsDebuggerPresent nous mène bien sur un return de la fonction main.

On voit aussi sur cette image que le malware stocke l'adresse pour récupérer notre argument à [ebp+var_6CC]

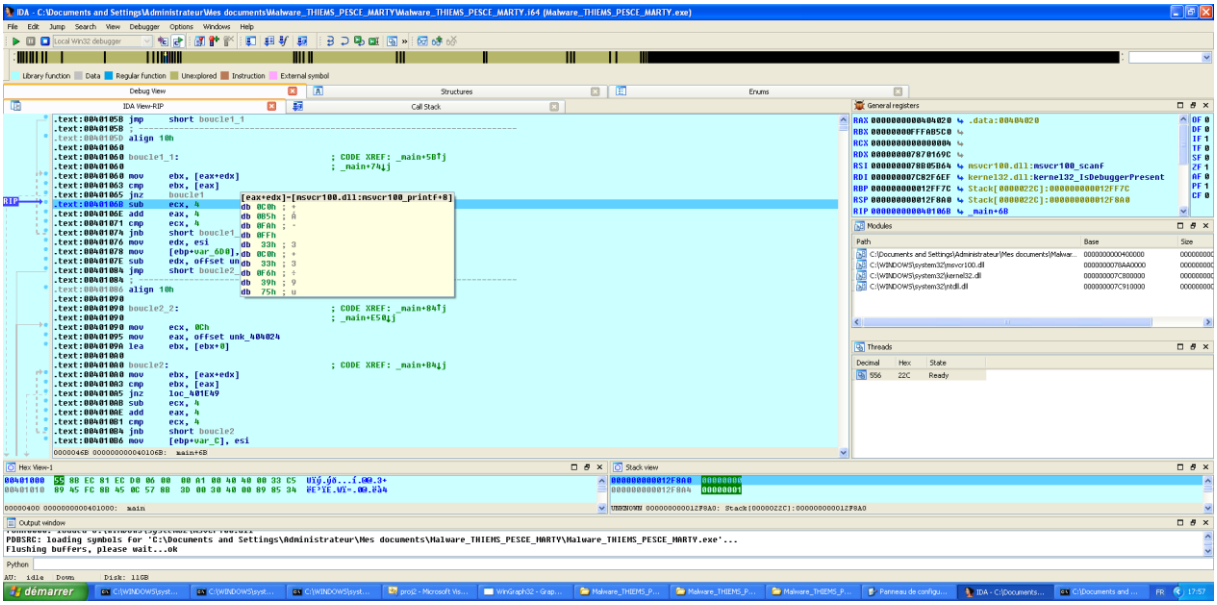
On a ensuite chercher dans le graph du malware si il y avait des structures qui nous sont familières.

On remarque donc 3 boucles d'affilées qui ressemble à une recherche d'un pointeur sur une fonction à partir d'une autre fonction. Ici on part donc de scanf. On remarque qu'on a des instruction dec sur edx et [ebp+var_6C8] et edx prend la valeur de esi qui vaut l'adresse de scanf donc on peut dire que l'adresse de la fonction recherché serra dans [ebp+var_6C8].

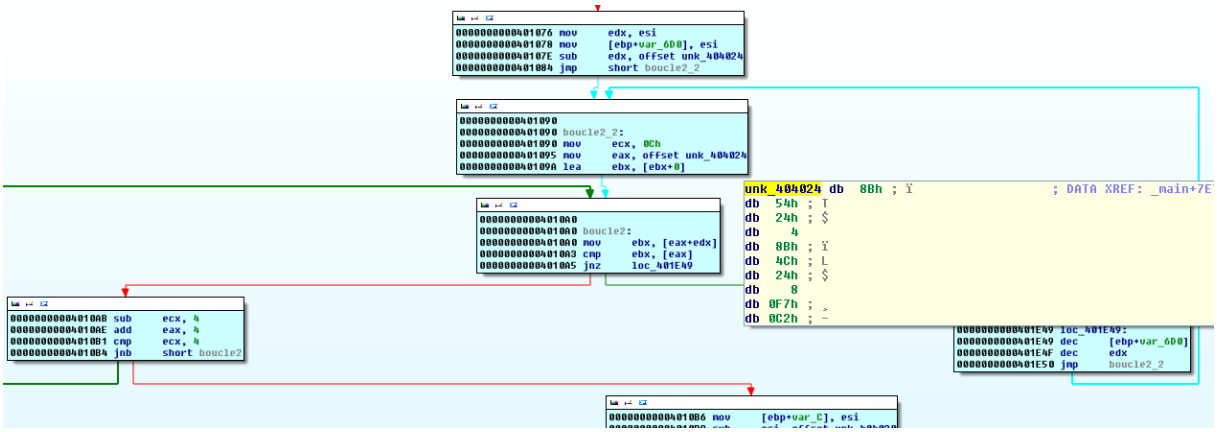


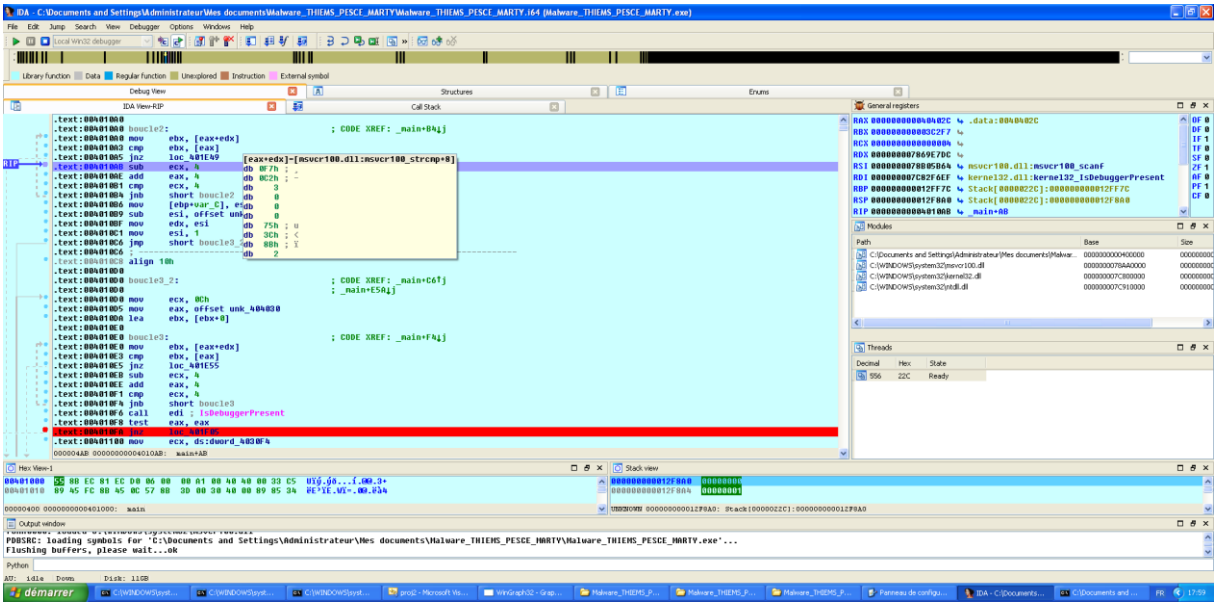
Ici on voit que unk_404010 correspond aux premiers octets de la fonction printf qu'on a vu en cours quand on voyait cette technique justement.

Le fait que l'adresse de printf est stockée dans [ebp+var_6C8] est confirmé quand on a lancé le décompilateur en passant les IsDebuggerPresent en modifiant le flag ZF. (La seule méthode anti-debug utilisé par le groupe est la fonction IsDebuggerPresent)

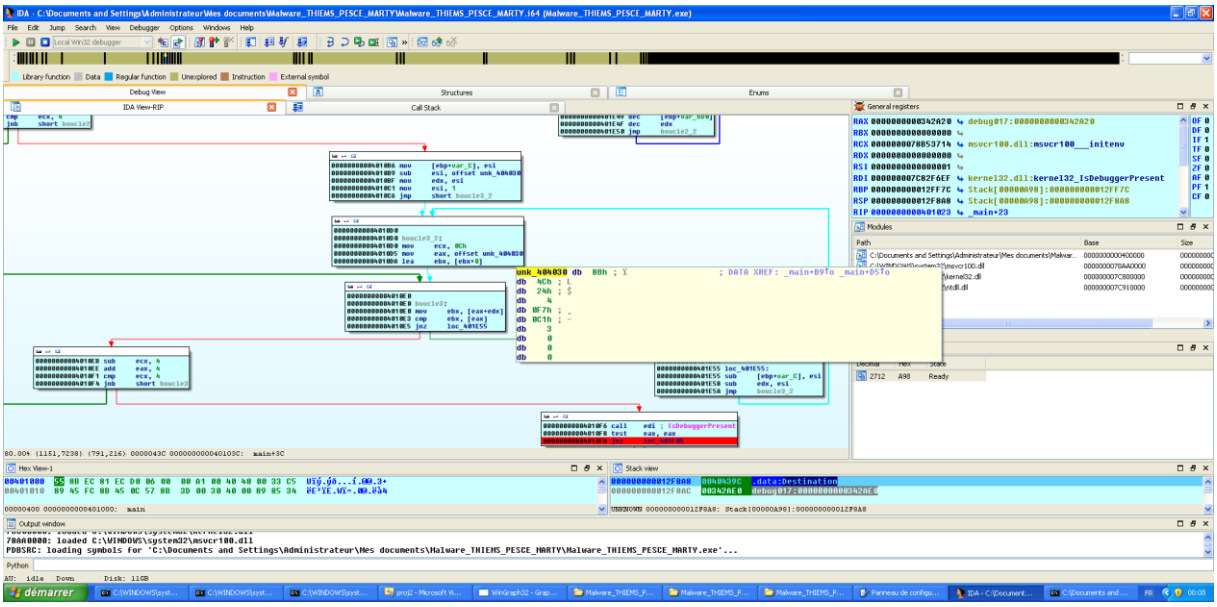


Pour la deuxième boucle on repart encore de scanf (stocké dans esi précédemment) et le malware utilise la même technique que précédemment mais cette fois en comparent avec unk_404024 et en stockant l'adresse obtenu dans [ebp+var_6D0]. Au début on ne savait pas juste en regardant à quelle fonction correspondait cette suite d'octets mais IDA nous en informe ensuite en debuggant et en regardant vers quoi ca pointe quand on est à la fin de la boucle. La fonction recherchée dans la 2eme boucle est donc strcmp





La même situation se produit pour la troisième boucle que pour la deuxième et on trouve donc que c’est la fonction strlen qui est recherché et que l’adresse est stocké dans [ebp+var_C]



On a donc 3 fonctions qui pourront être appelé avec un call sans les appelés directement et que ce soit bien visible avec IDA:

boucle1: pointeur sur printf +8 [ebp+var_6C8]

boucle2: pointeur sur strcmp +8 [ebp+var_6D0]

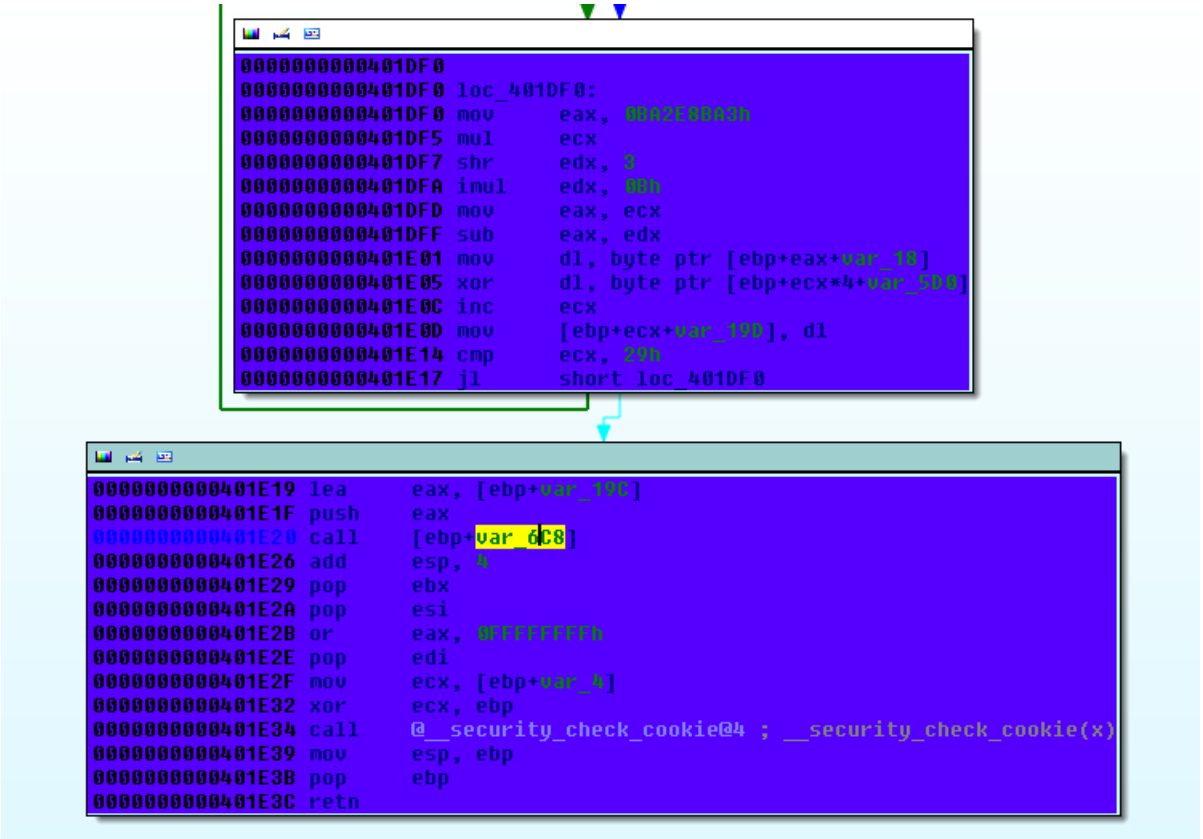
boucle3: pointeur sur strlen +8 [ebp+var_C]

On va donc pouvoir regarder dans le malware les différentes instructions call qui appellent ces fonctions.

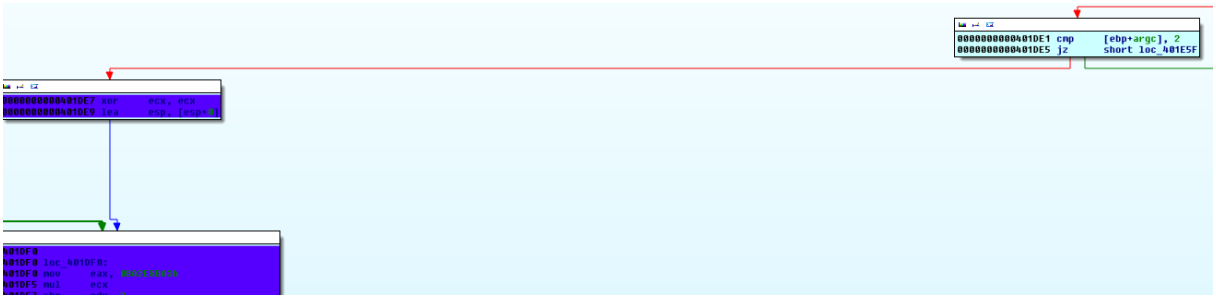
On remarque que printf n'est jamais directement appelé. Le malware utilise toujours [ebp+var_6C8].

On regarde donc à la fin du programme pour chercher par où le programme devrait passer pour afficher autre chose que l'argument.

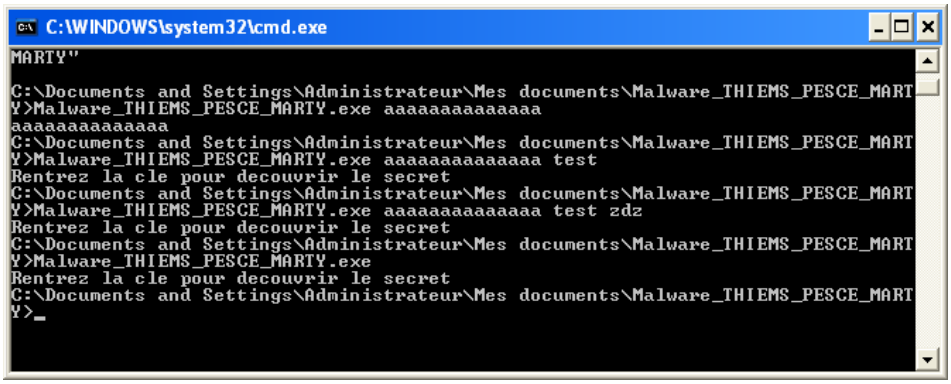
Le printf le plus haut dans le graphe ne printf pas l'argument car elle push à la place eax qui a l'instruction lea eax, [ebp+var_19C] juste avant.



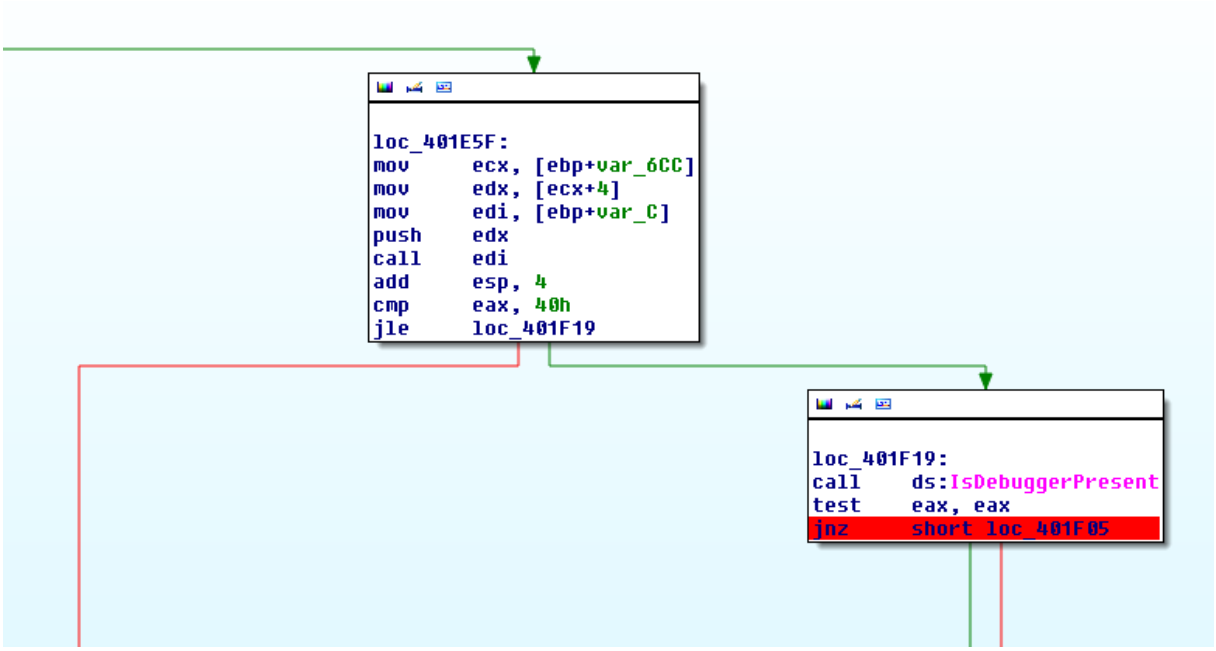
Il y a une boucle avec un xor juste avant qui modifie autour de l'adresse utilisé pour ce printf. On fait évidement fausse route car si on regarde le chemin qui mène ici il n'y a aucune comparaison avec notre argument et si on dézoome un peu on voit qu'on atterrit ici après un test pour voir si le nombre d'argument est diffèrent de 2.

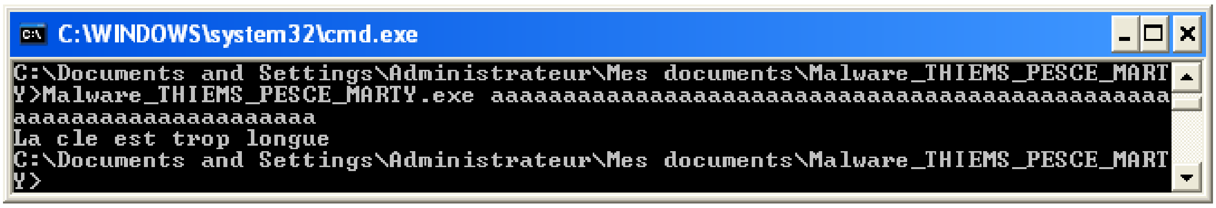


Ce chemin correspond donc plutôt au printf qui nous affiche un message quand on ne met pas le bon nombre d'argument. Ce raisonnement nous est confirmé si on utilise le débugger IDA et qu'on atterrit ici.



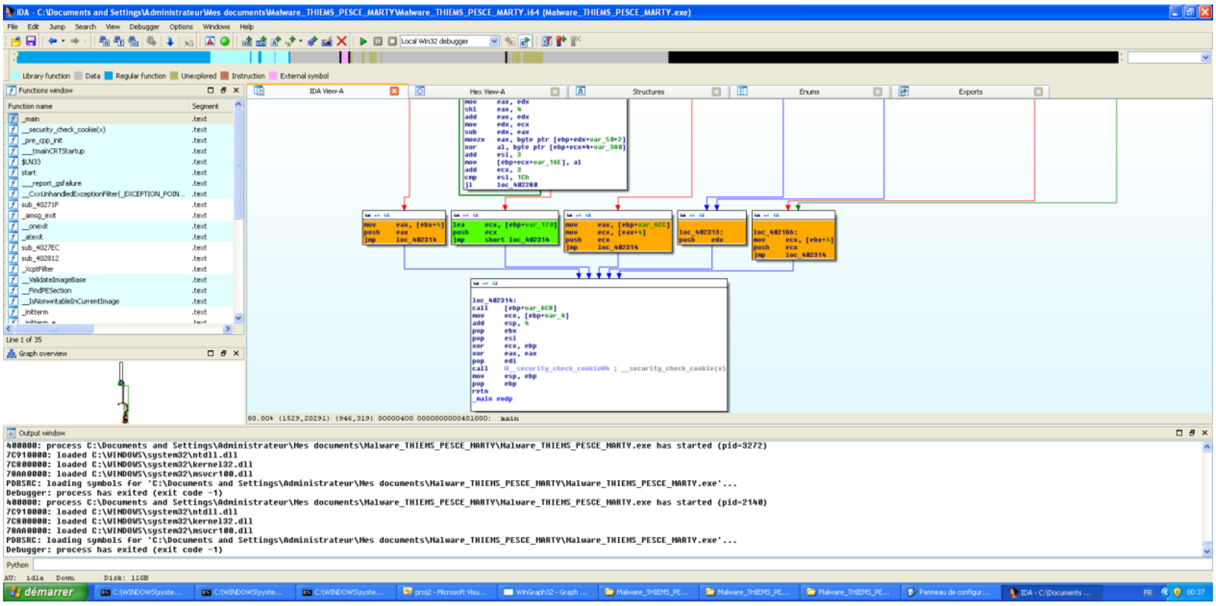
On a la même situation qui s'est reproduit mais cette fois ci c'est pour le message quand l'argument est trop grand. On le voit avec la node qui mènent à cette boucle de xor utilise une instruction call [ebp+var_C], donc un appel à la fonction strlen et on voit une comparaison entre le résultat de la fonction strlen et 40h (qui correspond à 64 en décimale). Cela mène à cette boucle seulement si la valeur retourné par strlen est supérieur à 64 (car on a jle : jump if lower or egal le false mène à cette boucle)





On peut aussi confirmer en utilisant le debugger IDA.

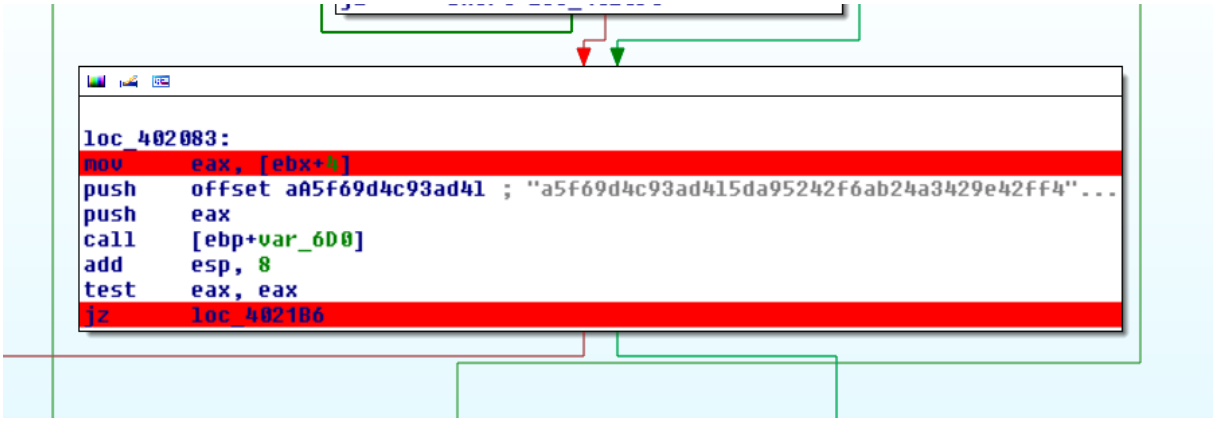
On continue donc de regarder les utilisations de printf et on trouve tout en bas du graph le dernier printf dans le node terminal du graph. On remarque qu'il n'y a pas de push dans le node lui-même donc c'est un node au-dessus qui s'occupe de push la valeur à passer à printf. Pour 3 nodes on voit qu'on push l'argument pour une autre faut remonter un peu mais on push aussi l'argument et on a une seule node (en vert) qui push autre chose que l'adresse ou est stocké l'argument. On fait donc l'hypothèse que si le programme passe par ce chemin il affichera un message de félicitation.

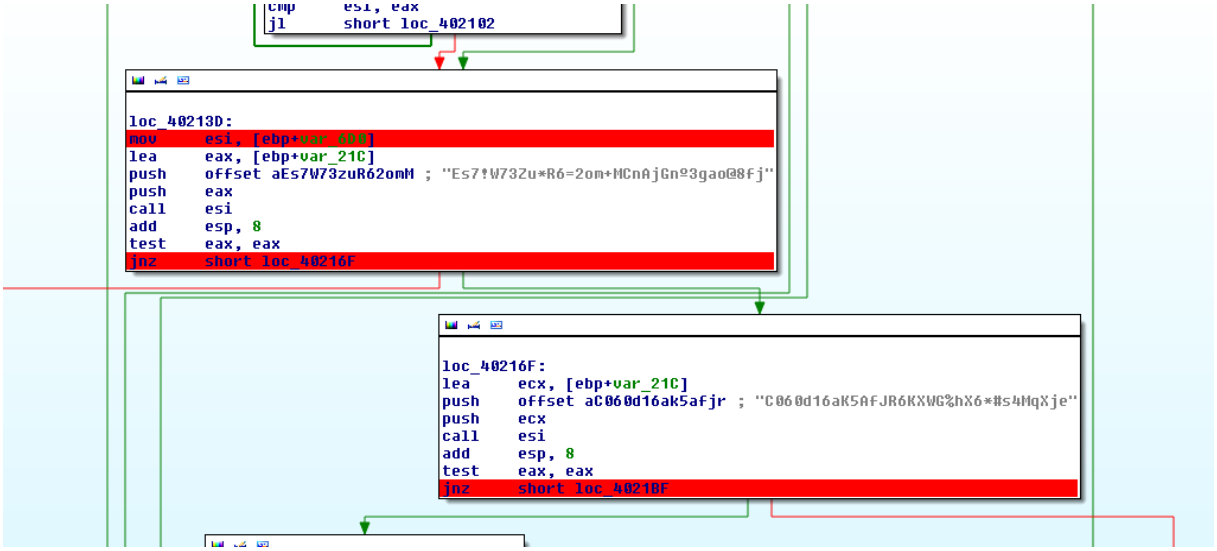


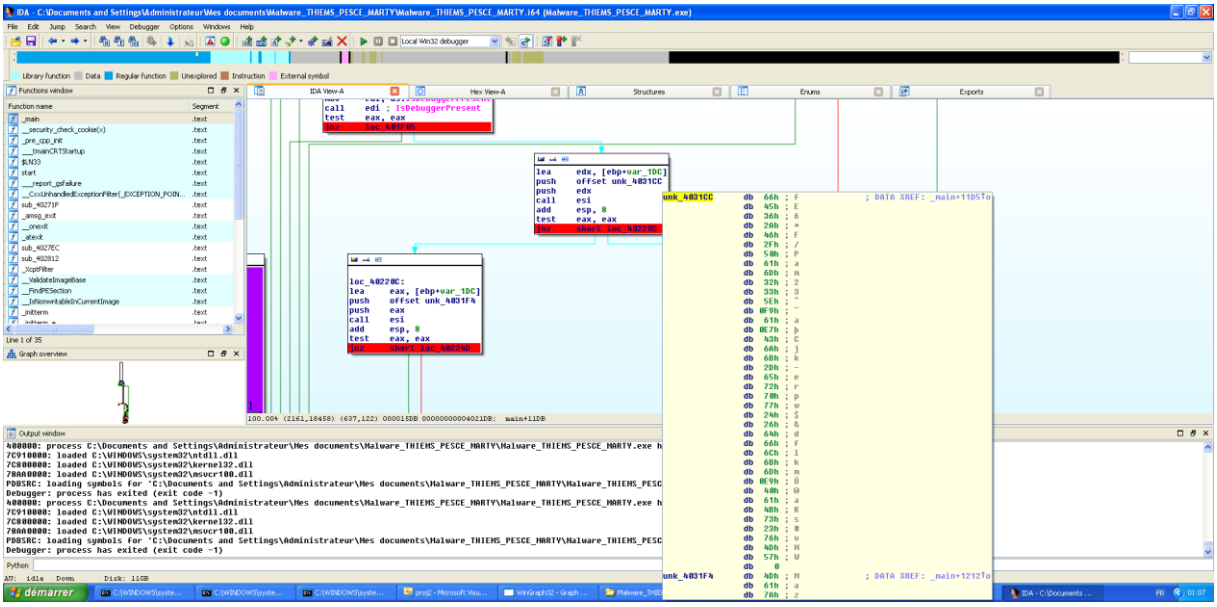
En remontant ce chemin on voit une boucle avec un xor mais pas de comparaison après, on fait donc l'hypothèse que le xor sert pour l'affichage du message de félicitation.

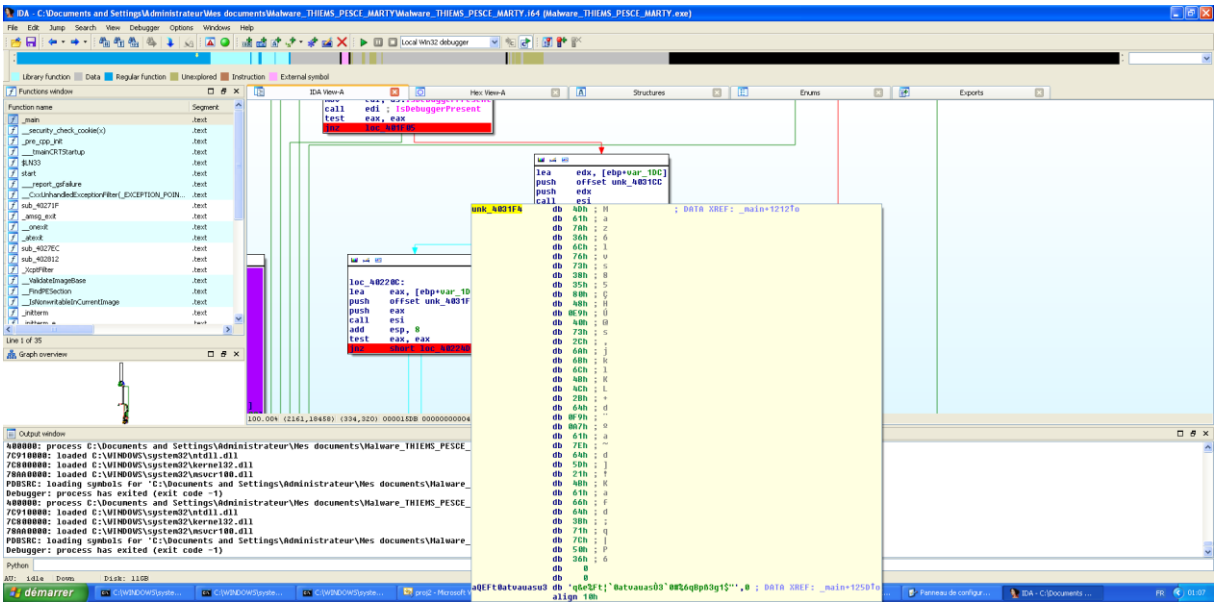
On se concentre ensuite sur les instructions call [ebp+var_6D0] donc quand on appelle la fonction strcmp car cette fonction est surement utilisée pour comparer notre argument avec autre chose ou surement notre argument chiffré avec la clé chiffré.

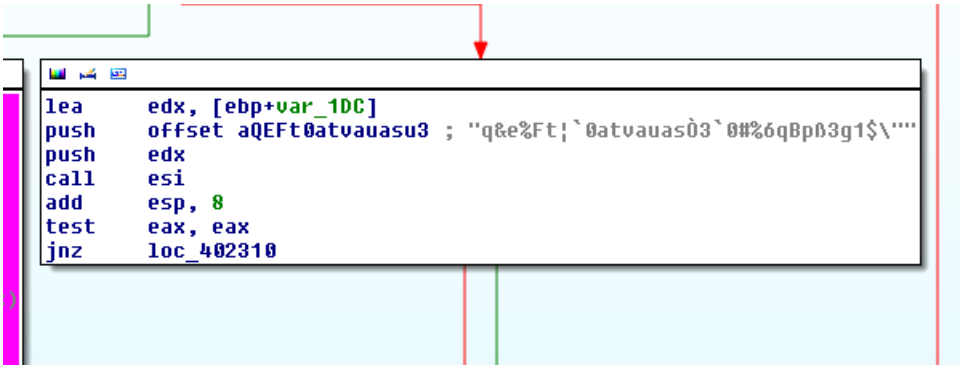
On trouve directement un call [ebp+var_6D0] et des fois c'est un peu plus caché en passant [ebp+var_6D0] dans un registre au-dessus, avant d'utiliser se registre pour appeler la fonction strcmp :


loc_402083:
mov eax, [ebx+4]
push offset aA5f69d4c93ad41 ; "a5f69d4c93ad415da95242f6ab24a3429e42ff4"...
push eax
call [ebp+var_6D0]
add esp, 8
test eax, eax
jz loc_4021B6


cmp esi, eax
jl short loc_402102
loc_40213D:
mov esi, [ebp+var_6D0]
lea eax, [ebp+var_21C]
push offset aEs7W73zuR62omM ; "Es7!W73Zu*R6=2om+MCnAjGn93gao@8fj"
push eax
call esi
add esp, 8
test eax, eax
jnz short loc_40216F
loc_40216F:
lea ecx, [ebp+var_21C]
push offset aC060d16ak5afjr ; "C060d16aK5AfJR6KXWC%hX6*#s4MqXje"
push ecx
call esi
add esp, 8
test eax, eax
jnz short loc_4021BF


IDA - C:\Documents and Settings\Administrateur\Mes documents\Malware_THIEMS_PESCE_MARTY\Malware_THIEMS_PESCE_MARTY.i64 (Malware_THIEMS_PESCE_MARTY.exe)
File Edit Jump Search View Debugger Options Windows Help
Functions window
Function name
Segment
IDA View-A
Hex View-A
Structures
Enums
Exports
Output window
démarrer





On a donc 6 clés chiffrées potentielles :

a5f69d4c93ad4l5da95242f6ab24a3429e42ff463

C060d16aK5AfJR6KXWG%hX6#s4MqXje

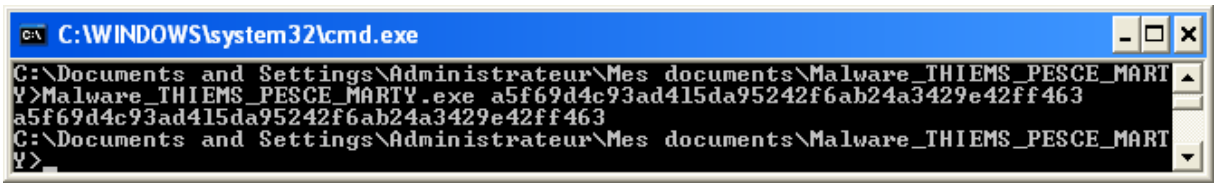
Es7!W73ZuR6=2om+MCnAjGnº3gao@8fj

unk_4031CC

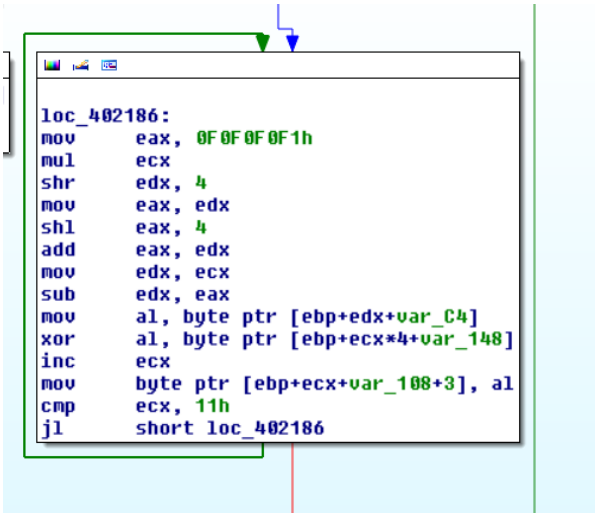
unk_4031F4

q&e%Ft¦0atvauasÒ30#%6qBpß3g1$"

On peut facilement en éliminer une (a5f69d4c93ad4l5da95242f6ab24a3429e42ff463) car elle est comparée directement à l'argument ce qui n'est pas le cas des autres et si strcmp renvoi true ça nous envoi sur un chemin qui affiche juste l'argument. En regardant avec le debugger on voit que l'argument n'as pas été altéré. (On peut aussi aller jusqu'au bout du programme…). On peut aussi tester d'entrer directement cette clé comme argument.



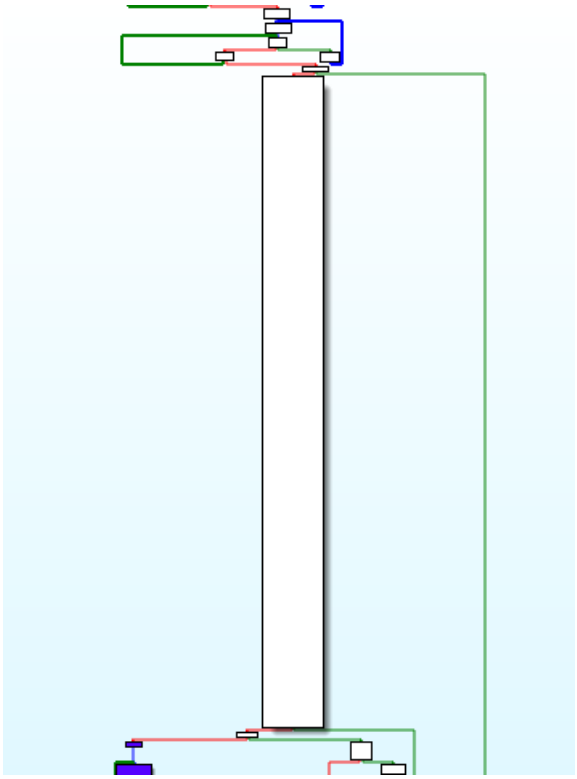
La boucle avec xor qui suit est donc juste un leurre :



Les autres strcmp comparent avec une valeur se trouvant soit dans [ebp+var_21C] ou [ebp+var_1DC].

On a ensuite décider de regarder les autres nodes en commençant par le haut.

On a donc juste après les trois boucles pour récupérer un pointeur sur printf, strcmp et strlen (et après un autre IsDebuggerPresent()) un énorme pavé d'instruction composé exclusivement d'instruction mov, movsdb et encore d'un autre IsDebuggerPresent().



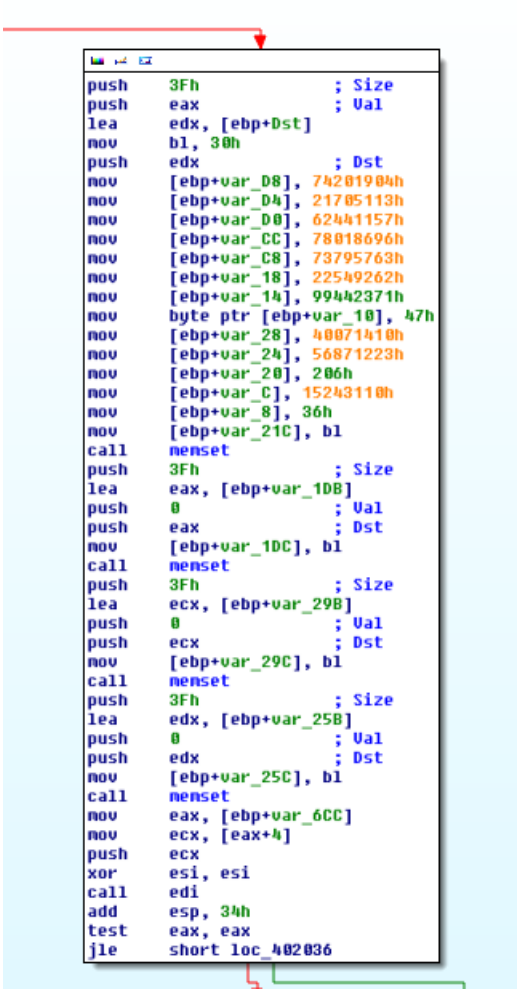
Il n'y a pas d'adresse qui nous intéresse donc cela semble être des leurres ou alors des valeurs qui seront utilisé comme constante pour un chiffrement ou plus simplement utilisé pour afficher un message avec printf.

Après cela on a le test sur le nombre d'argument puis celui sur la longueur de la clé.

On a encore un autre IsDebuggerPresent.

On arrive alors sur un node où le programme fait deux memset.

Il y a des instruction mov qui ne servent clairement à rien et qui sont donc surement là juste pour nous distraire.



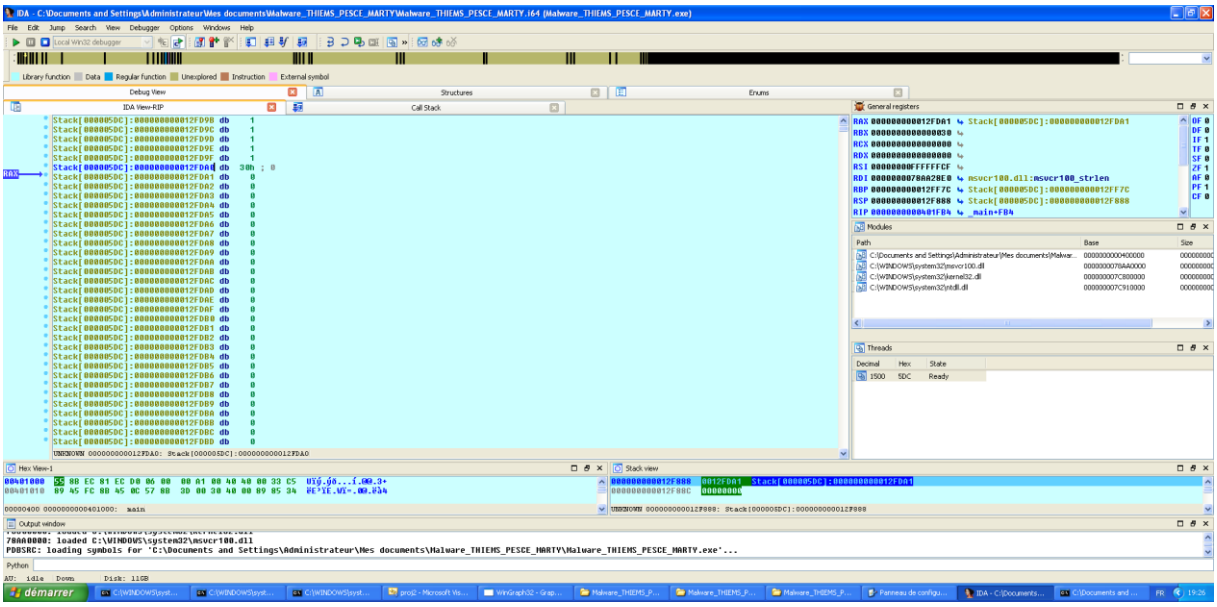
```
push    3Fh             ; Size
push    eax             ; Val
lea     edx, [ebp+Dst]
mov     bl, 30h
push    edx             ; Dst
mov     [ebp+var_D8], 7420190Ah
mov     [ebp+var_D4], 21705113h
mov     [ebp+var_D0], 62441157h
mov     [ebp+var_CC], 78018696h
mov     [ebp+var_C8], 73795763h
mov     [ebp+var_18], 22549262h
mov     [ebp+var_14], 99442371h
mov     byte ptr [ebp+var_10], 47h
mov     [ebp+var_28], 40071410h
mov     [ebp+var_24], 56871223h
mov     [ebp+var_20], 206h
mov     [ebp+var_C], 15243110h
mov     [ebp+var_8], 36h
mov     [ebp+var_21C], bl
call    memset
push    3Fh             ; Size
lea     eax, [ebp+var_1DB]
push    0               ; Val
push    eax             ; Dst
mov     [ebp+var_1DC], bl
call    memset
push    3Fh             ; Size
lea     ecx, [ebp+var_29B]
push    0               ; Val
push    ecx             ; Dst
mov     [ebp+var_29C], bl
call    memset
push    3Fh             ; Size
lea     edx, [ebp+var_25B]
push    0               ; Val
push    edx             ; Dst
mov     [ebp+var_25C], bl
call    memset
mov     eax, [ebp+var_6CC]
mov     ecx, [eax+4]
push    ecx
xor     esi, esi
call    edi
add     esp, 34h
test    eax, eax
jle     short loc_402036
```

Le premier memset met plein de 1 dans la mémoire à partir de [ebp+var_21C] 30h puis 1... (63 de taille (3Fh))

Cela correspond à l'adresse : Stack[00000494]:000000000012FD6. Il y a la valeur 30h sur l'adresse juste avant (000000000012FD5)

Le deuxième memset met plein de 0 dans la mémoire à partir de [ebp+var_1DC] 30h puis 0... (63 de taille (3Fh))

Cela correspond à l'adresse : Stack[00000494]:000000000012FDA1 Il y a la valeur 30h sur l'adresse juste avant (000000000012FDA0)



On confirme ces résultats en utilisant le debugger.

Le troisième memset met plein de 0 dans la mémoire à partir de [ebp+var_29C] : 30h puis 0... (63 de taille (3Fh))

Cela correspond à l'adresse : Stack[00000494]: 000000000012FCE1 Il y a la valeur 30h sur l'adresse juste avant (000000000012FCE0)

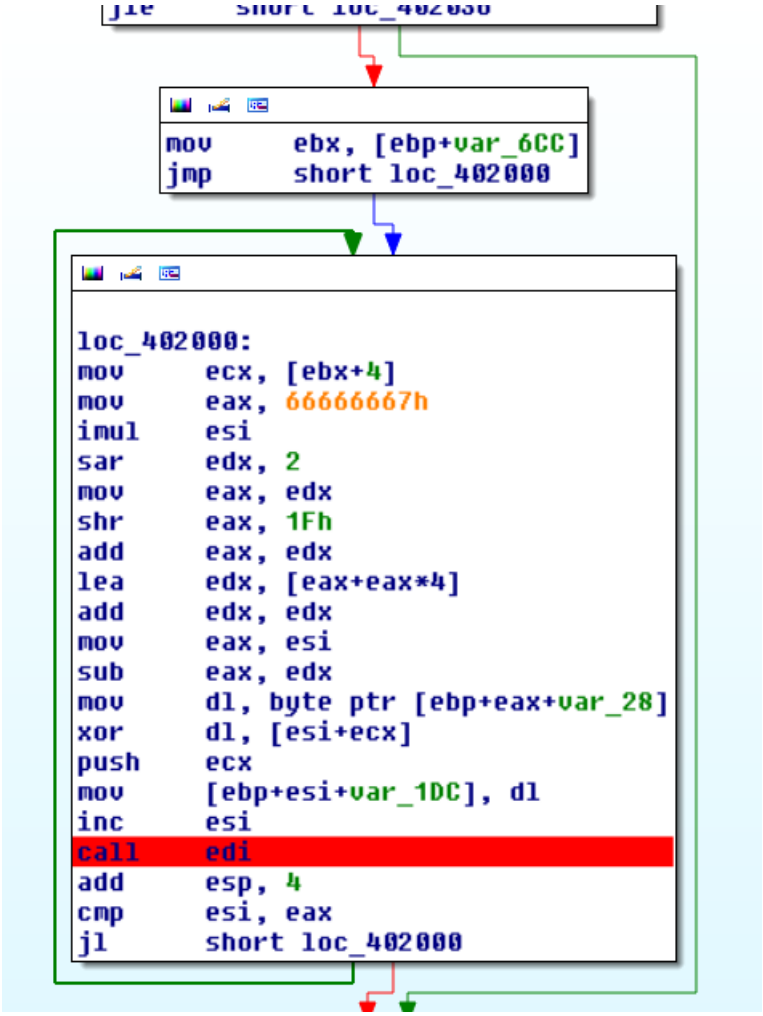
Le quatrième memset met plein de 0 dans la mémoire à partir de [ebp+var_25C] : 30h puis 0... (63 de taille (3Fh))

Cela correspond à l'adresse : Stack[00000494]: 000000000012FD21Il y a la valeur 30h sur l'adresse juste avant (000000000012FD20)

[ebp+var_1DC] et [ebp+var_21C] correspondent aux adresses que l'on recherchait. Ces memset prépare donc le terrain pour un potentiel future chiffrement.

Cela se confirme sur la node suivante quand on récupère la valeur de l'argument avec ebx et le stocke dans ecx et qu'on rencontre une boucle avec un xor qui utilise l'adresse [ebp+var_1DC].

On a donc ici un chiffrement.



On regarde donc ce que fait ce chiffrement:

(edi a prit la valeur de l'adresse pour strlen plus haut)

ebx <- adresse pour argument

Premier passage:

ecx <- debut argument (61h)

eax <- 66666667h

imul esi (esi = 0 et imul: multiplication signé : EDX:EAX = EAX * esi donc ici EDX:EAX = 66666667h * 0)

sar edx, 2 (sar/shr: bouge les bits sur la droite de 2 ici. Les bits qui sont envoyé trop loins sont envoyé dans le flag CF. shr rempli les blancs par des zeros et sar utilise le bit de signe: edx & 0x80) équivalent de diviser par 2^3

mov eax, edx (eax et edx égal à 0 dans le premier passage...)

shr eax, 1Fh (eax = 0)

add eax, edx (eax et edx = 0 premier passage)

lea edx, [eax+eax*4] (edx = 0 premier passage)

add edx, edx (edx = 0)

mov eax, esi (edx = 0 et esi = 0)

sub eax, edx (eax = 0)

mov dl, byte ptr [ebp+eax+var_28] (dl = 10h, ebp+eax+var_28 = 000000000012FF54 donc ebp+var_28 = 000000000012FF54)

xor dl, [esi+ecx] (xor entre dl et le premier character de notre argument ici 61h) dl = 71h

push ecx

mov [ebp+esi+var_1DC], dl (esi = 0 [ebp+var_1DC] donc on range dans la premiere adresse de la plange modifié par un memset précedement)

inc esi (esi++)

call edi (On appelle la fonction strlen)

add esp, 4

cmp esi, eax (On regarde si on a traverser tous notre argument. Si ce n'est pas le cas on recommence la boucle)

jl short loc_402000

Deuxième passage:

ecx <- debut argument (61h)

eax <- 66666667h

EDX:EAX = eax * esi donc EDX:EAX = 66666667h * 1 eax = 66666667h

sar edx, 2

mov eax, edx (eax = edx = 0)

shr eax, 1Fh

mov eax, edx

lea edx, [eax+eax*4] edx = 0 car eax = 0

add edx, edx (edx = 0)

mov eax, esi (eax = 1 car esi = 1)

sub eax, edx (eax = 1 car edx = 0)

mov dl, byte ptr [ebp+eax+var_28] (dl = 14h, ebp+eax+var_28 = 000000000012FF55)

xor dl, [esi+ecx] (xor entre dl et le deuxieme character de notre argument ici 61h) dl = 75h

push ecx

mov [ebp+esi+var_1DC], dl (esi = 0 [ebp+var_1DC] donc on range dans la deuxième adresse de la plange modifié par un memset précedement)

inc esi (esi++)

call edi (On appelle la fonction strlen)

add esp, 4

cmp esi, eax (On regarde si on a traversé tous notre argument. Si ce n'est pas le cas on recommence la boucle)

jl short loc_402000

3eme: dl = 7 puis dl = 66h (resultat avec arg)

dl = 40h puis dl= ... (resultat avec arg)

dl = 23h

dl = 12h

dl = 87h

56h

6

2

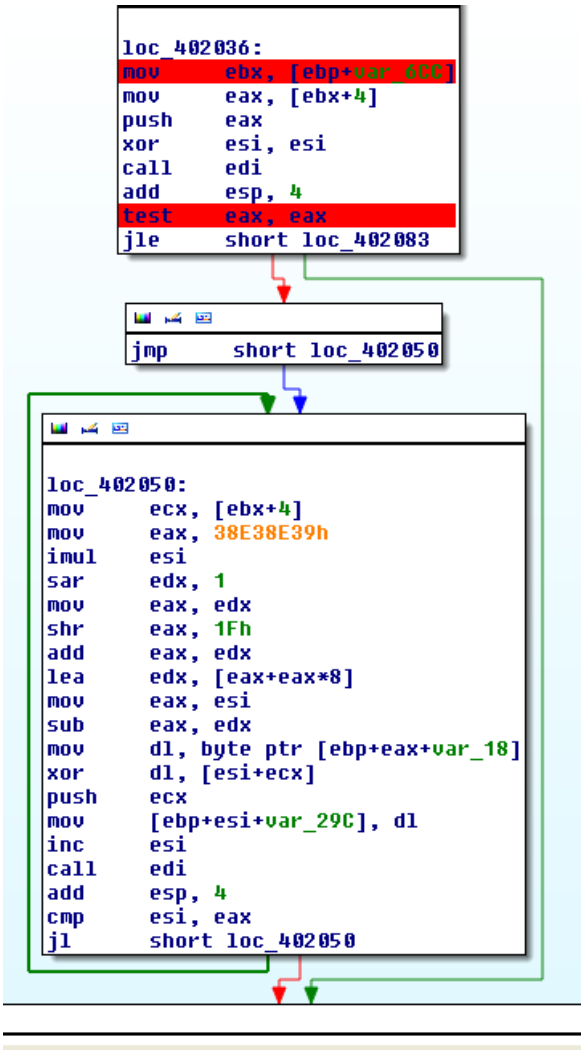
11eme:

sub eax, edx (eax = edx = Ah donc eax = 0)

mov dl, byte ptr [ebp+eax+var_28] (dl = 10h eax = 0 edx = Ah donc eax = 0 [ebp+eax+var_28]=000000000012FF54)

Le chiffrement effectue donc un xor de notre argument avec les valeurs de l'adresse 000000000012FF54 à 000000000012FF5D répété le nombre de fois nécessaire pour couvrir toute la longueur de notre argument. Ici les valeurs xoré à notre argument sont {10h, 14h, 7, 40h, 23h, 12h, 87h, 56h, 6, 2} et si l'argument dépasse le nombre de valeurs on recommence à la première valeur.

(Après ça on a un strlen qui ne sert à rien, toujours faux)

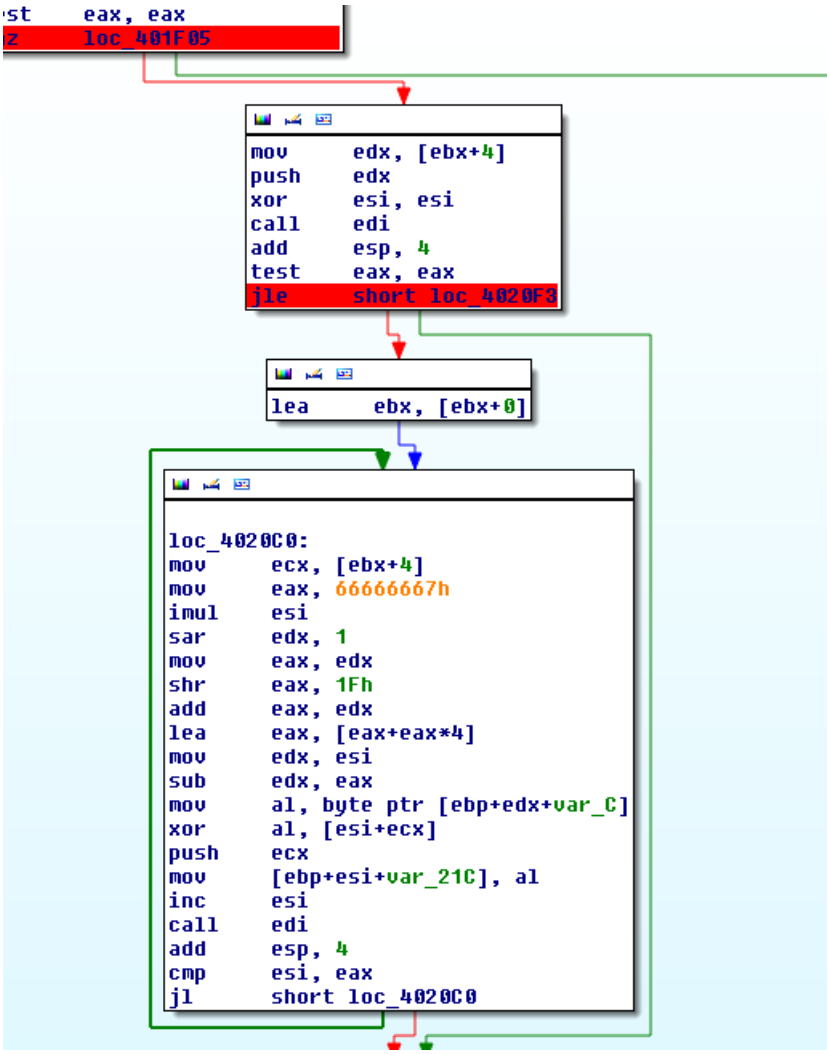


On a alors un autre chiffrement qui chiffre notre argument et le range sur les adresses que le memset a initialisé à partir de l'adresse 000000000012FCE0. (à la place de [ebp+esi+var_1DC] on a [ebp+esi+var_29C] et à la place de [ebp+eax+var_28] on a [ebp+eax+var_18])

C'est le même principe que le chiffrement précédant mais on utilise les valeurs allant de l'adresse 000000000012FF64 à l'adresse 000000000012FF6C.

On a donc la liste {62h, 92h, 54h, 22h, 71h, 23h, 44h, 99h, 47h}

(On sait que le strcmp avec a5f69d4c93ad4l5da95242f6ab24a3429e42ff463 est un leurre du coup on suit la flèche false et on trouve un énième IsDebuggerPresent()puis un autre strlen qui sert à rien)

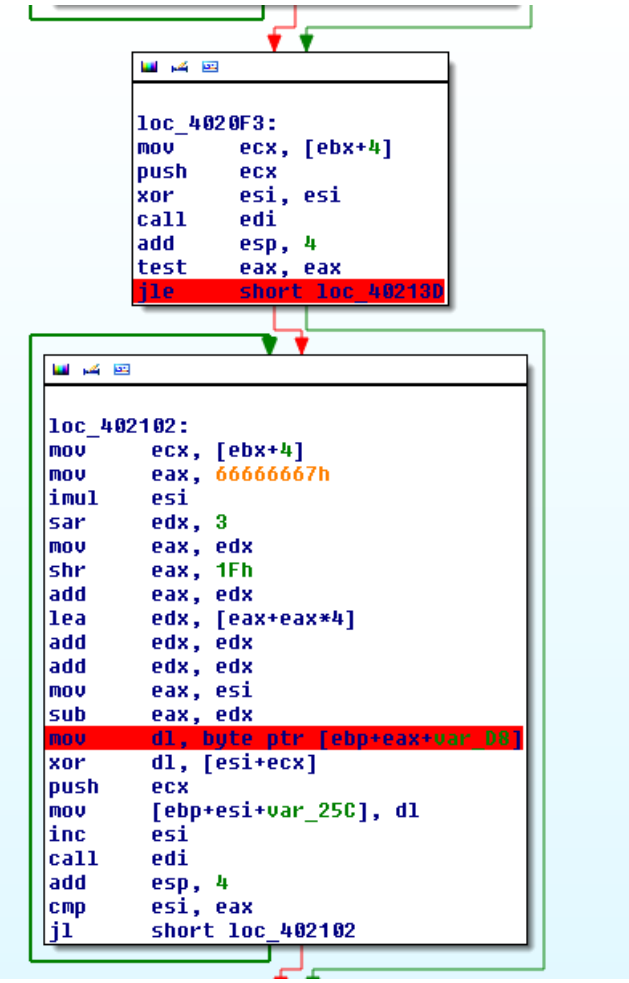


On a alors un autre chiffrement qui chiffre notre argument et le range sur les adresses que le memset a initialisé à partir de l'adresse 000000000012FD60. (À la place de [ebp+esi+var_29C] on a [ebp+esi+var_21C] et à la place de [ebp+eax+var_18] on a [ebp+eax+var_C])

C'est le même principe que le chiffrement précédant mais on utilise les valeurs allant de l'adresse 000000000012FF70 à l'adresse 000000000012FF74.

On a donc la liste {10h, 31h, 24h, 15h, 36h}

(Après ça on a un strlen qui ne sert à rien, toujours faux)



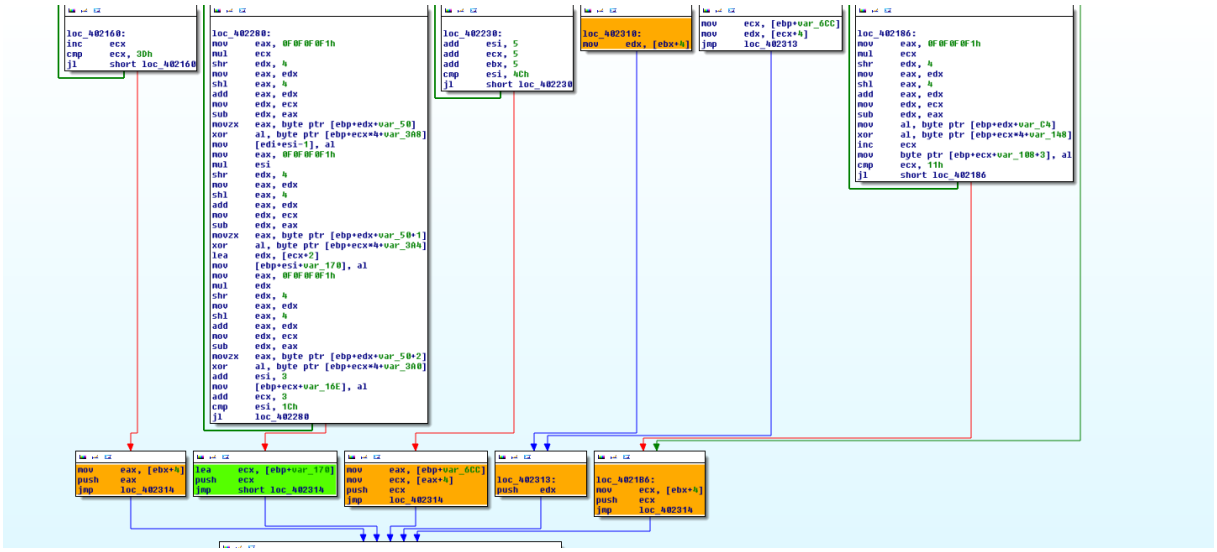
On a alors un autre chiffrement qui chiffre notre argument et le range sur les adresses que le memset a initialisé à partir de l'adresse 000000000012FD60. (À la place de [ebp+esi+var_21C] on a [ebp+esi+var_25C] et à la place de [ebp+eax+var_C] on a [ebp+eax+var_D8])

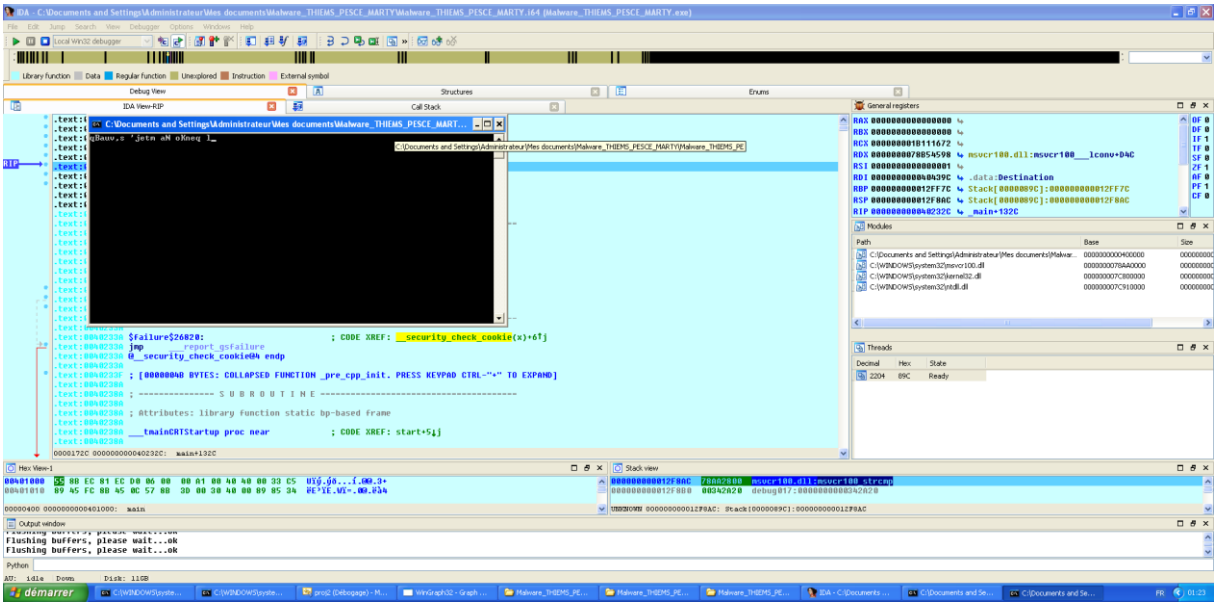
C'est le même principe que le chiffrement précédant mais on utilise les valeurs allant de l'adresse 000000000012FEA4 à l'adresse 000000000012FEB7.

On a donc la liste {4, 19h, 20h, 74h, 13h, 51h, 70h, 21h, 57h, 11h, 44h, 62h, 96h, 86h, 1, 78h, 63h, 57h, 79h, 73}

On doit donc ensuite chercher quel est le strcmp qui compare l'un des argument chiffré avec la bonne clé chiffré. Pour ca on confirme notre hypothese du bon chemin en utilisant le debugger de IDA et en allant le plus loin possible en essayant de passer par ce chemin en changeant les flags.

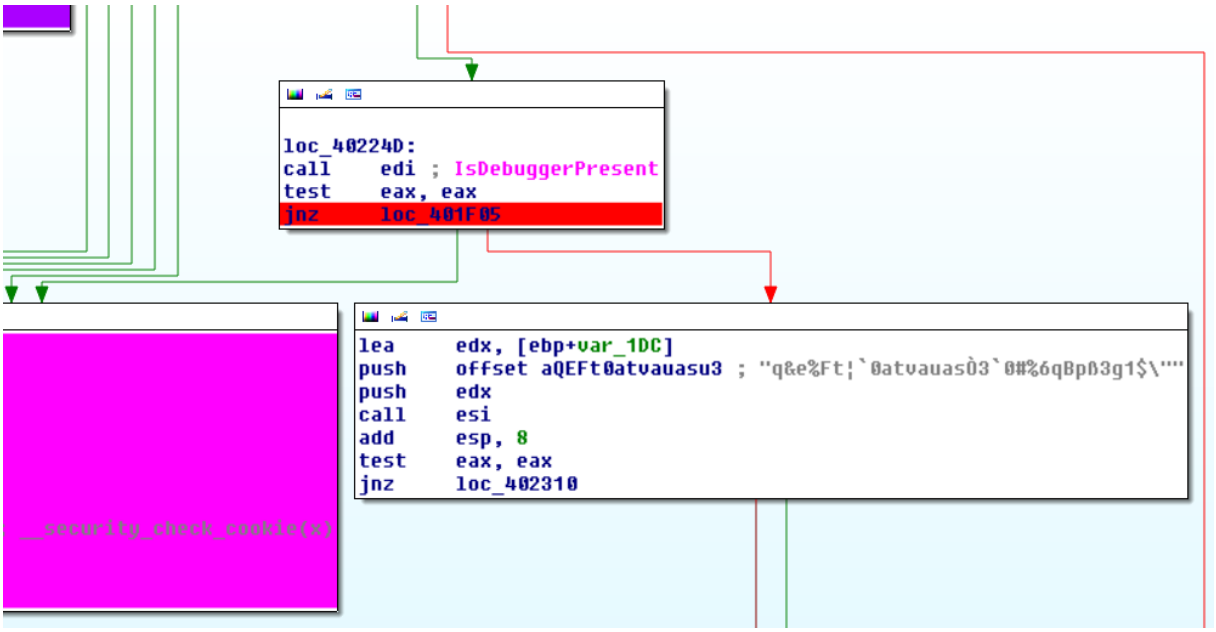
On y arrive et on obtient le message suivant :





Le message final est brouillé (Bravo quelque chose…) et doit donc dépendre en partie de la bonne clé entré en argument.

Mais cela confirme que c'est le bon chemin et pour l'atteindre il faut que toutes les strcmp retourne false sauf un. Ce strcmp compare donc avec la bonne clé chiffrée :



On sait donc que la fonction strcmp est appelé et compare entre edx et q&e%Ft¦`0atvauasÒ3`0#%6qBpß3g1$"

Juste avant edx prend la valeur allant de Stack[00000494]:000000000012FDA0 à 000000000012FDBD (s'arrête avant si valeur = 0)

edx prend donc une valeur qu'on a vu précédemment : [ebp+var_1DC]. C'est celle du premier chiffré calculé par le premier node de chiffrement qu'on a vu précédemment.

On résume alors plusieurs informations sur le chiffrement. Le chiffrement chiffre charactere par charactere et chaque chiffrement de charactere est indépendant et nécessite juste un xor avec une autre valeur et dans une liste qui se répète. Cela implique donc que la clé en claire à la même taille que la clé chiffré (ici 32 characteres). Il n'y a pas non plus d'aléatoire, de IV, ou de nonce…

On peut alors utiliser plusieurs méthodes pour récupérer la clé.

On peut appliquer le chiffrement qui a créé ce chiffré à ce chiffré ce qui nous dévoile le claire.

Pour cela on peut utiliser un scripte python :

```
def main():
    listeChiffrement = ['0x10', '0x14', '0x7', '0x40', '0x23', '0x12',
'0x87', '0x56', '0x6', '2']
    i = 0
    cle = ''
    for char in 'q&e%Ft¦`0atvauasÒ3`0#%6qBpß3g1$"':
        print(hex(int(listeChiffrement[i], 16) ^ int(hex(ord(char)), 16)))
        cle += chr(int(listeChiffrement[i], 16) ^ int(hex(ord(char)), 16))
        i += 1
        if i >= len(listeChiffrement):
            i = 0
    print(cle)
if __name__ == '__main__':
    main()
```

On obtient la possible clé : a2beef!66cdbf5BaUef23111abXea346 mais elle ne fonctionne pas. On regarde alors la clé chiffré et on voit quand on l'entre comme argument qu'i y a des char qui ont

comme valeur des overflow pour un char et donc pas valide pour une base 16. (plusieurs hex correspond à ce char par exemple ¦ peut être égale à 0DDh ou 0B1h…)

```
debug012:000000000342A98 db  2Eh ; .
debug012:000000000342A99 db  65h ; e
debug012:000000000342A9A db  78h ; x
debug012:000000000342A9B db  65h ; e
debug012:000000000342A9C db    0
debug012:000000000342A9D db  71h ; q
debug012:000000000342A9E db  26h ; &
debug012:000000000342A9F db  65h ; e
debug012:000000000342AA0 db  25h ; %
debug012:000000000342AA1 db  46h ; F
debug012:000000000342AA2 db  74h ; t
debug012:000000000342AA3 db 0DDh ; ¦
debug012:000000000342AA4 db  60h ; `
debug012:000000000342AA5 db  30h ; 0
debug012:000000000342AA6 db  61h ; a
debug012:000000000342AA7 db  74h ; t
debug012:000000000342AA8 db  76h ; v
debug012:000000000342AA9 db  61h ; a
debug012:000000000342AAA db  75h ; u
debug012:000000000342AAB db  61h ; a
debug012:000000000342AAC db  73h ; s
debug012:000000000342AAD db 0E3h ; ã
debug012:000000000342AAE db  33h ; 3
debug012:000000000342AAF db  60h ; `
debug012:000000000342AB0 db  30h ; 0
debug012:000000000342AB1 db  23h ; #
debug012:000000000342AB2 db  25h ; %
debug012:000000000342AB3 db  36h ; 6
debug012:000000000342AB4 db  71h ; q
debug012:000000000342AB5 db  42h ; B
debug012:000000000342AB6 db  70h ; p
debug012:000000000342AB7 db 0E1h ; ß
debug012:000000000342AB8 db  33h ; 3
debug012:000000000342AB9 db  67h ; g
debug012:000000000342ABA db  31h ; 1
```

(X veut dire qu'on ne connait pas le charactere correspondant)

Notre clé vaut donc : a2beefX66cdbf5BaXef23111abXea346

Il y a 3 charactere que l'on ne connait pas.

Pour résoudre ce problème le plus simple reste de brute force.

Pour cela on installe le module python exrex ainsi que setuptools afin de l'installer. Cela nécessite de copier-coller ces fichiers dans la VM et de les installer manuellement. Exrex nous permet d'utiliser les expressions régulière et de générer des strings à partir d'une expression régulière.

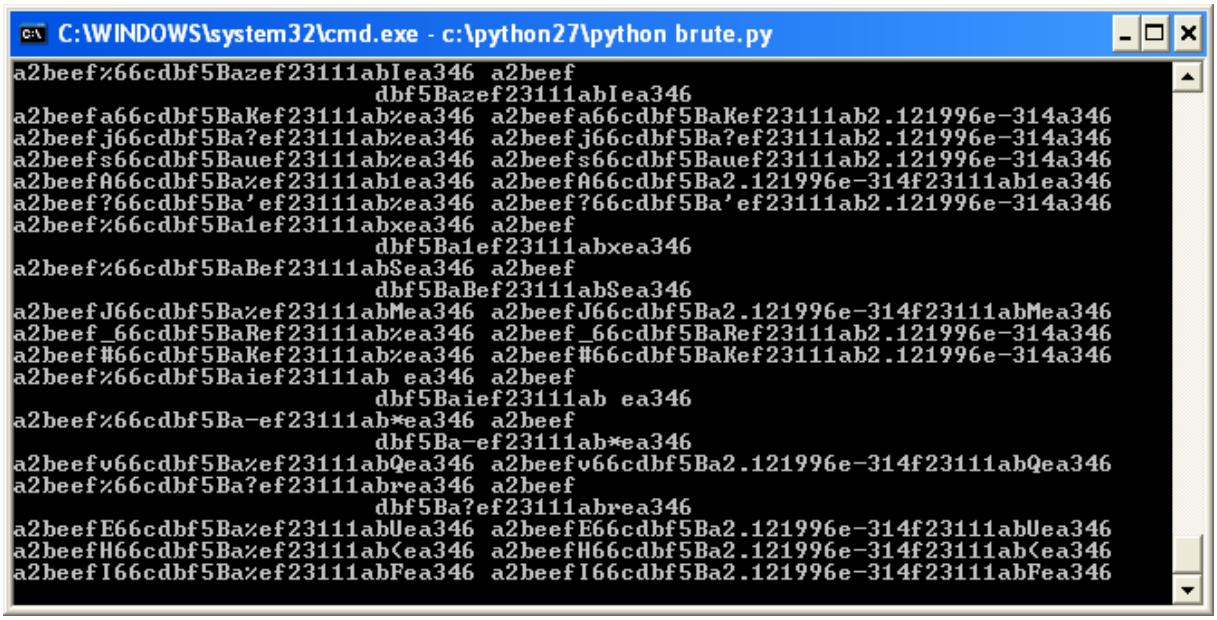
On utilise donc le script python suivant :

```
import subprocess
import exrex


def main():
    while 1:
        key = exrex.getone('a2beef.66cdbf5Ba.ef23111ab.ea346')
        output = subprocess.check_output(['C:\Documents and
Settings\Administrateur\Mes
documents\Malware_THIEMS_PESCE_MARTY\Malware_THIEMS_PESCE_MARTY.exe ',
key])
        if output != key:
            print(key + ' ' + output)
```

```
if __name__ == '__main__':
    main()
```

Le programme nous renvoie beaucoup de fois notre string mais déformé.



On finit par trouver la clé : a2beef666cdbf5Badef23111abfea346

Une autre façon de faire car un brute force avec 256^3 possibilités ça prend quand même un peu de temps, c'est de rentrer une chaine de caractère unique de longueur 32 (par exemple 32 fois 'a'), de regarder le résultat et de noté là où le résultat correspond (par exemple si on rentre 32 fois 'a' qu'il y a un charactere qui correspond ça veut dire qu'a cette indice la clé a 'a').

On essaie ca avec plusieurs characteres afin de réduire au maximum les possibilité d'un possible brute force :

```
clé potentiel   a2beef366cdbf5?adef23111abfea346
clé potentiel   a2beef466cdbf5?adef23111abfea346
clé potentiel   a2beef566cdbf5?adef23111abfea346
clé potentiel   a2beef666cdbf5?adef23111abfea346
clé potentiel   a2beef766cdbf5?adef23111abfea346

clé chiffrée    q&e%Ft¦`0atvauasÒ3`0#%6qBpß3g1$"
pour 32*'f':    vra&Etß0`dvra&Etß0`dvra&Etß0`dvr
pour 32*'e':    uqb%FwÔ3cguqb%FwÔ3cguqb%FwÔ3cguq
pour 32*'d':    tpc$GvÒ2bftpc$GvÒ2bftpc$GvÒ2bftp
pour 32*'c':    swd#@qõ5easwd#@qõ5easwd#@qõ5easw
pour 32*'b':    rve"ApÕ4d`rve"ApÕ4d`rve"ApÕ4d`rv
pour 32*'9':    )->yh+¥o?;)->yh+¥o?;)->yh+¥o?;)-
pour 32*'8':    (,?xh*+n>:(,?xh*+n>:(,?xh*+n>:(,
pour 32*'7':    '#0wh%¦a15'#0wh%¦a15'#0wh%¦a15'#
pour 32*'6':    &"1vh$¦`04&"1vh$¦`04&"1vh$¦`04&"
pour 32*'5':    %!2uh'¦c37%!2uh'¦c37%!2uh'¦c37%!
pour 32*'4':    $h3th&¦b26$h3th&¦b26$h3th&¦b26$h
pour 32*'3':    #'4sh!¦e51#'4sh!¦e51#'4sh!¦e51#'
pour 32*'2':    "&5rhhÁd40"&5rhhÁd40"&5rhhÁd40"&
pour 32*'0':    h$7ph"Àf62h$7ph"Àf62h$7ph"Àf62h$
pour 32*'1':    !%6qh#Âg73!%6qh#Âg73!%6qh#Âg73!%
pour 32*'a':    quf!Bsµ7gcquf!Bsµ7gcquf!Bsµ7gcqu
```

(? veut dire qu'on a pas trouver le charactere correspondant)

On a donc 5 clés potentielles :

a2beef366cdbf5?adef23111abfea346

a2beef466cdbf5?adef23111abfea346

a2beef566cdbf5?adef23111abfea346

a2beef666cdbf5?adef23111abfea346

a2beef766cdbf5?adef23111abfea346

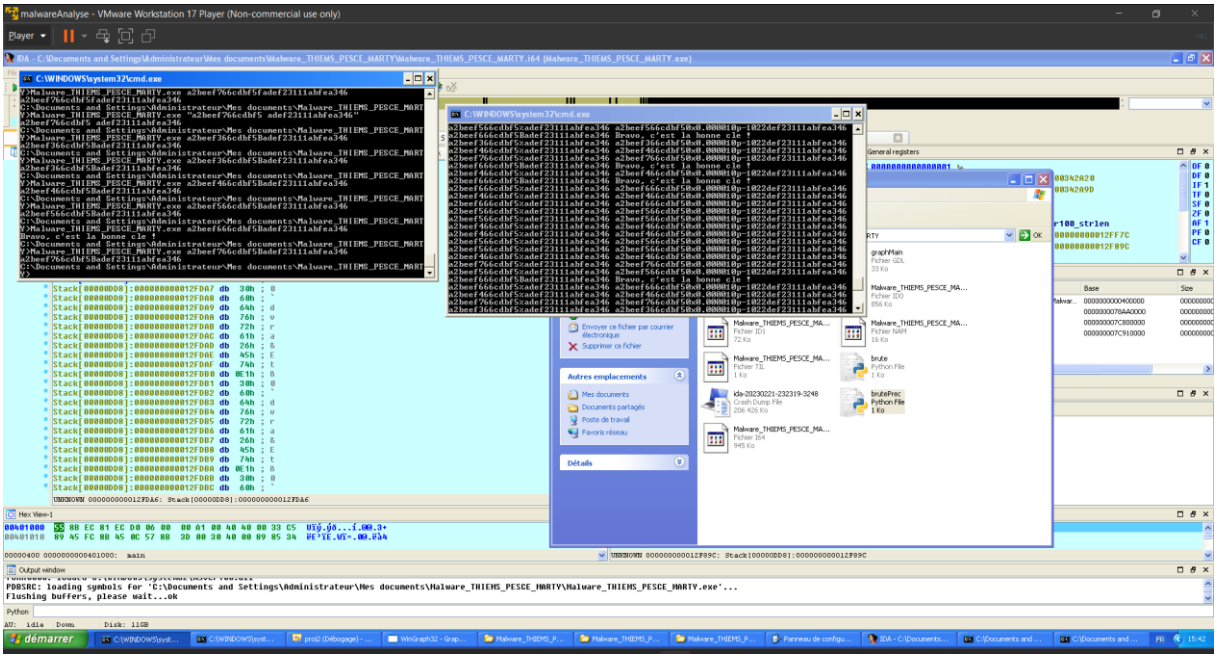
On a qu'un seul charactere à brute force (juste 256 possibilités)

On utilise alors un scripte python pour brute force :

```
import subprocess
import exrex


def main():
    while 1:
        key = exrex.getone('a2beef[3-7]66cdbf5.adef23111abfea346')
        output = subprocess.check_output(['C:\Documents and
Settings\Administrateur\Mes
documents\Malware_THIEMS_PESCE_MARTY\Malware_THIEMS_PESCE_MARTY.exe ',
key])
        if output != key:
            print(key + ' ' + output)

if __name__ == '__main__':
    main()
```



On trouve alors la bonne clé : a2beef666cdbf5Badef23111abfea346

Résumé : techniques repérées dans le malware :

- Anti-debug : utilisation de beaucoup de IsDebuggerPresent() tout le long du malware
- Obfuscation : de fonction en partant de scanf pour les fonctions printf, strcmp et strlen
- Utilisation de plein de variables négatives pour rémonter à partir de ebp
- Utilisation de leurres avec strcmp
- Beaucoup de mov ou d'instructions inutiles pour essayer de brouiller l'analyse
- Chiffrement faible charactere à charactere avec un xor qui ne dépend pas d'un autre charactere, d'un IV, d'un nonce ou de l'aléatoire…