Malware Analysis

(ransomware “policía federal”)

6/6/2013

Autor: Alejandro Torres Ramírez (TorresCrack)
T: @torrescrack248
http://Torrescrack.blogspot.com

Resumen

La muestra analizada es sobre un ransomware que usurpa la identidad de la policía federal. Ransomware es un tipo de malware (software malintencionado) que los criminales instalan en su PC sin su consentimiento y les da a los criminales la capacidad de bloquear su equipo, luego presentará una ventana emergente con un aviso que dice que su ordenador que se encuentra bloqueado y afirma que no podrá acceder al mismo a no ser que pague.

El análisis dinámico con el sandbox Anubis nos devuelve información incompleta ya que el malware hace uso de la API denominada “Sleep”, la cual duerme durante algunos minutos lo cual altera el funcionamiento de Anubis, pues únicamente deja en ejecución el proceso durante poco tiempo y lo cierra por lo tanto no da oportunidad al malware de ejecutarse por completo y de capturar por completo sus funciones.

El malware analizado se copia en diferentes carpetas dentro del sistema, modifica un path para que se ejecute al iniciar Windows, copia y borra algunas llaves del registro encargadas del reinicio “modo seguro” impidiendo el mismo, guarda la copia del contenido real de las mismas con un nombre diferentes.

Utiliza únicamente la encriptación XOR para desencriptar todos o gran parte de los strings y el mismo método lo utiliza para desencriptar una gran parte del código que utiliza en diferentes capas.

Lanza el proceso “svchost.exe” en modo suspendido, el cual es un proceso del sistema que aloja varios servicios de Windows, crea una sección nueva dentro del proceso y coloca dentro de ella otra parte del código del malware, después sobrescribe el “entry point” o punto de entrada que es donde inicia la ejecución el proceso y coloca un salto directo a la sección creada y escrita por el malware para seguir con la ejecución del malware en el proceso inyectado.

Las conexiones cliente – servidor las realiza en dos servidores diferentes en el primer servidor de ucrania únicamente descarga la imagen .jpg que muestra y en el segundo servidor manda la información que se colocó en el espacio donde solicita la clave del pago que se debe realizar.

Se analizaron rápidamente los servidores, encontrando en uno de ellos información de la base de datos SQL en la cual se registraban las IP’s de las víctimas, la información del pago y la fecha en que mando la Informacion.

Tabla de Contenidos

1. Introduccion

1.1 Contexto

Este documento fue creado para dar un ejemplo de un análisis exhaustivo de malware. El documento puede ser compartido a todos los interesados en el mismo.

1.2 Objetivo

El objetivo es hacer un análisis completo de un "ransomware" que usurpa la identidad de la policía federal y entender cómo funciona el malware, controlarlo y si es posible recuperar los daños causados por el mismo.

2. Información del binario

2.1 Ejecutable encriptado

2.1.1 Hash del archivo

SHA256:	0975e128554064c484237aba089153e0fb0200e01a346ea8fe28026c1354f41a
SHA1:	49627b699a37864375ff44098a879e6b71f690e5
MD5:	bf26ea9dc4af592c658b5af66c14c11d

2.1.2 Tipo de Archivo

Win32 EXE executable PE for MS Windows (GUI) Intel 80386 32-bit

2.1.3 Informacion PE

Fecha de compilación: 2013-03-12 14:24:35

EntryPoint: 0x00009590

Secciones:

Name	Virtual address	Virtual size	Raw size	Entropy	MD5
.textRR	4096	591	1024	3.38	57b2ada72a312cd3d1820c15965af89f
.text	8192	35166	35328	7.11	86a7122622ffd7d15502ef964d19a7b6
.data	45056	100	512	0.87	5501f3f560b3455b778fcbbda0df04c6
.rdata3	49152	100	512	0.00	bf619eac0cdf3f68d496ea9344137e8b
.rdata2	53248	100	512	0.00	bf619eac0cdf3f68d496ea9344137e8b

2.2 Informacion Virus Total

Antivirus	Resultado
AhnLab-V3	Trojan/Win32.Blocker
AVG	Generic32.FYN 20130320
BitDefender	Trojan.Generic.KDZ.10799
DrWeb	Trojan.MulDrop4.26873
ESET-NOD32	Win32/LockScreen.AQT
F-Secure	Trojan.Generic.KDZ.10799
GData	Trojan.Generic.KDZ.10799
Kaspersky	Trojan-Ransom.Win32.Blocker.avzn
Kingsoft	Win32.Troj.Undef.(kcloud)
Malwarebytes	Trojan.Ransom
McAfee	PWS-Zbot-FAKU!BF26EA9DC4AF
McAfee-GW-Edition	PWS-Zbot-FAKU!BF26EA9DC4AF
MicroWorld-eScan	Trojan.Generic.KDZ.10799
nProtect	Trojan/W32.Blocker.47896
Panda	Trj/Dtcontx.C
PCTools	HeurEngine.MaliciousPacker
Symantec	Packed.Generic.401
TrendMicro	Mal_Ransom-1
TrendMicro-HouseCall	Mal_Ransom-1
VBA32	Hoax.Blocker.auxi
VIPRE	Trojan.Win32.Reveton.a (v)
ViRobot	Trojan.Win32.Ransom.47896

2.3 Informacion Anubis

- Registry Values Modified:

Key	Name	New Value
HKU\S-1-5-21-842925246-1425521274-308236825- 00\ SOFTWARE\Microsoft\Windows\Current ersion	DNS	C:\ MigAutoPla.exe

MigAutoPlay.exe - File Activities

- File System Control Communication:

File	Control Code	Times
C:\Program Files\Common Files\	0x00090028	1

- Processes Created:

Executable	Command Line
C:\WINDOWS\system32\svchost.exe	

svchost.exe - Registry Activities

- Registry Keys Created:

HKLM\System\CurrentControlSet\Control\SafeBoot\mini

Registry Keys Deleted:

HKLM\System\CurrentControlSet\Control\SafeBoot\Minimal

- Files Deleted:

C:\MigAutoPlay.exe

- Files Created:

C:\Documents and Settings\All Users\Application Data\MigAutoPlay.exe

- Files Read:

C:\Documents and Settings\All Users\Application Data\MigAutoPlay.exe

3. Herramientas

3.1 Preparando el escenario

las herramientas utilizadas para este análisis:

- Olly debugger
- VMware – Windows XP (x86)
- RDG Packer Detector

3.2 Escaneando el binario

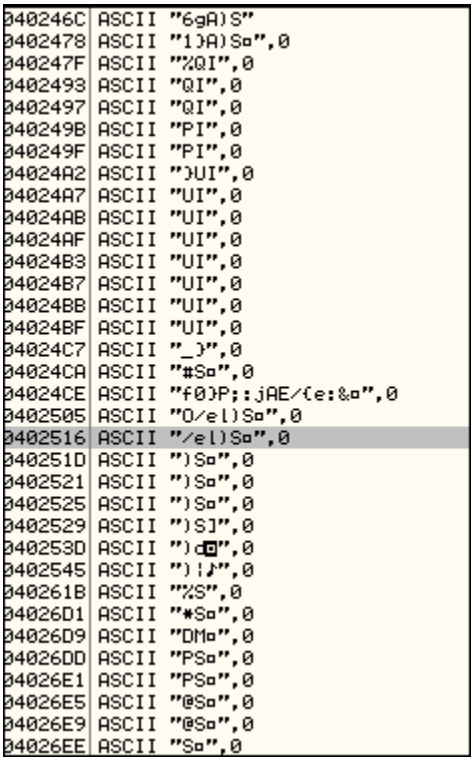
No detecto ningún algoritmo de encriptación pero no hay que descartar que pudiera ocupar algún método que no fue detectado, pero este usa el tipo de encriptación/desencriptación XOR para strings y algo de código.

4. Desempacando

4.1 Primer capa

Al cargarlo con el debugger podemos ir viendo un poco más detallado todo y como va cargando las cosas, bien seguimos.

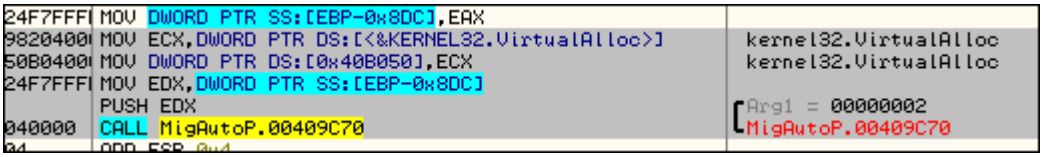
Veamos las strings a ver si hay algo que nos pueda ayudar, nombres de archivos, carpetas o algo que ocupe para hacer destrozos en el sistema veamos:



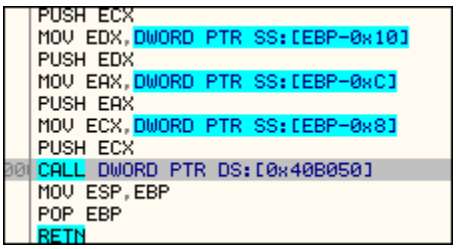
No hay nada, únicamente strings encriptadas, que seguramente deben desencriptarse al momento de utilizarlas, también desde aquí puedo deducir que cargara otra sección de memoria y parte de todo esto es código lo colocara ahí dentro.

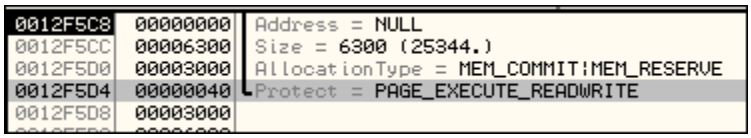
4.1.1 Guardando Datos Encriptados

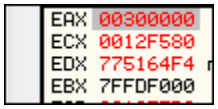
Seguimos viendo directo a las zonas importantes para no alargar demasiado intentado explicar las vueltas que da para llegar a cargar una API, seguimos:



Quiero mostrarles como guarda el offset de *VirtualAlloc* en el buffer y después entra a un call en la dirección 0x409C70 :







Llama a la función *VirtualAlloc* que reserva o asigna una región de páginas del espacio de direcciones virtual del proceso que la invoca.

Su sintaxis es:

Syntax

LPVOID VirtualAlloc(

LPVOID lpAddress, // dirección de la región a reservar o asignar

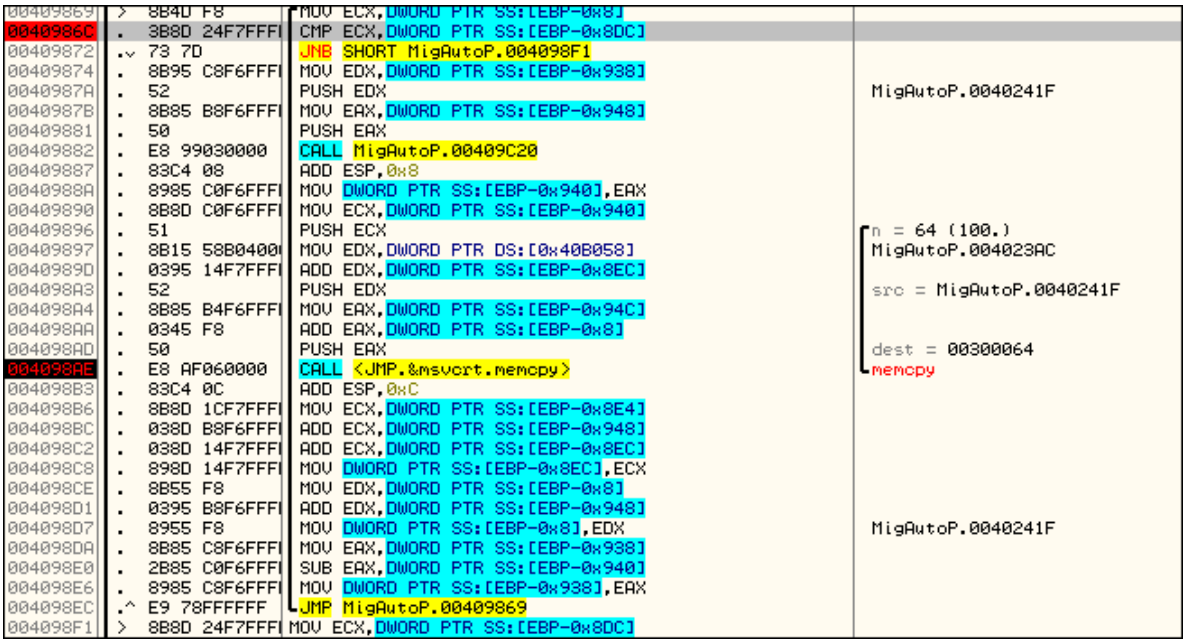
DWORD dwSize, // tamaño de la región

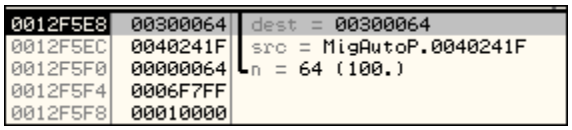
DWORD flAllocationType, // tipo de petición

DWORD flProtect // tipo de protección de acceso

);

Crea una sección de memoria vacía (en la dirección 0x300000) en la cual más adelante meterá algunos datos encriptados, seguimos y al avanzar vemos esta rutina:

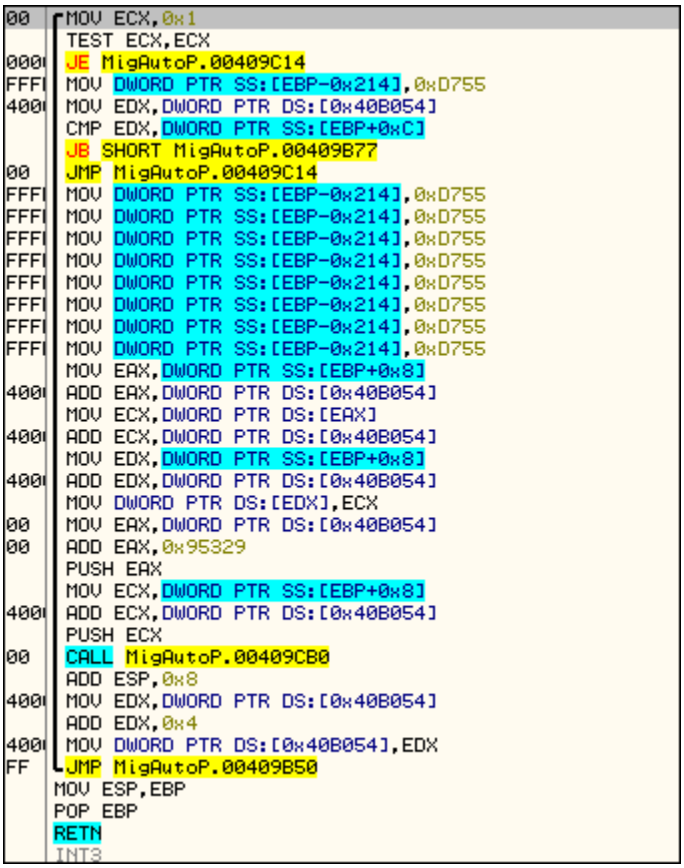




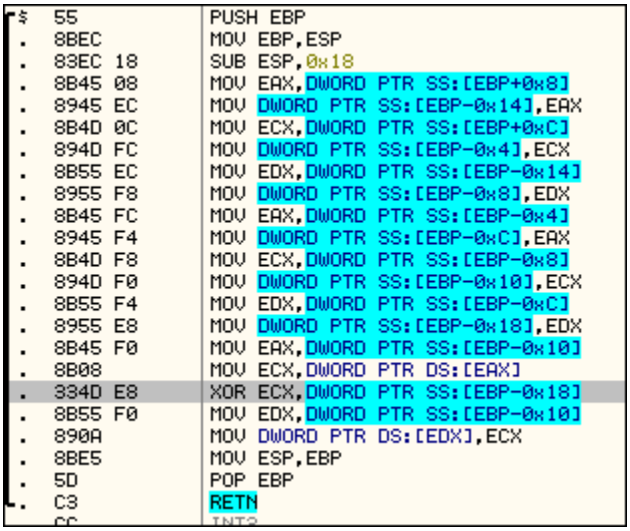
Lo que hace aquí en esta rutina en sencillo, en el offset 0x40986C compara ecx con el tamaño de la sección que creo anteriormente que era de 6300h y de ser igual o mayor te saca de la rutina y si no pasa a copiar el contenido de 0x40241f con ayuda de Memcpy . ***Copia los primeros n caracteres del objeto apuntado por s2 al objeto apuntado por s1.***

Seguimos:

4.1.2 Rutina de Desencriptación



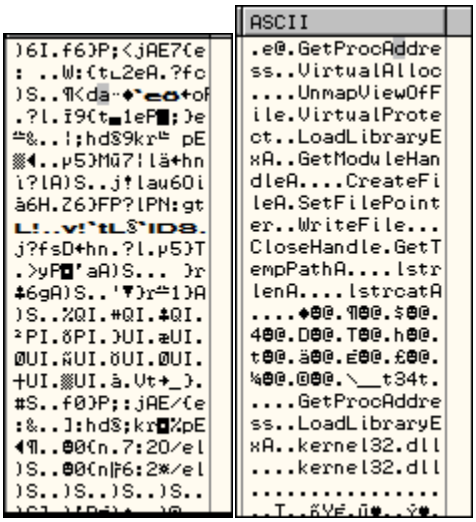
```
00   MOV ECX,0x1
     TEST ECX,ECX
000  JE MigAutoP.00409C14
FFF  MOV DWORD PTR SS:[EBP-0x214],0xD755
400  MOV EDX,DWORD PTR DS:[0x40B054]
     CMP EDX,DWORD PTR SS:[EBP+0xC]
     JB SHORT MigAutoP.00409B77
00   JMP MigAutoP.00409C14
FFF  MOV DWORD PTR SS:[EBP-0x214],0xD755
FFF  MOV DWORD PTR SS:[EBP-0x214],0xD755
FFF  MOV DWORD PTR SS:[EBP-0x214],0xD755
FFF  MOV DWORD PTR SS:[EBP-0x214],0xD755
FFF  MOV DWORD PTR SS:[EBP-0x214],0xD755
FFF  MOV DWORD PTR SS:[EBP-0x214],0xD755
FFF  MOV DWORD PTR SS:[EBP-0x214],0xD755
FFF  MOV DWORD PTR SS:[EBP-0x214],0xD755
     MOV EAX,DWORD PTR SS:[EBP+0x8]
400  ADD EAX,DWORD PTR DS:[0x40B054]
     MOV ECX,DWORD PTR DS:[EAX]
400  ADD ECX,DWORD PTR DS:[0x40B054]
     MOV EDX,DWORD PTR SS:[EBP+0x8]
400  ADD EDX,DWORD PTR DS:[0x40B054]
     MOV DWORD PTR DS:[EDX],ECX
00   MOV EAX,DWORD PTR DS:[0x40B054]
00   ADD EAX,0x95329
     PUSH EAX
     MOV ECX,DWORD PTR SS:[EBP+0x8]
400  ADD ECX,DWORD PTR DS:[0x40B054]
     PUSH ECX
00   CALL MigAutoP.00409CB0
     ADD ESP,0x8
400  MOV EDX,DWORD PTR DS:[0x40B054]
     ADD EDX,0x4
400  MOV DWORD PTR DS:[0x40B054],EDX
FF   JMP MigAutoP.00409B50
     MOV ESP,EBP
     POP EBP
     RETN
     INT3
```



```
$  55          PUSH EBP
.  8BEC        MOV EBP,ESP
.  83EC 18     SUB ESP,0x18
.  8B45 08     MOV EAX,DWORD PTR SS:[EBP+0x8]
.  8945 EC     MOV DWORD PTR SS:[EBP-0x14],EAX
.  8B4D 0C     MOV ECX,DWORD PTR SS:[EBP+0xC]
.  894D FC     MOV DWORD PTR SS:[EBP-0x4],ECX
.  8B55 EC     MOV EDX,DWORD PTR SS:[EBP-0x14]
.  8955 F8     MOV DWORD PTR SS:[EBP-0x8],EDX
.  8B45 FC     MOV EAX,DWORD PTR SS:[EBP-0x4]
.  8945 F4     MOV DWORD PTR SS:[EBP-0xC],EAX
.  8B4D F8     MOV ECX,DWORD PTR SS:[EBP-0x8]
.  894D F0     MOV DWORD PTR SS:[EBP-0x10],ECX
.  8B55 F4     MOV EDX,DWORD PTR SS:[EBP-0xC]
.  8955 E8     MOV DWORD PTR SS:[EBP-0x18],EDX
.  8B45 F0     MOV EAX,DWORD PTR SS:[EBP-0x10]
.  8B08        MOV ECX,DWORD PTR DS:[EAX]
.  334D E8     XOR ECX,DWORD PTR SS:[EBP-0x18]
.  8B55 F0     MOV EDX,DWORD PTR SS:[EBP-0x10]
.  890A        MOV DWORD PTR DS:[EDX],ECX
.  8BE5        MOV ESP,EBP
.  5D          POP EBP
.  C3          RETN
.  CC          INT3
```

Esta rutina (imagen de arriba) hace lo mismo que la anterior un solo que esta va sacando los datos encriptados que guardo anteriormente, va colocando los primeros 4 bytes en ECX (en la imagen de arriba está señalada en gris esa instrucción) y hace un **XOR** con 95329h y los vuelve a colocar en la

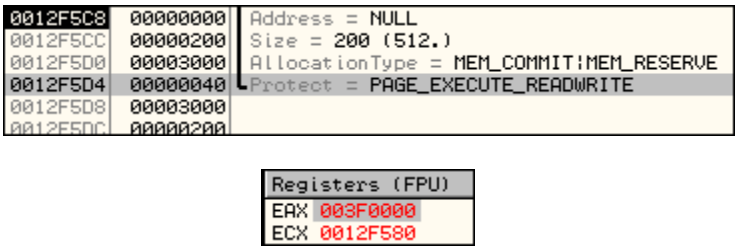
misma dirección donde se encontraban pero ahora ya desencriptados, para encriptar o desencriptar el código solo basta usar el mismo **XOR** (bytes,95329h) ya que vimos cómo funciona y que método usaba para encriptar/desencriptar los datos que copio anteriormente, coloquemos un BreakPoint al final de la rutina y veamos lo que desencripto:



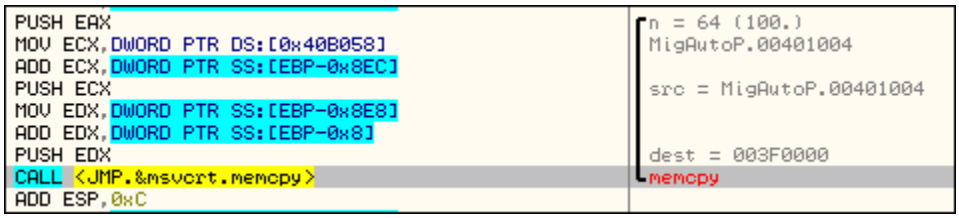
Antes **Después**

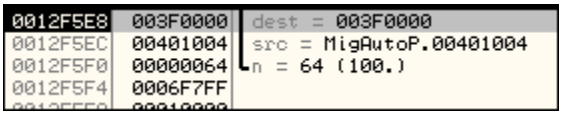
Encontramos algo interesante, son nombres de algunas *APIS* lo cual es muy probable las utilizara en conjunto con *GetProcAddress* para sacar las direcciones de las *APIS* e irlas cargando para poder utilizarlas más adelante así también como código que aún no sabemos para que lo ocupara, seguimos y encontramos esto:

4.1.3 Guardando Datos Encriptados

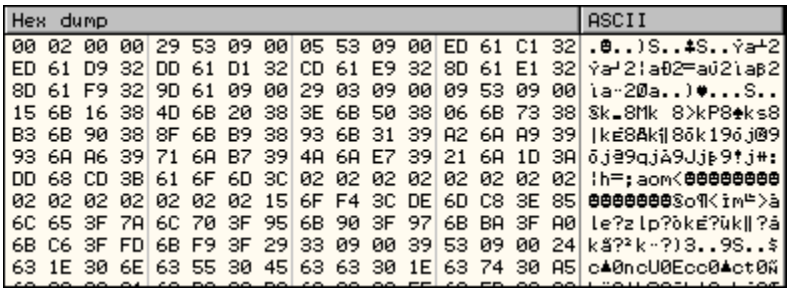


Aquí crea una zona vacía en 0x3F0000 con tamaño de 200h para posteriormente llenarla como vimos anteriormente donde guardo los strings encriptados, seguimos más adelante para ver qué datos guardara en esta sección:



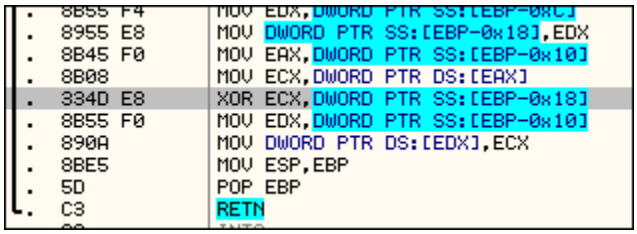


Estamos en la zona donde va a rellenar la sección vacía que creo anteriormente, pero veamos qué datos guardara seguimos avanzando y vemos esto:

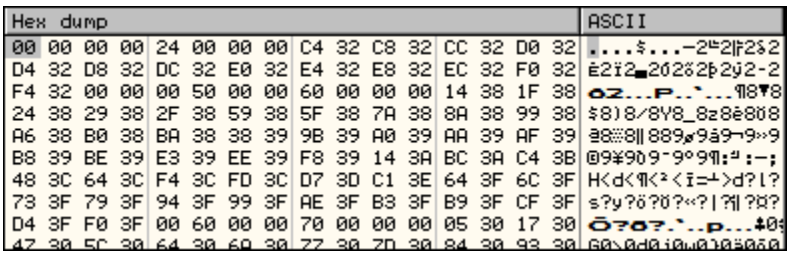


4.1.4 Rutina de Desencriptación

Igual que anteriormente guarda algunos datos que al parecer están encriptados, si seguimos podemos encontrar la rutina que desencriptará y podremos ver el contenido real



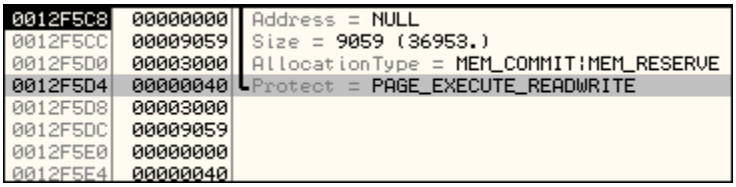
Usa la misma rutina que anteriormente uso para desencriptar los datos anteriores, es la misma forma de encriptación/desencriptación **XOR**, seguimos y al finalizar la rutina y ver el resultado de las operaciones vemos esto:



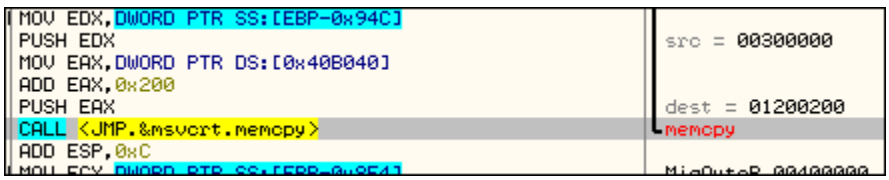
Son los datos desencriptados que a simple viste no tienen sentido pero quizás los ocupe o haga alguna otra operación con estos.

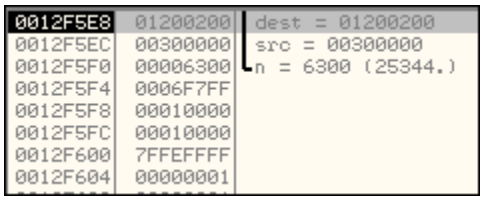
4.1.5 Copiando Datos Desencriptados

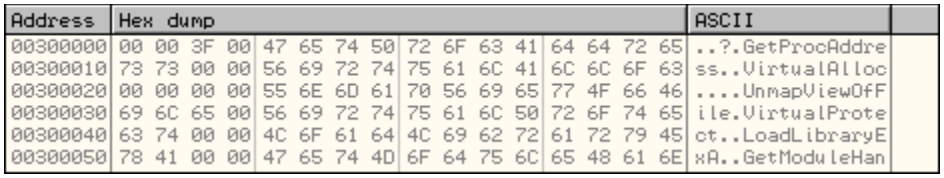
Seguimos:



Está creando otra zona con dirección 0x1200000 y el tamaño de 9059h, avanzamos un poco más para encontrar lo que guarda en esa dirección:

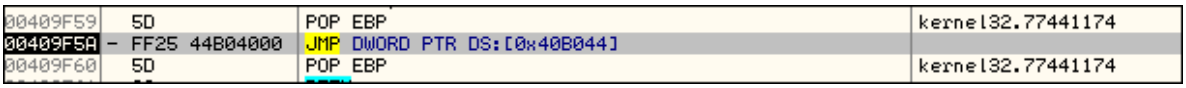


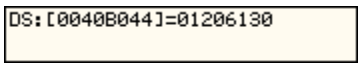




Ahora con *Mempcy* (imagen arriba) rellena la zona que acaba de crear y coloca dentro del zonavacia+200 mete los datos que desencripto que eran los nombres de las *API's* y parte de código que encontramos anteriormente.

Seguimos y un poco más adelante vemos un RET, si lo cruzamos nos encontramos con esta zona algo interesante y es la que nos llevara a la zona con parte del código que desencripto anteriormente:





4.2 Segunda capa

Vamos a esa dirección y empezamos a analizar ahora en la segunda capa de código.

Comenzando con la segunda capa y seguimos analizando:

4.2.1 Obteniendo direcciones API's

```
009459CC  EB 09          JMP SHORT 009459D7
009459CE  8B55 FC        MOV EDX,DWORD PTR SS:[EBP-0x4]
009459D1  83C2 01        ADD EDX,0x1
009459D4  8955 FC        MOV DWORD PTR SS:[EBP-0x4],EDX
009459D7  837D FC 0D     CMP DWORD PTR SS:[EBP-0x4],0xD
009459DB  73 21          JNB SHORT 009459FE
009459DD  8B45 FC        MOV EAX,DWORD PTR SS:[EBP-0x4]
009459E0  8B0C85 C402940 MOV ECX,DWORD PTR DS:[EAX*4+0x9402C4]
009459E7  51             PUSH ECX
009459E8  8B55 F8        MOV EDX,DWORD PTR SS:[EBP-0x8]          kernel32.7C800000
009459EB  52             PUSH EDX
009459EC  FF15 40639400  CALL DWORD PTR DS:[0x946340]            kernel32.GetProcAddress
009459F2  8B4D FC        MOV ECX,DWORD PTR SS:[EBP-0x4]
009459F5  89048D 4063940 MOV DWORD PTR DS:[ECX*4+0x946340],EAX
009459FC  EB D0          JMP SHORT 009459CE
009459FE  8BE5           MOV ESP,EBP
00945A00  5D             POP EBP                                 kernel32.7C800000
```

```
00 00 00 00 00 00 00 00  ........
00 00 00 00 00 00 3F 00  ......?.
47 65 74 50 72 6F 63 41  GetProcA
64 64 72 65 73 73 00 00  ddress..
56 69 72 74 75 61 6C 41  VirtualA
6C 6C 6F 63 00 00 00 00  lloc....
55 6E 6D 61 70 56 69 65  UnmapVie
77 4F 66 46 69 6C 65 00  wOfFile.
56 69 72 74 75 61 6C 50  VirtualP
72 6F 74 65 63 74 00 00  rotect..
4C 6F 61 64 4C 69 62 72  LoadLibr
61 72 79 45 78 41 00 00  aryExA..
47 65 74 4D 6F 64 75 6C  GetModul
65 48 61 6E 64 6C 65 41  eHandleA
00 00 00 00 43 72 65 61  ....Crea
74 65 46 69 6C 65 41 00  teFileA.
53 65 74 46 69 6C 65 50  SetFileP
6F 69 6E 74 65 72 00 00  ointer..
57 72 69 74 65 46 69 6C  WriteFil
65 00 00 00 43 6C 6F 73  e...Clos
65 48 61 6E 64 6C 65 00  eHandle.
47 65 74 54 65 6D 70 50  GetTempP
61 74 68 41 00 00 00 00  athA....
6C 73 74 72 6C 65 6E 41  lstrlenA
00 00 00 00 6C 73 74 72  ....lstr
```

Es muy fácil entender esta rutina en la cual va cargando los nombres de las *API's* y las utiliza en conjunto con *GetProcAdress* para ir obteniendo las direcciones de cada una, haciendo un incremento en *"ecx"* y después por 4 y sumando el offset 0x9402C4 para sacar los nombres poco a poco conforme va incrementando 1 y hace lo mismo de vuelta con el offset 0x946340 para guardar la direcciones devueltas por *GetProcAddress* y las vuelve a acomodar formando una pequeña tabla, que posteriormente va a utilizar, como vemos no tiene demasiadas *API's* por lo tanto a mi parecer sospecho que desencriptará otra capa más adelante y seguirá con su labor, seguimos avanzando para ver si encontramos algo interesante.

4.2.2 Guardando Datos Encriptados

Seguidamente vemos que con la API *VirtualAlloc* nos creara una zona vacía de size 0x5400 con dirección 0x950000 y después pasa por una rutina que no tiene sentido pero que solo coloca 0 en toda la zona que nos creó lo cual como acabamos de crearla siempre estará vacía por lo tanto es una parte que a mi parecer no tiene mucho sentido, (imagen abajo)

```
0946110   55              PUSH EBP
0946111   8BEC            MOV EBP,ESP
0946113   6A 40          PUSH 0x40
0946115   68 00300000    PUSH 0x3000
094611A   8B45 08        MOV EAX,DWORD PTR SS:[EBP+0x8]
094611D   50              PUSH EAX
094611E   6A 00          PUSH 0x0
0946120   FF15 44639400  CALL DWORD PTR DS:[0x946344]     kernel32.VirtualAlloc
0946126   5D              POP EBP
0946127   C3              RETN
0946128   CC              INT3
0946129   CC              INT3
094612A   CC              INT3
```

```
0012FF80  00000000  Address = NULL
0012FF84  00005400  Size = 5400 (21504.)
0012FF88  00003000  AllocationType = MEM_COMMIT|MEM_RESERVE
0012FF8C  00000040  Protect = PAGE_EXECUTE_READWRITE
0012FF90  0012FFC0
0012FF94  0094621D  RETURN to 0094621D from 00946110
0012FF98  00005400
```

```
Registers (FPU)
EAX 00950000
ECX 7C809B49 kernel32.7C809B49
```

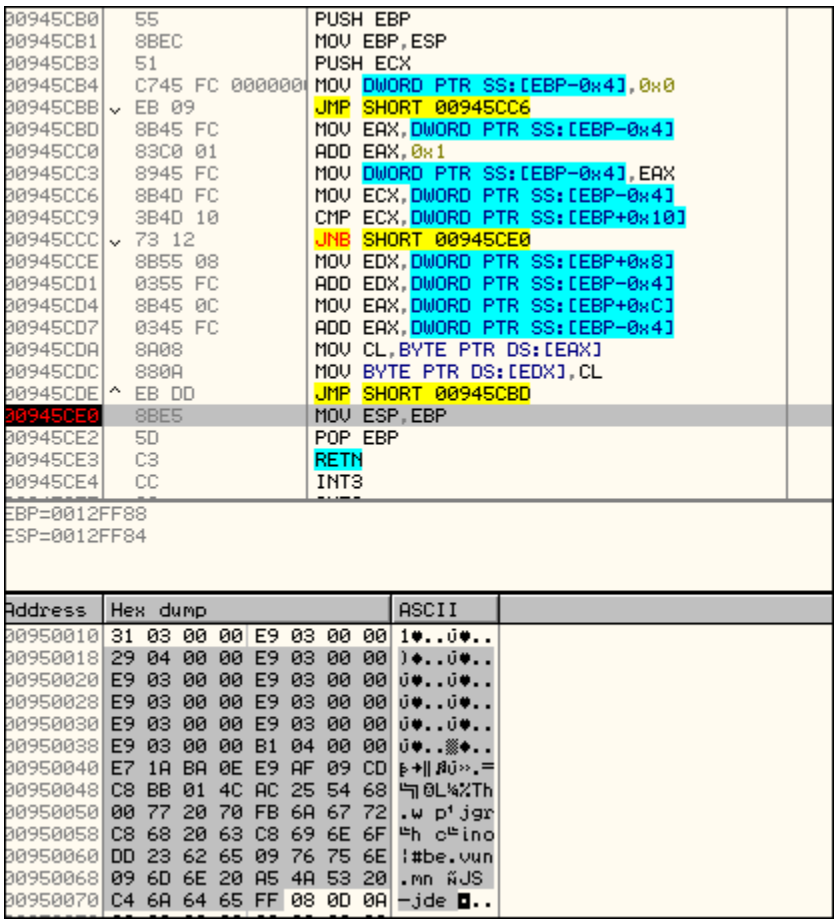
SECCION QUE PONE A 0 O LIMPIA EL CONTENIDO DE LA NUEVA SECCION

```
5A34   C745 FC 000000(  MOV DWORD PTR SS:[EBP-0x4],0x0
5A3B   EB 09            JMP SHORT 00945A46
5A3D   8B45 FC          MOV EAX,DWORD PTR SS:[EBP-0x4]
5A40   83C0 01          ADD EAX,0x1
5A43   8945 FC          MOV DWORD PTR SS:[EBP-0x4],EAX
5A46   8B4D FC          MOV ECX,DWORD PTR SS:[EBP-0x4]
5A49   3B4D 10          CMP ECX,DWORD PTR SS:[EBP+0x10]
5A4C   73 0D            JNB SHORT 00945A5B
5A4E   8B55 08          MOV EDX,DWORD PTR SS:[EBP+0x8]
5A51   0355 FC          ADD EDX,DWORD PTR SS:[EBP-0x4]
5A54   8A45 0C          MOV AL,BYTE PTR SS:[EBP+0xC]
5A57   8802             MOV BYTE PTR DS:[EDX],AL
5A59   EB E2            JMP SHORT 00945A3D
```

Entramos como a una pequeña rutina en la cual si “eax”es mayor al size 5400 entonces nos saca fuera de la rutina, lo cual si seguimos avanzando podremos saber con exactitud el contenido que guarda y que movimientos hace:

```
623D   8955 E4          MOV DWORD PTR SS:[EBP-0x1C],EDX
6240   8B45 FC          MOV EAX,DWORD PTR SS:[EBP-0x4]
6243   3B45 F0          CMP EAX,DWORD PTR SS:[EBP-0x10]
6246   73 53            JNB SHORT 0094629B
6248   8B4D E4          MOV ECX,DWORD PTR SS:[EBP-0x1C]
624B   51               PUSH ECX
624C   8B55 DC          MOV EDX,DWORD PTR SS:[EBP-0x24]
624F   52               PUSH EDX
6250   E8 0BFCFFFF      CALL 00945E60
6255   83C4 08          ADD ESP,0x8
6258   8945 E0          MOV DWORD PTR SS:[EBP-0x20],EAX
625B   8B45 E0          MOV EAX,DWORD PTR SS:[EBP-0x20]
625E   50               PUSH EAX
625F   8B0D A8649400    MOV ECX,DWORD PTR DS:[0x9464A8]
6265   034D E8          ADD ECX,DWORD PTR SS:[EBP-0x18]
6268   51               PUSH ECX
6269   8B15 88639400    MOV EDX,DWORD PTR DS:[0x946388]
```

Seguimos avanzando dentro y podemos encontrar lo siguiente:



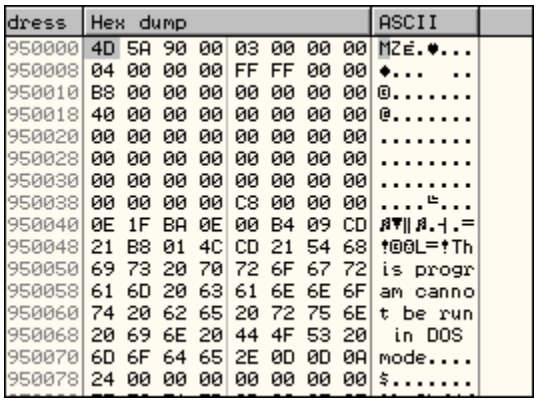
Esta rutina nos guarda datos encriptados, en específico solo guarda 78 bytes y los coloca dentro de la sección que anteriormente fue creada, si nos fijamos bien podemos observar que pareciera ser el comienzo o parte de la cabecera de un .exe (un poco ofuscada por la encriptación), seguirá guardan de 78 bytes hasta llenar por completo la zona o que eax valga 5400 que es el tamaño de la nueva zona creada anteriormente, seguimos avanzando hasta que podamos encontrar alguna rutina que desencripte esto y lograr ver si el contenido lo ocupara par algún destrozo, hasta ahora no tenemos al parecer el código que hace cambios en el sistema, sigamos..

4.2.3 Rutina de Desencriptación

Bien seguidamente llegamos a la zona que nos desencripta lo que guardo:

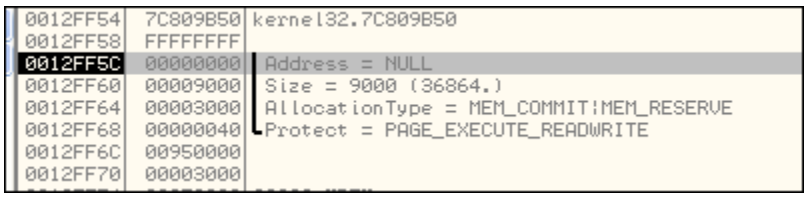
```
009462A4   8B45 F8          MOV EAX,DWORD PTR SS:[EBP-0x8]
009462A7   83C0 04          ADD EAX,0x4
009462AA   8945 F8          MOV DWORD PTR SS:[EBP-0x8],EAX
009462AD   8B4D F8          MOV ECX,DWORD PTR SS:[EBP-0x8]
009462B0   3B4D F0          CMP ECX,DWORD PTR SS:[EBP-0x10]
009462B3   73 39            JNB SHORT 009462EE
009462B5   8B15 88639400    MOV EDX,DWORD PTR DS:[0x946388]
009462BB   0355 F8          ADD EDX,DWORD PTR SS:[EBP-0x8]
009462BE   8B02             MOV EAX,DWORD PTR DS:[EDX]
009462C0   0345 F8          ADD EAX,DWORD PTR SS:[EBP-0x8]
009462C3   8B0D 88639400    MOV ECX,DWORD PTR DS:[0x946388]
009462C9   034D F8          ADD ECX,DWORD PTR SS:[EBP-0x8]
009462CC   8901             MOV DWORD PTR DS:[ECX],EAX
009462CE   8B55 F8          MOV EDX,DWORD PTR SS:[EBP-0x8]
009462D1   81C2 E9030000    ADD EDX,0x3E9
009462D7   A1 88639400      MOV EAX,DWORD PTR DS:[0x946388]
009462DC   0345 F8          ADD EAX,DWORD PTR SS:[EBP-0x8]
009462DF   3310             XOR EDX,DWORD PTR DS:[EAX]
009462E1   8B0D 88639400    MOV ECX,DWORD PTR DS:[0x946388]
009462E7   034D F8          ADD ECX,DWORD PTR SS:[EBP-0x8]
009462EA   8911             MOV DWORD PTR DS:[ECX],EDX
009462EC   EB B6            JMP SHORT 009462A4
009462EE   E8 BDF7FFFF      CALL 00945AB0
009462F3   8B15 88639400    MOV EDX,DWORD PTR DS:[0x946388]
```

En esta rutina lo único que hace es tomar es sacar de la dirección donde están los bytes encriptados y sacar de 4 bytes (word) y los coloca en “eax” y después le suma 8 y los vuelve a guardar el resultado en la zona de donde lo saco , después a 8 le suma 0x3EA y al resultado 0x3F1 le hace un **XOR** con el contenido de “eax” que actualmente contendría el word que esta desencriptando, el resultado lo contiene edx que posteriormente serían los 4 bytes desencriptados y los coloca en la misma dirección de donde saco los encriptados, lo cual si revertimos estas mimas operación con restas y utilizando **XOR** podríamos así encriptar otra vez los datos , por lo tanto esta sería fácil la rutina de encriptación/desencriptación, si vemos los datos que quedaron al final podemos observar un binario limpio:

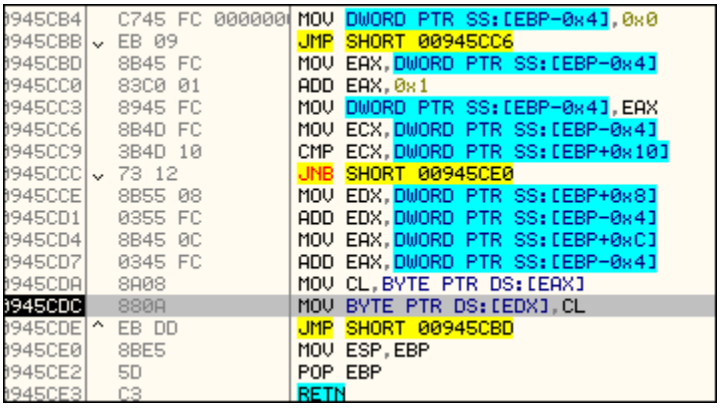


4.2.4 Copiando datos desencriptados

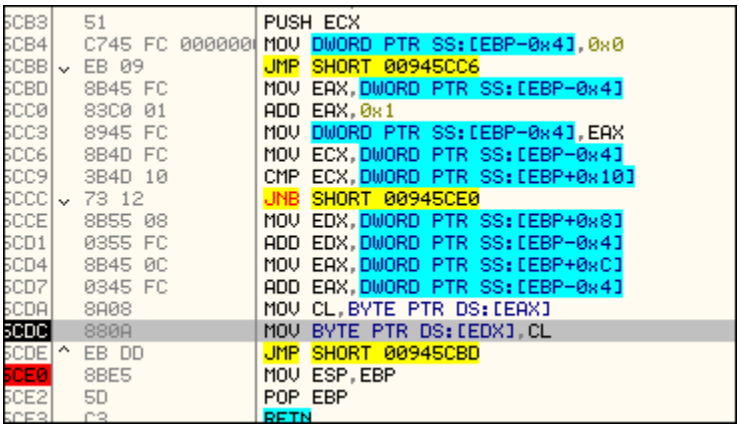
Adelante nos encontramos con otro *VirtualAlloc* que nos creara otra zona pero esta vez de un tamaño diferente 0x9000 con dirección en 0x960000



Ahora sigamos hasta encontrar lo que guardara en la zona vacía, hasta que llegamos a aquí:

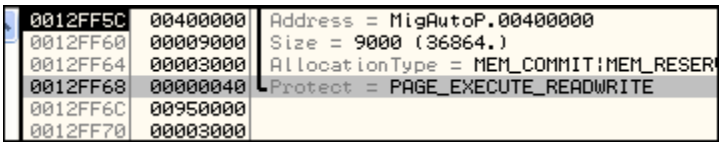


En esa rutina que muestro en la imagen de arriba, lo que hace es que mete 400 bytes de la zona que anteriormente desencripto y los mete en esta nueva zona que acaba de crear en la dirección 0x960000, al parecer solo guarda los datos de la cabecera sin guardar mas parte del código, seguimos...

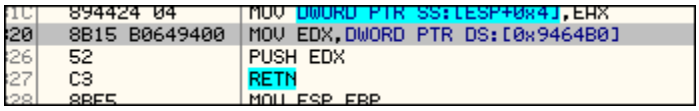


En la misma rutina colocara otros 4400 bytes y empieza a colocar únicamente desde la sección .code del binario anterior, localizo la sección *.code* y la copia en esta nueva sección en la dirección 961000, después vuelve a entrar en esta misma rutina y guarda la sección *.rdata* con el size de 0x600 en la dirección 0x966000, después la sección *.data* con de tamaño 200 en la dirección 0x967000, y sigue con la sección *.reloc* de tamaño 400 en la dirección 0x968000,

Seguimos avanzando y podemos ver que crea otra sección con *VirtualAlloc* pero esta ocasión ocupa la dirección 0x400000 así que pone a 0 toda esa sección con el tamaño de 9000.



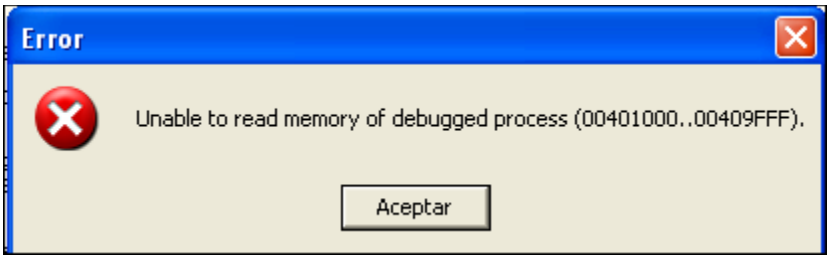
Más adelante podemos ver que utiliza la misma rutina que utilizo para guardar las secciones, la utiliza para poder copiar todo el contenido a de la dirección 0x966000 a la dirección 0x400000 sobrescribiendo su propia cabecera y ahora este mismo binario tiene una cabecera nueva con contenido nuevo.



Saca el entry point y lo coloca en “edx” y hace un “push” al entry point de la zona que acaba de copiar en 0x400000 y pasa por un ret , por lo cual estaríamos ya por caer en la tercer capa.

4.3 Tercer capa

Sigamos más adelante y veamos que más tiene, pero en caso de perdernos o de reiniciar para eso ya habíamos hecho un dumped atrás con esa misma información por lo tanto ya tenemos el binario que sospecho es el que se encarga de hacer los destrozos, busquemos strings para ver algo que nos ayude a saber si tenemos mayores referencias y vemos



Nos muestra un error y no podemos ver los strings, pero intentemos con el dumped que teníamos y vemos:

ress	Disassembly	Text string
01C33	PUSH dumped1.00406000	ASCII "%s\MigAutoPlay.exe"
01CF1	MOV ESI,dumped1.004061C8	ASCII "PheeragIrefvba\Cbyvpvrf\Flfgrz"
01D01	PUSH dumped1.004061B4	ASCII "SOFTWARE\MICROSOFT\"
01D85	PUSH dumped1.004061A8	ASCII "EnableLUA"
01DCF	PUSH dumped1.00406020	ASCII "user32.dll"
01DDF	PUSH dumped1.00406014	ASCII "wsprintfA"
01E0F	PUSH dumped1.004061F4	ASCII """%s\MigAutoPlay.exe""
01E26	MOV ESI,dumped1.0040612C	ASCII "FBSGJNER\Zvpebfbsg\Jvaqbjf\PheeragIrefvba\Eha"
01E76	PUSH dumped1.00406120	ASCII "MigAutoPlay"
01EDF	PUSH dumped1.00406020	ASCII "user32.dll"
01EEF	PUSH dumped1.00406014	ASCII "wsprintfA"
01F1F	PUSH dumped1.004061F4	ASCII """%s\MigAutoPlay.exe""
01F36	MOV ESI,dumped1.0040612C	ASCII "FBSGJNER\Zvpebfbsg\Jvaqbjf\PheeragIrefvba\Eha"
01F86	PUSH dumped1.00406120	ASCII "MigAutoPlay"
0253B	PUSH dumped1.00406258	ASCII "Ole32.dll"
02552	PUSH dumped1.00406248	ASCII "OleAut32.dll"
02562	PUSH dumped1.00406234	ASCII "OleSavePictureFile"
02576	PUSH dumped1.0040621C	ASCII "CreateStreamOnHGlobal"
02590	PUSH dumped1.0040620C	ASCII "OleLoadPicture"
0317E	PUSH dumped1.00406270	ASCII "GET"
0322E	PUSH dumped1.00406270	ASCII "GET"
032ED	PUSH dumped1.00406298	ASCII "Urlmon.dll"
03300	PUSH dumped1.00406284	ASCII "URLDownloadToFileA"
0331B	PUSH dumped1.00406020	ASCII "user32.dll"
0332B	PUSH dumped1.00406014	ASCII "wsprintfA"
0335B	PUSH dumped1.00406278	ASCII "%s\1.jpg"
033FE	PUSH dumped1.004062A4	ASCII "google.com"
03410	PUSH dumped1.004062A4	ASCII "google.com"
03425	PUSH dumped1.004062A4	ASCII "google.com"
03BC0	PUSH EBP	(Initial CPU selection)
03BFF	PUSH dumped1.004062CC	ASCII "ZwAllocateVirtualMemory"

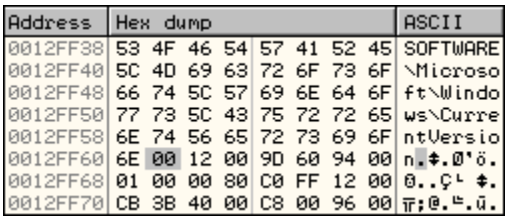
Bien vemos que tenemos al bicho después de todo esto lo hemos sacado, ahora falta analizar lo que hace este, ya que a simple vista se pueden ver algunos strings aun encriptados, algunas URL's y demás cosas, seguimos poco a poco, y empezamos a analizar este bicho por donde estábamos...

4.3.1 Modificación en el registro de Windows

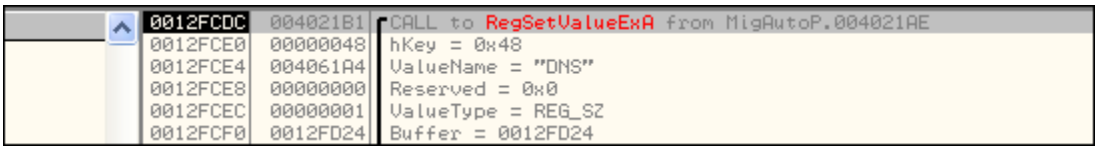
Seguimos analizando y encontramos otra rutina en la cual empezara a desencriptar con XOR algunos strings para posteriormente usarlos:

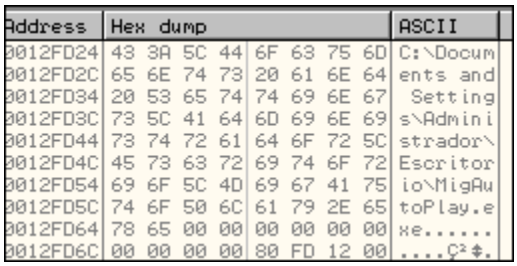
```
Registers (FPU)
EAX 00000046
ECX 0012FF38 ASCII "FBSGJNER\Zvpebfbsg\Jvaqbjf\PheeragIrefvba"
EDX 0012FF38 ASCII "FBSGJNER\Zvpebfbsg\Jvaqbjf\PheeragIrefvba"
EBX 7FFD6000
ESP 0012FD00
EBP 0012FD04
ESI 004061A2 MigAutoP.004061A2
EDI 0012FF62
```

Toma el primer digito y le resta 0x34, después duplica el valor del operando en el registro con CDQ, idiv, le suma 0x41, y hace lo mismo con el resto de los caracteres hasta obtener un string desencriptado:



Seguimos avanzando hasta ver donde qué datos guardara en el registro pues aquí esta desencriptando la ruta para posteriormente guardar en esa dirección:

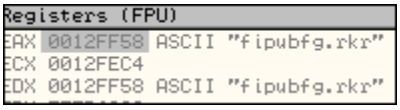


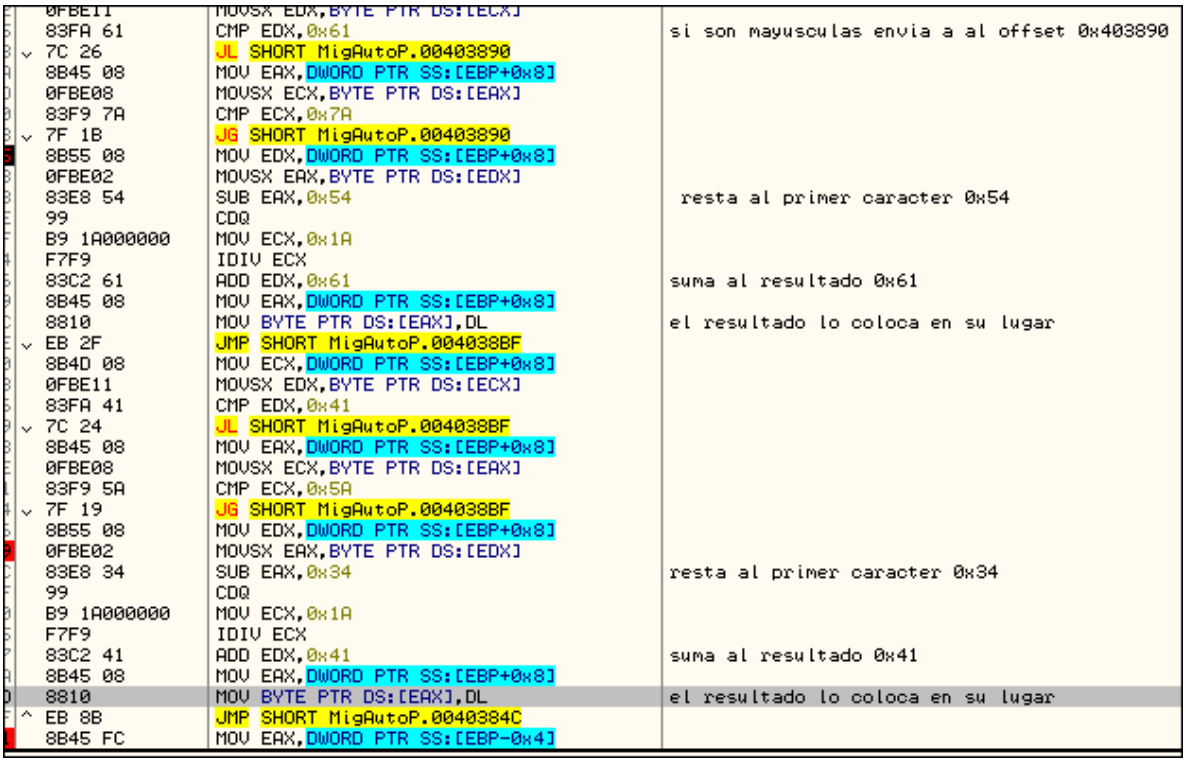


Esta guardando un path en HKCU software\Microsoft\Windows\currentversion, el path lo obtuvo con GetModuleFileName el cual devuelve el path de donde fue ejecutado el proceso para posteriormente guardarlo en esa llave del registro, seguimos

4.3.2 Rutina de Desencriptación

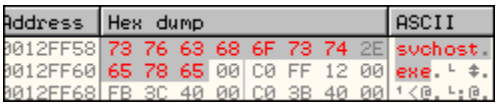
Seguimos avanzando poco a poco y llegamos a una rutina algo similar pero con diferentes operaciones para el proceso de desencriptación de un string esta vez algo pequeño





```
0FBE11      MOVSX EDX,BYTE PTR DS:[ECX]
83FA 61     CMP EDX,0x61                    si son mayusculas envia a al offset 0x403890
7C 26       JL SHORT MigAutoP.00403890
8B45 08     MOV EAX,DWORD PTR SS:[EBP+0x8]
0FBE08      MOVSX ECX,BYTE PTR DS:[EAX]
83F9 7A     CMP ECX,0x7A
7F 1B       JG SHORT MigAutoP.00403890
8B55 08     MOV EDX,DWORD PTR SS:[EBP+0x8]
0FBE02      MOVSX EAX,BYTE PTR DS:[EDX]
83E8 54     SUB EAX,0x54                     resta al primer caracter 0x54
99          CDQ
B9 1A000000 MOV ECX,0x1A
F7F9        IDIV ECX
83C2 61     ADD EDX,0x61                    suma al resultado 0x61
8B45 08     MOV EAX,DWORD PTR SS:[EBP+0x8]
8810        MOV BYTE PTR DS:[EAX],DL        el resultado lo coloca en su lugar
EB 2F       JMP SHORT MigAutoP.004038BF
8B4D 08     MOV ECX,DWORD PTR SS:[EBP+0x8]
0FBE11      MOVSX EDX,BYTE PTR DS:[ECX]
83FA 41     CMP EDX,0x41
7C 24       JL SHORT MigAutoP.004038BF
8B45 08     MOV EAX,DWORD PTR SS:[EBP+0x8]
0FBE08      MOVSX ECX,BYTE PTR DS:[EAX]
83F9 5A     CMP ECX,0x5A
7F 19       JG SHORT MigAutoP.004038BF
8B55 08     MOV EDX,DWORD PTR SS:[EBP+0x8]
0FBE02      MOVSX EAX,BYTE PTR DS:[EDX]
83E8 34     SUB EAX,0x34                    resta al primer caracter 0x34
99          CDQ
B9 1A000000 MOV ECX,0x1A
F7F9        IDIV ECX
83C2 41     ADD EDX,0x41                    suma al resultado 0x41
8B45 08     MOV EAX,DWORD PTR SS:[EBP+0x8]
8810        MOV BYTE PTR DS:[EAX],DL        el resultado lo coloca en su lugar
EB 8B       JMP SHORT MigAutoP.0040384C
8B45 FC     MOV EAX,DWORD PTR SS:[EBP-0x4]
```

En esta rutina (imagen arriba) podemos observar que toma cada carácter y compara de la “A” hasta la “Z” si es mayúscula o minúscula , de acuerdo a la a eso hace diferentes operaciones, en este caso todas son minúsculas así que las operaciones que realiza como muestro en la imagen seria resta al primer carácter 0x54 y al resultado le suma 0x61, el cual vuelve a colocar en su lugar y hace lo mismo con los próximos caracteres, si vamos al final de la rutina podremos ver que es lo que desencripto:

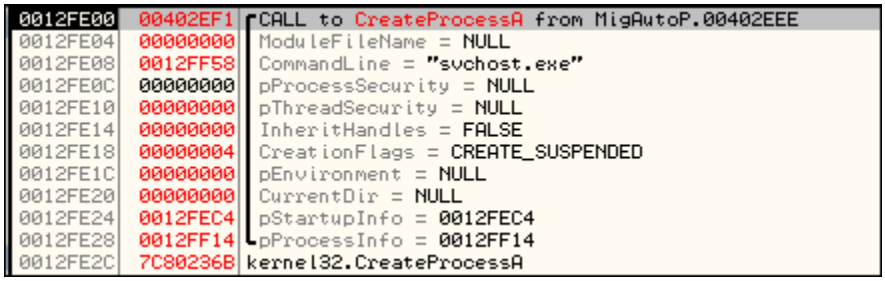


Address	Hex dump	ASCII
0012FF58	73 76 63 68 6F 73 74 2E	svchost.
0012FF60	65 78 65 00 C0 FF 12 00	exe.└ $.
0012FF68	FB 3C 40 00 C0 3B 40 00	¹<@.└;@.

4.3.3 Creación del proceso suspendido

Finalmente desencripto “svchost.exe”, el cual es un proceso del sistema que aloja varios servicios de Windows, posiblemente adelante inyectara el proceso o algo similar, seguimos avanzando...

Encontramos algo interesante y es la API que se encargara de crear el proceso svchost = CreateProcessA, aquí es algo importante para poder seguirle el paso y saber qué es lo que hará con él, veamos:



Hay tenemos los parámetro que utiliza y lo que es de gran importancia es conocer el PID que es el identificador del proceso que creara en modo suspendido, es importante conocerlo para poder seguir el paso a lo que hace en caso de que inyecte código en el mismo, en pProcessInfo nos retornara información de acuerdo al Process_Information structure en el tercer dword tendremos a nuestro identificador del proceso el cual nos devolverá en hexadecimal:

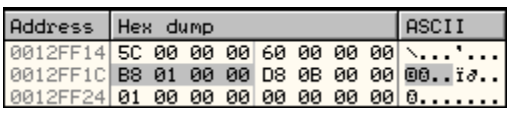
typedef struct _PROCESS_INFORMATION {

HANDLE hProcess;

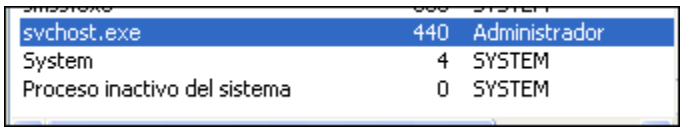
HANDLE hThread;

DWORD dwProcessId;

DWORD dwThreadId;



El valor del identificador es 01B8h = 440d :



4.3.4 Detección del Sistema Operativo

Más adelante invoca a GetversionEx lo cual recupera información del sistema operativo,

La estructura es la siguiente:

typedef struct _OSVERSIONINFO {

DWORD dwOSVersionInfoSize;

DWORD dwMajorVersion;

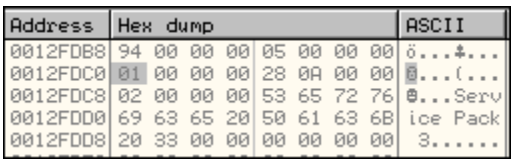
DWORD dwMinorVersion;

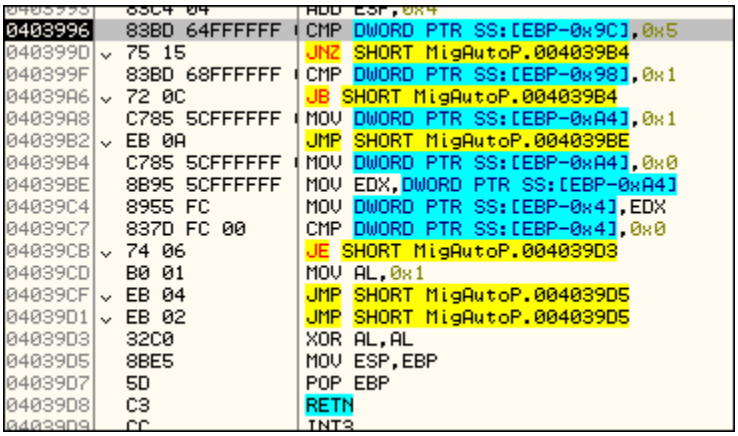
DWORD dwBuildNumber;

DWORD dwPlatformId;

TCHAR szCSDVersion[128];

} OSVERSIONINFO;





hace una comparación con dwMajorVersion sea igual a 5 y dwMinorVersion sea igual a 1 , (imagen arriba) para saber si el sistema operativo es Windows XP y de ser así sale de la rutina con eax valiendo 1

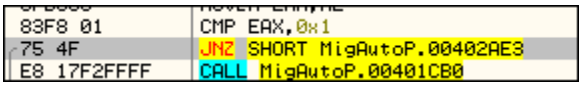
Ejemplo:

.if dwMajorVersion ==5 Then

.if dwMinorVersion == 1 Then

Version = Windows XP"

Return = True



Así podrá decidir qué camino tomar, en este caso al ser Windows XP toma un camino el cual seguiremos avanzando posteriormente haremos pruebas con Windows 7, seguimos

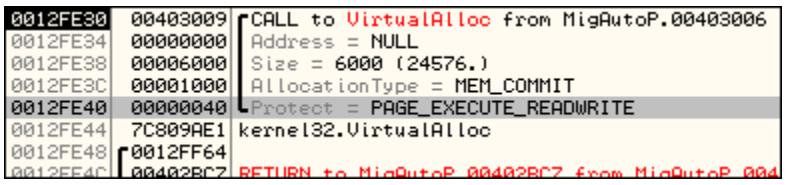
Con Windows 7:

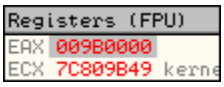
Salta a una dirección de memoria lo cual trata de evitar pasar por la API zwSetInformationProcess que ocasionaría un DoS del sistema en alguna versiones de Windows 7

4.3.5 Inyectando en el proceso suspendido

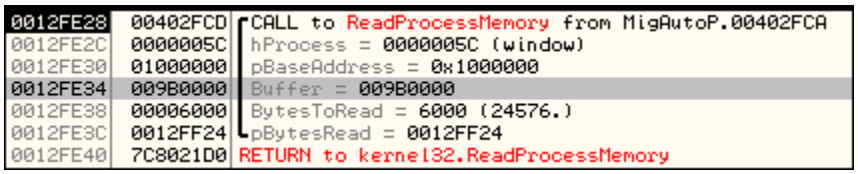
Seguimos Windows XP:

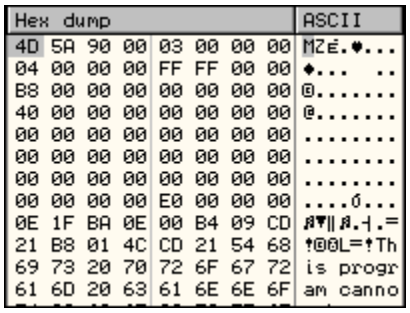
Después de un rato seguir avanzando y pasar por diferentes API's que verificaban si el usuario estaba dentro del grupo administradores y ajustar los permisos, encontramos una zona donde utiliza VirtualAlloc para reservar un espacio vacío en memoria de size 0x6000 en la dirección 0x9B0000:





Después con ReadProcessMemory guarda contenido en esa dirección de memoria creada, en la cual guarda la información a partir del BaseAddress 0x1000000



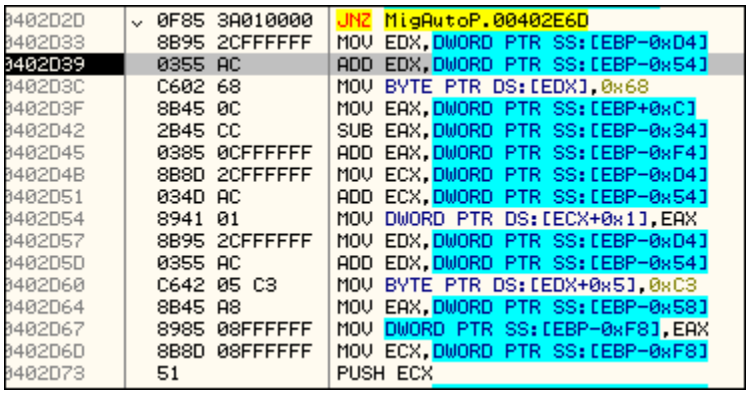


Más adelante Crea una sección con algunas API's de la librería .ntdll *ZwCreateSection* y posteriormente la lee con *ZwMapViewOfsection*, creando un mapa de memoria en la dirección

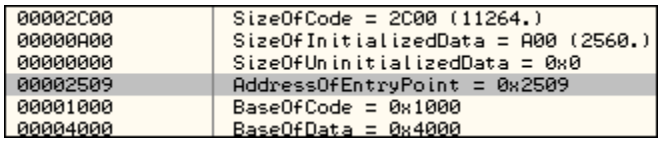
0x9c0000 el cual lo rellenara con *RtlMoveMemory* y copiara su propio contenido, es decir, 0x400000 , al crear el mapa en memoria cualquier cambio que se le haga a la memoria mapeada se verá reflejada en el contenido real del proceso , por lo tanto el proceso ya esta inyectado.

4.3.6 Modificando el entrypoint

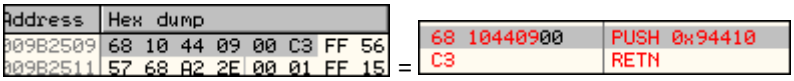
Seguimos avanzando podemos encontrar esta zona y es muy importante:



Lo que hace aquí es llegar a la dirección donde se encuentra ubicado el EntryPoint del proceso suspendido de acuerdo a la información de la cabecera es "*AdressOfEntryPoint*" = 0x2509



Lo cual hace la suma a la dirección mapeada y ubicar el entrypoint que sobrescribirá con una redirección:



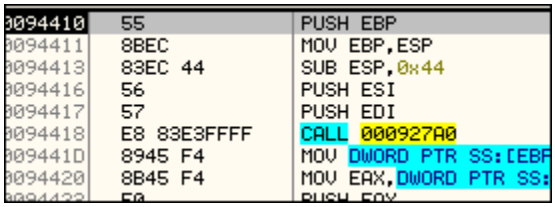
Bien ya tenemos el EntryPoint que sobrescribe, lo cual es muy importante ya que en adelante solo se esperara algún *ResumeThread* para que continúe la ejecución del proceso inyectado, por lo tanto modificaremos algunos bytes de tal manera que se ejecute un loop infinito con el cual nos dé tiempo y podamos adjuntar el proceso y poder seguir la ejecución del malware en el proceso inyectado, por lo tanto quedaría algo así:

Con eso basta para poder hacer el loop, seguimos avanzado y encontramos el *ResumeThread* pero ya tenemos modificado el EntryPoint así que no hay problema se quedara corriendo el loop, si avanzamos un poco mas llegamos a un ExitProcess y fin del proceso.

5 Proceso inyectado

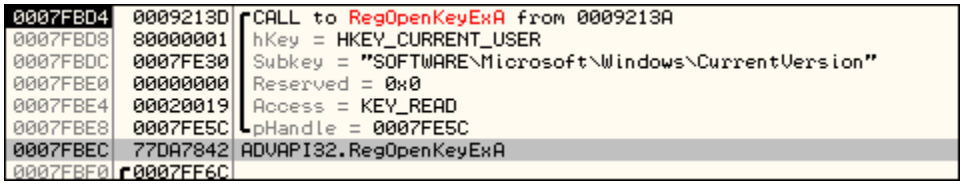
Adjuntamos ahora el proceso y ponemos pausa a la ejecución y caemos en el loop que colocamos:

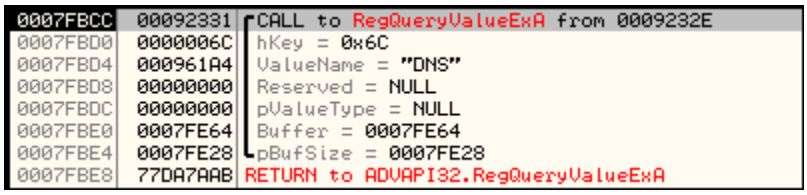
Colocamos el push 0x94410 que el malware quería escribir y llegamos al entrypoint del injerto que hizo el malware:



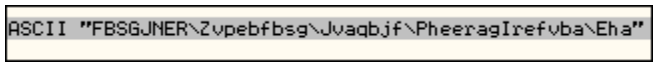
5.1 Modificación en el registro de Windows

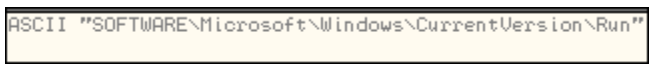
Seguimos analizando a partir del entrypoint y avanzamos, un poco más adelante podremos encontrarnos una rutina que desencriptará otro string, la rutina es similar a la anteriormente mostrada comparando si son mayúsculas o minúsculas y a partir de ahí hace las operaciones XOR correspondientes, y después ocupara el string desencriptado para usarlo con *RegOpenKey* y retirar los valores que anteriormente había guardado



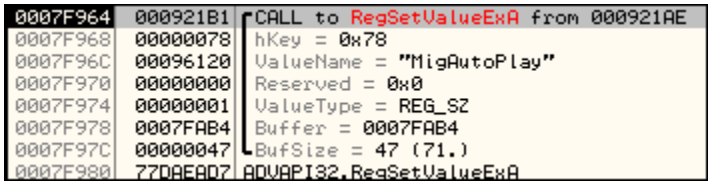


Verifica su existencia con CreateFile y como no encuentra nada, sigue su ejecución pasando primero a desencriptar otro string:



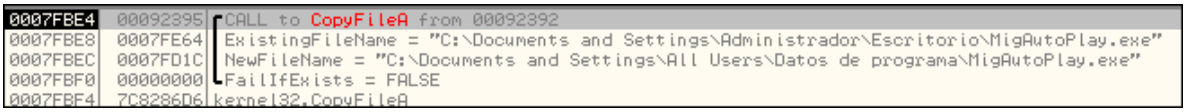


El cual mismo se establecerá en la llave de registro encargada de cargar los programas para iniciar con Windows

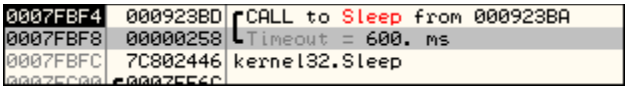


5.2 Copiando y eliminando Ficheros

Guarda el path obtenido por *SHGetSpecialFolderPath* y al cual le concateno “MigAutoPlay.exe”, en esa dirección aún no se encuentra ningún ejecutable con ese nombre así que debe copiarlo:



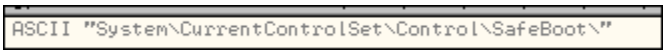
Pasando esto, invoca a “sleep” con el parámetro 600 ms



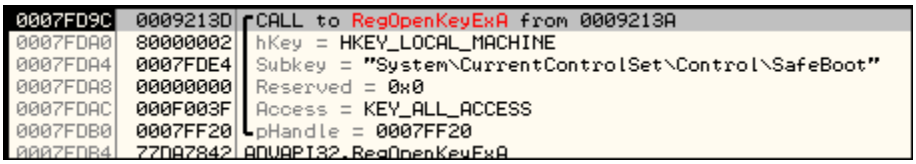
Seguido elimina el binario con *DeleteFile*, lo cual es la muestra principal de donde se había empezado a analizar.

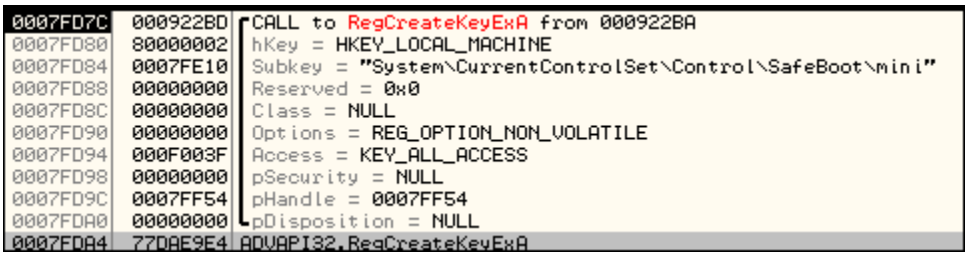
5.3 Copiando y eliminando llaves del registro de Windows

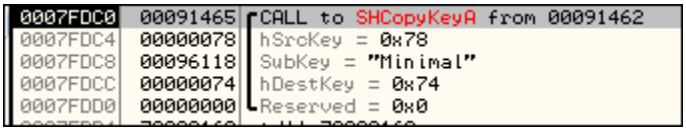
Poco más adelante desencripta otro string con la misma rutina que ha utilizado:

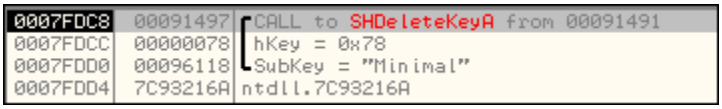


En esa llave se encuentran las configuraciones para entrar en “Modo Seguro”, este es otro de los movimientos importantes que hará en el sistema, si seguimos vemos:







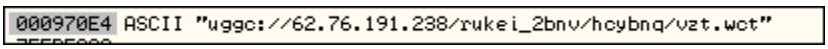


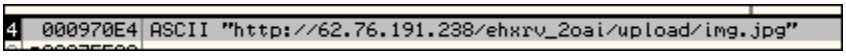
Bien como muestro en las imágenes de arriba crea una entrada nueva en el registro con el nombre de “System\CurrentControlSet\Control\SafeBoot\ mini” y copia hay dentro el contenido de “System\CurrentControlSet\Control\SafeBoot\Minimal” para posteriormente borrarla.

Seguimos y hace las mismas operaciones con “System\CurrentControlSet\Control\SafeBoot\NetWork” copiándolo a “System\CurrentControlSet\Control\SafeBoot\net” y posteriormente eliminando la llave, estas son muy importantes ya que son las que se encargan de arrancar el sistema en "*Modo seguro*" normal y con funciones de red.

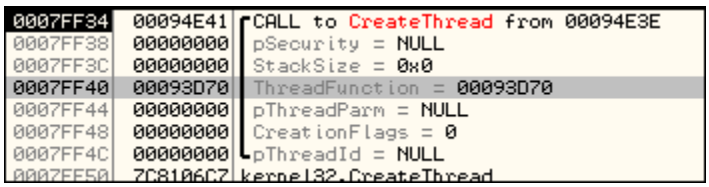
5.4 Desencriptando datos de conexión la red

Seguimos avanzando y ahora desencriptará con la misma rutina que ha venido desencriptando strings, lo que parece ser una dirección URLURL de una imagen:





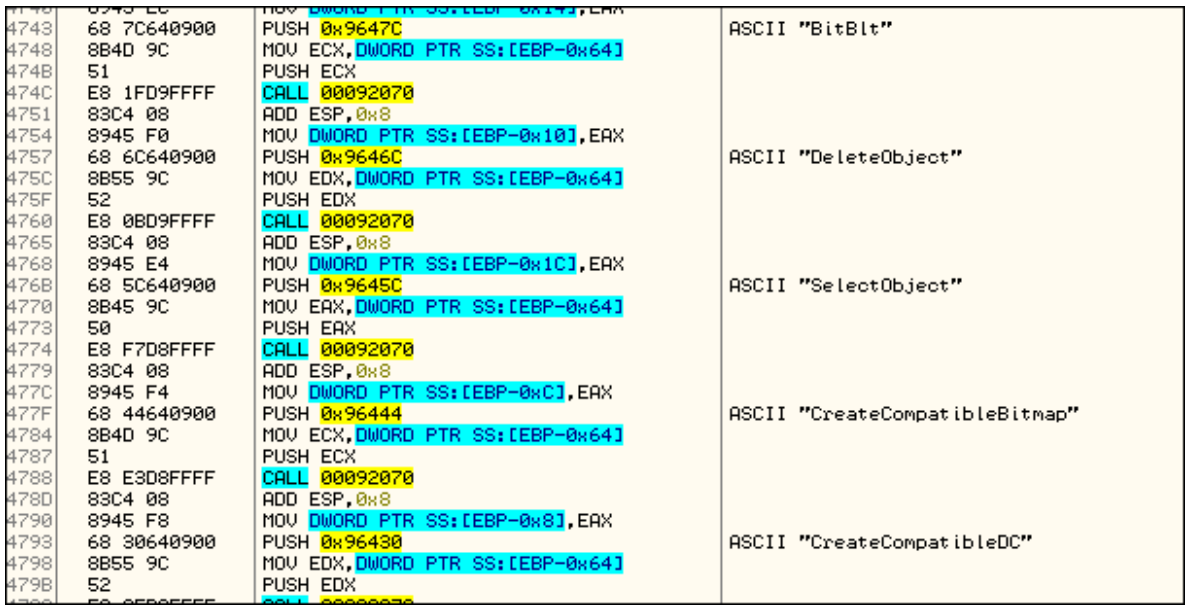
Mas adelante empezara con la creación de un Thread en la dirección 093D70



Sigue su ejecución normal e invoca un Sleep con el parámetro 600000ms = 600 segundos, si esperamos a que pasa por esta instrucción, entonces entra la función del Thread en la cual lo que hace es hacer una serie de comprobaciones en los registros o los datos que guardo anteriormente.

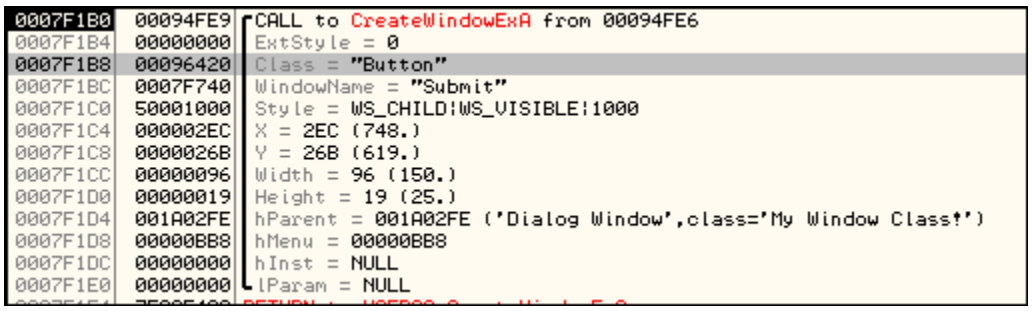
5.5 Estableciendo controles de la ventana de dialogo

Siguiendo el camino por el que íbamos empieza a crear la ventana con CreateWindowExA lo cual cambiamos los parámetros para que el width height sea de 0x30 y podamos manipular en caso de que se interponga en la pantalla, seguidamente empieza a cargar nombres de algunas APIS para poder usarlas con *GetProcAdress.*



```
4743   68 7C640900      PUSH 0x9647C                    ASCII "BitBlt"
4748   8B4D 9C          MOV ECX,DWORD PTR SS:[EBP-0x64]
474B   51               PUSH ECX
474C   E8 1FD9FFFF      CALL 00092070
4751   83C4 08          ADD ESP,0x8
4754   8945 F0          MOV DWORD PTR SS:[EBP-0x10],EAX
4757   68 6C640900      PUSH 0x9646C                    ASCII "DeleteObject"
475C   8B55 9C          MOV EDX,DWORD PTR SS:[EBP-0x64]
475F   52               PUSH EDX
4760   E8 0BD9FFFF      CALL 00092070
4765   83C4 08          ADD ESP,0x8
4768   8945 E4          MOV DWORD PTR SS:[EBP-0x1C],EAX
476B   68 5C640900      PUSH 0x9645C                    ASCII "SelectObject"
4770   8B45 9C          MOV EAX,DWORD PTR SS:[EBP-0x64]
4773   50               PUSH EAX
4774   E8 F7D8FFFF      CALL 00092070
4779   83C4 08          ADD ESP,0x8
477C   8945 F4          MOV DWORD PTR SS:[EBP-0xC],EAX
477F   68 44640900      PUSH 0x96444                    ASCII "CreateCompatibleBitmap"
4784   8B4D 9C          MOV ECX,DWORD PTR SS:[EBP-0x64]
4787   51               PUSH ECX
4788   E8 E3D8FFFF      CALL 00092070
478D   83C4 08          ADD ESP,0x8
4790   8945 F8          MOV DWORD PTR SS:[EBP-0x8],EAX
4793   68 30640900      PUSH 0x96430                    ASCII "CreateCompatibleDC"
4798   8B55 9C          MOV EDX,DWORD PTR SS:[EBP-0x64]
479B   52               PUSH EDX
```

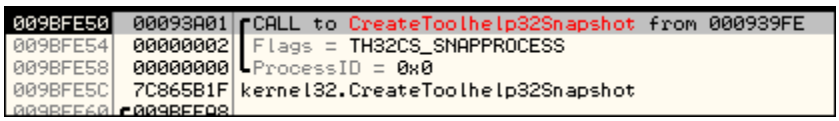
Seguidamente a crear los botones:



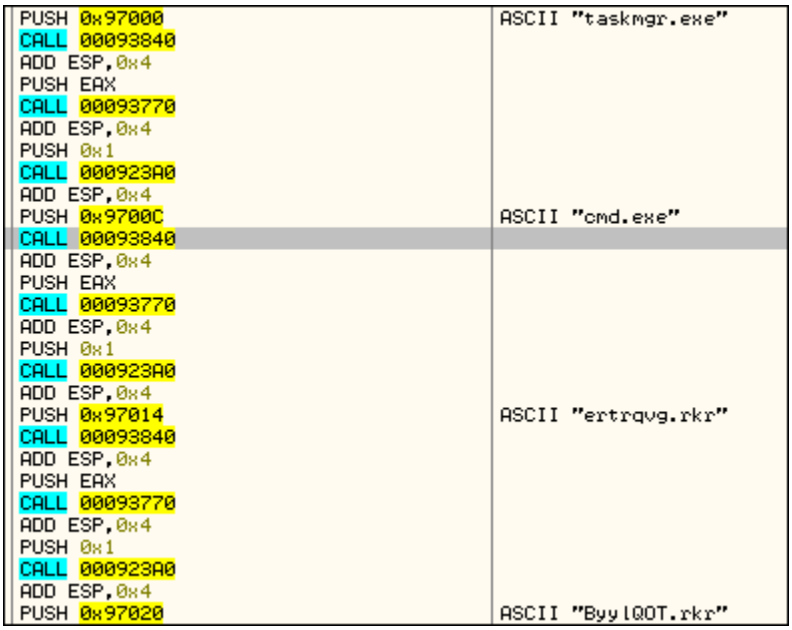
```
0007F1B0   00094FE9   CALL to CreateWindowExA from 00094FE6
0007F1B4   00000000   ExtStyle = 0
0007F1B8   00096420   Class = "Button"
0007F1BC   0007F740   WindowName = "Submit"
0007F1C0   50001000   Style = WS_CHILD|WS_VISIBLE|1000
0007F1C4   000002EC   X = 2EC (748.)
0007F1C8   0000026B   Y = 26B (619.)
0007F1CC   00000096   Width = 96 (150.)
0007F1D0   00000019   Height = 19 (25.)
0007F1D4   001A02FE   hParent = 001A02FE ('Dialog Window',class='My Window Class!')
0007F1D8   00000BB8   hMenu = 00000BB8
0007F1DC   00000000   hInst = NULL
0007F1E0   00000000   lParam = NULL
```

5.6 Monitor y cierre de procesos en el sistema

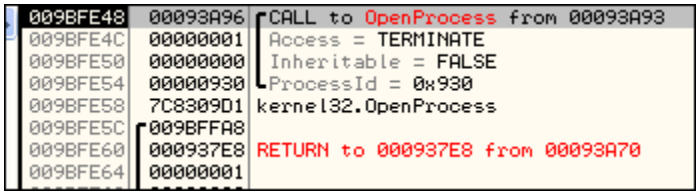
Crea un snapshot de los procesos corriendo actualmente:

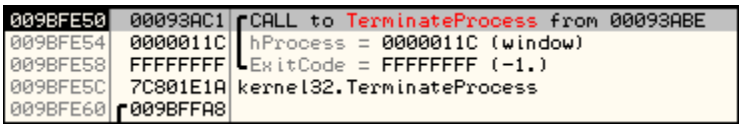


Seguido empieza a desencriptar de nuevo algunos strings que si nos damos cuenta son algunas cosas que seguramente impedirá ejecutar en el sistema, por ejemplo:



Conforme va desencriptando tiene una rutina en la cual va cargando proceso por proceso hasta encontrarse con él y en caso de encontrarlo lo termina:





6 Palabras finales

Hasta aquí llego humildemente la primer parte de este análisis de malware, en próximos días se publicara la segunda parte en donde se incluirá a detalle las conexiones remotas, el análisis de los servidores con los resultados obtenidos con la base de datos, las conclusiones finales y la posible solución.

7 Agradecimientos

Debo agradecer a: Nahuel Riva (NCR+/CRC!), Christian Navarrete (Chr1x), Erick (+Erisoft)

Por leer el análisis y darme feedback para mejorar las cosas.

8 Contacto

Twitter: https://twitter.com/TorresCrack248

e-mail: tora_248@hotmail.com

Blog: http://torrescrack.blogspot.com