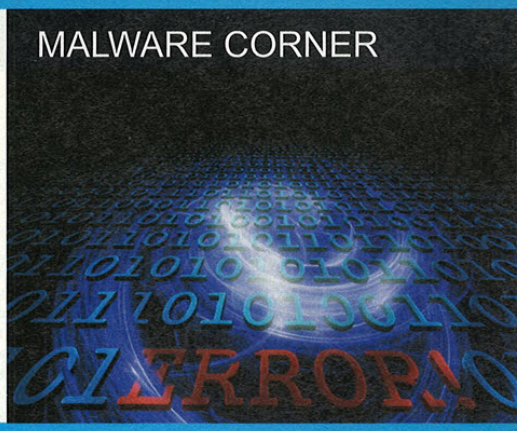
VOTRE MBR PRIS EN OTAGE !

Nicolas Brulez – nicolas.brulez@kaspersky.fr
Senior Malware Researcher
Global Research and Analysis Team – Kaspersky Lab



mots-clés : CODES MALICIEUX / REVERSE ENGINEERING / RANSOMWARE / ANALYSE DE CODE / MBR / BOOT / MD5 : 1E7A4A518C91432C816917BD14AB323B

Les Ransomwares s'appuient sur de la cryptographie - du chiffrement - pour empêcher l'utilisation d'une machine. Ils existent depuis plusieurs années. En général, il s'agit d'une application demandant l'envoi d'un SMS surtaxé pour obtenir un code de déblocage (MISC 47, p. 14 à 17) à entrer sous Windows. Cette fois-ci, le blocage s'effectue bien avant le démarrage de l'OS, directement dans le MBR (Master Boot Record).

1 Analyse du malware

Lors de l'analyse de notre *malware*, nous allons rencontrer rapidement des indices qui nous dirigent vers une infection du MBR. En effet, la première sous-routine effectue la requête WQL (*WMI Query Language*) comme suivante :

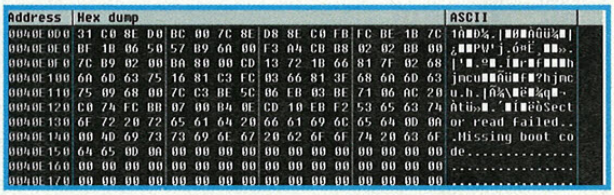
```
SELECT * FROM Win32_DiskPartition Where BootPartition = true
```

Cela permet à notre malware de récupérer la partition bootable.

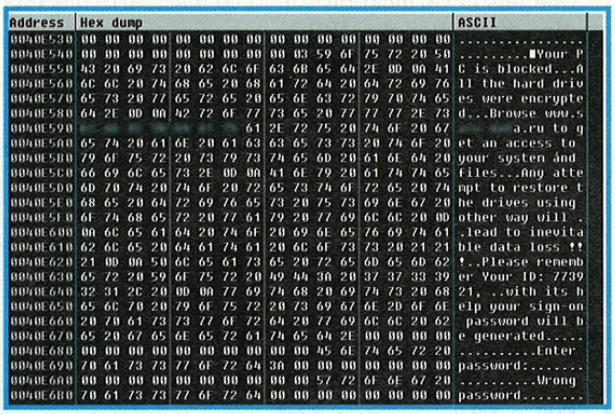
Le ransomware accède ensuite aux ressources de l'exécutable pour récupérer le code à injecter dans le MBR de la machine :

```
push    RT_RCDATA       ; lpType
mov     ebx, eax
mov     eax, [ebp+hModule]
push    0DCh            ; lpName
push    eax             ; hModule
call    ds:FindResourceA
mov     edi, eax
push    edi             ; hResInfo
push    0               ; hModule
call    ds:LoadResource
push    eax             ; hResData
call    ds:LockResource
push    edi             ; hResInfo
push    0               ; hModule
mov     esi, eax
call    ds:SizeofResource
cmp     eax, 600h       ; size of res
jz      short loc_4013CD
```

À l'aide d'un débogueur (ou d'un éditeur de ressources), il est possible de visionner et de dumper cette ressource :



et de voir une partie très intéressante un peu plus bas :



Nous trouvons dans cette ressource un message nous informant du blocage de la machine et du chiffrement des disques. Le message indique qu'il est nécessaire de se connecter à un site web pour obtenir un code de déblocage et récupérer les données.

L'exécution continue avec une copie du MBR malicieux en mémoire allouée.

Nous nous trouvons ensuite en présence d'une routine intéressante qui confirme l'infection du MBR :

```
lea     eax, [ebp+FileName]
push    offset a_Physicaldrive ; "\\\\.\\PHYSICALDRIVE%d"
push    eax             ; char *
call    _sprintf
add     esp, 0Ch
push    ebx             ; hTemplateFile
push    ebx             ; dwFlagsAndAttributes
push    3               ; dwCreationDisposition
push    ebx             ; lpSecurityAttributes
push    1               ; dwShareMode
push    0C0000000h      ; dwDesiredAccess
lea     ecx, [ebp+FileName]
push    ecx             ; lpFileName
call    ds:CreateFileA
```

La routine ci-dessus utilise le résultat de la requête WQL pour obtenir un « handle » pointant vers le bon « PHYSICALDRIVE », celui qui contient le MBR à infecter.

Notre malware vérifie la présence d'un marqueur d'infection (présent à deux endroits) pour s'assurer de l'état du disque. Si celui-ci est déjà infecté, il ne tentera aucune infection :

```
push    ebx             ; lpOverlapped
lea     eax, [ebp+NumberOfBytesRead]
push    eax             ; lpNumberOfBytesRead
push    512             ; nNumberOfBytesToRead
lea     ecx, [ebp+MBR]
push    ecx             ; lpBuffer
push    esi             ; hFile
call    ds:ReadFile
test    eax, eax
jz      short not_found
mov     eax, 'cmjh'     ; EAX = marker
cmp     [ebp+pos_marker1], eax
jnz     short not_found
cmp     [ebp+pos_marker2], eax
mov     al, 1
jz      short found
```

Une fois la vérification effectuée, nous arrivons à la routine d'infection du MBR.

Le MBR original est d'abord sauvegardé, puis écrasé par la routine suivante :

```
push    ecx             ; lpNumberOfBytesWritten
push    600h            ; nNumberOfBytesToWrite
push    edx             ; lpBuffer
mov     [ebp+NumberOfBytesWritten], ebx
mov     ebx, ds:WriteFile
push    esi             ; hFile
call    ebx ; WriteFile
test    eax, eax
jz      short loc_4012C0
push    0               ; dwMoveMethod
push    0               ; lpDistanceToMoveHigh
push    2048            ; lDistanceToMove
mov     eax, 0AFBEh     ; some marker
push    esi             ; hFile
mov     [ebp+var_56], ax
call    edi ; SetFilePointer
push    0               ; lpOverlapped
lea     ecx, [ebp+NumberOfBytesWritten]
push    ecx             ; lpNumberOfBytesWritten
push    512             ; nNumberOfBytesToWrite
lea     edx, [ebp+Buffer]
push    edx             ; lpBuffer
push    esi             ; hFile
call    ebx ; WriteFile
```

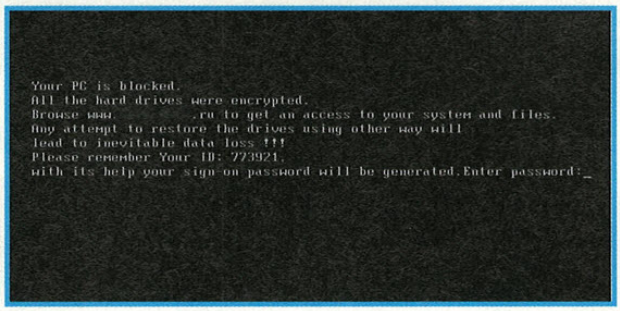
Le premier appel à la fonction **WriteFile** écrase le MBR en écrivant 0x600 octets.

(Secteurs 0, 1, 2). La fonction **SetFilePointer** est ensuite utilisée pour se déplacer à l'offset 0x800 (Secteur 4) pour l'écraser avec le MBR original. Le « handle » du « physicaldrive » est ensuite fermé.

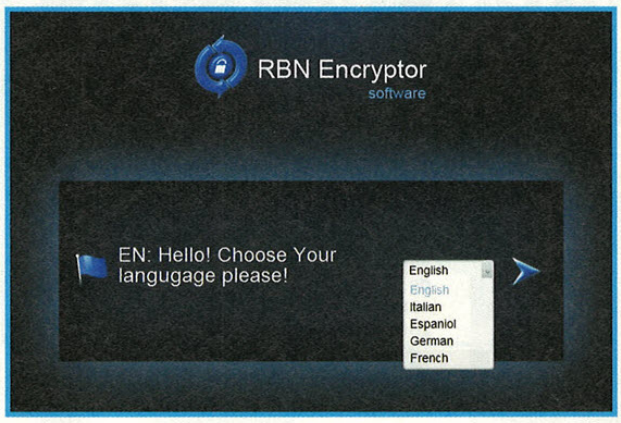
À partir de cet instant, le MBR est infecté par le ransomware. Non content de l'avoir infecté, il va ensuite forcer le redémarrage de la machine après l'obtention du privilège **SeShutdownPrivilege**. La fonction **ExitWindowsEx** est appelée pour exécuter le redémarrage :

```
lea     ecx, [ebp+NewState.Privileges]
push    ecx             ; lpLuid
push    offset Name     ; "SeShutdownPrivilege"
push    0               ; lpSystemName
call    ds:LookupPrivilegeValueA
mov     eax, [ebp+TokenHandle]
push    0               ; ReturnLength
push    0               ; PreviousState
push    0               ; BufferLength
lea     edx, [ebp+NewState]
push    edx             ; NewState
push    0               ; DisableAllPrivileges
push    eax             ; TokenHandle
mov     [ebp+NewState.PrivilegeCount], 1
mov     [ebp+NewState.Privileges.Attributes], 2
call    ds:AdjustTokenPrivileges
call    ds:GetLastError
test    eax, eax
jnz     short failed
push    80020003h       ; dwReason
push    6               ; uFlags
call    ds:ExitWindowsEx
```

Lors du redémarrage d'une machine infectée, nous pouvons lire la demande de rançon suivante :



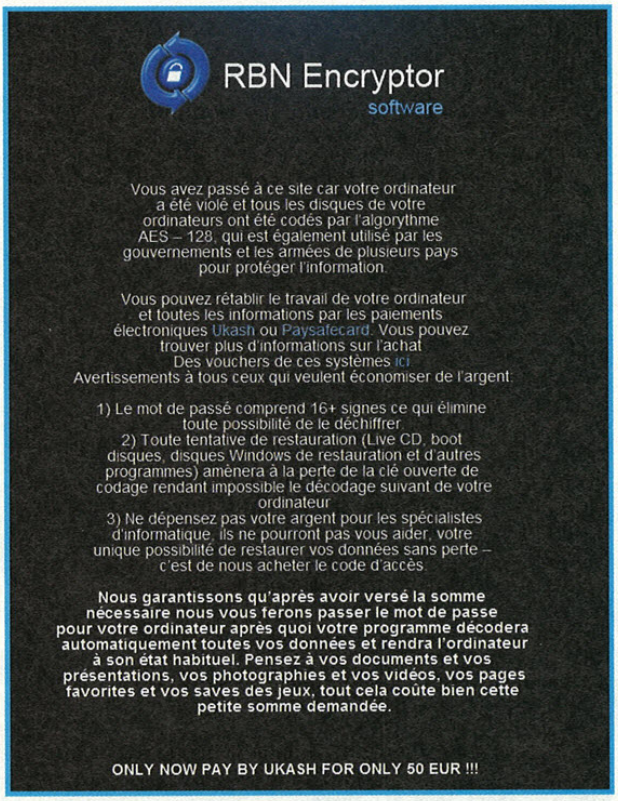
Lors de la visite du site de paiement de la rançon, il est possible de choisir parmi 5 langues : Anglais, Italien, Espagnol, Allemand et Français :



Voici la version française, probablement traduite à l'aide d'un traducteur en ligne compte tenu du nombre d'erreurs présentes dans le texte.

Les données ont soi-disant été chiffrées à l'aide de l'algorithme AES-128. La clé de déchiffrement contiendrait plus de 16 caractères.

Aucune cryptographie n'a été employée par notre ransomware, il s'agit simplement d'effrayer les utilisateurs pour obtenir le paiement d'une rançon : 100$ ou 50 euros (étrange taux de conversion).



2 Analyse du MBR

Étudions maintenant le code injecté dans le MBR pour démarrer la machine. Pour ce faire, j'ai utilisé IDA Pro **[1]** et son débogueur pour Bochs **[2]**.

2.1 Configuration

Je vous invite à lire le blog **[3]** de l'éditeur pour obtenir plus d'informations sur le débogage de MBR ainsi que les scripts nécessaires à son chargement.

Pour déboguer un MBR infecté, nous avons besoin de plusieurs éléments :

- un *dump* du MBR ;
- une image disque créée à l'aide de l'aide de l'utilitaire **dximages** de l'émulateur Bochs ;
- du script **mbr.py** et le fichier de configuration de Bochs, que vous pourrez télécharger sur le blog **[3]** ;
- IDA Pro avec le *plugin* Bochs et IDA Python.

Pour effectuer le dump du MBR, il est en général préférable de démarrer sur un *live CD* Linux et d'utiliser, par exemple, la commande :

```
dd if=/dev/sda of=mbr.dump bs=512 count=5
```

Dans la commande donnée en exemple, j'utilise **count=5** pour dumper 5 secteurs. Bien qu'un MBR classique fasse 512 octets (1 secteur), il est préférable d'en dumper un peu plus. Vous pouvez adapter le nombre de secteurs à dumper en fonction de la menace que vous analysez.

Il vous faudra ensuite modifier votre fichier de configuration Bochs pour qu'il utilise votre image disque :

```
ata0-master: type=disk, path="votreimage.img", mode=flat,
cylinders=20, heads=16, spt=63
```

2.2 Utilisation du script mbr.py

Le script est nécessaire pour charger correctement notre MBR et créer un fichier IDB valide. La seule chose à modifier dans le script est le nom de votre dump, le nom de votre image disque, ainsi que le nombre de secteurs que vous voulez copier dans votre image.

```
BOOT_SIZE   = 0x7C00 + SECTOR_SIZE * 4
MBRNAME     = "votredumpmbr "
IMGNAME     = "votreimage.img"
```

Dans le cas de notre MBR, j'utilise *4 pour copier 4 secteurs dans l'image disque.

Il suffit ensuite d'exécuter le script comme ceci : **mbr.py update** pour mettre à jour l'image.

2.3 Création d'une IDB valide pour débogage

Pour obtenir une IDB valide, le fichier de configuration **bochsrc** doit être ouvert dans IDA Pro, qui détectera automatiquement le type de fichier.

À l'aide d'IDA Python, il nous reste à charger **mbr.py** pour réarranger les segments correctement. Par défaut, le script enregistra l'IDB et se terminera.

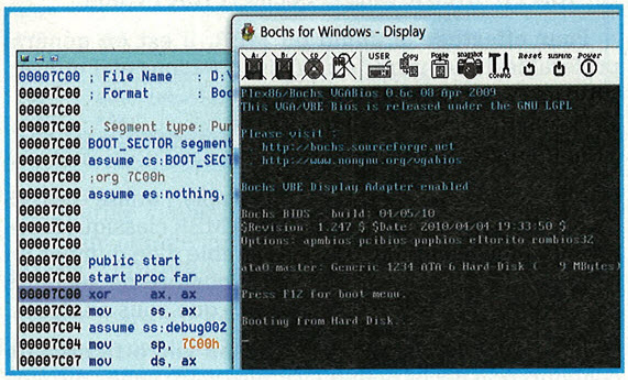
Vous pourrez l'ouvrir à nouveau et lancer le débogueur Bochs pour analyser le MBR pas à pas.

Note

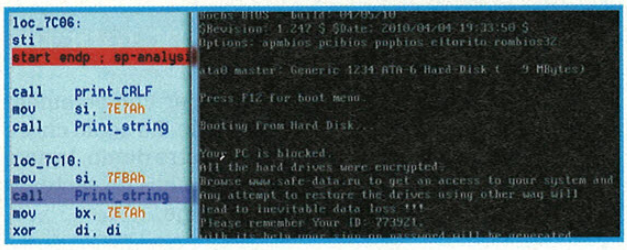
Ne pas oublier de créer une variable d'environnement dximage, comme décrit dans le blog [3].

2.4 Débogage

Voici une capture d'écran une fois le débogueur lancé :



La première routine intéressante recherche deux marqueurs d'infection, puis continue l'exécution jusqu'à l'affichage de la demande de rançon :



Une fois la demande de rançon affichée, la routine de gestion des entrées clavier est exécutée.

L'int 0x16 est utilisée pour gérer les entrées clavier.

```
not_allowed:                            ; CODE XREF: BOOT_SECTOR:7C35↓j
                                        ; BOOT_SECTOR:7C39↓j ...
mov     ah, 10h
int     16h                             ; KEYBOARD - GET ENHANCED KEYSTROKE (AT
                                        ; Return: AH = scan code, AL = character
cmp     ah, 1
jz      short SCANCODE_ESCAPE           ; ESCAPE was pressed
cmp     ah, 0Eh
jz      short SCANCODE_DEL              ; DEL was pressed
cmp     ah, 1Ch
jz      short SCANCODE_ENTER            ; ENTER was pressed
cmp     ah, 0E0h ; 'Ó'
jz      short SCANCODE_ENTER            ; ENTER was pressed
cmp     al, '!'                         ; chars before "!" are not allowed
jb      short not_allowed
cmp     al, '~'
ja      short not_allowed               ; chars above "~" not allowed
cmp     di, 16
jnb     short not_allowed               ; 16 chars max
```

La routine filtre les entrées clavier. Au-delà de 16 caractères, il est impossible d'entrer de nouvelles lettres. Il est possible d'annuler (**ESCAPE**), de corriger (**DEL**) ou de valider (**ENTER**).

Une fois le code entré (attention avec la configuration de clavier QWERTY pour les personnes utilisant un clavier AZERTY), nous arrivons à la routine suivante :

```
BOOT_SECTOR:7C7D mov     al, 20h ; ' '
BOOT_SECTOR:7C7F
BOOT_SECTOR:7C7F add_spaces:
BOOT_SECTOR:7C7F cmp     di, 16
BOOT_SECTOR:7C82 jnb     short done_filling_pass_buffer
BOOT_SECTOR:7C84 mov     [bx+di], al
BOOT_SECTOR:7C86 inc     di
BOOT_SECTOR:7C87 jmp     short add_spaces
```

Le bout de code ci-dessus s'assure de la taille du mot de passe final. En effet, ci celui-ci fait moins de 16 caractères, les caractères manquants deviendront des espaces.

Nous avons maintenant un code de déverrouillage de 16 caractères, et la routine suivante le vérifie :

```
hash_serial proc near
push    ax
push    cx
mov     ah, al
xor     al, al
xor     dx, ax
mov     cl, 8

loc_7D72:
shl     dx, 1
jnb     short no_carry
xor     dx, 1021h

no_carry:
dec     cl
jnz     short loc_7D72
pop     cx
pop     ax
retn
hash_serial endp
```

Cette routine est appelée pour chaque caractère du mot de passe et génère un *checksum* de 16 bits qui est ensuite comparé à un checksum hardcodé dans le MBR :

```
loop_all_chars:
lodsb
call    hash_serial
dec     cl
jnz     short loop_all_chars
cmp     dx, ds:7FFAh  ; 0x3C01
jz      short Good_Password
mov     si, 7FDAh
call    Print_string
call    print_CRLF
dec     byte ptr ds:7E79h ; dec counter
jnz     tries_left
jmp     short reboot_machine
```

Le checksum du mot de passe entré doit être égal à 0x3C01. En cas d'égalité, le MBR malicieux utilise la copie du MBR original et désinfecte la machine.

Dans le cas contraire, un compteur est décrémenté. Au troisième essai, la machine est rebootée.

Note

Ne surtout pas utiliser de fix mbr, la table de partition étant aussi déplacée, vous ne pourriez plus démarrer la machine.

3 Brute force

Après écriture de la routine de checksum en Python (spéciale dédicace à Phil :-)... Il comprendra !), il fut possible de brute forcer le checksum.

```
def hash(serial):
    global CF
    DX = 0
    for x in range(0,16):
        AX = SHLw(ord(serial[x]),8)
        DX ^= AX
        for i in range(0,8):
            DX = SHLw(DX,1)
            if CF:
                DX = DX ^ 0x1021

return DX

print "Brute Forcing MBR ransomware"
print "Nicolas Brulez - Kaspersky Lab\n"
print "Valid passwords to unlock machine:\n"


for char1 in range(97,123):
    for char2 in range(97,123):
        for char3 in range(97,123):
            for char4 in range(97,123):
                serial = "%c%c%c%ckaspersky    " %
(char1,char2,char3,char4)
                if hash(serial) == 0x3C01:
                    print serial
```

Le *brute force* se fait sur 4 octets, le reste est fixé à **'kaspersky'**.

L'algorithme original remplit le *buffer* par des espaces jusqu'à l'obtention d'un code de 16 caractères, j'ai donc ajouté des espaces après la partie codée en dur.

Après quelques secondes d'exécution, nous obtenons :

```
eirdkaspersky
exptkaspersky
jypkkaspersky
qunnkaspersky
```

En entrant un de ces mots de passe, le MBR sera corrigé et l'ordinateur redémarre normalement. (Rappel QWERTY !)

GAME OVER :-) ■

■ RÉFÉRENCES

[1] IDA Pro - http://www.hex-rays.com/idapro/

[2] Bochs : Emulateur IA-32- http://bochs.sourceforge.net/

[3] *Develop your master boot record and debug it with IDA Pro and the Bochs debugger plugin* - http://www.hexblog.com/?p=103