# Bootkit REvisited

Samuel Chevet
s.chevet@gmail.com

Sogeti ESEC

**Résumé** La première preuve de concept de Bootkit a été présentée à BlackHat en 2005 par Derek Soeder [3]. C'était un simple infecteur de Master Boot Record (MBR) permettant de s'authentifier sur une machine sans en connaitre le mot de passe. En 2007, les auteurs de malwares ont décidé d'utiliser cette technique sur les plateformes Windows x64 pour outrepasser le mécanisme de signature des drivers. Depuis 2005 plusieurs projets ont vus le jour, mais les techniques utilisées pour reproduire ce genre d'attaque sont restées les mêmes : mise en place de Hook et utilisation de signature pour patcher en mémoire les binaires. Cet article va décrire une nouvelle technique, ne se basant pas sur les anciens projets connus. La technique présentée nous permet de contrôler le processus de boot de toutes les versions de Windows, avec ou sans systèmes de chiffrement de disques, car nous ne nous ne réalisons aucune modification du code de démarrage en mémoire. Il est à noter que cette solution ne fonctionne pas avec Secureboot ou équivalent. Les techniques décrites dans cet article reposent sur des fonctionnalités offertes par le processeur comme le mode v8086, les breakpoints de type matériel et l'isolation de privilèges, technique qui fût utilisé par VMware pour faire de la virtualisation dans les années 2000. L'intérêt de ces techniques est qu'aucune signature mémoire n'est nécessaire, ni la mise en place de Hook. Nous proposons un exemple qui se trouve sous la forme d'une image ISO bootable sur clef USB permettant de charger n'importe quel driver non signé. Notre driver d'exemple permet d'outrepasser la fenêtre de login sans infection.

## 1 Introduction

Le mot bootkit est originaire de la fusion des mots : rootkit et boot. Un rootkit, qu'on appelle aussi outil de dissimulation d'activité est un malware souvent sous la forme d'un driver, s'exécutant en espace noyau. Il s'exécute avec les privilèges les plus élevés, il a un accès total au matériel, il peut ajouter ou supprimer des fonctionnalités au noyau. Ces possibilités lui permettent de se dissimuler plus facilement sur la machine. Microsoft Windows, jusqu'à l'arrivée de Windows 2003 Server, ne se souciait pas de ce qu'allait faire un driver : il pouvait être chargé en mémoire, s'exécuter et n'avait aucune restriction. Maintenant, un driver sur un système 64bits doit obligatoirement être signé par une autorité de certification. De plus, des mécanismes comme PatchGuard ont vu le jour, permettant d'empêcher

certains drivers de modifier/supprimer certaines fonctionnalités du noyau. Donc, pour charger un driver, les auteurs de malwares ont dû trouver de nouvelles solutions comme l'infection des secteurs de démarrage de l'ordinateur. L'objectif est d'exécuter du code dès le démarrage de la machine qui persiste tout au long des différentes étapes de boot jusqu'au lancement de Windows pour finalement charger un driver non signé dans le noyau.

### 1.1 La chaîne de boot

Afin de mieux comprendre l'intégralité de cet article, nous allons présenter les différents composants de la chaîne de démarrage d'un poste de travail sur environnement Windows de type 64 bits.

**Basic Input Output System (BIOS).** Le BIOS est un programme contenu dans la mémoire morte de la carte mère qui va être chargé en mémoire lorsque l'on mettra l'ordinateur sous tension. Ce composant est exécuté en mode réel (16 bits) et aura pour but d'initialiser tous les composants matériels du système. Le BIOS va ensuite donner la main à un média comme un CD-ROM ou un disque dur. Il va charger normalement le premier secteur d'un disque dur ayant la signature correcte à l'adresse `0x0000:0x7C00`. Ce premier secteur en question est appelé Master Boot Record (MBR). Sur la figure 2, on peut voir la disposition de la mémoire lorsque l'ordinateur démarre.

**Master Boot Record (MBR).** Le MBR a été recopié par le BIOS à l'adresse `0x0000:0x7C00`. Ce composant sera exécuté en mode réel. Le MBR standard de Windows, va :

1. se recopier à l'adresse `0x000:0x0600` ;
2. parcourir la table des partitions pour trouver une partition avec le flag *bootable* ;
3. recopier le premier secteur de cette partition à l'adresse `0x000:0x7C00` ;
4. l'exécuter.

Sur le listing 1, on peut voir la disposition du premier secteur d'un disque de type MBR.

```
+0x000 : Code
+0x1B8 : Disk signature
+0x1BC : Unknown (usually null)
+0x1BE : Partition table (4 entries)
```

```
+0x1FE : Signature (0xAA55)
```

**Listing 1.** Disposition du MBR

```
+0x00 : Boot ID
+0x01 : CHS address of first sector
+0x04 : System ID
+0x05 : CHS address of last sector
+0x08 : LBA first sector
+0x0C : Number of sectors
```

**Listing 2.** Disposition de la table des partitions

Un MBR doit toujours avoir la signature 0xAA55. Le champ *Boot ID* dans une entrée de la table de partition permet de préciser si la partition est active et si elle est bootable. *System ID* spécifie le type de la partition : NTFS, FAT32, AIX, etc.

**Volume Boot Record (VBR).** Le VBR se trouve sur les premiers secteurs d'une partition. Ces secteurs contiennent du code qui va servir à charger des composants pour le démarrage du système d'exploitation. Sous Windows, le VBR se trouve sous la forme de 16 secteurs contigus au début de la partition NTFS. Le premier secteur contenant le Bios Parameter Block (BPB) est le code qui charge les 15 autres secteurs.

Les 15 autres secteurs servent à parcourir la partition NTFS à la recherche des fichiers : NTLDR ou BOOTMGR. Ces 15 secteurs s'appellent Initial Program Loader (IPL). À l'aide du champ *HiddenSectors* présent à l'offset 0x01C du BPB, le VBR sait à combien de secteurs du début du disque se trouve l'IPL (les 15 autres secteurs).

**Bootmgr.** Bootmgr.exe est un binaire compressé au sein d'un chargeur 16 bits. C'est un binaire deux en un.

Le chargeur 16 bits va préparer l'environnement nécessaire comme le passage en mode protégé pour l'exécution de Bootmgr.exe 32 bits. La partie 16 bits calcule également la somme de contrôle de Bootmgr qu'il décompresse à l'adresse 0x400000. La partie 32 bits va initialiser :

- un débogueur de boot ;
- l'affichage (Framebuffer) ;
- la gestion de la mémoire ;
- le système de fichiers ;
- le réseau.

Ce binaire va ensuite vérifier si l'ordinateur était en mode hibernation, si c'est le cas le binaire Winresume.exe sera chargé, ou sinon Winload.exe. Dans le cas où nous ne sortons pas d'une hibernation, la base BCD (base de registre) va être montée. S'il n'existe qu'une seule entrée de programme de démarrage, elle sera démarrée automatiquement. Dans le cas contraire, un menu s'affichera avec les différentes entrées pour que l'utilisateur puisse choisir son système. Un fichier bootstat.dat, est aussi lu ou mis à jour pour savoir si le dernier démarrage s'est bien déroulé, et propose la réparation de l'ordinateur si besoin. Pour passer la main à l'étape suivante c'est la fonction ImgArchPcatStartBootApplication() qui est appelée, et qui se chargera de passer notre processeur en mode 64 bits avant d'exécuter le binaire Winload.exe.

**Winload.** Bootmgr.exe appelle le chargeur de système d'exploitation Windows, qui est localisé dans %SystemRoot%\System32\Winload.exe. Ce binaire est un système d'exploitation minimaliste de type 64 bits qui va avoir les tâches suivantes :

- mettre en place la pagination ;
- charger les clés de registre système ;
- charger le noyau (ntoskrnl.exe) ;
- charger ces dépendances et les drivers avec l'option SERVICE_BOOT_START ;
- récupérer les options de démarrage.

La chose la plus importante que fait ce binaire, est la préparation de la structure : LOADER_PARAMETER_BLOCK. Cette structure est passée en argument à ntoskrnl.exe.

```
+0x000 LoadOrderListHead : _LIST_ENTRY
+0x008 MemoryDescriptorListHead : _LIST_ENTRY
+0x010 BootDriverListHead : _LIST_ENTRY
+0x018 KernelStack : Uint4B
+0x01C Prcb : Uint4B
...
```

**Listing 3.** Dump LOADER_PARAMETER_BLOCK

Elle est initialisée par la fonction OslInitializeLoaderBlock et sera remplie d'informations en rapport avec l'état du système. Ici, nous sommes intéressés par le champ BootDriverListHead qui est une liste chaînée de type _LDR_DATA_TABLE_ENTRY. Cette liste va contenir tous les drivers ayant l'option SERVICE_BOOT_START, qui sont destinés au driver lancé au moment du démarrage. C'est la future

PsLoadedModuleList qui va être utilisée par ntoskrnl.

À cette étape du lancement du système d'exploitation notre Global Descriptor Table (GDT) contient les entrées suivantes :

- une entrée de code pour le mode long (64 bits) ;
- une entrée de code pour le mode protégé ;
- une entrée de data pour le mode protégé ;
- une entrée pour la TSS en mode long ;
- une entrée de code pour le mode réel ;
- une entrée de data pour le mode réel ;
- une entrée de data pour le Framebuffer (0x000B8000).

**Interruption BIOS en mode Long.** Si Winload garde des entrées de code ou data pour le mode protégé ou le mode réel, c'est qu'il va continuer à faire appel aux exécutables précédents (bootmgr_32, bootmgr_16). Winload a besoin d'exécuter des interruptions du BIOS pour lire et écrire des fichiers, afficher des informations à l'écran ou connaître l'état du clavier. À ce stade du chargement du système d'exploitation, aucun driver destiné à effectuer ces actions n'est alors chargé. Pour effectuer une interruption BIOS depuis le mode Long, ils vont remplir une structure qui est stockée en 0x030000 décrite sur le listing 4.

```
struct reg_frame
{
        DWORD dwIntNumber;      /* + 0x00 */
        DWORD dwEAX;            /* + 0x04 */
        DWORD dwEBX;            /* + 0x08 */
        DWORD dwECX;            /* + 0x0C */
        DWORD dwEDX;            /* + 0x10 */
        DWORD dwEBP;            /* + 0x14 */
        DWORD dwESP;            /* + 0x18 */
        DWORD dwESI;            /* + 0x1C */
        DWORD dwEDI;            /* + 0x20 */
        DWORD dwCS;             /* + 0x24 */
        DWORD dwDS;             /* + 0x28 */
        DWORD dwSS;             /* + 0x2C */
        DWORD dwES;             /* + 0x30 */
        DWORD dwFS;             /* + 0x34 */
        DWORD dwGS;             /* + 0x38 */
        DWORD dwEFLAGS;         /* + 0x3C */
};
```

**Listing 4.** Structure interruption BIOS

Pour faire un appel à une interruption du BIOS, Winload fait appel à la fonction PcatX64SuCallback, qui va passer en mode protégé et exécuter la fonction PcatX64LmSuCallback sur un segment 32 bits. Cette fonction fait

l'équivalent d'un appel à call [PcatServicesTable], où PcatServicesTable est une variable initialisée dans la fonction InitializeLibrary. La destination de ce call arrive dans Bootmgr_32 et effectue un trampoline pour revenir en mode réel sur le segment de code de Bootmgr_16. La figure 5 explique tout le cheminement.

**Ntoskrnl.** Ntoskrnl est le dernier maillon de notre chaîne de boot. Ce binaire est l'image noyau pour la famille des systèmes d'exploitation Microsoft Windows NT. Il fournit le noyau qui est responsable de diverses actions telles que la virtualisation du matériel, la gestion des processus et de la mémoire. Il contient le gestionnaire de cache, le moniteur de sécurité, le planificateur, etc.

**Chaîne de confiance** Une chaîne de confiance est installée entre certains composants du processus de boot.

- Bootmgr_16 va calculer une somme de contrôle sur l'exécutable qu'il contient : Bootmgr_32

- Bootmgr_32 va calculer une somme de contrôle sur lui-même à l'aide de la fonction BmFwVerifySelfIntegrity

Au sein de Bootmgr_32, la fonction BlImgLoadBootApplication va être responsable du chargement de winload.exe. Pour vérifier son intégrité, la fonction ImgpValidateImageHash est utilisée.
Winload ne calcule pas de somme de contrôle sur lui-même, mais tous les exécutables qu'il chargera (ntsokrnl.exe, hal.dll, ...) vont être vérifiés lors de l'appel à la fonction BlImgLoadPEImageEx.

## 2 État de l'art

Les premiers bootkits sont apparus en 2005 et ont été utilisés comme preuve de concept par des chercheurs en sécurité informatique, leur but n'était pas de nuire. Ce n'est qu'en 2007 que le premier bootkit malveillant voit le jour, nommé par les compagnies antivirus : "Mebroot". Il ne visait que des systèmes 32 bits et les techniques utilisées n'étaient pas très évoluées. La figure 7 montre l'évolution des bootkits.

Au cours des années 2010, le premier bootkit malveillant visant des systèmes **64 bits** a été découvert, il se nomme *TDSS*, *Alureon* ou encore *Olmarik* en fonction des éditeurs d'antivirus. Il se peut que ce ne soit pas

le plus récent, car un malware répondant au nom de Turla, a été découvert récemment, et serait plus vieux que TDSS. Il existe à ce jour une liste bien plus conséquente sur des familles de bootkit pouvant toucher des systèmes 64 bits :

- Cidox ;
- XPAJ ;
- YURN ;
- Prioxer ;
- Gapz et d'autres.

Les techniques utilisées par les Bootkits depuis de nombreuses années n'ont pas évoluées et suivent le même cheminement. Le BIOS expose des interruptions à l'aide de l'Interrupt Vector Table (IVT). Une fonction très utilisée par le processus de boot de Windows est l'interruption 0x13 (19) et plus particulièrement la fonction 0x42 (66) qui permet la lecture de disque étendue. Cette fonction est utilisée à chaque lecture effectuée sur le disque pour chacun des fichiers nécessaires au lancement du système d'exploitation. Les bootkits vont mettre en place un *Hook* (Figure 8) sur cette interruption et vérifier le numéro de fonction appelée.

Dans le cas d'un appel à la fonction 0x42, le code mis en place au sein du *Hook* exécutera la fonction original, scannera le buffer contenant le résultat de la lecture à la recherche d'un motif mémoire afin de placer de nouveau un *Hook* au sein du fichier qui vient d'être lu pour garder la main sur la prochaine étape du processus de boot. Par exemple :

1. placer un *Hook* sur l'IVT ;
2. exécution du MBR original ;
3. le *Hook* détecte la lecture du VBR ;
4. un nouveau *Hook* va être placé au sein du VBR ;
5. ce nouveau *Hook* détecte la lecture de Bootmgr ;
6. mise en place d'un nouveau *Hook* au sein de Bootmgr ;
7. etc.

La mise en place de Hook au sein des différents composants requiert le patch de la chaîne de confiance. Le but final est d'être en mesure d'exécuter du code jusqu'au chargement de ntoskrnl.exe. Arrivé à ce stade, ils peuvent charger un driver non signé ou effectuer des patchs dans le noyau comme désactiver la protection PatchGuard. La figure 9 récapitule la modification de tout le processus de boot par un malware.

Il existe de nombreux projets Open-Source utilisant exactement la même approche :

- KON-BOOT
- Stoned Bootkit
- vBootkit

DreamBoot [1], un des premiers bootkits UEFI présenté a SSTIC en 2013, quant à lui met en place des hooks à partir de Winload.

### 2.1 Problématiques des solutions existantes

Que ce soit du côté des personnes malveillantes ou des preuves de concept, tous les Bootkits reposent sur ces mêmes techniques.

- La mise en place de Hook : ce genre de technique peut être facilement détectable. De plus, la mise en place de Hook au sein des binaires qui constituent la chaîne de boot requiert de patcher la chaîne de confiance entre chacun des composants.
- L'utilisation de signature : pour mettre en place des Hooks au sein des composants du processus de boot, les Hooks précédents doivent scanner la mémoire à la recherche de signature. Mais ces signatures peuvent changer d'une version à une autre (d'un Windows 7 à Windows 8), ou lors d'installation de mise à jour de type service pack, ce qui pourrait empêcher l'ordinateur de redémarrer.

Ces deux techniques nécessitent lors de la lecture des fichiers sur le disque que les fichiers soient en clair et non chiffrés par des outils comme Bitlocker ou encore TrueCrypt. Certains projets open-source utilisent 2 layers de Hook, pour la recherche de signatures, mais ils supportent au mieux un seul système de chiffrement de disque.

## 3 REboot

Dans la suite, nous allons présenter une solution innovante pour implémenter des Bootkits pour Windows 64 bits pour contourner ces problèmes précédents. Nous avons appelé cette solution "REboot".

**Réponses aux problématiques.** Nous voulons une solution capable de supporter les différentes versions de Windows (7, 8, 8.1 ou encore 2003 Server), sans avoir à modifier notre code ou les tests et signatures prédéfinis au sein de notre Bootkit. Dans le cas d'une mise à jour de type Service Pack modifiant le processus de boot de Windows nous voulons que la machine soit encore capable de démarrer "correctement". De plus nous voulons être capable d'avoir un Bootkit fonctionnel indépendamment d'un système de chiffrement de disque.

### 3.1 Solution

Afin d'être en mesure d'exécuter du code durant le processus de boot, nous ne nous focalisons pas sur les binaires en eux mêmes, comme vu dans les précédents projets, où des hooks sont mis en place sur chacun des composants :

- le MBR qui Hook le VBR
- le VBR qui Hook Bootmgr
- Bootmgr qui Hook Windload
- Winload qui patch Ntoskrnl

Nous allons nous appuyer seulement sur le changement de mode du processeur, et nous n'utiliserons pas de Hook, ni de scan mémoire à la recherche de signatures prédéfinies, pour n'avoir à faire aucune modification dans notre code d'une version du système d'exploitation à une autre. La solution n'effectuera aucun patch en mémoire afin d'éviter d'avoir à patcher la chaine de confiance.

### 3.2 Quatre grandes étapes

La solution est découpée en 4 étapes pour arriver au chargement d'un driver non signé. Ces 4 étapes sont les suivantes :

1. le passage du mode réel (16 bits) au mode protégé (32 bits) ;
2. le passage du mode protégé (32 bits) au mode long (64 bits) ;
3. le passage de Winload à Ntoskrnl ;
4. le chargement du driver non signé.

**Virtual 8086.** Il existe un mode du processeur qui peut nous permettre de conserver le contrôle jusqu'à l'exécution de bootmgr et le passage en mode protégé.

Le mode virtuel 8086 est un sous-mode du mode protégé. Ce mode permet d'exécuter une tâche 16 bits en mode réel dans un environnement protégé. Ceci permet par exemple, de faire tourner des binaires DOS (16 bits) depuis un Windows 9.X jusqu'à un Windows Vista. Le virtual mode est activé lorsque le Virtual Machine (VM) flag dans les EFLAGS (bit#17) est à 1. Il est à noter que la mise en place de ce flag ne peut pas se faire à l'aide de l'instruction *popf* mais peut se faire en utilisant l'instruction *iret* ou en passant par l'utilisation d'une TSS 386. La partie la plus intéressante pour notre projet est la gestion des interruptions

et exceptions dans ce mode. Il existe trois niveaux (IOPL) pour définir le comportement du processeur lors d'interruptions et exceptions. Les instructions *cli*, *sti*, *pushf*, *popf*, *int n*, *iret* sont sensibles aux niveaux d'IOPL. Si l'IOPL est inférieur à 3 alors l'instruction privilégiée lance une exception de type "General Protection Fault". La dernière chose à noter est le fait qu'ici nous n'avons besoin que d'une seule tâche virtuelle 8086. Nous choisirons un privilège d'IOPL de level 1 permettant au code exécuté en mode réel d'avoir un maximum de possibilités, sauf l'exécution d'interruptions. Nous expliquerons par la suite l'objectif de ce choix. Le but est d'avoir un gestionnaire récupérant une fault lorsque le code voudra exécuter des instructions privilégiées tel que *lgdt*, pour charger une GDT lors du passage du mode réel en mode protégé au sein du binaire Bootmgr. Pour cette idée le cheminement à suivre est le suivant :

- mettre en place le mode protégé ;
- mettre en place le mode 8086 ;
- charger le MBR original ;
- exécuter ce code en mode 8086.

Grâce à cette solution nous conservons le contrôle sans avoir modifié du code en mémoire jusqu'au chargement du binaire Bootmgr. Nous filtrons aussi les interruptions car la capture des instructions de type *lgdt* peut apporter des faux positifs lorsque nous sommes sur une machine avec TPM car les instructions BIOS en rapport avec TPM passent le processeur en mode protégé. De ce fait, lorsqu'une interruption veut être exécutée, nous repassons le processeur en mode réel et nous exécutons l'interruption "normalement".

**Debug Register.** Pour la deuxième étape du processus de démarrage, nous voulons être en mesure de garder la main lors du passage en mode protégé au mode long.

Le binaire Bootmgr, avant de donner la main à Winload, va devoir préparer le passage en mode Long (64 bits), mettre en place une nouvelle GDT et IDT, ainsi qu'activer la pagination. C'est la fonction ImgArchPcatStartBootApplication() au sein de Bootmgr_32 qui va :

- réserver une page mémoire ;
- récupérer la structure de description de la GDT et IDT à l'aide de *sgdt* et *sidt* ;
- copier le contenu de la GDT et IDT dans cette page mémoire grâce à l'adresse de base et de la limite ;

- tester si le binaire winload est de type 32 bits ou 64 bits grâce à l'en-tête PE ( IMAGE_FILE_HEADER->Machine ) ;
- mettre à zéro toute les entrées de l'IDT (dans le cas d'un binaire 64 bits) qu'il vient de copier, car elles ne sont pas valides en mode Long (64 bits).

Toutes ces opérations sont effectuées par le binaire Bootmgr_32 juste avant l'exécution de Winload. Nous nous servons de ces informations, dans le but d'exécuter du code durant cette étape du lancement du système d'exploitation.
À cette étape du boot, nous sommes en mode protégé donc nous utilisons les registres de débogage de type matériel. Grâce à l'utilisation de ces registres de débogage, nous allons tracer le passage entre Bootmgr_32 et Winload. Ce que nous allons faire c'est :

1. mettre l'adresse de l'IDT dans le registre dr0 ;
2. mettre la condition break on access dans dr7 ;
3. mettre en place un gestionnaire d'interruption de type 0x1.

On reprend l'exécution et notre gestionnaire d'exception sera appelé au moment de l'exécution de la fonction ImgArchPcatStartBootApplication(), lors de la recopie de l'IDT, car un accès en lecture va être effectué. Pour continuer à être en mesure d'exécuter du code, nous allons remettre un break-on-access sur le nouvel emplacement de l'IDT car nous récupérons l'adresse de destination du memcpy() sur lequel nous venons de breaker. Ceci nous permettra de breaker au milieu de la fonction memset, lorsqu'il met à 0 toutes les entrées de l'IDT n'étant pas valide pour le mode Long (64 bits). La figure 12 récapitule toutes les opérations effectuées par le bootkit.

Winload en mode 64 bits démarre avec une IDT_64 vide, c'est la fonction BlpArchInstallTrapVectors qui est responsable d'installer les gestionnaires pour le mode long. En particulier il va installer un gestionnaire pour l'exception "General Protection Fault", entrée 0xD (13). Ce que nous faisons est de placer un nouveau registre de debug, sur l'adresse de l'entrée 0xD. De cette façon nous pourrons continuer à exécuter du code jusqu'à être au sein du code de Winload. Nous sommes en mesure d'exécuter du code durant la transition du mode protégé au mode long.

**Ring1.** L'avant dernière étape est de garder le contrôle sur le processus de boot pendant la transition entre Winload et ntoskrnl.

Winload utilisera des instructions privilégiées pour mettre à jour la GDT ou l'IDT nécessaire à ntoskrnl. En modifiant le level de privilège du segment de code de Winload, nous sommes capable de monitorer toutes les instructions privilégiées. Pour cela il suffit de :

1. installer un nouveau segment de code avec un DPL = 1 ;
2. installer un nouveau segment de data avec un DPL = 1 ;
3. mettre en place notre propre gestionnaire pour les "General Protection Fault" ;
4. mettre à jour la structure TSS_64.

Le gestionnaire pour les "General Protection Fault" devra se charger de :
- récupérer l'adresse de l'instruction qui a générée l'exception ;
- vérifier le type de l'instruction ;

Si l'instruction ne nous intéresse pas :
- la recopier dans un espace mémoire ;
- l'exécuter.

Nous émulerons l'instruction dans le cas contraire.

Pour calculer la taille d'une instruction afin de la recopier en mémoire nous utilisons un LDE (Length Disassembly Engine), qui est une librairie en assembleur. Il est à noter qu'il existe trois cas spéciaux d'instruction que nous devons émuler :

1. mov ds, ax : Dans la fonction PcatX64SuCallback, winload veut mettre à jour le registre de segment ds, pour passer en mode protégé. Dans ce cas nous restaurons le DPL à 0 pour éviter tout futur probème.
2. jmp far `seg:offset` : Cette instruction est utilisé pour mettre à jour le segment de code pour revenir en mode long car le RPL != DPL. Nous allons mettre à jour les registres de segment cs, ss et le registre général rip, pour remettre winload en ring1.
3. mov ss, ax : La mise à jour du registre de segment arrive juste après le jmp far, nous avons déjà mis à jour ss lors du précédent cas et donc nous ignorons cette instruction.

Tous les autres cas peuvent être recopiés et exécutés en mémoire. Le dernier cas sera l'instruction *lgdt* permettant la mise en place de la nouvelle GDT, juste avant l'exécution de ntoskrnl.

**Payload.** Maintenant que nous sommes juste avant le lancement de ntoskrnl.exe, nous voulons être en mesure de charger un driver non signé. Nous avons accès à toutes les APIs exportées par ntsokrnl.exe. Pour récupérer l'adresse de chacune nous parcourons la table des exports. Notre

charge va être injectée dans kdcom.dll, une dll permettant le débogage à travers l'interface série. Elle va remplacer le code de KdDebuggerInitialize1, fonction exportée par cette DLL. La charge va appeler la fonction IoCreateDriver(PUNICODE_STRING DriverName, PDRIVER_INITIALIZE InitializationFunction), nous permettant de créer un DEVICE_OBJECT. C'est un gain de temps, surtout avec les problèmes des tables IRP, car il nous aurait fallu récupérer un objet déjà existant. La fonction InitializationFunction passée en argument sera en charge de :

1. ouvrir et lire un driver (PE) ;
2. mapper les sections en mémoire ;
3. résoudre les fonctions d'imports ;
4. résoudre les relocations ;
5. mettre à jour le DEVICE_OBJECT, avec la taille mémoire du driver, son adresse de base, et son point d'entrée ;
6. appeler le point d'entrée.

### 3.3 Résultats

Notre solution permet d'avoir de l'exécution de code durant tout le process de boot de Windows, en utilisation le mode v8086 pour surveiller les composants s'exécutant en mode réel, puis les registres de débogage matériel lors du mode protégé et enfin l'isolation de privilèges pour le mode long. Notre solution est actuellement fonctionnelle, sur des systèmes de virtualisation comme VMware et VirtualBox avec des machines virtuelles de type Windows 7, 8 ou 8.1 64 bits. Notre solution fonctionne également avec Windows Server 2012 x64 en machine virtuelle. Elle fonction également sur des machines physique qui utilisent un BIOS ou en UEFI avec le mode BIOS legacy activé, ainsi que des émulateurs tels que Bochs.

### 3.4 Évolutions et projets

La solution ne fonctionne pas avec l'UEFI (sans SecureBoot[2] activé) mais cette modification ne représente pas énormément de temps de développement, car l'UEFI rend la main au système directement en long mode, il nous suffit d'utiliser la méthode d'isolation de privilège dès le départ. Cette solution se focalise seulement sur les versions 64 bits, la partie sur les registres de débogage de type matériel ne fonctionne que sur 64 bits, car dans notre implémentation nous jouons sur le fait que l'IDT est mise à zéro car les entrées 32 bits ne sont pas valides en 64 bits. Il

peut être envisageable d'utiliser 2 registres de débogage pour permettre de différencier le fait que l'IDT soit mise à zéro où qu'une entrée soit mise à jour. Notre projet ne s'est focalisé que sur des environnement Windows, mais ces solutions peuvent surement s'appliquer à d'autres systèmes. La partie 8086 peut être applicable pour prendre la main sur des chargeurs de démarrage. La partie sur les registres de débogage de type matériel a été développée après avoir effectué de la rétro-ingénierie sur le processus de boot de Windows, mais ne se déroule peut être pas de la même manière sur d'autres systèmes d'exploitation. Par contre la méthode de pseudo-virtualisation à travers l'isolation de privilèges utilisée peut permettre sur n'importe quel système de surveiller son démarrage en appliquant les cas spécifiques au sytème d'exploitation visé.

### 3.5 Prévention

Pour se protéger de ce type d'attaque, il est nécessaire d'empêcher le chargement de codes non signés au démarrage. Il est conseillé d'utiliser les technologies offertes par Trusted Platform Module (TPM), en mode statique (Static Root of Trust for Measurements : SRTM), permettant de stocker une somme de contrôle de chacun des composants de la chaîne de boot afin qu'elle ne soit pas compromise. Cette solution est généralement utilisé conjointement un avec système de chiffrement de disque comme Bitlocker, qui ne déverrouille la clé maitre de déchiffrement uniquement si tous les éléments de la chaine de boot sont intègres. Windows SecureBoot permet le chargement de système d'exploitation signé, Windows Trusted-Boot permet de vérifier l'intégrité de chacun des composants de la chaîne de démarrage. Mais ces fontionalités ne sont disponibles que sur des PC possédant un boot UEFI et le composant TPM.

## Références

1. Sébastien Kaczmarek. Dreamboot et uefi. *https://www.sstic.org/2013/presentation/dreamboot_et_uefi/*, 2013.
2. Microsoft. Secured boot and measured boot. *http://download.microsoft.com/download/4/0/D/40DF6E51-DAD0-4CF8-B768-8E3B4A848D9C/secured-boot-measured-boot.docx*, 2013.
3. Derek Soeder. eeye bootroot. *https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf*, 2005.

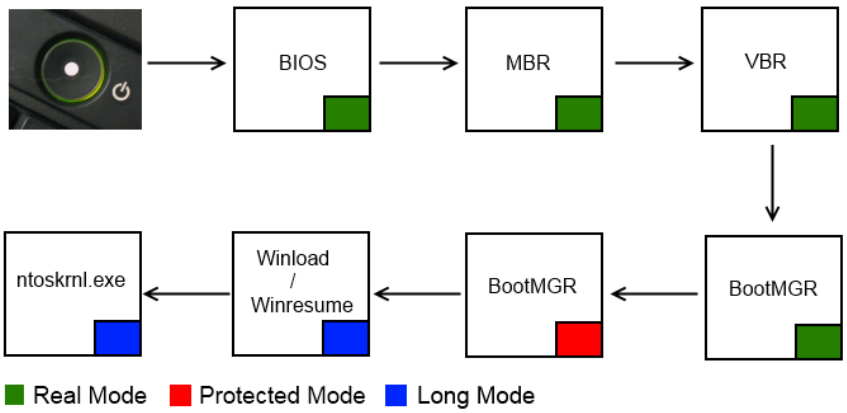


**Figure 1.** Processus de boot

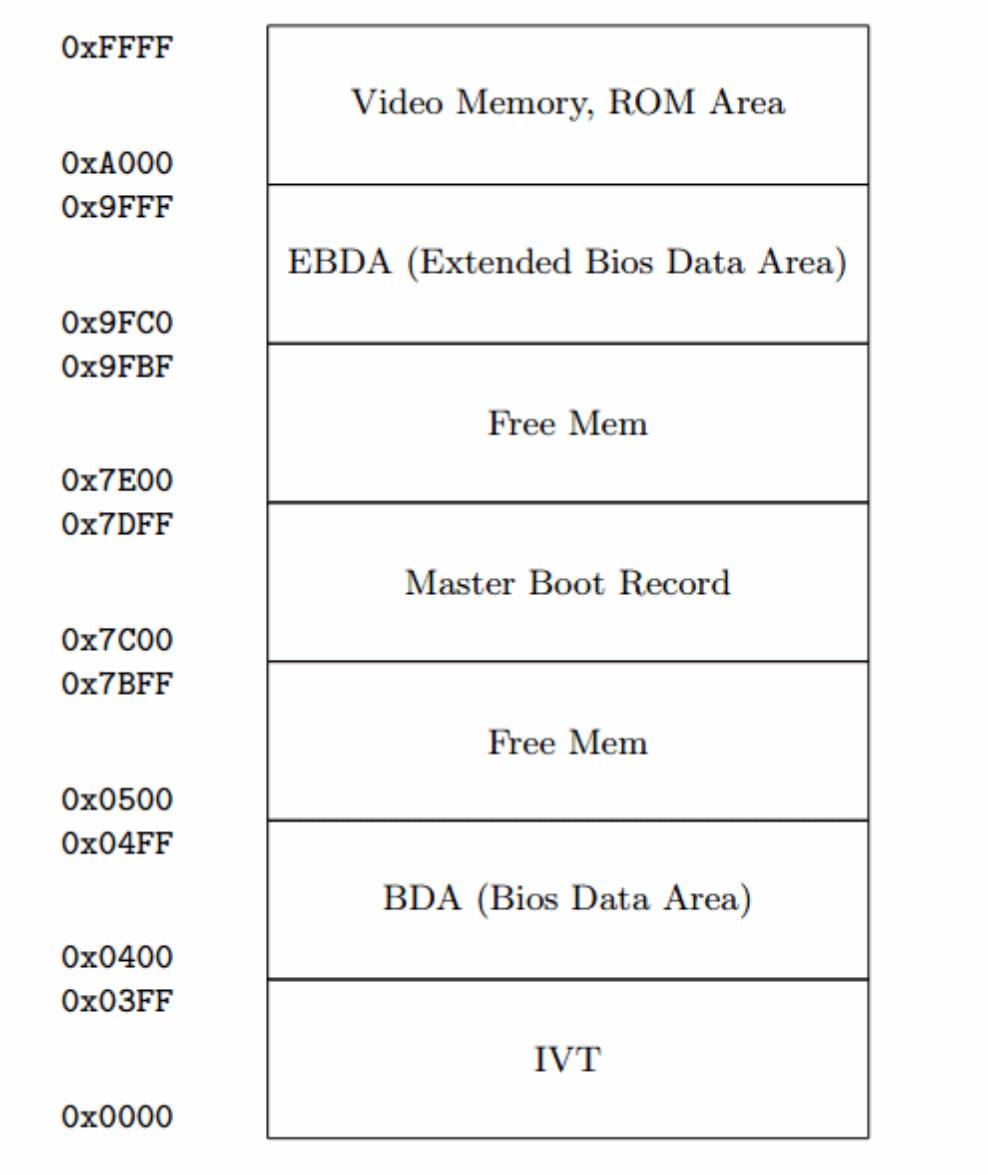


**Figure 2.** Disposition mémoire du BIOS

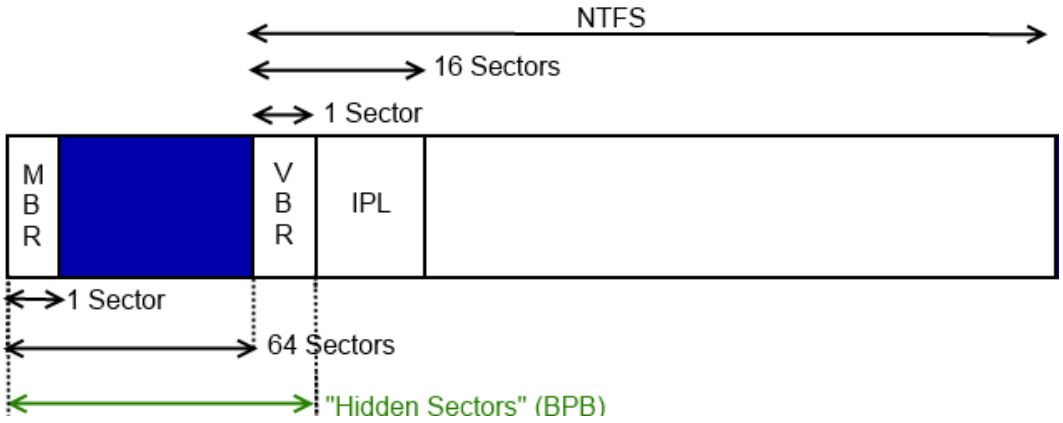


**Figure 3.** Disposition disque dur

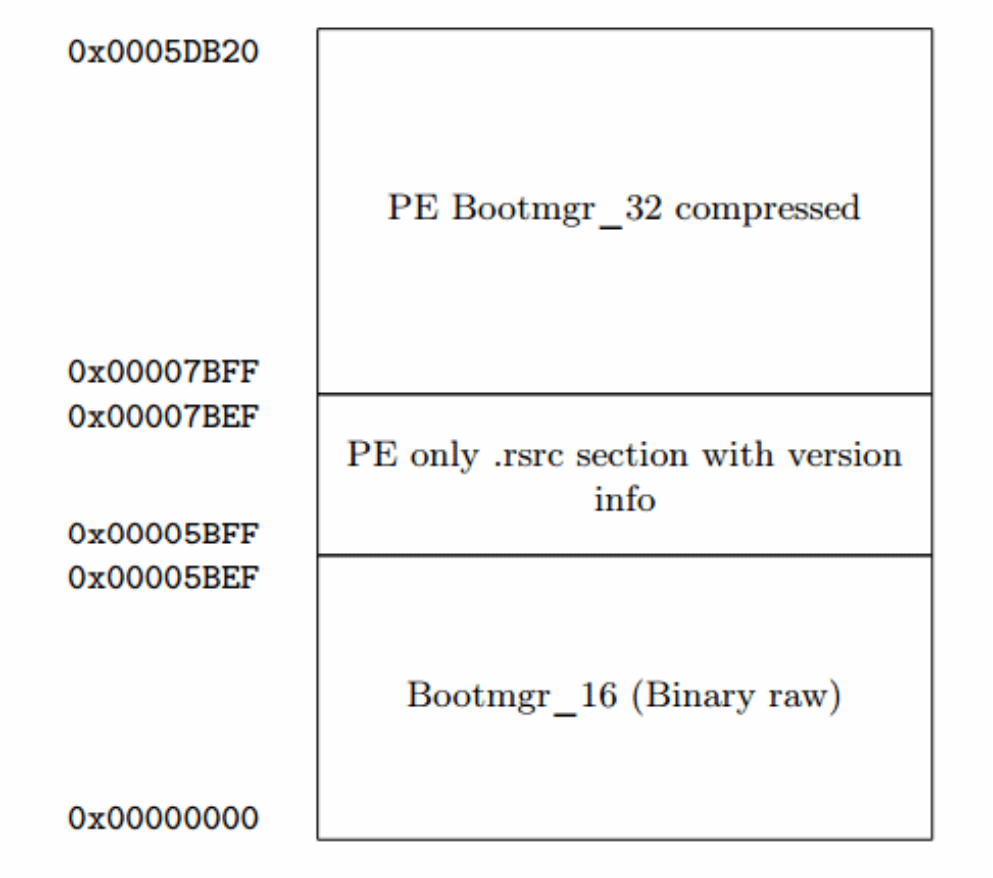


**Figure 4.** Disposition de Bootmgr.exe

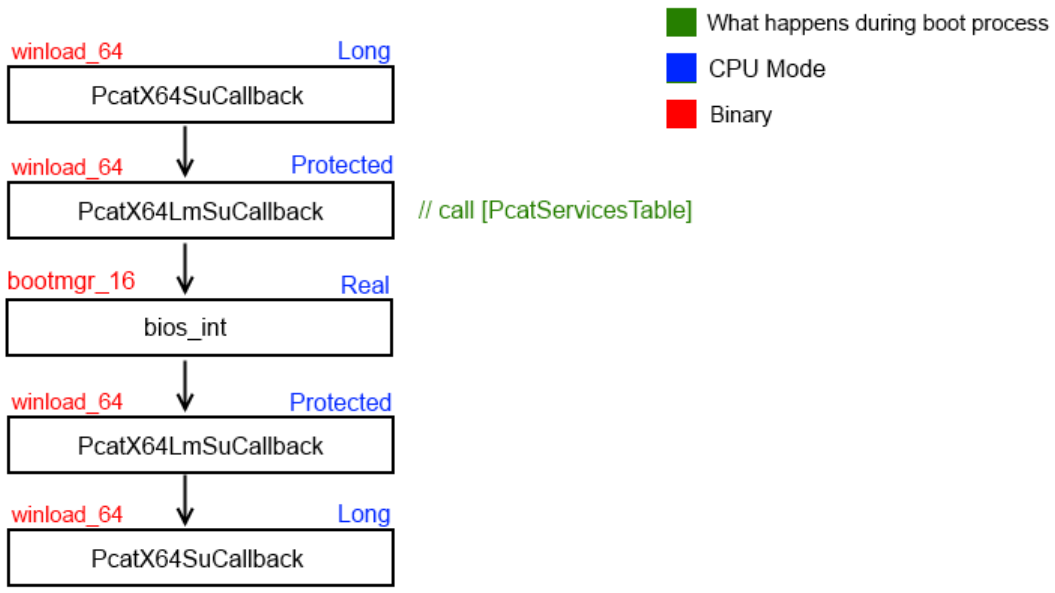


**Figure 5.** Appel d'une interruption BIOS depuis le mode 64 bits

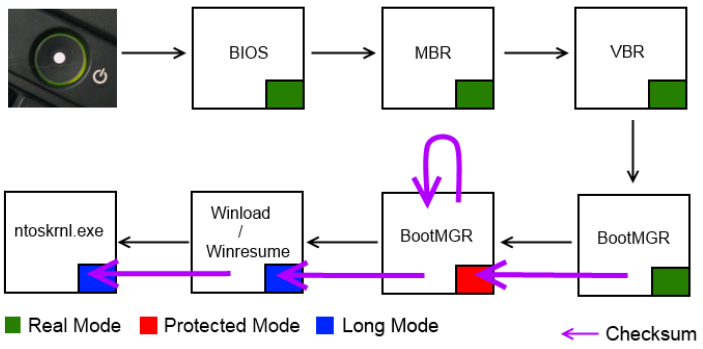


**Figure 6.** Chaîne de confiance

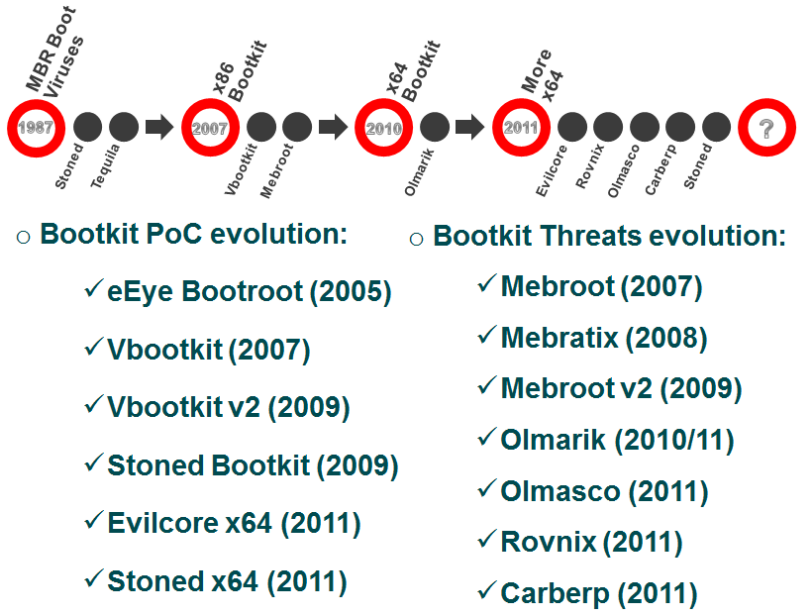


**Figure 7.** Évolution des bootkits (http ://www.welivesecurity.com/)

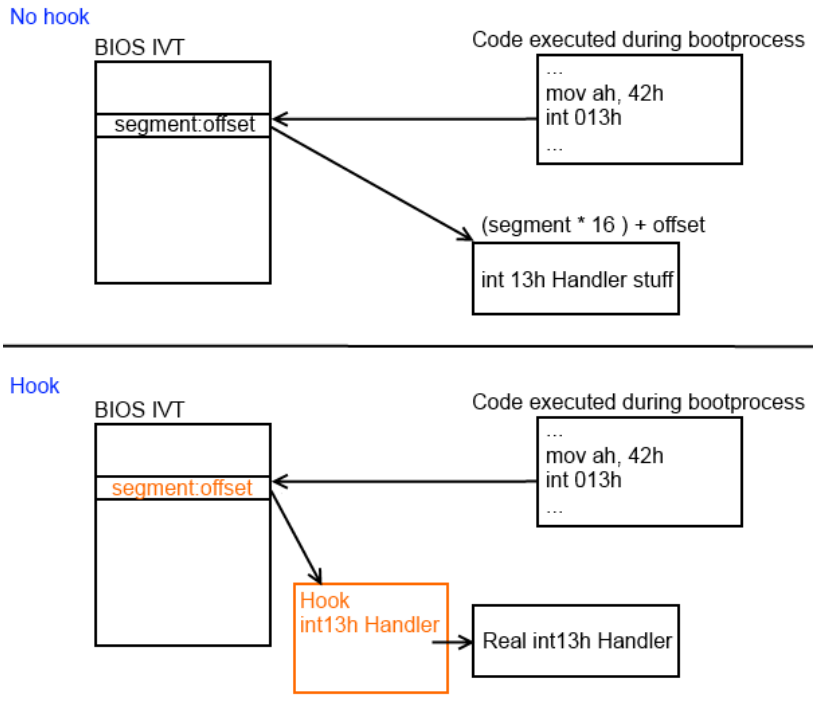


**Figure 8.** Hook Interruption BIOS

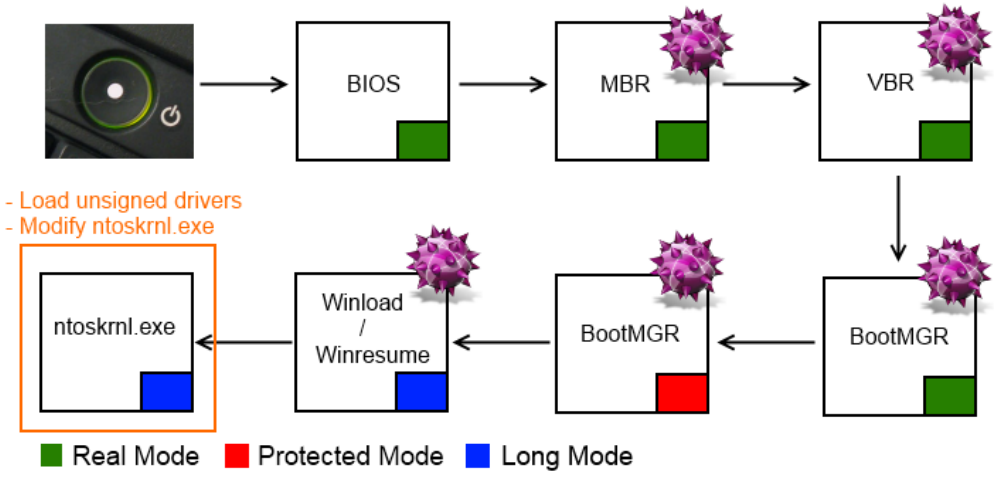


**Figure 9.** Processus de boot infecté

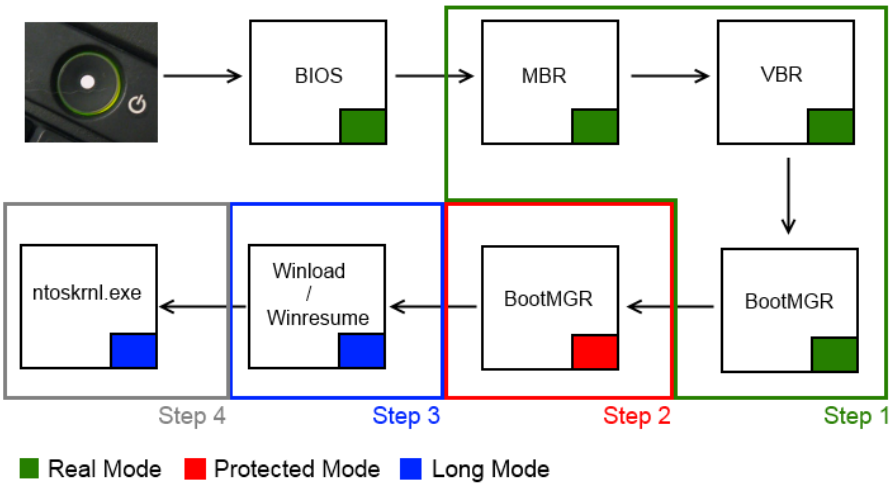


**Figure 10.** Quatre étapes

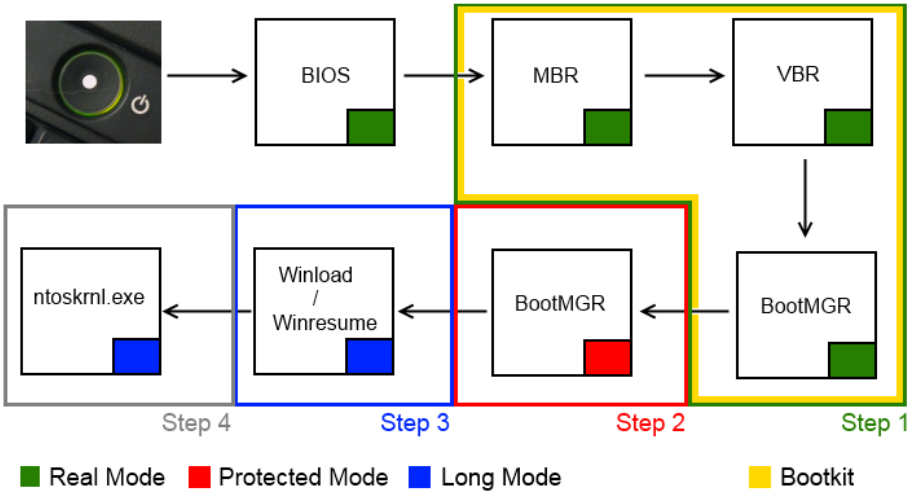


**Figure 11.** Première étape résolue avec le mode v8086

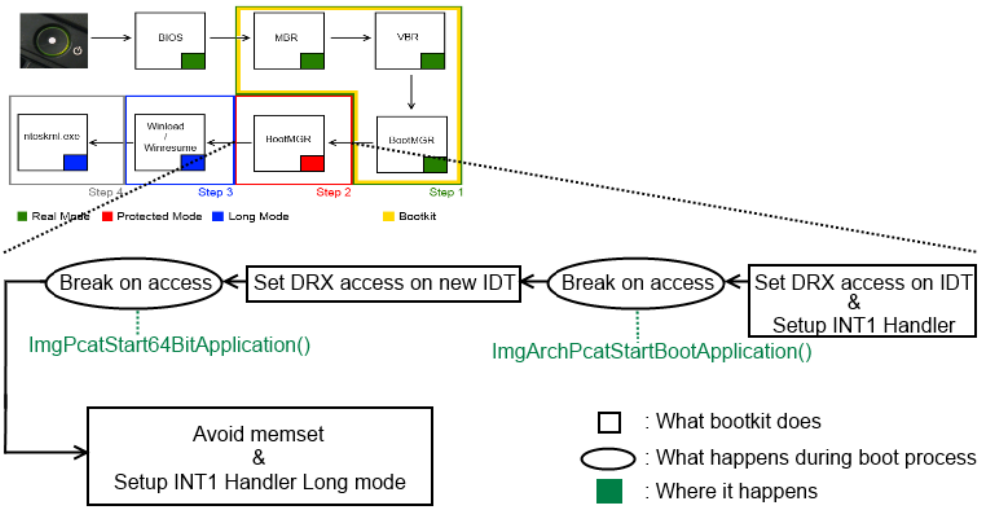


**Figure 12.** Technique avec les registres de debug

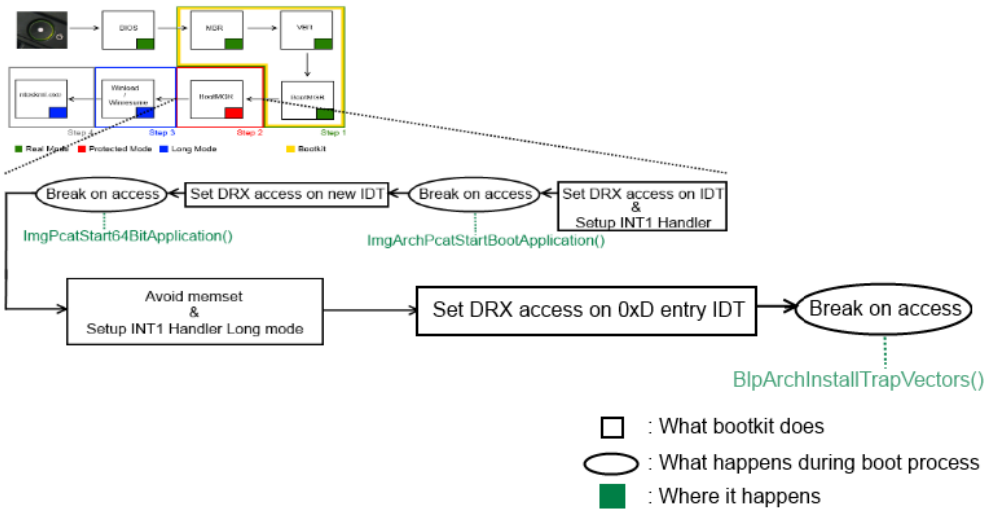


**Figure 13.** Passage du mode protégé au mode long

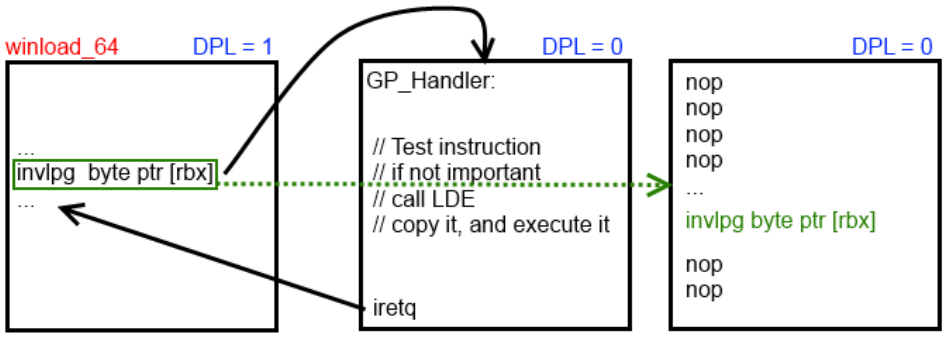


**Figure 14.** Action du gestionnaire de General Protection Fault