REboot: Bootkits Revisited

Samuel Chevet

Jeudi 5 Juin 2014

Agenda

- Qu'est ce qu'un bootkit
- Processus de boot Windows
- État de l'art des bootkits
- Le projet REboot
- Conclusion

Plan

Rootkit

- Type de logiciel malveillant
- Réside en espace kernel
- Avec le plus de privilèges
- Dissimulation d'activité
- Ajoute / Remplace des portions de code de l'OS
- Utilisé parfois dans des solutions de protection logicielle

Bootkit

Problèmes depuis Windows 64 bits

- Signature des drivers obligatoire
- Achat ou vol de certificat
- Protection kernel

Nouvelles attaques

- Compromettre le système de démarrage
- Outrepasser la signature des drivers en 64 bits
- Charger des drivers non signés
- Botnets: Spam, vol d'informations d'identification, DDOS, . . .

Bootkit

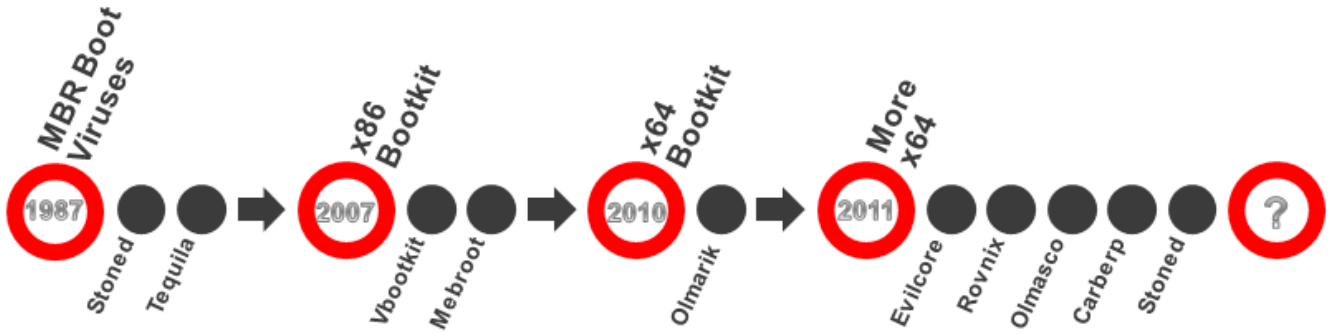
Problèmes depuis Windows 64 bits

- Signature des drivers obligatoire
- Achat ou vol de certificat
- Protection kernel

Nouvelles attaques

- Compromettre le système de démarrage
- Outrepasser la signature des drivers en 64 bits
- Charger des drivers non signés
- Botnets: Spam, vol d'informations d'identification, DDOS, . . .

Bootkit



- **Bootkit PoC evolution:**
  - ✓ eEye Bootroot (2005)
  - ✓ Vbootkit (2007)
  - ✓ Vbootkit v2 (2009)
  - ✓ Stoned Bootkit (2009)
  - ✓ Evilcore x64 (2011)
  - ✓ Stoned x64 (2011)
- **Bootkit Threats evolution:**
  - ✓ Mebroot (2007)
  - ✓ Mebratix (2008)
  - ✓ Mebroot v2 (2009)
  - ✓ Olmarik (2010/11)
  - ✓ Olmasco (2011)
  - ✓ Rovnix (2011)
  - ✓ Carberp (2011)

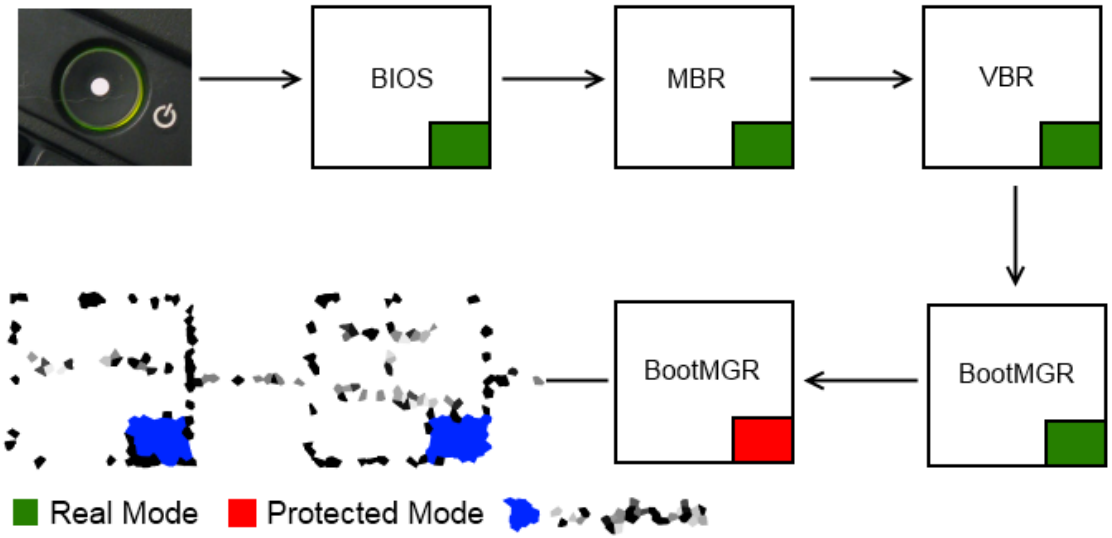
Évolution des bootkits (`http://www.welivesecurity.com/` ©)

Plan

2 Introduction

- Processus de démarrage
- Chaîne de confiance

Boot process



Winload

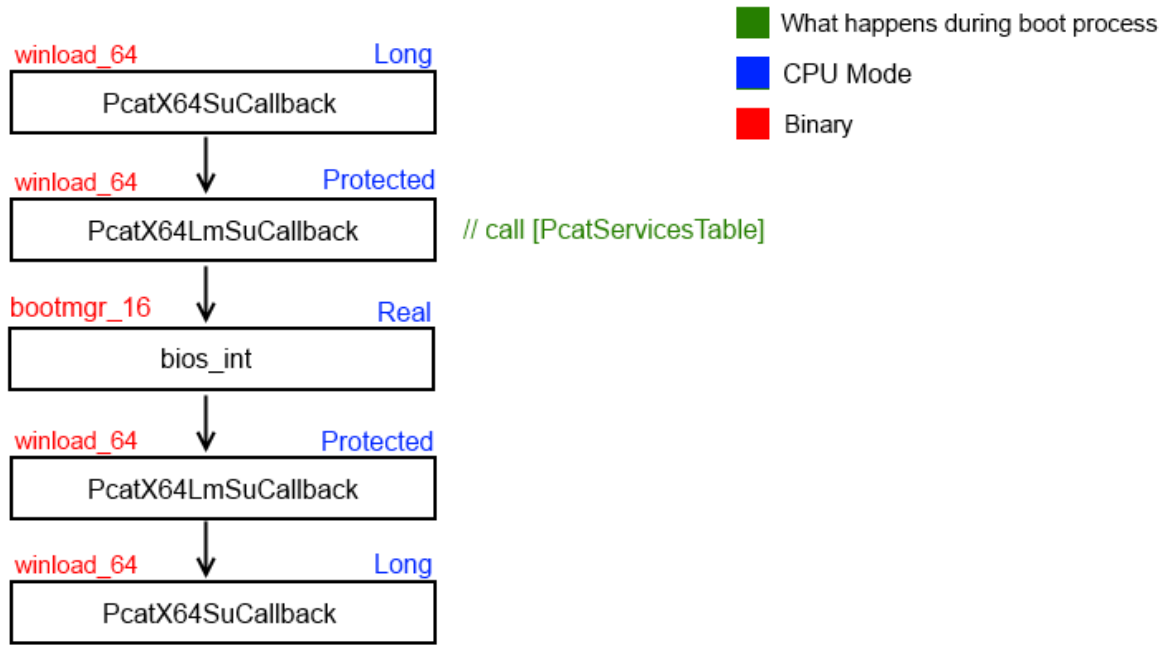
- Mise en place d'un kernel 64 bits minimaliste
- Activation de la pagination
- Récupération des options de démarrage (DISABLE_INTEGRITY_CHECKS, TESTSIGNING, ...)
- Chargement des entrées BCD
- Remplir la structure **LOADER_PARAMETER_BLOCK**
- Chargement des clés de registre (system32\config\system)
- Chargement de **Ntoskrnl.exe, hal.dll, drivers de type SERVICE_BOOT_START**
- Création de PsLoadedModuleList

Entrées dans la Global Descriptor Table(GDT)

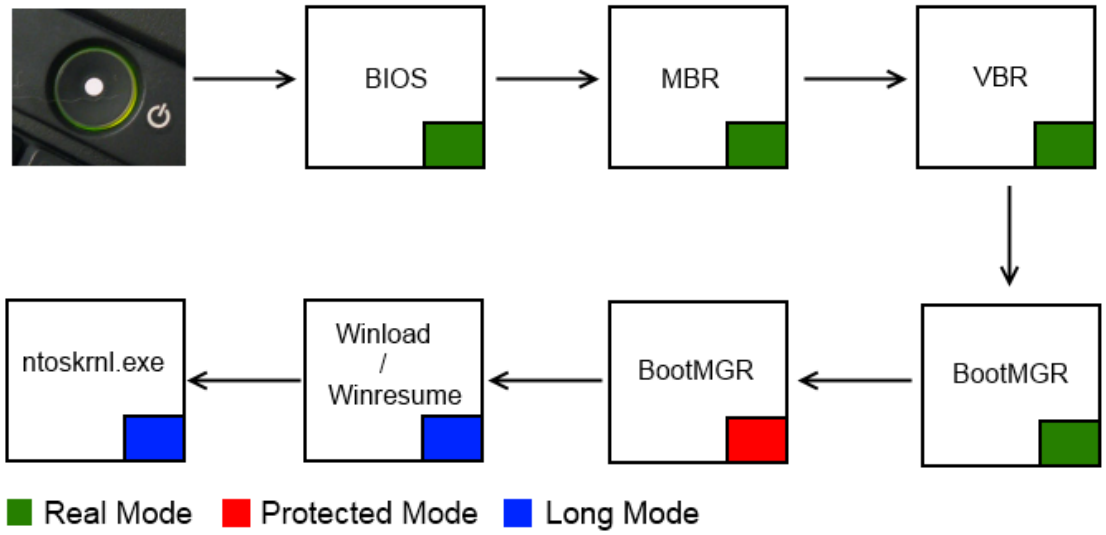
- Entrée de code pour le long mode
- Entrée de code pour le mode protégé
- Entrée de data pour le mode protégé
- TSS pour le long mode
- Entrée de code pour le mode réel
- Entrée de data pour le mode réel
- Entrée pour le framebuffer (0x000B8000)

Interruption BIOS depuis le long mode

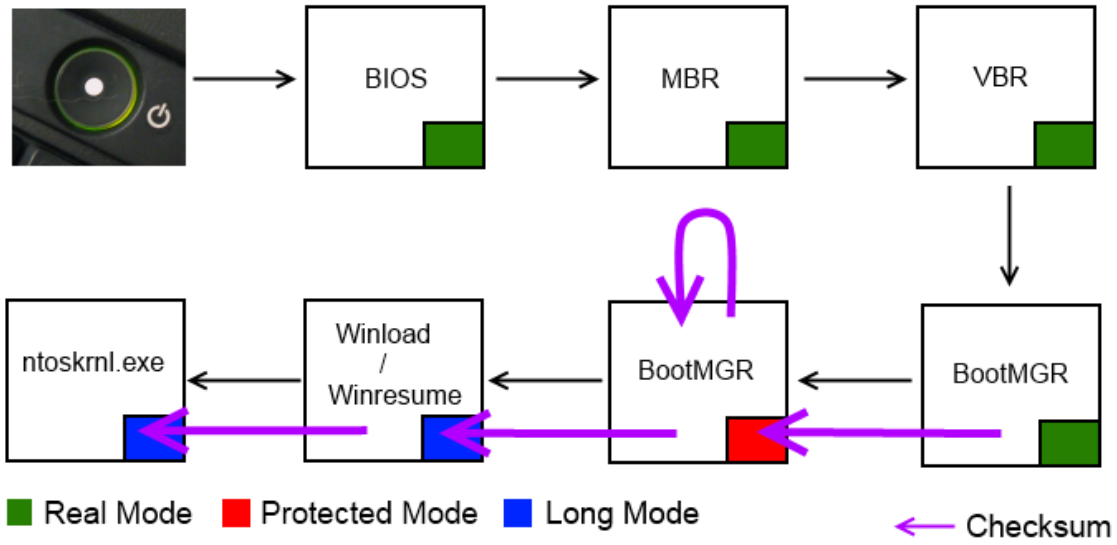
- Winload a besoin de lire / écrire des fichiers
- Écrire à l'écran, récupérer des entrées claviers, . . .
- **Winload est capable d'exécuter des interruptions BIOS**



Boot process



Chaîne de confiance



Plan

3 État de l'art

- Types d'infections
- Payload
- Problèmes

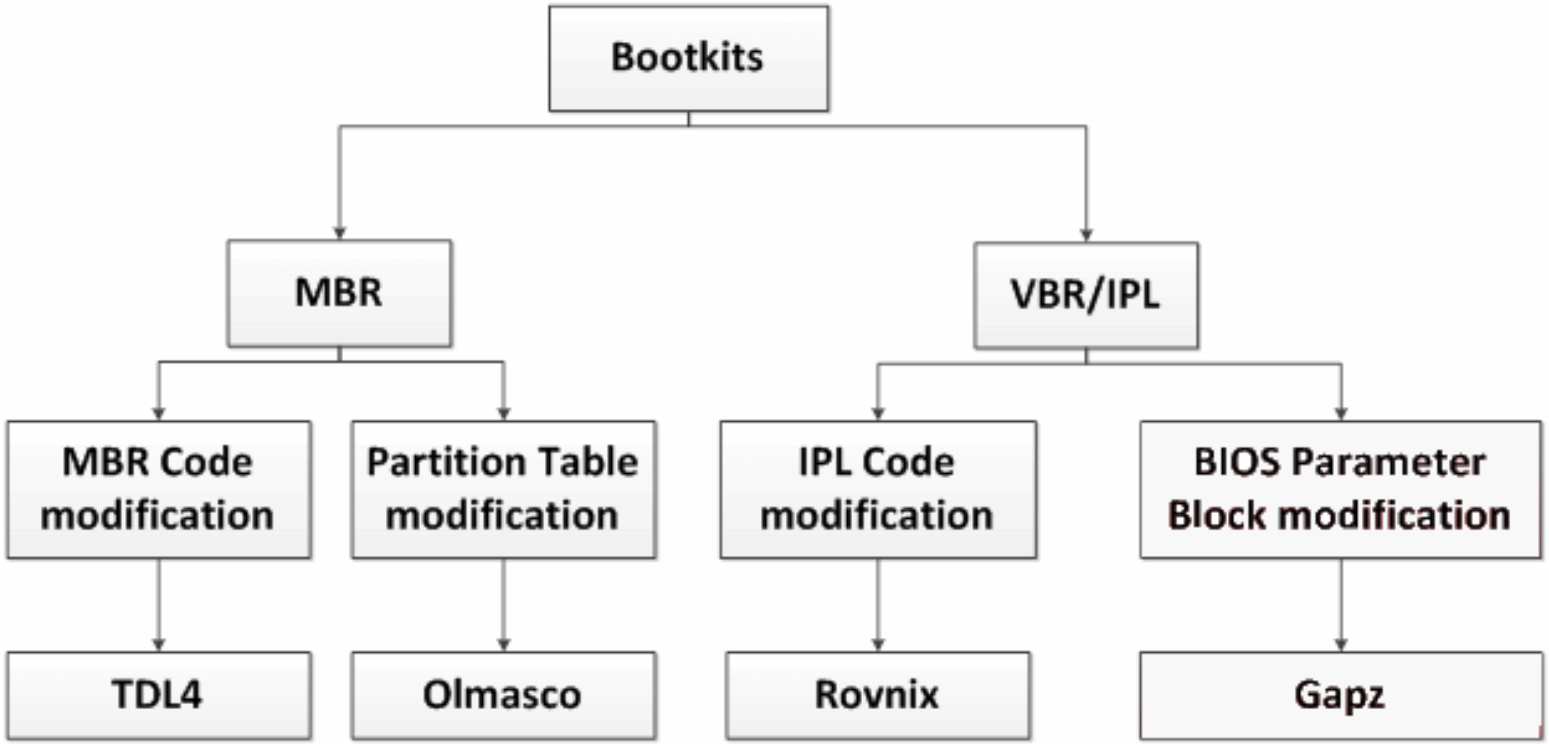
État de l'art

- En 2010, les auteurs de malwares ont commencés à attaquer les systèmes 64 bits
- TDL, aka Alureon, famille de malware

Exemples de bootkits

- TDL4
- Turla
- gapz
- xpaj
- Cidox
- yurn
- prioxer
- rovnix
- ...

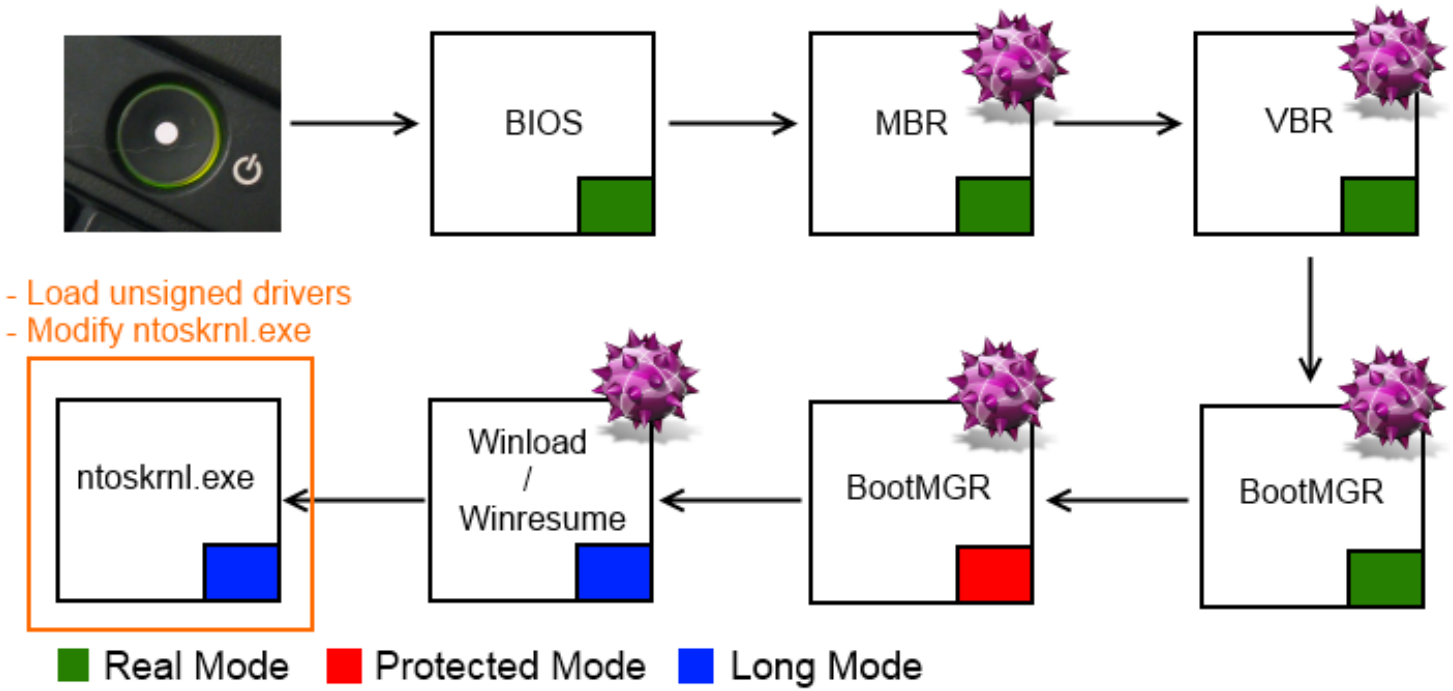
Types d'infections



Techniques d'infection (http://www.welivesecurity.com/ ©)

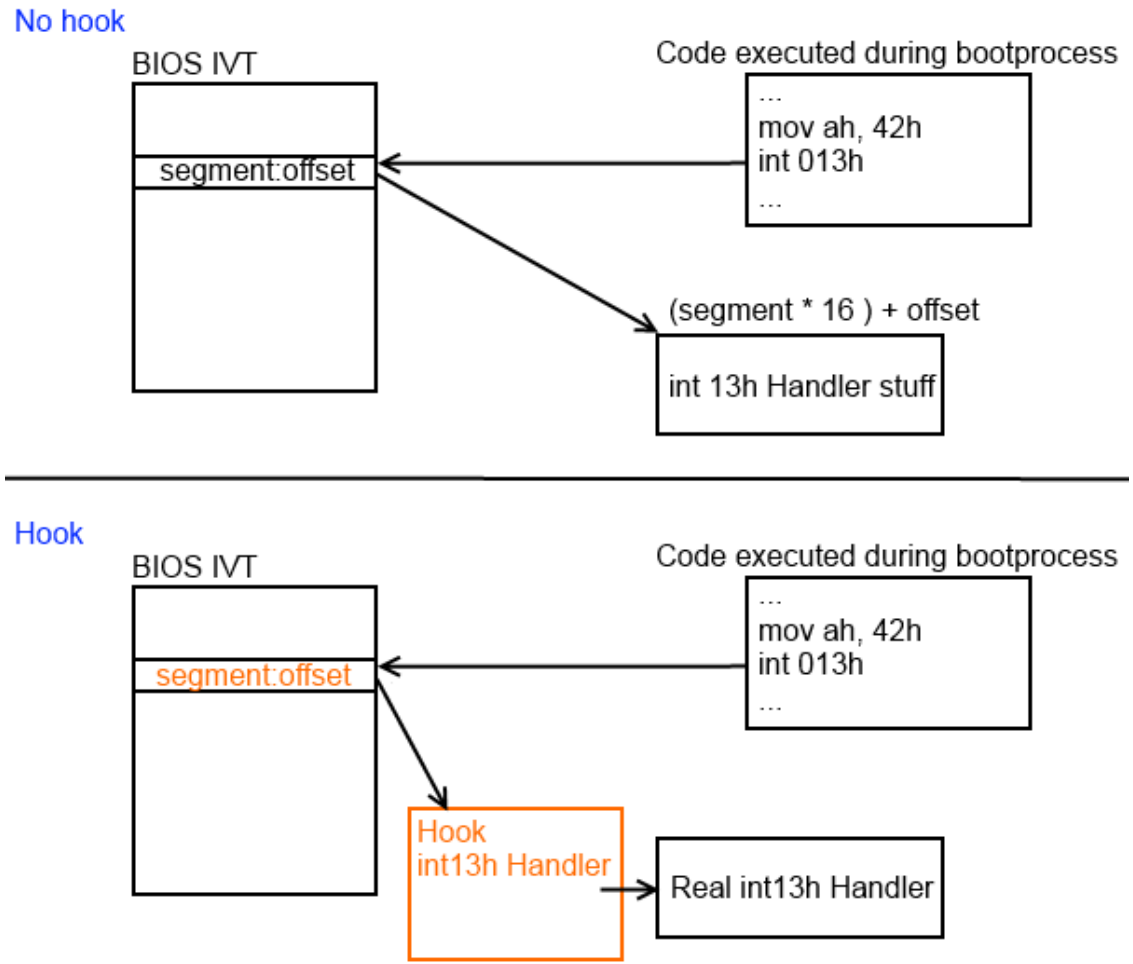
Payload

- Exécution de code durant le processus de boot jusqu'au chargement de ntoskrnl.exe
- Le payload final est injecté à l'étape de ntoskrnl.exe



Payload

- Le BIOS met à disposition des interruptions
- int 013h (Fcnt : 042h) : EXTENDED READ
- Mise en place d'un Hook sur cette interruption
- Utilisée pour n'importe quelle méthode d'infection



L'intérêt des Hooks

- Scanner le résultat de toutes les opérations de disque
- Patcher les fichiers en mémoire
- Mise en place de trampolines pour les prochaines étapes
- (Ex : MBR -> VBR, VBR -> BootMGR, . . .)
- Le but final est d'arriver au chargement de ntorksnl.exe
- Charger un driver non signé
- Désactiver les protections

Projet Open-Source

- StonedBootkit
- VBootkit
- DreamBoot
- . . .

Problèmes

- Se concentre seulement les binaires en eux mêmes (VBR, BootMGR_16, BootMGR_32, Windload)
- La plupart des bootkits ne se basent que sur la modification de code et la mise en place de Hooks:
  - Utilise des signatures et des offsets hardcodés
  - Exige le patch de la chaîne de confiance
- Ces techniques ne sont pas fiables:
  - Non générique en fonction des différentes versions de Windows
  - TrueCrypt & BitLocker ne sont pas supportés (sauf un projet: StonedBootkit)
  - Facilement détectable

Plan

4 REboot

- Recherches
- Du mode réel au mode protégé
- Du mode protégé au long mode
- De Winload à Ntoskrnl
- Payload

Recherches

- Créer une preuve de concept capable de contrôler chacune des étapes du processus de boot jusqu'au chargement du kernel
- N'utilisant pas les techniques déjà connues

Objectifs

- Trouver un moyen d'implémenter un bootkit de manière générique
- **Outrepasser la chaîne de confiance**
- **Charger un driver non signé**

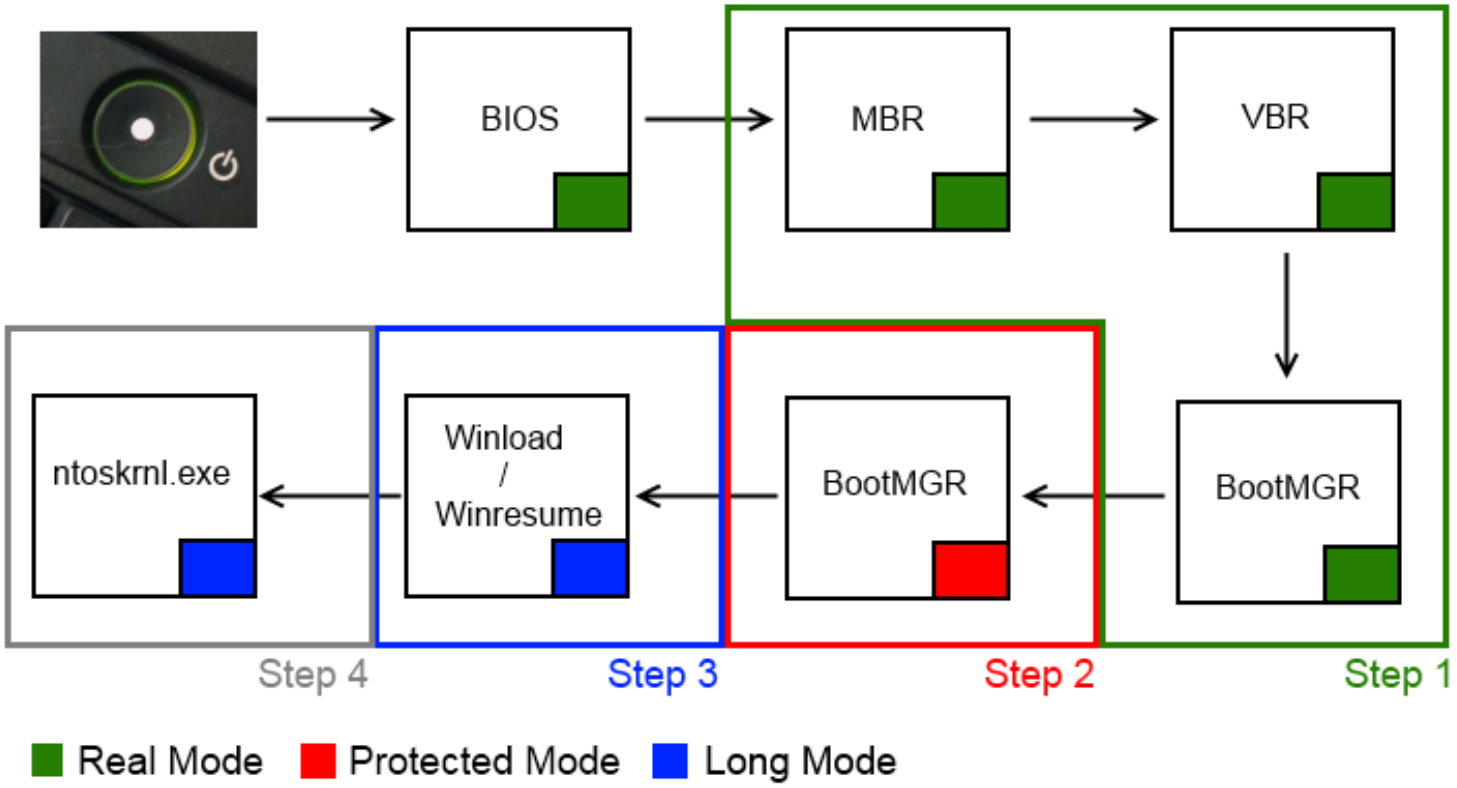
Idées

- Durant le boot, le seul probème est le changement de mode du CPU :
  - Mode réel (16 bits)
  - Mode protégé (32 bits)
  - Long mode (64 bits)
- Nous voulons être capable d'exécuter du code à chacune de ces étapes
- **Sans la recherche de signature ou mise en place de Hooks**
- Nous allons seulement utiliser les fonctionalités offertes par notre processeur

Quatre étapes

1. Passage du mode réel (16 bits) au mode protégé (32 bits)
2. Passage du mode protégé au long mode (64 bits, Winload)
3. Passage de Winload à ntoskrnl
4. Exécution du payload

Quatre étapes



Plan

4 REboot

- Recherches
- Du mode réel au mode protégé
- Du mode protégé au long mode
- De Winload à Ntoskrnl
- Payload

Virtual 8086 mode

- Virtual 8086 est un sous mode du mode protégé
- Possibilité d'exécuter du code 8086 en mode protégé
- NTVDM
- Virtual machine (VM) bit dans le registre EFLAGS (bit #17)
- Nous n'avons besoin que d'une seule tâche
- L'instruction popf ne marche pas, il faut utiliser iret ou une tss 386
- **Surveillance des instructions privilégiées comme lgdt**

Virtual 8086 mode

Problèmes rencontrés

- Utilisation d'un level de privilège I/O (IOPL) égal à 3
- Surveillance seulement des instructions privilégiés
- Les interruptions du BIOS pour TPM passent en mode protégé
- Faux positifs sur la détection de BootMGR

Virtual 8086 mode

Solution

- Utilisation d'IOPL égal à 1
- Quand une interruption veut être exécutée
  1. On revient en mode réel
  2. On exécute l'interruption voulue
  3. On repasse en mode 8086

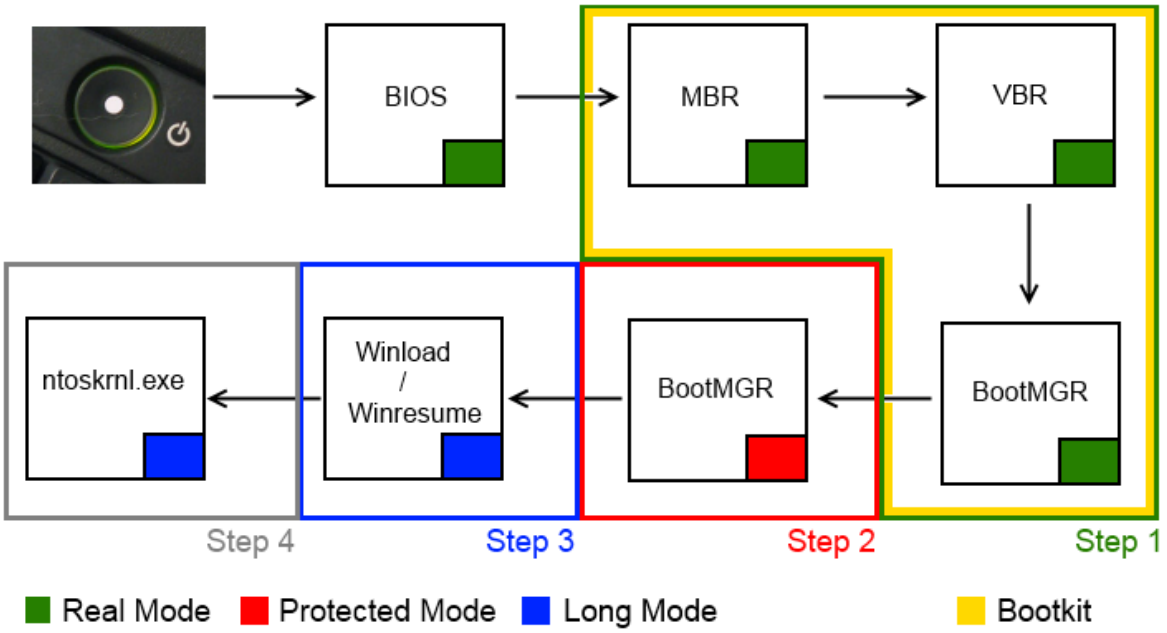
Virtual 8086 mode

Étape par étape

- Mise en place du mode protégé
- Chargement du MBR original
- Préparation du mode 8086
- Exécution du MBR original
- Gestion de chacune des execptions
- GP Handler executé sur l'instruction lgdt

Du mode réel au mode protégé

Première étape résolue en utilisant le mode 8086



Plan

4 REboot

- Recherches
- Du mode réel au mode protégé
- Du mode protégé au long mode
- De Winload à Ntoskrnl
- Payload

Du mode protégé au long mode

- Avec le mode v8086, nous contrôlons l'exécution jusqu'à BootMGR_32
- BootMGR_32 doit :
  - Préparer le passage en long mode dans le cas d'un kernel 64 bits
  - Mettre en place une nouvelle GDT et IDT
  - Activer la pagination
- **La GDT et l'IDT devra être placé sur une page allouée**
- Toutes ces opérations sont faites par la fonction ImgArchPcatStartBootApplication()

Mode protégé au long mode

ImgArchPcatStartBootApplication()

- Mise en place d'une page pour stocker la GDT et l'IDT
- Utilisation de l'instruction sidt pour charger et recopier les entrées dans ce nouvel emplacement
- Test du flag PIMAGE_FILE_HEADER->Machine pour le démarrage d'un os 32 ou 64 bits

ImgPcatStart64BitApplication()

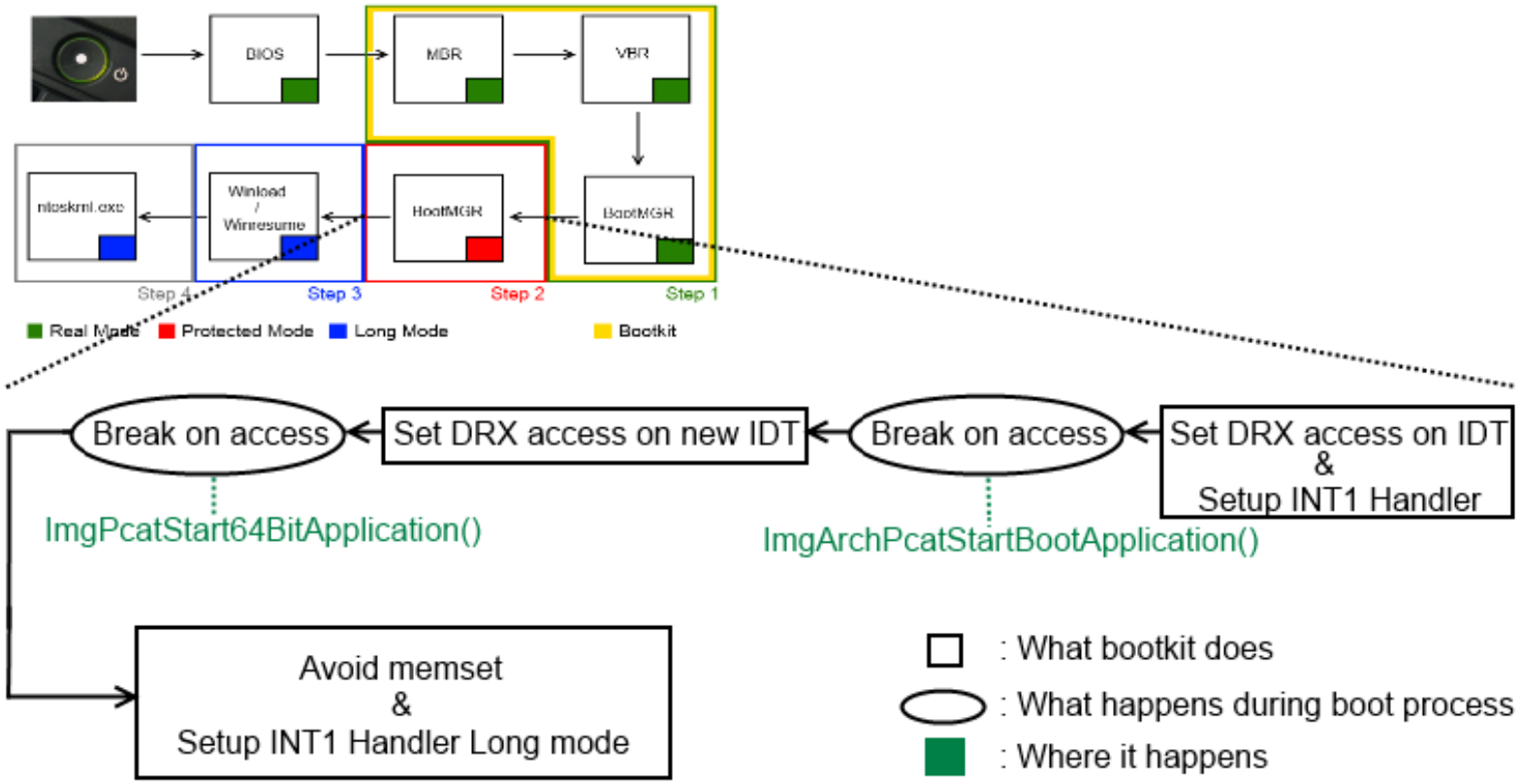
- Cas d'un OS 64 bits
- **Mise à zéro de toutes les entrées de l'IDT car elles sont invalides en long mode**

Mode protégé au long mode

Lorsque nous sommes en mode protégé nous pouvons :

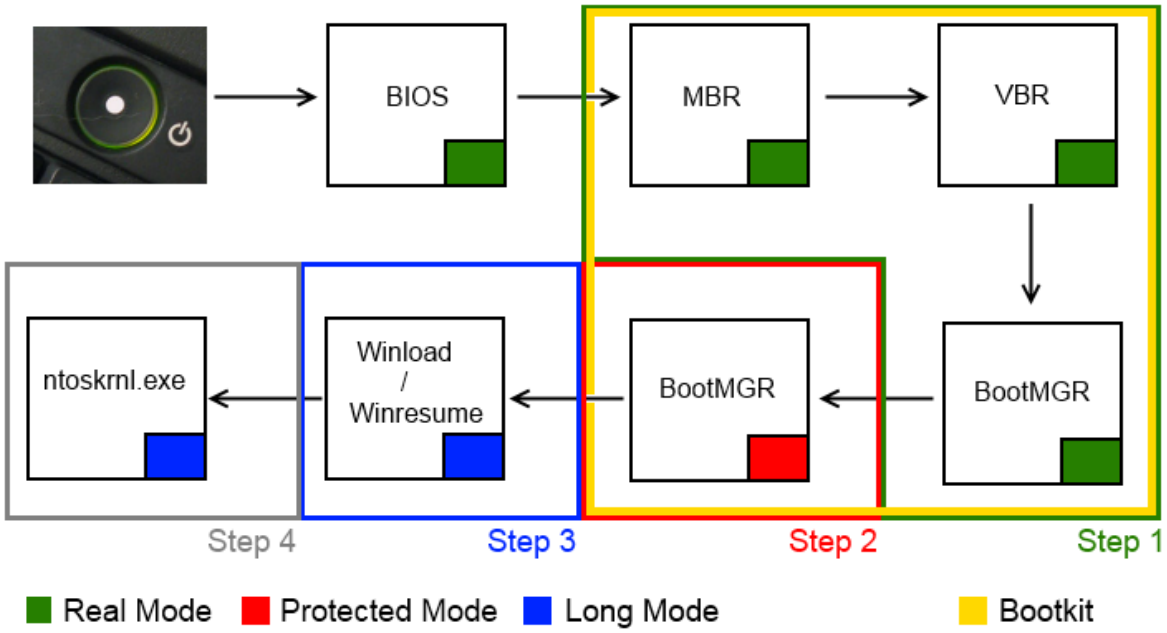
- Utiliser les registres de debug (dr0 . . . dr3)
- Installer un hanlder pour l'interruption 0x1
- On contrôle l'exécution jusqu'à Winload

Mode protégé au long mode



Mode protégé au long mode

Deuxième étape résolue en utilisant les registres de debug



Plan

Winload à Ntoskrnl

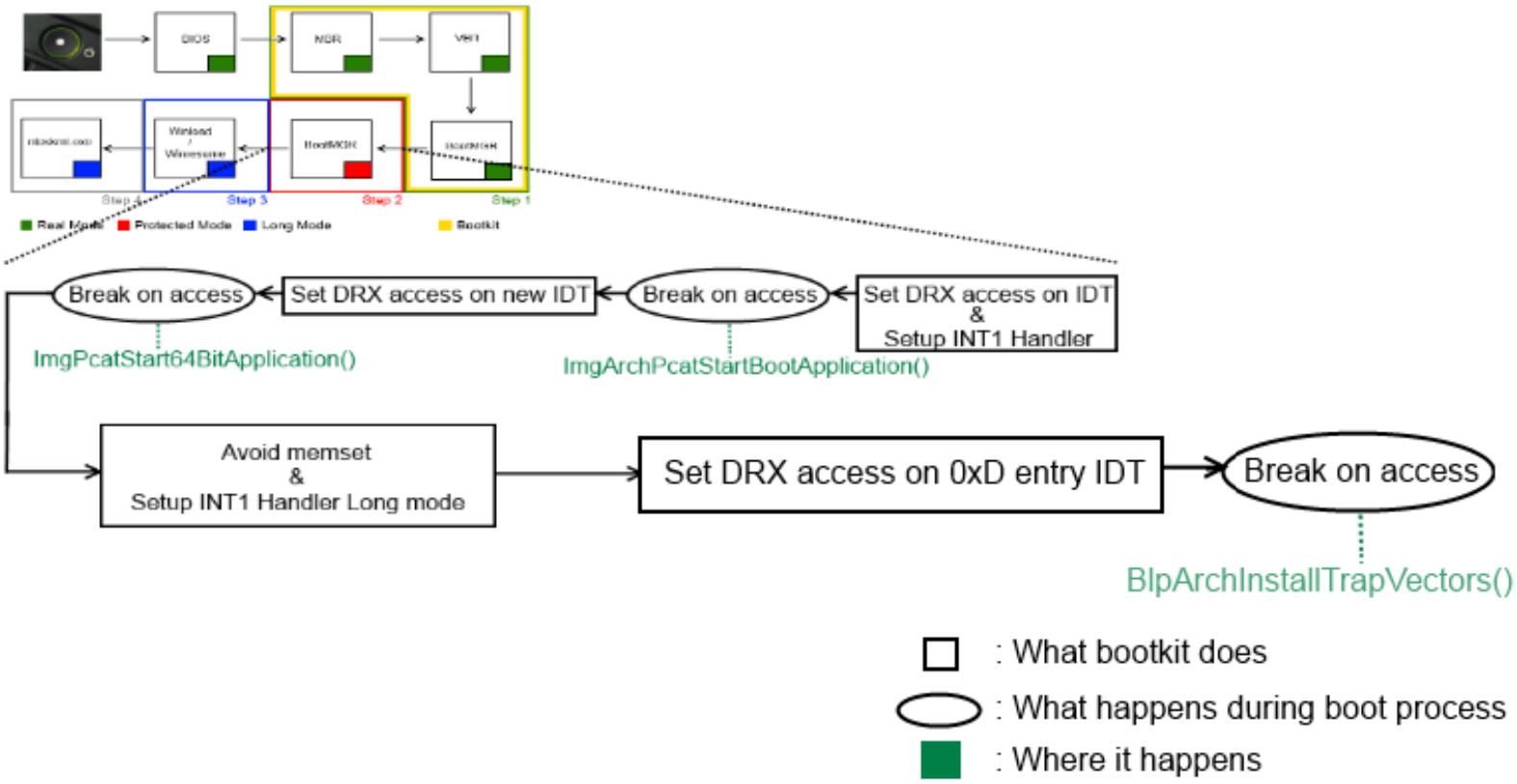
- Avec les registres de debug nous contrôlons jusqu'à l'exécution de Winload
- Windload démarre avec une IDT vide (memset 0)

BlpArchInstallTrapVectors()

- La fonction ArchGetIdtRegister() récupère l'IDTR et met en place des nouvelles entrés spécifiques

- Nous pouvons installer un DRX en écriture sur ces entrées avant de passer du mode protégé au long mode

Winload à Ntoskrnl



Winload à Ntoskrnl

Samuel Chevet

- Nous gardons le contrôle pendant l'exécution de Winload
- Nous voulons surveiller la transition entre Winload et Ntoskrnl
- **Winload va mettre en place une nouvelle GDT et IDT avant d'exécuter le "vrai" kernel**
- Nous pouvons tracer les instructions privilégiées
- Nous passons le segment de code de Windload en DPL égal à 1

Pourquoi ring 1?

- Le code de winload se situe sur des pages avec le bit supervisor set

The page-level protection mechanism allows restricting access to pages based on two privilege levels:

- Supervisor mode (U/S flag is 0)—(Most privileged) For the operating system or executive, other system software (such as device drivers), and protected system data (such as page tables).
- User mode (U/S flag is 1)—(Least privileged) For application code and data.

The segment privilege levels map to the page privilege levels as follows. If the processor is currently operating at a CPL of 0, 1, or 2, it is in supervisor mode; if it is operating at a CPL of 3, it is in user mode. When the processor is

Ring 1

- Mise en place de nouveaux segments de code et données, et modification des anciennes entrées en DPL 1
- Mise en place de notre General Protection Fault handler
- Mise à jour du champ rsp0 dans la TSS_64

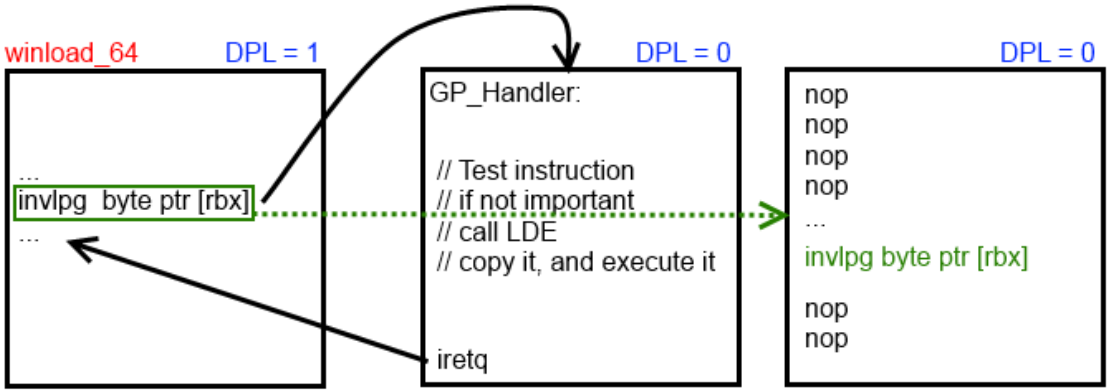
GP Handler

- Où est-ce que la GP fault a été déclenchée?
- Quelle instruction privilégiée a voulu être exécutée?
- Copie et exécution de cette instruction sur un segment avec un DPL de 0
- Ou dans des cas spécifiques nous allons l' "émuler"

Ring 1

Exemples

- mov ds, ax
- mov rax, cr3
- jmp far . . .
- . . .



Ring 1 : Cas spéciaux

mov ds, ax

- Au sein de la fonction PcatX64SuCallback
- Mise à jour du segment de data pour exécuter une interruption BIOS (passage de long mode à mode réel)
- Pour éviter tout problème nous restaurons tout en ring0
- Nous attendons le retour du mode réel (jmp far 10h:343D31h)

jmp far XX:YYYY

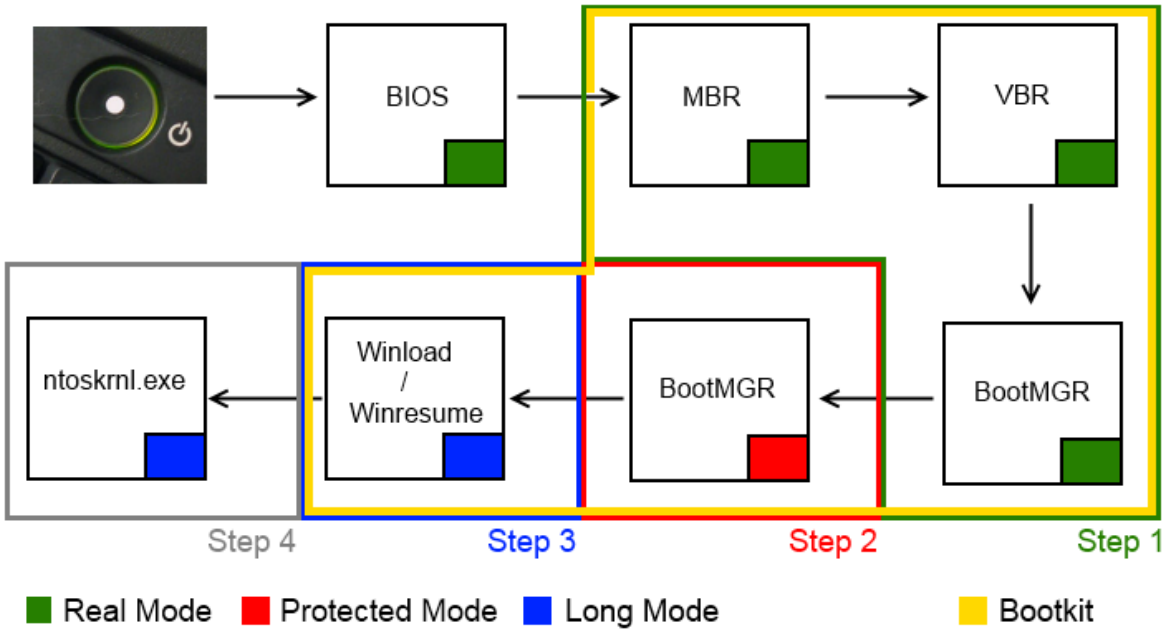
- GP handler appelé car DPL != RPL
- Mise à jour de cs, ss et ip avant iretq

Winload à Ntoskrnl

- Tous les autres cas peuvent être recopiés autre part et être exécutés
- Le dernier cas sera : lgdt fword ptr [rax]
- Dans la fonction : OslArchTransferToKernel
- Juste avant le passage à Ntoskrnl.exe
- Premier paramètre de KiSystemStartup() est LOADER_PARAMETER_BLOCK
- +0x10 : _LDR_DATA_TABLE_ENTRY (boot driver)

Winload à Ntoskrnl

Troisième étape résolue grâce à l'isolation de privilèges



Plan

4 REboot

- Recherches
- Du mode réel au mode protégé
- Du mode protégé au long mode
- De Winload à Ntoskrnl
- Payload

Juste avant Ntoskrnl.exe

- On injecte notre propre driver dans la liste PsLoadModuleList
- Nous avons accés aux APIs exportées par ntoskrnl
- Le kernel n'est pas complètement initialisé
- On remplace le point d'entrée d'un driver
- Mais la plupart des point d'entrée des drivers sont appelés par hal.dll
- On remplace la fonction exportée par kdcom.dll (KdDebuggerInitialize1)

Payload

- Nous ne voulons pas injecter un payload spécifique
- Notre but est de charger un driver non signé
- Utilisation d'une méthode non documentée pour outrepasser la vérification de signatures

Méthode non documentée

- IoCreateDriver(PUNICODE_STRING DriverName, PDRIVER_INITIALIZE InitializationFunction)
- Fonction exportée par Ntosrknl.exe pour créer un objet driver
- L'argument DriverName peut être null

Payload

- Nous ne voulons pas injecter un payload spécifique
- Notre but est de charger un driver non signé
- Utilisation d'une méthode non documentée pour outrepasser la vérification de signatures

Méthode non documentée

- IoCreateDriver(PUNICODE_STRING DriverName, PDRIVER_INITIALIZE InitializationFunction)
- Fonction exportée par Ntosrknl.exe pour créer un objet driver
- L'argument DriverName peut être null

Payload

InitializationFunction

- Ouverture et lecture d'un fichier PE type driver
- Mapper les sections en mémoire
- Résoudre les imports
- Fixer les rélocations
- Finir de remplir les informations dans le DRIVER_OBJECT
- On appelle le point d'entrée

5 Conclusion

Demo

Demo time !

TODO

TODO

- Implémentation avec UEFI (sans SecureBoot)
- Jouer plus avec BitLocker et TrueCrypt: Extraire les passphrases au démarrage

Conclusion

- Vrai intérêt d'utiliser les bootkits, pour le chargement de drivers non signés
- REBoot n'utilise pas de modifications en mémoire!
- La chaîne de confiance est intacte
- Fonctionne sur n'importe quelle version de Windows 64 bits
  - Environnement virtualisé ou émulateur
  - Machine physique avec un BIOS ou en mode UEFI legacy
- Ne fonctionne pas avec SecureBoot ou une solution de chiffrement en mode TPM

Questions ?

Merci de votre attention