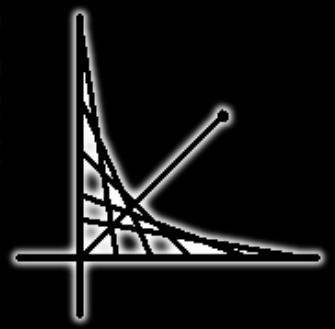
GoRing0

Le mode noyau à portée de main



Trance
Emilien Girault
trance@ghostsinthestack.org
www.ghostsinthestack.org
www.emiliengirault.fr

Introduction

- But : présenter GoRing0
  - Rootkit simple sous Windows
- Notions requises
  - Bases sur l'architecture x86
    - Pile, registres
- N'ayez pas peur !
  - Rappels en 1ère partie
  - Posez des questions !

Rootkits

\x89\x14\x24\xe8\xe8\x0c\x00\x00\x31\xd2\x89\x15\x00\x40\xc8\x6e\xc7\x04\x24\x00\x00

- Un rootkit n'est pas :
  - Un virus
  - Un exploit
- Un rootkit est :
  - Une technologie visant à rendre un code furtif et indétectable
- Un rootkit peut être utilisé pour :
  - L'attaque (piratage)
  - La défense (protection)

Furtivité

\x89\x14\x24\xe8\xe8\x0c\x00\x00\x31\xd2\x89\x15\x00\x40\xc8\x6e\xc7\x04\x24\x00\x00

- Pourquoi ?
  - Maintenir un accès sur un système
- Comment ?
  - Intégration dans le système d'exploitation
  - Manipulations très bas niveau
- Exemples
  - Attaque : backdoors, spywares
  - Défense : anti-virus, protection de contenu

Attirail nécessaire

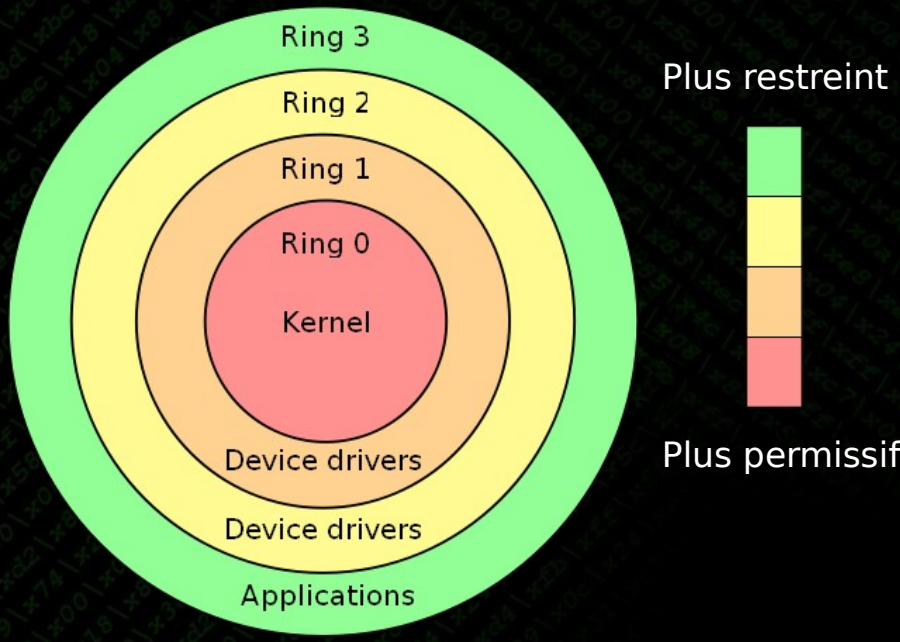
- Notions de bas-niveau
  - Système d'exploitation
  - Architecture matérielle
    - Processeur, mémoire, périphériques...
- Documentation
  - MSDN, Doc Intel, Docs non officielles...
- Environnement de test
  - Windbg, machine virtuelle (ou pas)...

Systèmes d'exploitation (OS)

- OS séparé en deux parties :
  - (1) Programmes utilisateur
  - (2) Modules noyau, *drivers*
- Un rootkit peut s'attaquer aux 2
  - Autrefois la (1), maintenant la (2)
- La complexité d'un rootkit est étroitement liée à celle de l'OS (et du processeur)

Privilèges sous x86

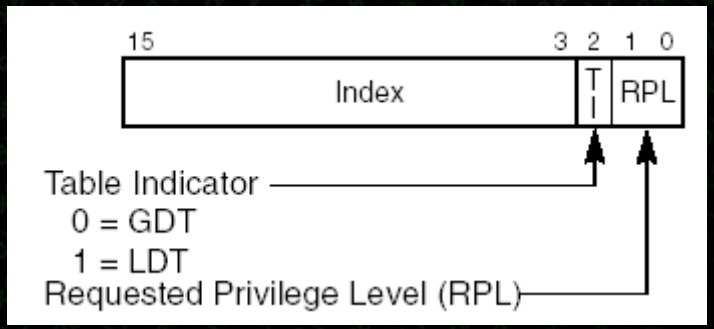
- En mode protégé, le processeur gère matériellement 4 privilèges, ou *rings*
- Les OS classiques ne gèrent souvent que le 0 et le 3
- Certaines instructions nécessitent un ring spécifique



Privilèges sous X86

- Le ring 0 a accès à toutes les instructions
- Le ring 3 ne peut pas appeler les instructions ring 0 sauf dans certains cas
  - Interruptions, SYSENTER, *call gates*
- Le ring courant est appelé CPL
  - CPL = les 2 1ers bits du registre CS

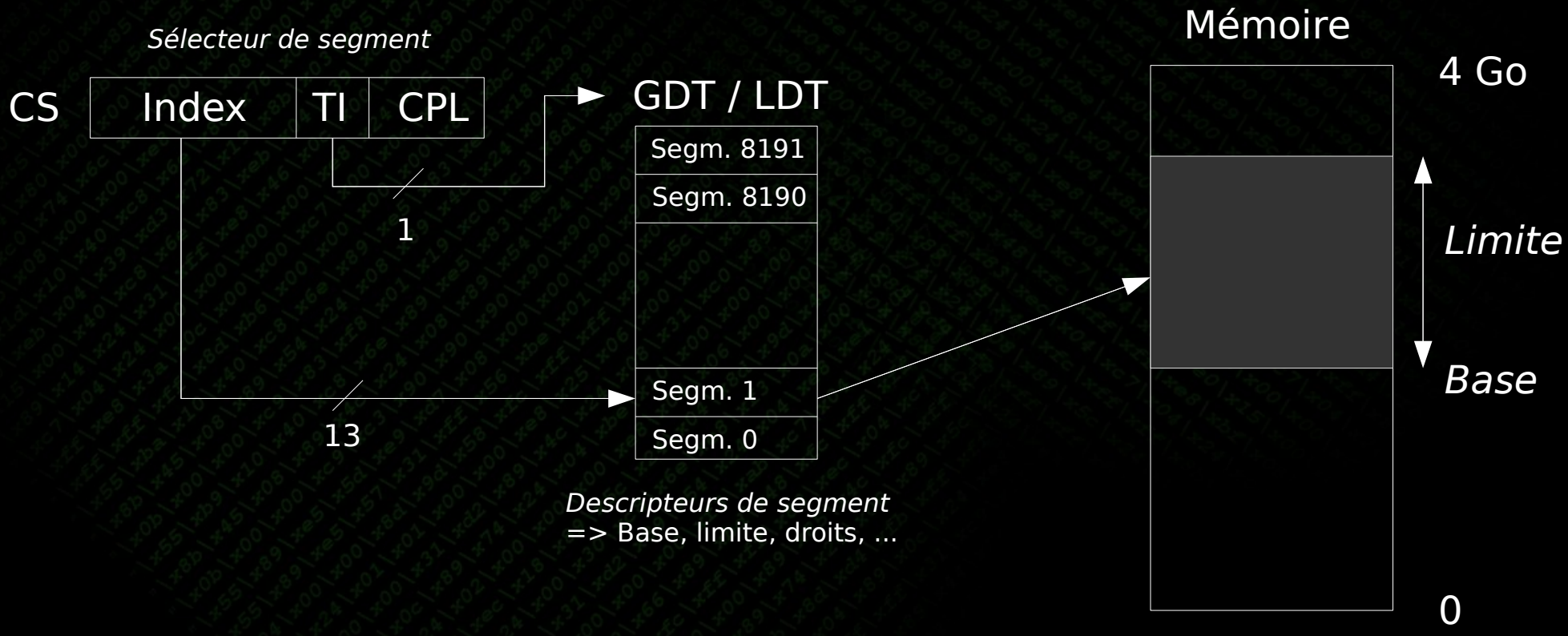
Un registre de segment :



Pour CS, RPL = CPL

Segmentation

- CS : registre de segment pour le code
- Segments = plages mémoire
  - Définis dans une table : la GDT (ou LDT)



Segmentation

- Windows utilise (quasiment) le *flat model*
  - Tous les segments décrivent toute la mémoire (de 0 à 4 Go), sauf un (FS)
  - Certains segment existent en 2 copies :
    - Une pour l'userland (RPL = 3)
    - Une pour le kernelland (RPL = 0)
- Exemple : CS vaut
  - 0x1b en ring 3 (0b11011)
  - 0x8 en ring 0 (0b1000)

GoRing0

- Idée n°1 : si un thread change les 2 1ers bits de CS, il peut passer en ring 0
  - Problème : impossible, pour des raisons de sécurité évidentes...
  - Un thread ring 3 ne peut pas passer en ring 0 *lui-même*
- Idée n°2 : Demander à un autre thread plus privilégié de le faire !
  - Le système, un candidat idéal...
  - Comment communiquer avec lui ?

Interruptions

- Technique visant à faire exécuter une routine lorsqu'un événement se produit
  - Déclenché par les périphériques ou le processeur lui-même
- Interruption = routine + vecteur (IV)
  - Vecteur = numéro entre 0 et 255
    - Certains sont réservés par Intel
  - Routine (ou handler) = code exécuté quand l'interruption se produit
    - Référencée dans une table : l'IDT

Interruptions

- IDT = tableau de descripteurs
- Descripteur d'interruption = structure
  - Pointeur vers la routine à exécuter
  - Segment selector de la routine (dont CPL)
  - DPL : Ring maximal de l'appelant
- Possibilité de déclencher une interruption logiciellement avec l'instruction INT
  - Si desc.DPL = 3 et desc.CPL = 0...
  - ... appel userland - kernelland !

Interruptions

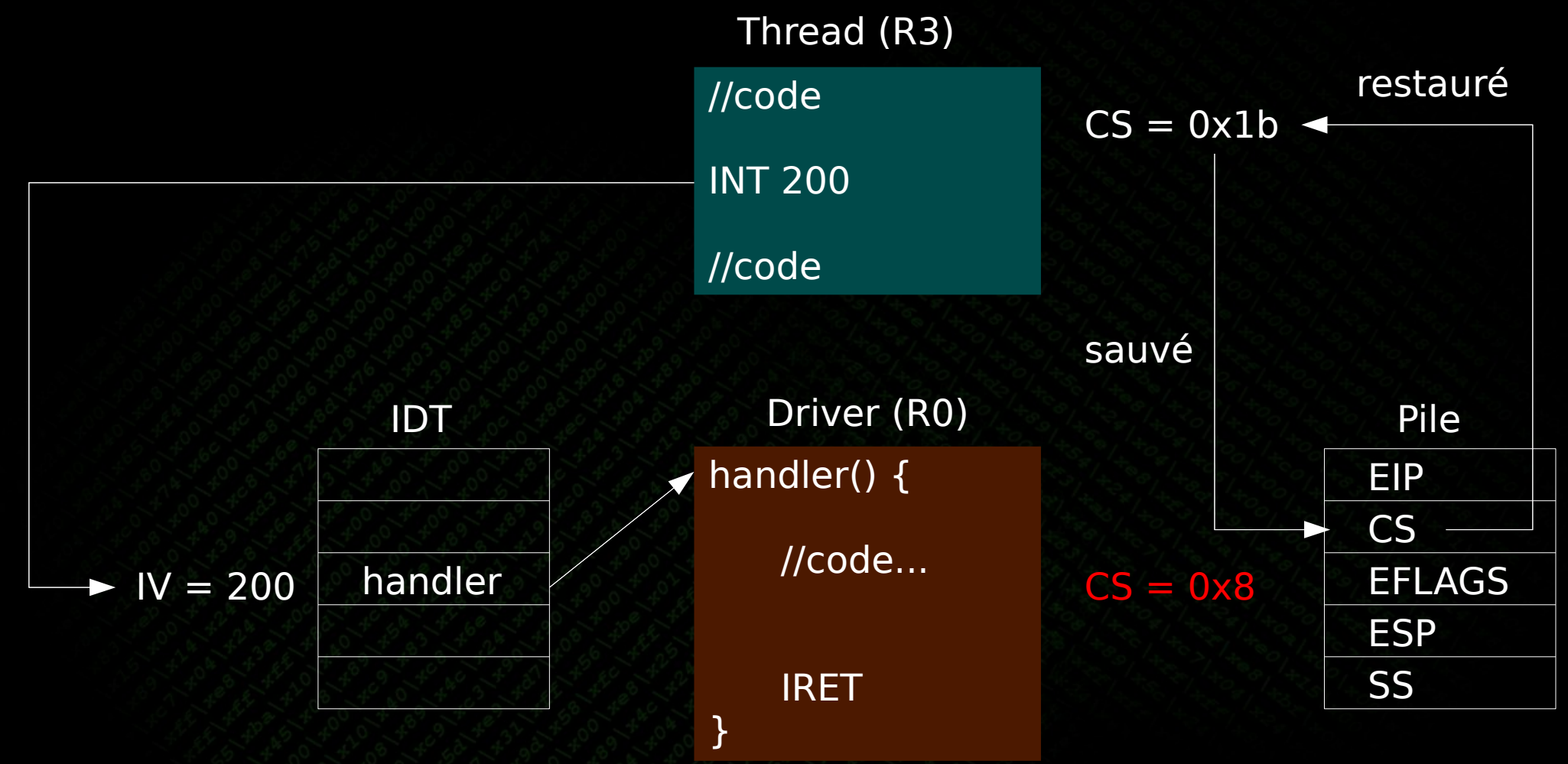
- IDT hooking : remplacer un descripteur
  - Le faire pointer vers notre routine
  - Appel de la routine depuis l'userland : INT
- Une IDT par processeur
  - Nécessité de hooker toutes les IDT
  - Deux méthodes
    - Appel de SIDT sur tous les processeurs
      - Ne fonctionne pas dans certaines VMs
    - Utiliser les structures internes de Windows
      - KPCR.IDT

GoRing0 - Concept

\x89\x14\x24\xe8\xe8\x0c\x00\x00\x31\xd2\x89\x15\x00\x40\xc8\x6e\xc7\x04\x24\x00\x00

- Appeler une interruption revient +/- à appeler une fonction
  - Sauvegarde d'EIP empilée...
  - ...avec en prime, une sauvegarde de CS
    - La valeur qu'il avait en userland
  - Tout cela est restauré à la fin
- Et si la routine modifiait la valeur de CS ?
  - Par exemple... CS = 8 ?
  - A la fin de la routine, le thread appelant se retrouve... en ring 0 :)

Interruptions



GoRing0

\x89\x14\x24\xe8\xe8\x0c\x00\x00\x31\xd2\x89\x15\x00\x40\xc8\x6e\xc7\x04\x24\x00\x00

- Exécution de code ring 0 dans le contexte même du thread courant
  - Même registres (sauf 3)
  - Même pile
- Bascule véritablement le thread en ring 0
  - Différent des appels via IOCTL

GoRing0 – Idée générale

- Un driver (ring 0)
  - my_handler() : MOV [backup CS], 8
  - IDT Hooking : mettre en place le handler
    - IDT[X] = my_handler()
- Une librairie (ring 3)
  - GoRing0() : Passe en ring 0
    - INT X
  - GoRing3() : Revient en ring 3
    - MOV CS, 0x1b

Problèmes (1)

- Le handler doit être appelable en ring 3
  - IDT[X].handler = my_handler()
  - IDT[X].DPL = 3
- X doit être un IV bien choisi
  - Ne correspondant pas à une int. matérielle
    - Comportement différent si appel software ou hardware !
  - Le plus simple : prendre un IV non utilisé
    - 255 :)

Problèmes (2)

- Retour en ring 3 : MOV CS, 0x1b ?
  - Instruction invalide !
  - Solution : Faire comme pour GoRing0
    - my_handler() : MOV [CS bakup], 0x1b
    - GoRing3() : INT 255
- FS écrasé après GoRing3()
  - Reste en kernel (0x30) après GoRing0()
  - Passe à 0 après GoRing3()
    - Restauration manuelle (0x3b) juste après

Problèmes (3)

- Changement de pile (*stack switch*)
  - Se produit lors de int et iret quand on change de ring
  - Or nous souhaitons garder le même ESP
  - GoRing0() : r3 → r0 → r0
    - Après iret, ESP reste en kernel (> 2Go)
    - Rebasculer manuellement sur l'ancienne
  - Goring3() : r0 → r0 → r3
    - ESP pas empilé lors du int → plante au iret
    - Empiler manuellement ESP avant int
      - Empiler aussi SS userland (0x23)

GoRing0 – Code simplifié

RING 0

```
Hook(){
    For_each_processor (p) {
        p.IDT[255].h = my_handler;
        p.IDT[255].DPL = 3;
    }
}


my_handler(){

    int* CSBackup = [esp+4];

    If (*CSbackup != 0x8)
        *CSbackup = 0x8; //CS kernel
    Else
        *CSbackup = 0x1B; //CS user

}
```

RING 3

```
GoRing0(){
    save(ESP);
    INT 255;
    restore(ESP);
}

GoRing3(){
    PUSH 0x23; //SS userland
    PUSH ESP;

    INT 255;

    PUSH 0x3B; //FS userland
    POP FS
}
```

Démo

Idées d'utilisation

- Scanner mémoire
- Manipulation d'objets kernel (DKOM)
  - Élévateur de privilèges
  - Camouflage de processus, fichiers, ...
- Hook (IDT, hardware breakpoints, ...)

→ Fonctionnalités rootkit classiques

Bilan

- Points forts
  - Plus besoin de drivers pour passer en R0
  - Portable sous Linux (en théorie)
  - Supporte le multicore (?)
- Limitations
  - Impossible d'appeler des fonctions kernel
    - Linkage exécutable / kernel ?
  - Non furtif
    - Hook IDT détectable par les AV

Bilan

- Points forts
  - Plus besoin de drivers pour passer en R0
  - Portable sous Linux (en théorie)
  - Supporte le multicore (?)
- Limitations
  - Impossible d'appeler des fonctions kernel
    - Linkage exécutable / kernel ?
  - Non furtif
    - Hook IDT détectable par les AV

Release

\x89\x14\x24\xe8\xe8\x0c\x00\x00\x31\xd2\x89\x15\x00\x40\xc8\x6e\xc7\x04\x24\x00\x00

- Sources et exécutables bientôt dispos sur
  - www.ghostsinthestack.org
  - www.emiliengirault.fr

Références

- Intel Software Developer's Manuals
  - 2A (INT et IRET), 3A (chap 5)
- Blog d'Ivanlef0u
  - http://www.ivanlef0u.tuxfamily.org
- Rootkits, Infiltration dans le noyau Windows
  - James Butler & Greg Hoglund

Questions ?

« Un anneau pour les gouverner tous,
Un anneau pour les trouver,
Un anneau pour les amener tous
Et dans les ténèbres les lier. »