Duqu contre Duqu : rétroconception du driver

Aurélien Thierry (aurelien.thierry@inria.fr), Fabrice Sabatier, Guillaume Bonfante, Jean-Yves Marion

SSTIC, le 7 juin 2013

Introduction

- Duqu, découvert en septembre 2011 par CrySys
- Lié à Stuxnet par certaines fonctionnalités et du code commun

Introduction

- Duqu, découvert en septembre 2011 par CrySys
- Lié à Stuxnet par certaines fonctionnalités et du code commun
- Objet de nombreuses études (Symantec[1], Kaspersky[2], Eset...) sur les fonctionnalités de la charge finale et les méthodes d'infection

[1] Symantec. W32.Duqu : The precursor to the Next Stuxnet, October 2011
[2] Kaspersky, The mystery of Duqu

Infection par Duqu

- Exploitation d'une faille 0-day du noyau (sur les polices de caractères TrueType) lors de l'ouverture d'un fichier *word*

- Écriture de fichiers chiffrés (DLL et fichier de configuration)
- Installation d'un driver (*nfrd965.sys*)
- Seul le driver est déchiffré sur le disque

Infection par Duqu

- Exploitation d'une faille 0-day du noyau (sur les polices de caractères TrueType) lors de l'ouverture d'un fichier *word*

- Écriture de fichiers chiffrés (DLL et fichier de configuration)
- Installation d'un driver (*nfrd965.sys*)
- Seul le driver est déchiffré sur le disque

- Au redémarrage, le driver surveille les modules lancés par l'OS
- Lorsque `services.exe` est lancé, la DLL est déchiffrée et y est injectée

Notre travail

On développe un outil de détection de malwares.

Notre travail

On développe un outil de détection de malwares.

- Capable de détecter la DLL principale de Duqu connaissant celle de Stuxnet[3]

[3] Recognition of binary patterns by Morphological analysis, REcon 2012

Notre travail

On développe un outil de détection de malwares.

- Capable de détecter la DLL principale de Duqu connaissant celle de Stuxnet[3]
- Travail fait par analyse statique, en post mortem
- Détecter une attaque en temps réel ? La déjouer ?

[3] Recognition of binary patterns by Morphological analysis, REcon 2012

Notre travail

On développe un outil de détection de malwares.

- Capable de détecter la DLL principale de Duqu connaissant celle de Stuxnet[3]
- Travail fait par analyse statique, en post mortem
- Détecter une attaque en temps réel ? La déjouer ?

On a besoin de la DLL déchiffrée, le moment privilégie est donc celui de l'injection.

[3] Recognition of binary patterns by Morphological analysis, REcon 2012

Notre travail

On développe un outil de détection de malwares.

- Capable de détecter la DLL principale de Duqu connaissant celle de Stuxnet[3]
- Travail fait par analyse statique, en post mortem
- Détecter une attaque en temps réel ? La déjouer ?

On a besoin de la DLL déchiffrée, le moment privilégie est donc celui de l'injection.

- On veut surveiller les processus lancés
- C'est ce que fait Duqu pour l'injection!

[3] Recognition of binary patterns by Morphological analysis, REcon 2012

Notre travail

On développe un outil de détection de malwares.

- Capable de détecter la DLL principale de Duqu connaissant celle de Stuxnet[3]
- Travail fait par analyse statique, en post mortem
- Détecter une attaque en temps réel ? La déjouer ?

On a besoin de la DLL déchiffrée, le moment privilégie est donc celui de l'injection.

- On veut surveiller les processus lancés
- C'est ce que fait Duqu pour l'injection!
- Réutiliser le code du driver et en faire une version défensive ?

Reverse et reconstruction du code source du driver en C++ !

[3] Recognition of binary patterns by Morphological analysis, REcon 2012

Décompilation du driver de Duqu

- Utilisation du plugin de décompilation d'IDA

Reconstruction d'un code lisible et compréhensible :

- Reconnaître les constantes
- Identifier les structures et les types

Décompilation : exemple

```
.text:00012F36 ; =============== S U B R O U T I N E =======================================
.text:00012F36
.text:00012F36
.text:00012F36 ; signed int __cdecl sub_12F36(int a1, PIMAGE_DOS_HEADER a2, int a3)
.text:00012F36 sub_12F36       proc near               ; CODE XREF: sub_11670+16↑p
.text:00012F36                                         ; sub_11B30+2E↑p ...
.text:00012F36
.text:00012F36 arg_0           = dword ptr  4
.text:00012F36 arg_4           = dword ptr  8
.text:00012F36 arg_8           = dword ptr  0Ch
.text:00012F36
.text:00012F36                 push    esi
.text:00012F37                 mov     esi, [esp+4+arg_4]
.text:00012F3B                 mov     eax, 'ZM'
.text:00012F40                 cmp     [esi], ax
.text:00012F43                 jz      short loc_12F4A
.text:00012F45
.text:00012F45 loc_12F45:                              ; CODE XREF: sub_12F36+27↓j
.text:00012F45                 xor     eax, eax
.text:00012F47                 inc     eax
.text:00012F48                 pop     esi
.text:00012F49                 retn
.text:00012F4A ; ---------------------------------------------------------------------------
.text:00012F4A
.text:00012F4A loc_12F4A:                              ; CODE XREF: sub_12F36+D↑j
.text:00012F4A                 mov     eax, [esi+3Ch]
.text:00012F4D                 add     eax, esi
.text:00012F4F                 mov     ecx, [eax]
.text:00012F51                 xor     ecx, 0F750F284h
.text:00012F57                 cmp     ecx, 0F750B7D4h
.text:00012F5D                 jnz     short loc_12F45
.text:00012F5F                 movzx   ecx, word ptr [eax+4]
.text:00012F63                 push    ebx
.text:00012F64                 push    edi
.text:00012F65                 mov     edi, ecx
.text:00012F67                 mov     edx, 5803h
.text:00012F6C                 xor     edi, 594Fh
.text:00012F72                 mov     ebx, edx
.text:00012F74                 cmp     bx, di
.text:00012F77                 jnz     short loc_12F9E
.text:00012F79                 mov     ecx, 5908h
.text:00012F7E                 xor     cx, [eax+18h]
.text:00012F82                 cmp     cx, dx
.text:00012F85                 jnz     short loc_13001
.text:00012F87                 mov     ecx, 0E0h
```

Décompilation : sortie d'IDA

```
signed int __cdecl sub_12F36(int a1, int a2, int a3){
  int v4; // eax@3
  unsigned __int16 v5; // cx@4
  int v6; // ecx@7
  int v7; // edx@7
  __int16 v8; // dx@12
  if ( *(_WORD *)a2 != 'ZM' ) return 1;
  v4 = a2 + *(_DWORD *)(a2 + 60);
  if ( (*(_DWORD *)v4 ^ 0xF750F284) != 0xF750B7D4 ) return 1;
  v5 = *(_WORD *)(v4 + 4);
  if ( 0x5803 == (v5 ^ 0x594F) ){
    if ( (*(_WORD *)(v4 + 24) ^ 0x5908) == 0x5803 && *(_WORD *)(v4 + 20) == 224 ){
      v6 = a1;
      *(_DWORD *)a1 = 0;
      v7 = v4 + 120;
LABEL_12:
      *(_DWORD *)(v6 + 16) = v7;
      *(_DWORD *)(v6 + 26) = *(_DWORD *)(v4 + 80);
      *(_DWORD *)(v6 + 20) = *(_WORD *)(v4 + 20) + v4 + 24;
      v8 = *(_WORD *)(v4 + 6);
      *(_DWORD *)(v6 + 12) = v4;
      *(_DWORD *)(v6 + 4) = a3;
      *(_WORD *)(v6 + 24) = v8;
      *(_DWORD *)(v6 + 8) = a2;
      return 0;
    }
  } // (...)
```

Décompilation : identification des constantes

```
signed int __cdecl sub_12F36(int a1, int a2, int a3){
  int v4; // eax@3
  unsigned __int16 v5; // cx@4
  int v6; // ecx@7
  int v7; // edx@7
  __int16 v8; // dx@12
  if ( *(_WORD *)a2 != 'ZM' ) return 1; (MZ)
  v4 = a2 + *(_DWORD *)(a2 + 60);
  if ( (*(_DWORD *)v4 ^ 0xF750F284) != 0xF750B7D4 ) return 1;
  v5 = *(_WORD *)(v4 + 4); 0xF750F284 XOR 0xF750B7D4 = 00004550 = 'PE\0\0'
  if ( 0x5803 == (v5 ^ 0x594F) ){
    if ( (*(_WORD *)(v4 + 24) ^ 0x5908) == 0x5803 && *(_WORD *)(v4 + 20) == 224 ){
      v6 = a1;
      *(_DWORD *)a1 = 0;
      v7 = v4 + 120;
LABEL_12:
      *(_DWORD *)(v6 + 16) = v7;
      *(_DWORD *)(v6 + 26) = *(_DWORD *)(v4 + 80);
      *(_DWORD *)(v6 + 20) = *(_WORD *)(v4 + 20) + v4 + 24;
      v8 = *(_WORD *)(v4 + 6);
      *(_DWORD *)(v6 + 12) = v4;
      *(_DWORD *)(v6 + 4) = a3;
      *(_WORD *)(v6 + 24) = v8;
      *(_DWORD *)(v6 + 8) = a2;
      return 0;
    }
  } // (...)
```

Décompilation : constantes et types

MZ, PE\0\0 : Strings présents dans les binaires Windows...

Décompilation : constantes et types

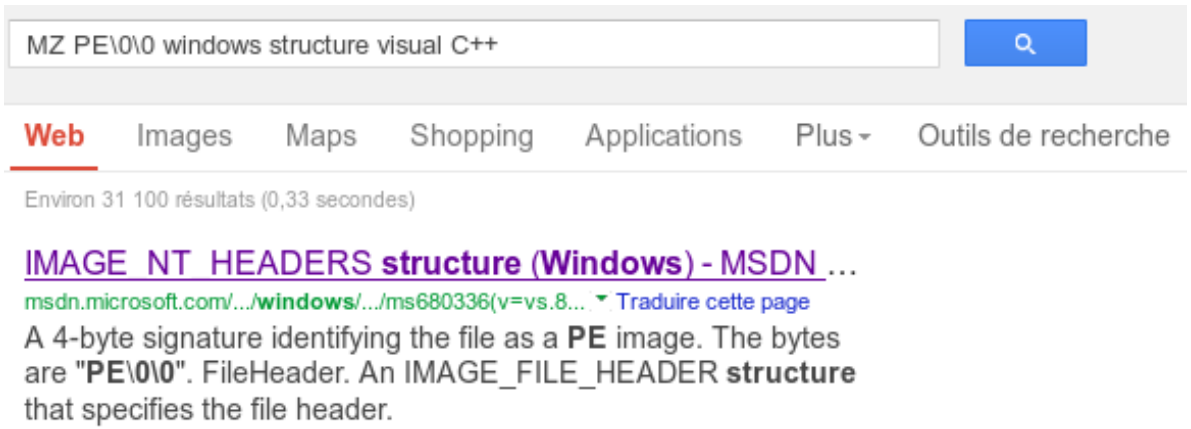
MZ, PE\0\0 : Strings présents dans les binaires Windows...

```
 if ( *(_WORD *)a2 != 'ZM' ) return 1;
v4 = a2 + *(_DWORD *)(a2 + 60);
if ( (*(_DWORD *)v4 ^ 0xF750F284) != 'PE\0\0' ^ 0xF750F284 ) return 1;
```

Décompilation : constantes et types

MZ, PE\0\0 : Strings présents dans les binaires Windows...

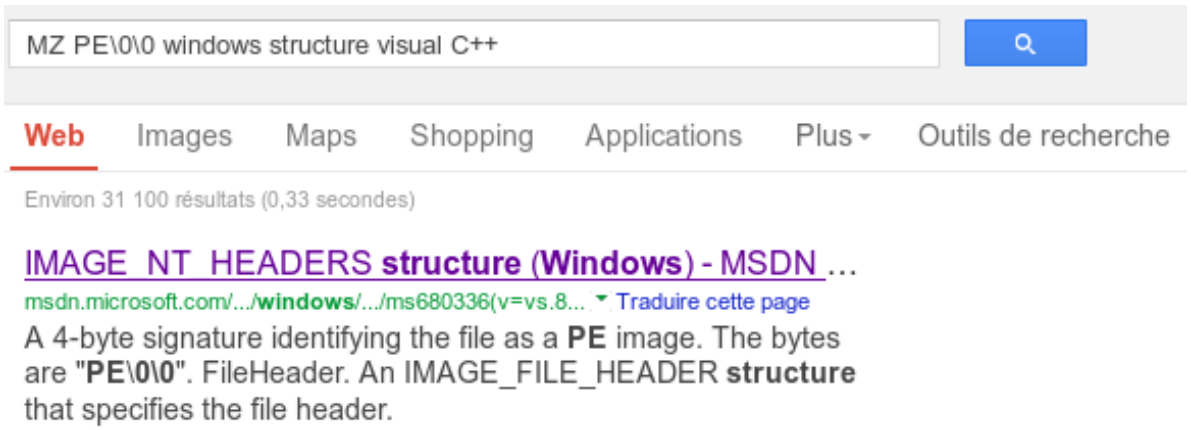
```
  if ( *(_WORD *)a2 != 'ZM' ) return 1;
v4 = a2 + *(_DWORD *)(a2 + 60);
if ( (*(_DWORD *)v4 ^ 0xF750F284) != 'PE\0\0' ^ 0xF750F284 ) return 1;
```



Décompilation : constantes et types

MZ, PE\0\0 : Strings présents dans les binaires Windows...

```
  if ( *(_WORD *)a2 != 'ZM' ) return 1;
v4 = a2 + *(_DWORD *)(a2 + 60);
if ( (*(_DWORD *)v4 ^ 0xF750F284) != 'PE\0\0' ^ 0xF750F284 ) return 1;
```

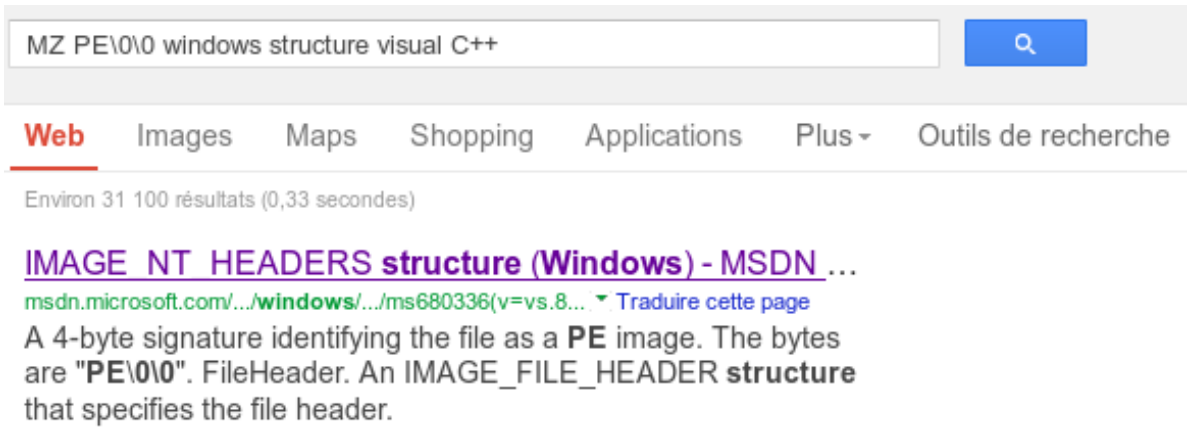



```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER   OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

Décompilation : constantes et types

MZ, PE\0\0 : Strings présents dans les binaires Windows...

```
  if ( *(_WORD *)a2 != 'ZM' ) return 1;
v4 = a2 + *(_DWORD *)(a2 + 60);
if ( (*(_DWORD *)v4 ^ 0xF750F284) != 'PE\0\0' ^ 0xF750F284 ) return 1;
```




```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER   OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

Signature est égal à 'PE\0\0' pour les binaires WIN32.
v4 de type PIMAGE_NT_HEADERS ?

Décompilation : types (merci IDA)

```
signed int __cdecl sub_12F36(int a1, int a2, int a3){
  PIMAGE_NT_HEADERS v4; // eax@3
  unsigned __int16 v5; // cx@4
  int v6; // ecx@7
  DWORD v7; // edx@7
  WORD v8; // dx@12
  if ( *(_WORD *)a2 != 'ZM' )
    return 1;
  v4 = (PIMAGE_NT_HEADERS)(a2 + *(_DWORD *)(a2 + 60));
  if ( (v4->Signature ^ 0xF750F284) != IMAGE_NT_SIGNATURE ^ 0xF750F284 )
    return 1;
  v5 = v4->FileHeader.Machine;
  if ( 22531 == (v5 ^ 0x594F) ){
    if ( (v4->OptionalHeader.Magic ^ 0x5908) == 22531 && v4->FileHeader.SizeOfOptionalHeader == 224 )
      v6 = a1;
      *(_DWORD *)a1 = 0;
      v7 = (DWORD)v4->OptionalHeader.DataDirectory;
LABEL_12:
      *(_DWORD *)(v6 + 16) = v7;
      *(_DWORD *)(v6 + 26) = v4->OptionalHeader.SizeOfImage;
      *(_DWORD *)(v6 + 20) = (char *)v4 + v4->FileHeader.SizeOfOptionalHeader + 24;
      v8 = v4->FileHeader.NumberOfSections;
      *(_DWORD *)(v6 + 12) = v4;
      *(_DWORD *)(v6 + 4) = a3;
      *(_WORD *)(v6 + 24) = v8;
      *(_DWORD *)(v6 + 8) = a2;
      return 0;
    }
  } // (...)
```

Décompilation : fonction finale

- Identification des autres structures de Windows
- Définition dans IDA des structures propres à Duqu (PEData)

Décompilation : fonction finale

- Identification des autres structures de Windows
- Définition dans IDA des structures propres à Duqu (PEData)

```
NTSTATUS __cdecl sub_12F36(__out PEDataPtr pPEData, __in PIMAGE_DOS_HEADER BaseAddress,
  PVOID infosPE;
  PIMAGE_DOS_HEADER  pDosHeader = NULL;
  PIMAGE_NT_HEADERS  pNtHeader  = NULL;
  PIMAGE_FILE_HEADER  pFileHeader = NULL;
  PIMAGE_OPTIONAL_HEADER32  pOptionHeader = NULL;
  PVOID pExportTableRVA;
  infosPE=(PIMAGE_DOS_HEADER)BaseAddress;
  if (((PIMAGE_DOS_HEADER)infosPE)->e_magic != 'ZM' ) return STATUS_WAIT_1;
  pNtHeader = (PIMAGE_NT_HEADERS32)((DWORD)infosPE + ((PIMAGE_DOS_HEADER)infosPE)->e_lfa
  if ( (pNtHeader->Signature ^ 0xF750F284) != (IMAGE_NT_SIGNATURE ^ 0xF750F284)) return
  pFileHeader = (PIMAGE_FILE_HEADER)&pNtHeader->FileHeader;
  pOptionHeader = (PIMAGE_OPTIONAL_HEADER32)&((PIMAGE_NT_HEADERS32)infosPE)->OptionalHea
  if ((pFileHeader->Machine ^ 0x594F) == (IMAGE_PE_i386_MACHINE ^ 0x594F)){
    if ((pOptionHeader->Magic ^ 0x5908) == (IMAGE_PE32_MAGIC ^ 0x5908) && (pFileHeader->
      pPEData->Status = 0;
      pExportTableRVA = (PVOID)&(pOptionHeader->DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPO
Continue:
      pPEData->ExportTableRVA=pExportTableRVA;
      pPEData->dwSizeOfImage = pOptionHeader->SizeOfImage;
      pPEData->lpDataDir=pFileHeader->SizeOfOptionalHeader + (BYTE *)&(pFileHeader->Cha
      pPEData->ResourceDataDir=(ULONG)infosPE;
      pPEData->PEAddress1=pPEData->PEAddress2;
      pPEData->wNumberOfSections=pFileHeader->NumberOfSections;
      pPEData->PEAddress2=(PIMAGE_DOS_HEADER)&((PIMAGE_NT_HEADERS32)infosPE)->Signature;
      return STATUS_SUCCESS;
    } } // (...)
```

Décompilation : fonction finale

```
NTSTATUS __cdecl ParsePE(__out PEDataPtr pPEData, __in PIMAGE_DOS_HEADER BaseAddress, __i
  // (...)
  infosPE=(PIMAGE_DOS_HEADER)BaseAddress;
  if (((PIMAGE_DOS_HEADER)infosPE)->e_magic != 'ZM')
    return STATUS_WAIT_1;
  pNtHeader = (PIMAGE_NT_HEADERS32)((DWORD)infosPE + ((PIMAGE_DOS_HEADER)infosPE)->e_lfa
  if ( (pNtHeader->Signature ^ 0xF750F284) != (IMAGE_NT_SIGNATURE ^ 0xF750F284))
    return STATUS_WAIT_1;
  pFileHeader = (PIMAGE_FILE_HEADER)&pNtHeader->FileHeader;
  pOptionHeader = (PIMAGE_OPTIONAL_HEADER32)&((PIMAGE_NT_HEADERS32)infosPE)->OptionalHea
  if ((pFileHeader->Machine ^ 0x594F) == (IMAGE_PE_i386_MACHINE ^ 0x594F)){
    if ((pOptionHeader->Magic ^ 0x5908) == (IMAGE_PE32_MAGIC ^ 0x5908) && (pFileHeader->
      pPEData->Status = 0;
      pExportTableRVA = (PVOID)&(pOptionHeader->DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPO
Continue:
      pPEData->ExportTableRVA=pExportTableRVA;
      pPEData->dwSizeOfImage = pOptionHeader->SizeOfImage;
      pPEData->lpDataDir=pFileHeader->SizeOfOptionalHeader + (BYTE *)&(pFileHeader->Cha
      pPEData->ResourceDataDir=(ULONG)infosPE;
      pPEData->PEAddress1=pPEData->PEAddress2;
      pPEData->wNumberOfSections=pFileHeader->NumberOfSections;
      pPEData->PEAddress2=(PIMAGE_DOS_HEADER)&((PIMAGE_NT_HEADERS32)infosPE)->Signature;
      return STATUS_SUCCESS;
    }
  } // (...)
```

- Vérification des entêtes d'un fichier (PE ?)
- Parser le fichier (points d'entrée, sections...)

Fail en 64 bits

```
if ((pFileHeader→Machine ∧ 0x594F) == (IMAGE_PE_i386_MACHINE ∧ 0x594F)) // 32 bits
{
    // traitement specifique
Continue:
    // traitement commun
    return STATUS_SUCCESS;
}
else if ((pFileHeader->Machine ^ 0xDE67) == (IMAGE_PE_x86_MACHINE ^ 0xDE67)
          && (pOptionHeader→Magic ∧ 0x5A08) == IMAGE_PE32_PLUS_MAGIC  // PE+ : 64 bits
          && pFileHeader->SizeOfOptionalHeader == 0xF0 )
{
  pPEData->Status=1;
  pExportTableRVA=pOptionHeader->DataDirectory[IMAGE_DIRECTORY_ENTRY].VirtualAddress;
  goto Continue;
}
return STATUS_WAIT_1;
```

Fail en 64 bits

```
if ((pFileHeader→Machine ∧ 0x594F) == (IMAGE_PE_i386_MACHINE ∧ 0x594F)) // 32 bits
{
    // traitement specifique
Continue:
    // traitement commun
    return STATUS_SUCCESS;
}
else if ((pFileHeader->Machine ^ 0xDE67) == (IMAGE_PE_x86_MACHINE ^ 0xDE67)
        && (pOptionHeader→Magic ∧ 0x5A08) == IMAGE_PE32_PLUS_MAGIC  // PE+ : 64 bits
        && pFileHeader->SizeOfOptionalHeader == 0xF0 )
{
  pPEData->Status=1;
  pExportTableRVA=pOptionHeader->DataDirectory[IMAGE_DIRECTORY_ENTRY].VirtualAddress;
  goto Continue;
}
return STATUS_WAIT_1;
```

- Il manque un XOR 0x5A08 !
- Pas de 32 bits = pas d'injection

Décompilation

- Obtention d'un code lisible

Décompilation

- Obtention d'un code lisible

Construction d'un code fonctionnel et proche de l'orginal une fois compilé :

- Trouver les conventions d'appel manquantes
- Tester différentes options de compilation

Décompilation

- Obtention d'un code lisible

Construction d'un code fonctionnel et proche de l'orginal une fois compilé :

- Trouver les conventions d'appel manquantes
- Tester différentes options de compilation

Code récupéré $\rightarrow$ analyse du code C++ et ASM

Lancement du driver

- Driver de type 'network', chargé avant la couche d'abstraction matérielle (HAL)
- Vérification du mode d'exécution (arrêt en cas de débogage)
- Attente de la fin du chargement du noyau (grâce à hal.dll)

Préparation de l'injection

Une fois actif, le driver met en place des fonctionnalités furtives de type *rootkit*.

- Le but est d'injecter la DLL déchiffrée dans le process `services.exe`

Préparation de l'injection

Une fois actif, le driver met en place des fonctionnalités furtives de type *rootkit*.

- Le but est d'injecter la DLL déchiffrée dans le process `services.exe`
- `ZwAllocateVirtualMemory` : allouer de la mémoire dans n'importe quel processus
- `ZwProtectVirtualMemory` : modifier les droits de la mémoire (RW→RWX ou RX→RWX)

Préparation de l'injection

Une fois actif, le driver met en place des fonctionnalités furtives de type *rootkit*.

- Le but est d'injecter la DLL déchiffrée dans le process `services.exe`
- `ZwAllocateVirtualMemory` : allouer de la mémoire dans n'importe quel processus
- `ZwProtectVirtualMemory` : modifier les droits de la mémoire (RW→RWX ou RX→RWX)
- `ZwProtectVirtualMemory` n'est utilisable (normalement) que par le noyau (elle est absente de la table d'exports du noyau)
- En vérifiant qu'elles ne sont pas la cible d'un *hook* défensif

Recherche de ZwProtectVirtualMemory dans le noyau

```
(01) PAGE:004ED1AD                    loc_4ED1AD: [...]
(02) PAGE:004ED1BC 50                 push    eax             ; BaseAddress
(03) PAGE:004ED1BD 57                 push    edi             ; ProcessHandle
(04) PAGE:004ED1BE E8 19 8C F1 FF     call DS:ZwAllocateVirtualMemory
(05) PAGE:004ED1C3 3B C3              cmp     eax, ebx
(06) PAGE:004ED1C5 8B 4D FC           mov     ecx, [ebp+BaseAddress]
(07) PAGE:004ED1C8 89 4E 0C           mov     [esi+0Ch], ecx
(08) PAGE:004ED1CB 7C 2E              jl      short loc_4ED1FB
(09) PAGE:004ED1CD 38 5D 0B           cmp     byte ptr [ebp+ProcessHandle+3], bl
(10) PAGE:004ED1D0 74 27              jz      short loc_4ED1F9
(11) PAGE:004ED1D2 8B 45 D0           mov     eax, [ebp+var_30]
(12) PAGE:004ED1D5 89 45 F8           mov     [ebp+ProtectSize], eax
(13) PAGE:004ED1D8 8D 45 F4           lea     eax, [ebp+OldProtect]
(14) PAGE:004ED1DB 50                 push    eax             ; OldProtect
(15) PAGE:004ED1DC 68 04 01 00 00     push 104h
(16) PAGE:004ED1E1 8D 45 F8           lea     eax, [ebp+ProtectSize]
(17) PAGE:004ED1E4 50                 push    eax             ; ProtectSize
(18) PAGE:004ED1E5 8D 45 FC           lea     eax, [ebp+BaseAddress]
(19) PAGE:004ED1E8 50                 push    eax             ; BaseAddress
(20) PAGE:004ED1E9 57                 push    edi             ; ProcessHandle
(21) PAGE:004ED1EA E8 93 96 F1 FF     call loc_406882 ; ZwProtectVirtualMemory
(22) PAGE:004ED1EF 3B C3              cmp     eax, ebx
```

- Recherche d'un pattern : appel à `ZwAllocateVirtualMemory` (04), puis présence d'un *push 104h* (15) et enfin un *call* (21)

Recherche de ZwProtectVirtualMemory dans le noyau

```
(01) PAGE:004ED1AD                    loc_4ED1AD: [...]
(02) PAGE:004ED1BC 50                 push    eax             ; BaseAddress
(03) PAGE:004ED1BD 57                 push    edi             ; ProcessHandle
(04) PAGE:004ED1BE E8 19 8C F1 FF     call DS:ZwAllocateVirtualMemory
(05) PAGE:004ED1C3 3B C3              cmp     eax, ebx
(06) PAGE:004ED1C5 8B 4D FC           mov     ecx, [ebp+BaseAddress]
(07) PAGE:004ED1C8 89 4E 0C           mov     [esi+0Ch], ecx
(08) PAGE:004ED1CB 7C 2E              jl      short loc_4ED1FB
(09) PAGE:004ED1CD 38 5D 0B           cmp     byte ptr [ebp+ProcessHandle+3], bl
(10) PAGE:004ED1D0 74 27              jz      short loc_4ED1F9
(11) PAGE:004ED1D2 8B 45 D0           mov     eax, [ebp+var_30]
(12) PAGE:004ED1D5 89 45 F8           mov     [ebp+ProtectSize], eax
(13) PAGE:004ED1D8 8D 45 F4           lea     eax, [ebp+OldProtect]
(14) PAGE:004ED1DB 50                 push    eax             ; OldProtect
(15) PAGE:004ED1DC 68 04 01 00 00     push 104h
(16) PAGE:004ED1E1 8D 45 F8           lea     eax, [ebp+ProtectSize]
(17) PAGE:004ED1E4 50                 push    eax             ; ProtectSize
(18) PAGE:004ED1E5 8D 45 FC           lea     eax, [ebp+BaseAddress]
(19) PAGE:004ED1E8 50                 push    eax             ; BaseAddress
(20) PAGE:004ED1E9 57                 push    edi             ; ProcessHandle
(21) PAGE:004ED1EA E8 93 96 F1 FF     call loc_406882 ; ZwProtectVirtualMemory
(22) PAGE:004ED1EF 3B C3              cmp     eax, ebx
```

- Recherche d'un pattern : appel à `ZwAllocateVirtualMemory` (04), puis présence d'un *push 104h* (15) et enfin un *call* (21)
- Le *call* (21) est considéré comme un appel vers `ZwProtectVirtualMemory`

Présence d'un hook ?

- Les adresses des fonctions `ZwAllocateVirtualMemory` et `ZwProtectVirtualMemory` sont connues

Présence d'un hook ?

- Les adresses des fonctions `ZwAllocateVirtualMemory` et `ZwProtectVirtualMemory` sont connues
- Le driver vérifie qu'elles sont dans un espace d'adresse réservé au noyau

Présence d'un hook ?

- Les adresses des fonctions `ZwAllocateVirtualMemory` et `ZwProtectVirtualMemory` sont connues
- Le driver vérifie qu'elles sont dans un espace d'adresse réservé au noyau

Un test d'intégrité (masque commun sur 11 octets) est réalisé sur chacune de ces fonctions :

B8	00	00	00	00	8D	54	24	04	9C
6A	08	E8	00	00	00	00	C2	14	00

ZwAllocateVirtualMemory :

.text:00405DDC	B8	11	00	00	00	mov eax 11h
.text:00405DE1	8D	54	24	04		lea edx [esp+ProcessHandle]
.text:00405DE5	9C					pushf
.text:00405DE6	6A	08				push 8
.text:00405DE8	E8	B9	20	00	00	call sub_407EA6
.text:00405DED	C2	14	00			retn 14h

ZwProtectVirtualMemory :

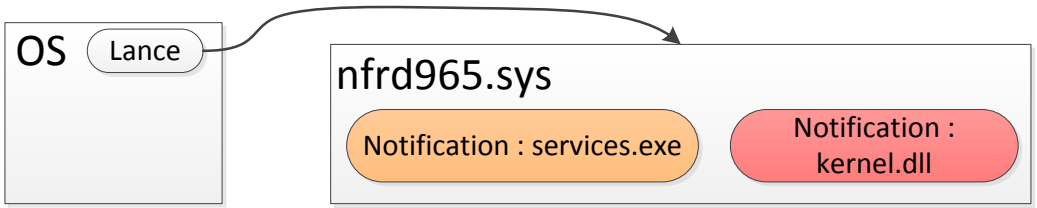
.text:00406882	B8	89	00	00	00	mov eax 89h
.text:00406887	8D	54	24	04		lea edx [esp+ProcessHandle]
.text:0040688B	9C					pushf
.text:0040688C	6A	08				push 8
.text:0040688E	E8	13	16	00	00	call sub_407EA6
.text:00406893	C2	14	00			retn 14h

Fin de l'initialisation

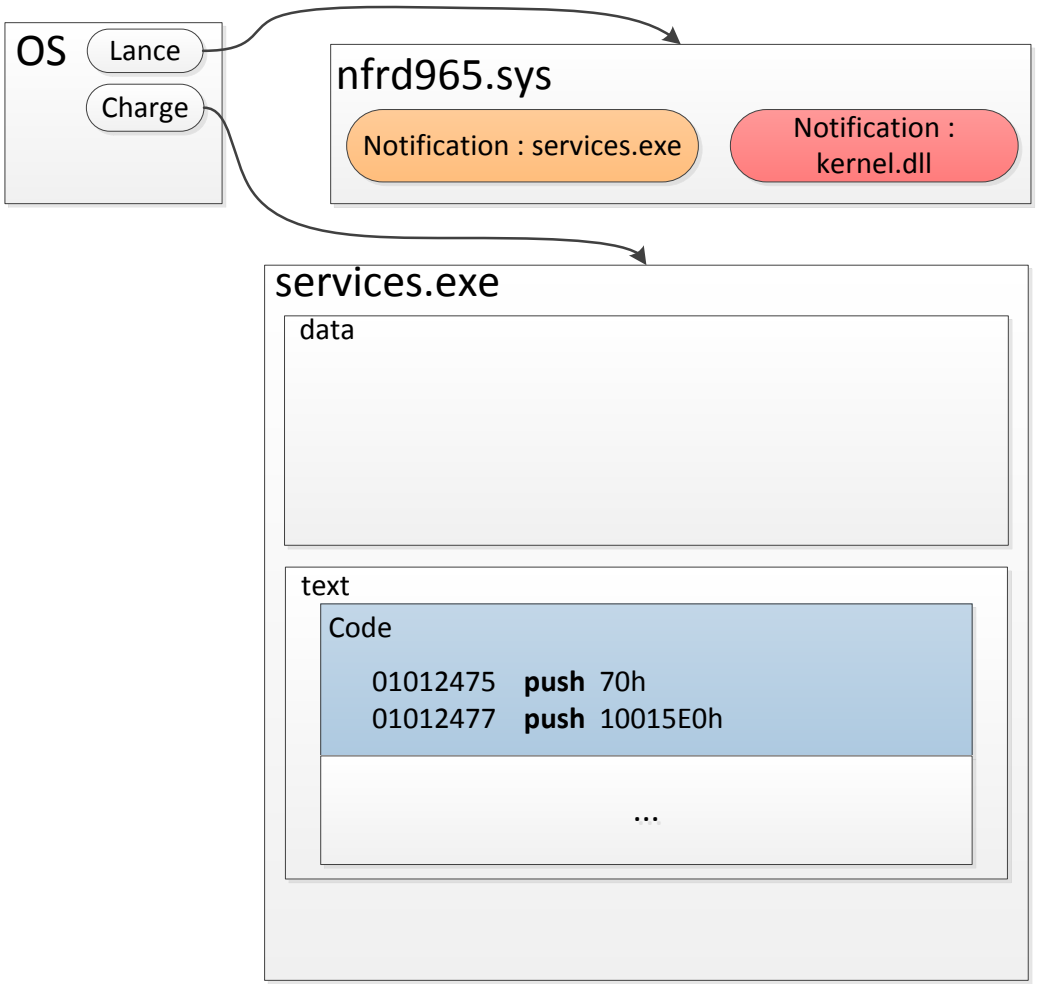
- Stockage de paramètres (clés de registre) déchiffrés dans la mémoire partagée
- Constitution d'une table d'imports utilisable par les charges finales
- Mise en place d'une notification dès qu'un module (DLL ou exécutable) est chargé en mémoire (`PsSetLoadImageNotifyRoutine`)

Injection au démarrage

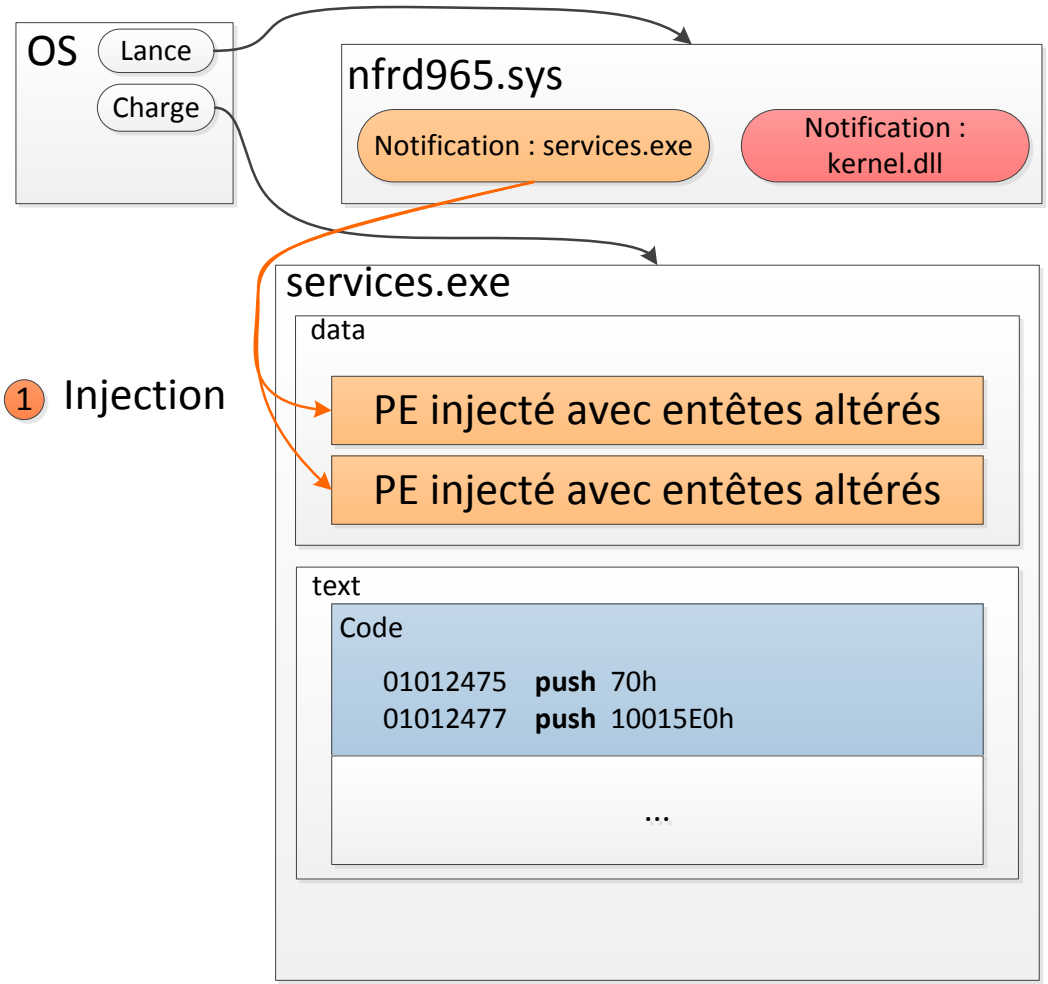
Injection au démarrage



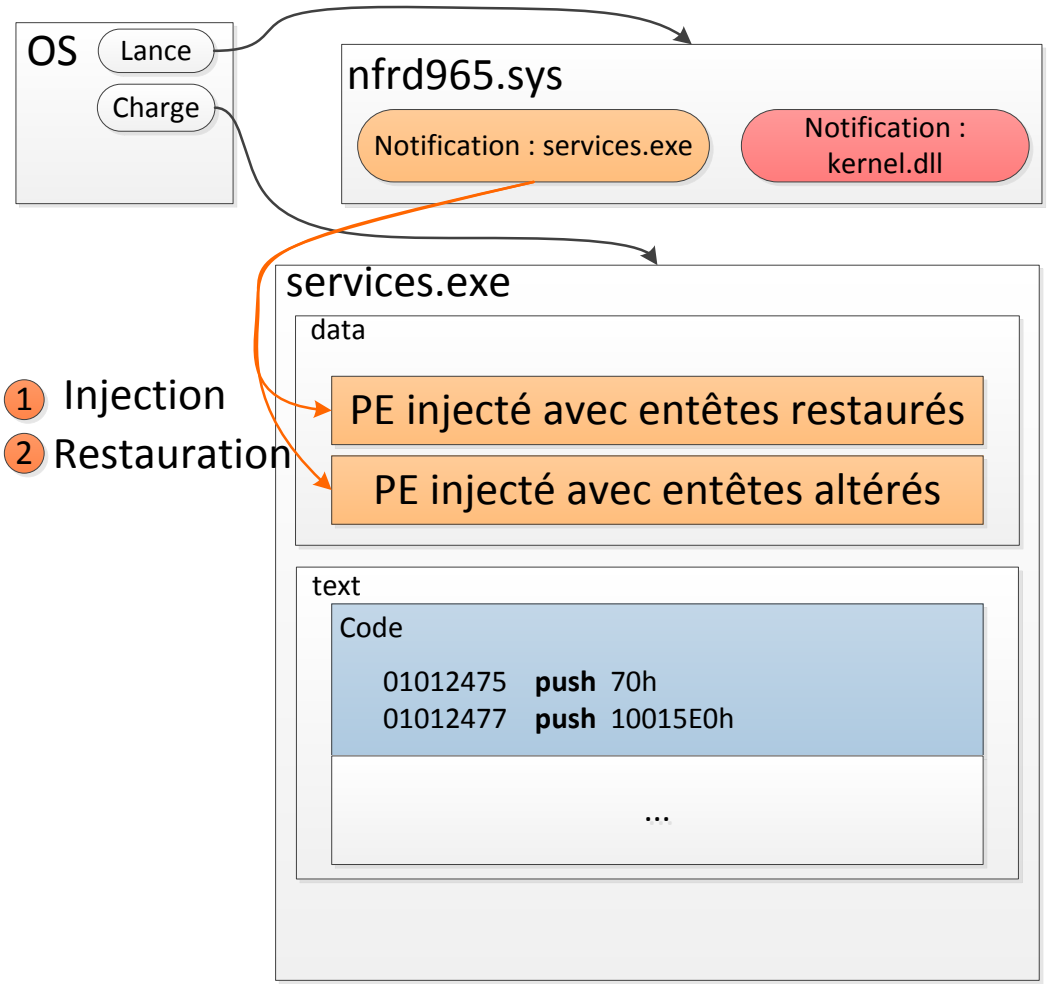
Injection au démarrage



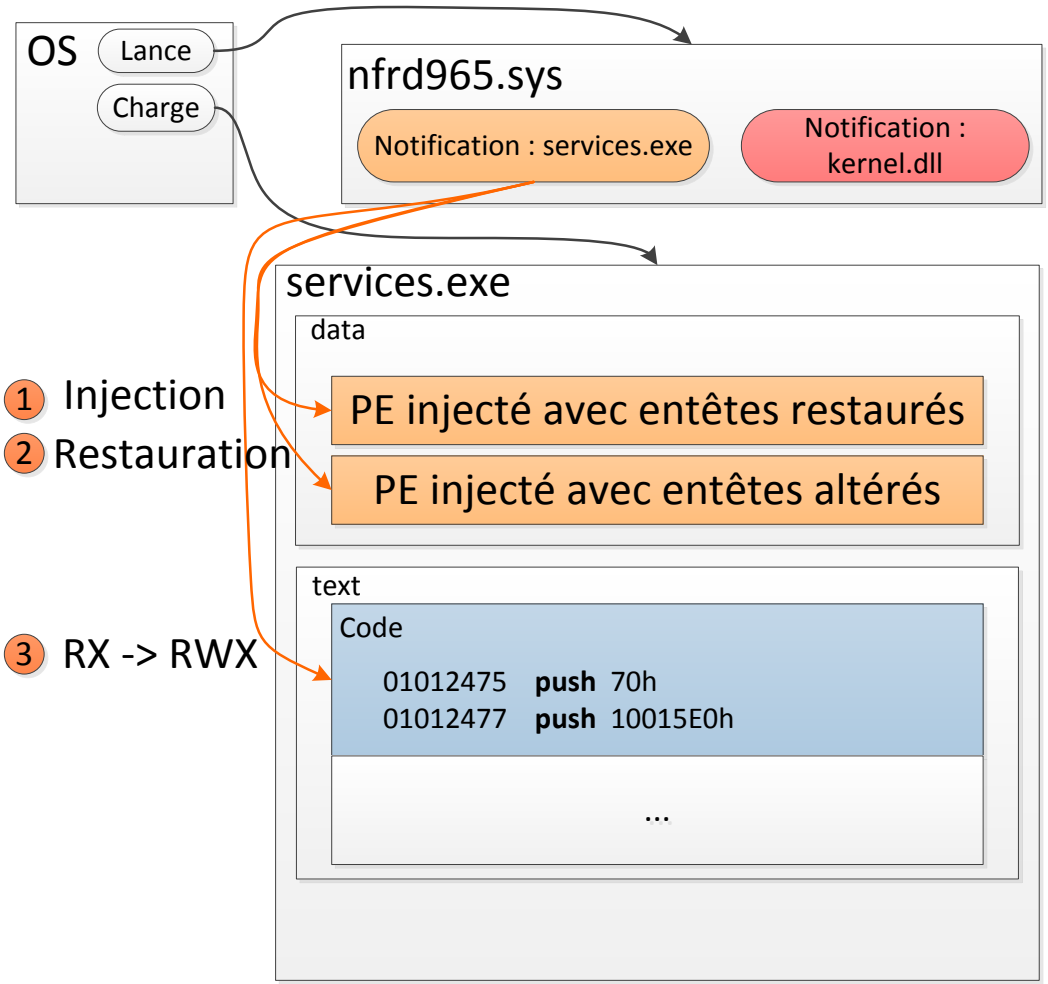
Injection au démarrage



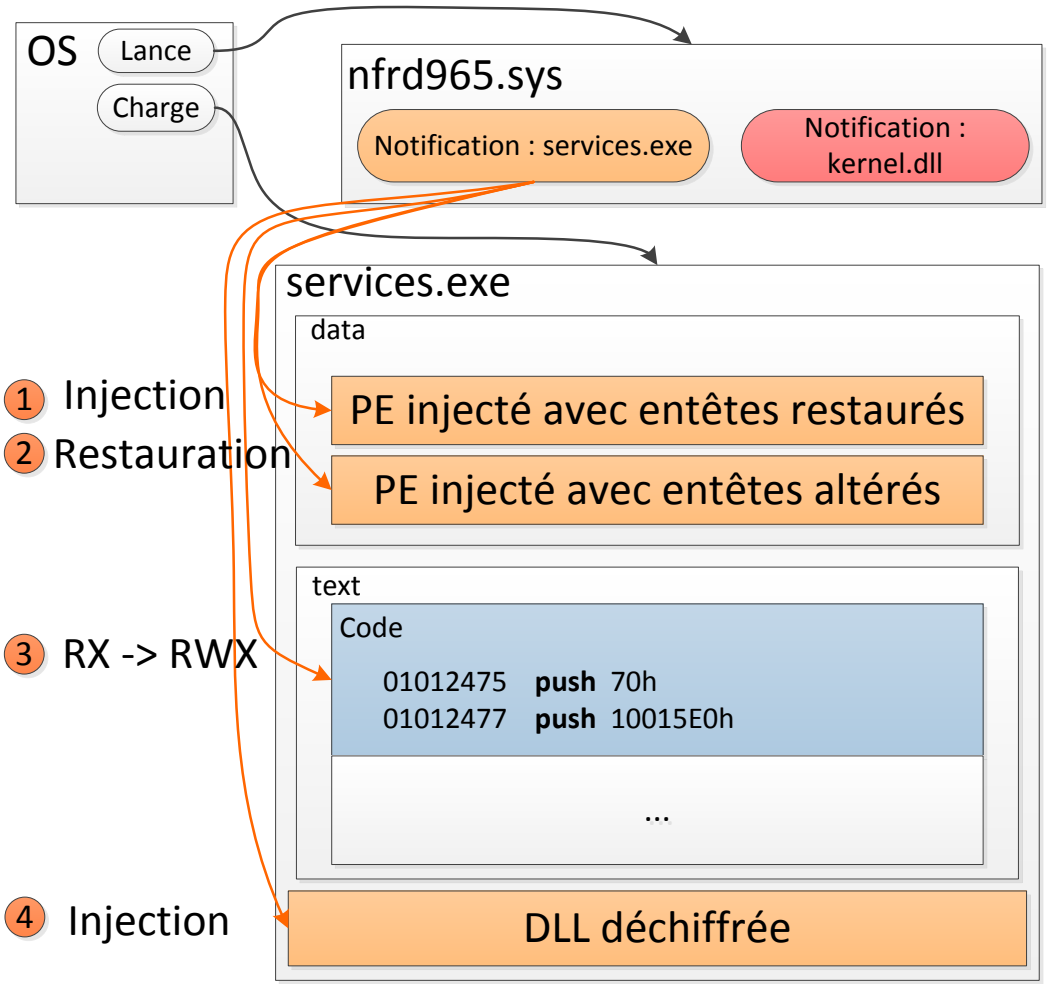
Injection au démarrage



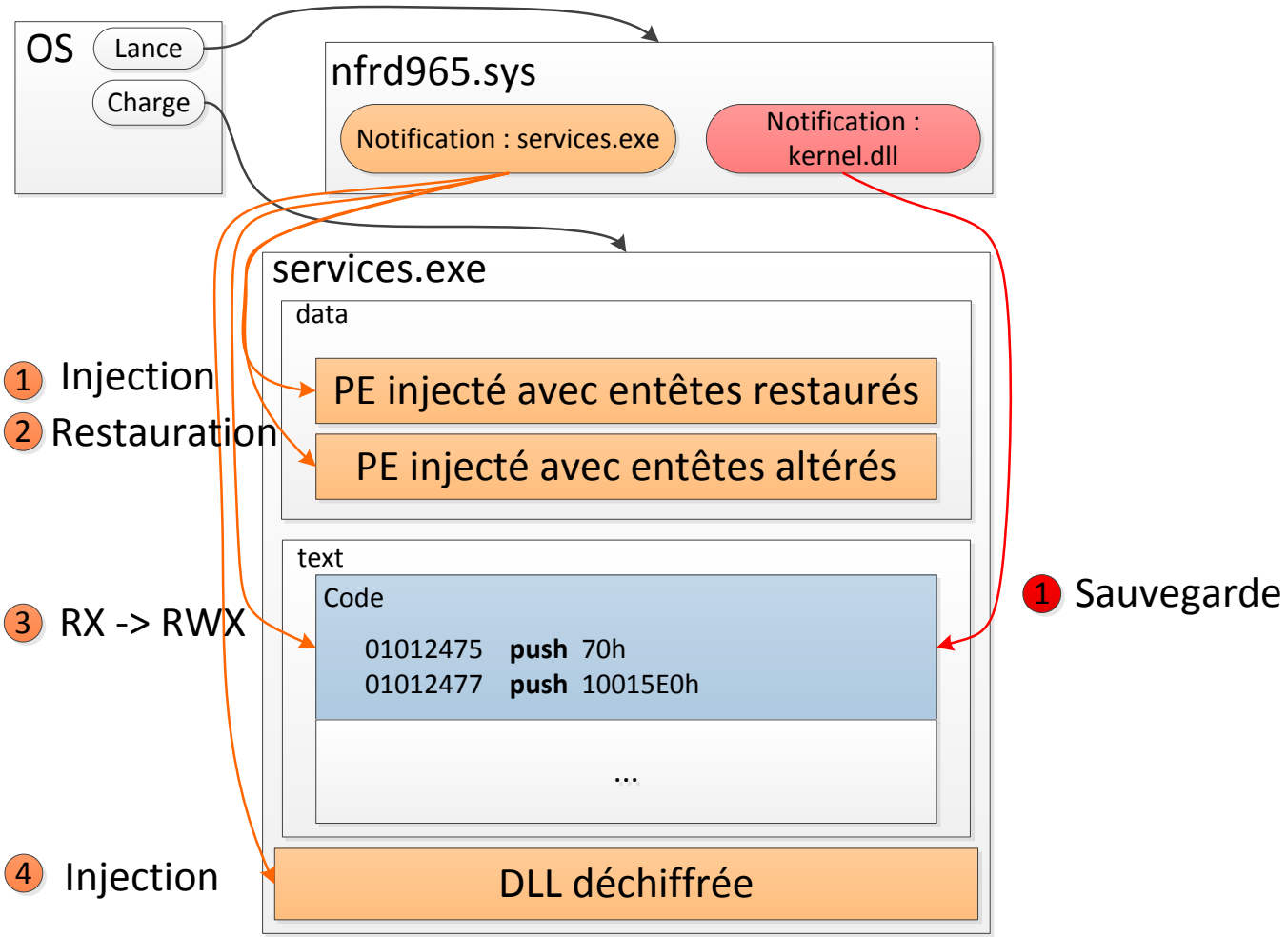
Injection au démarrage



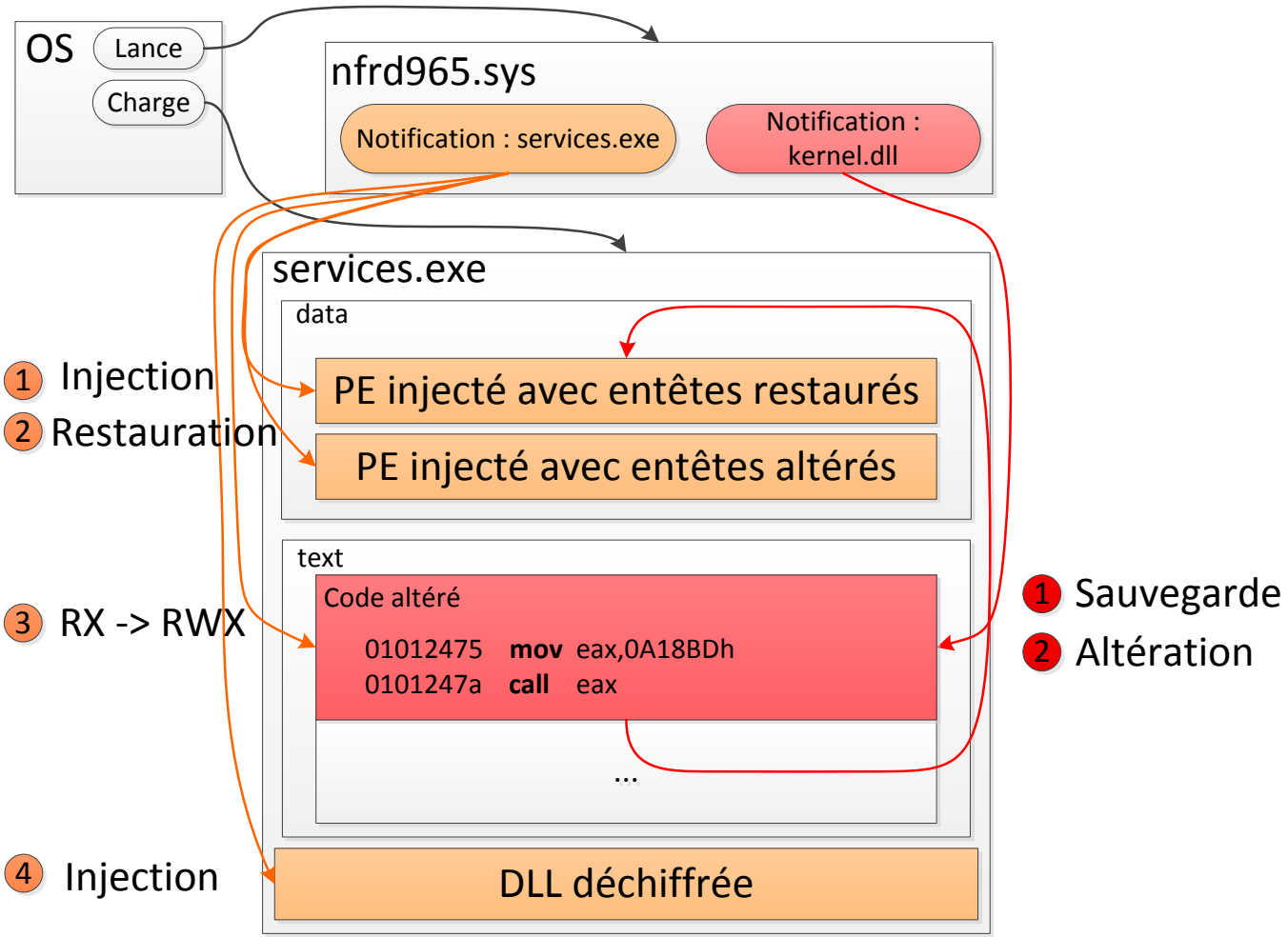
Injection au démarrage



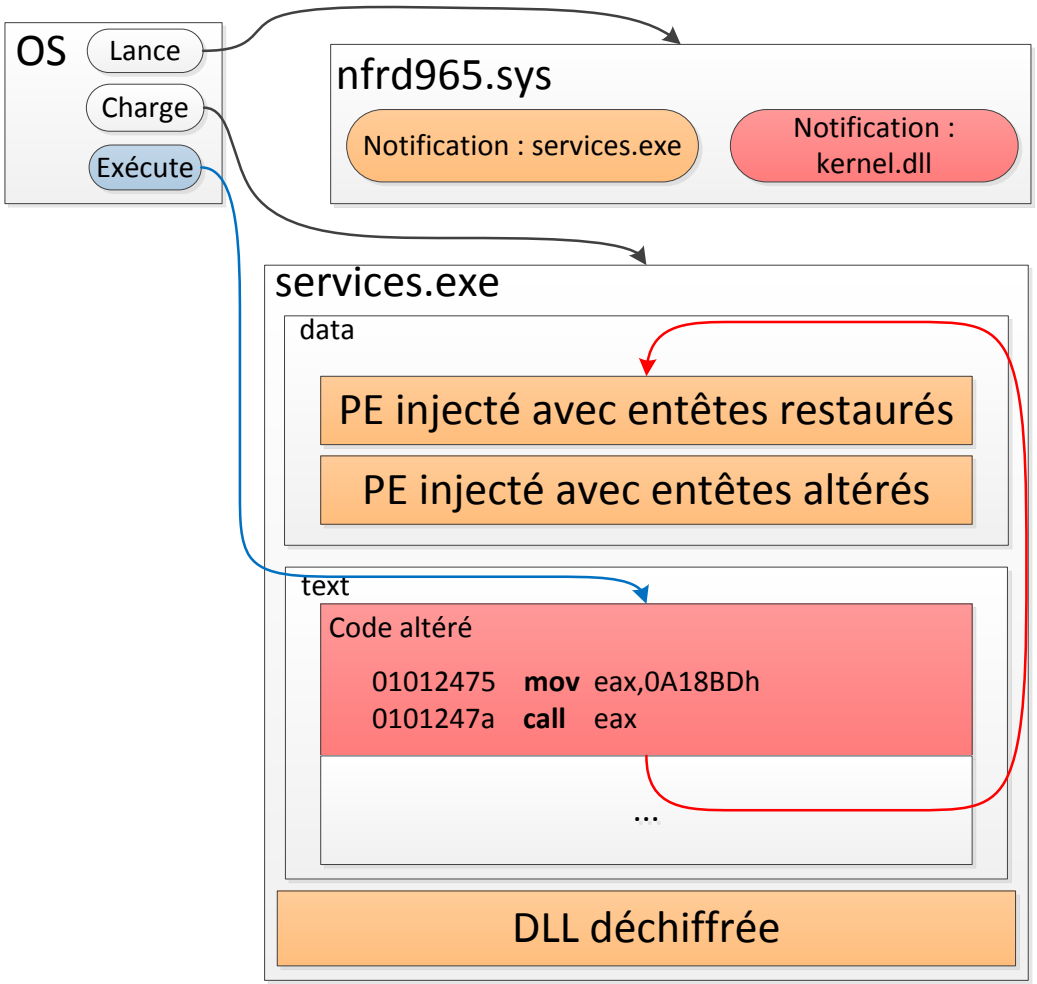
Injection au démarrage



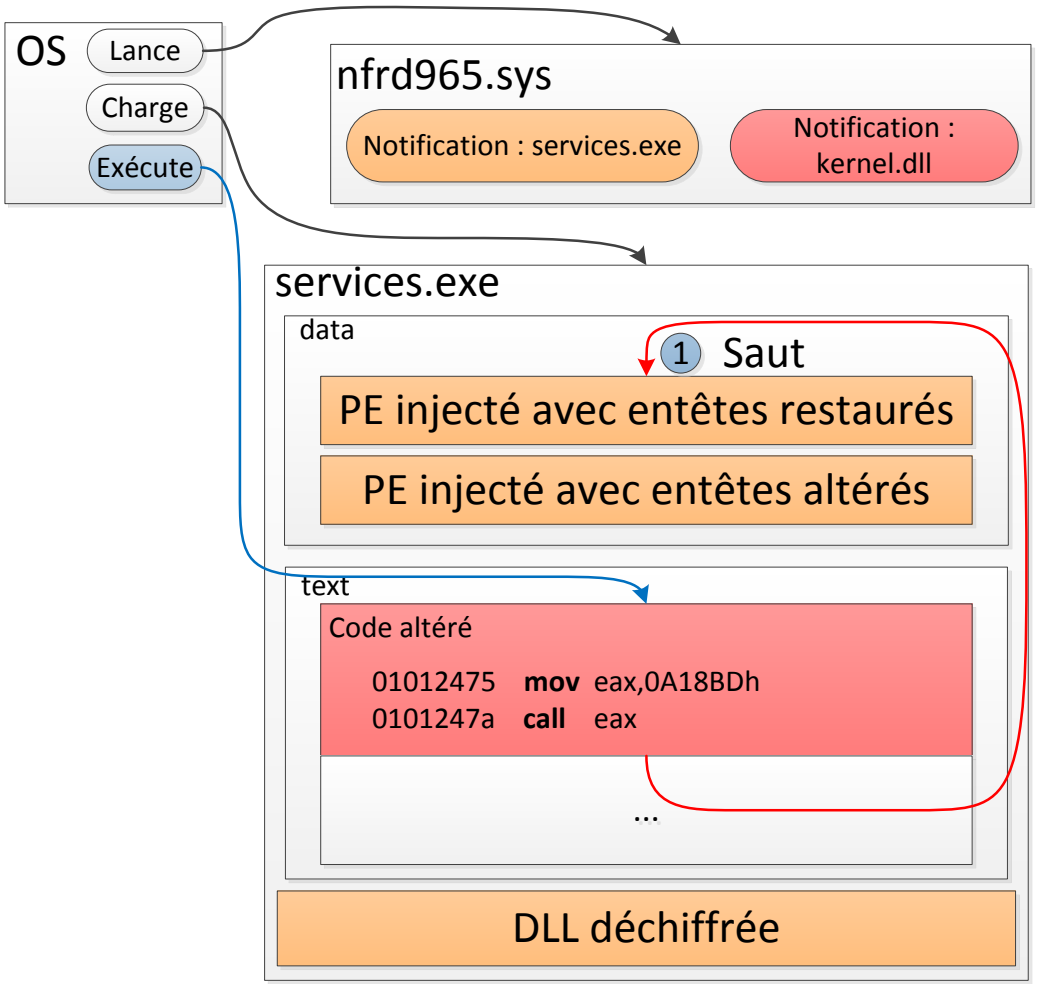
Injection au démarrage



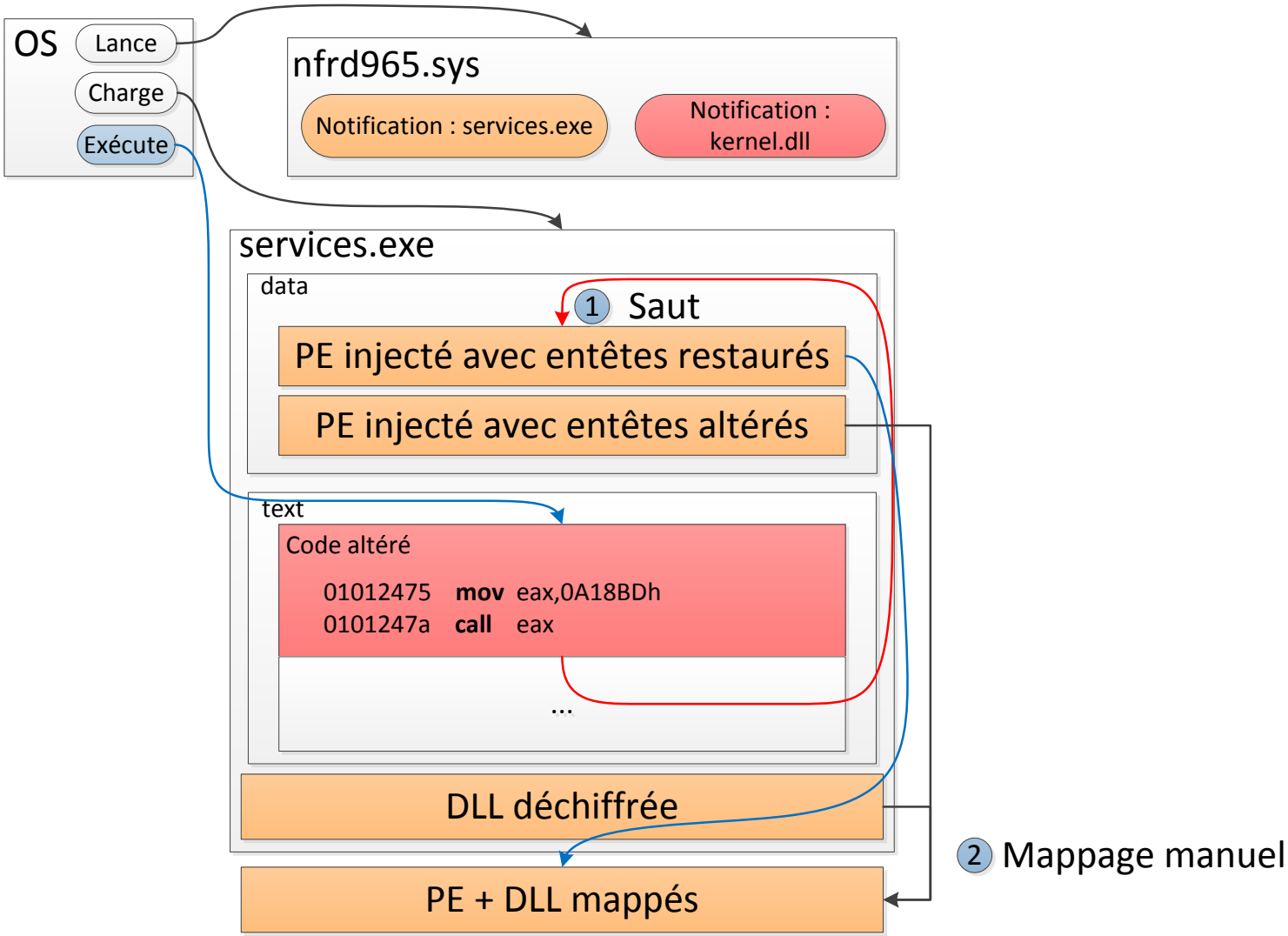
Injection au démarrage



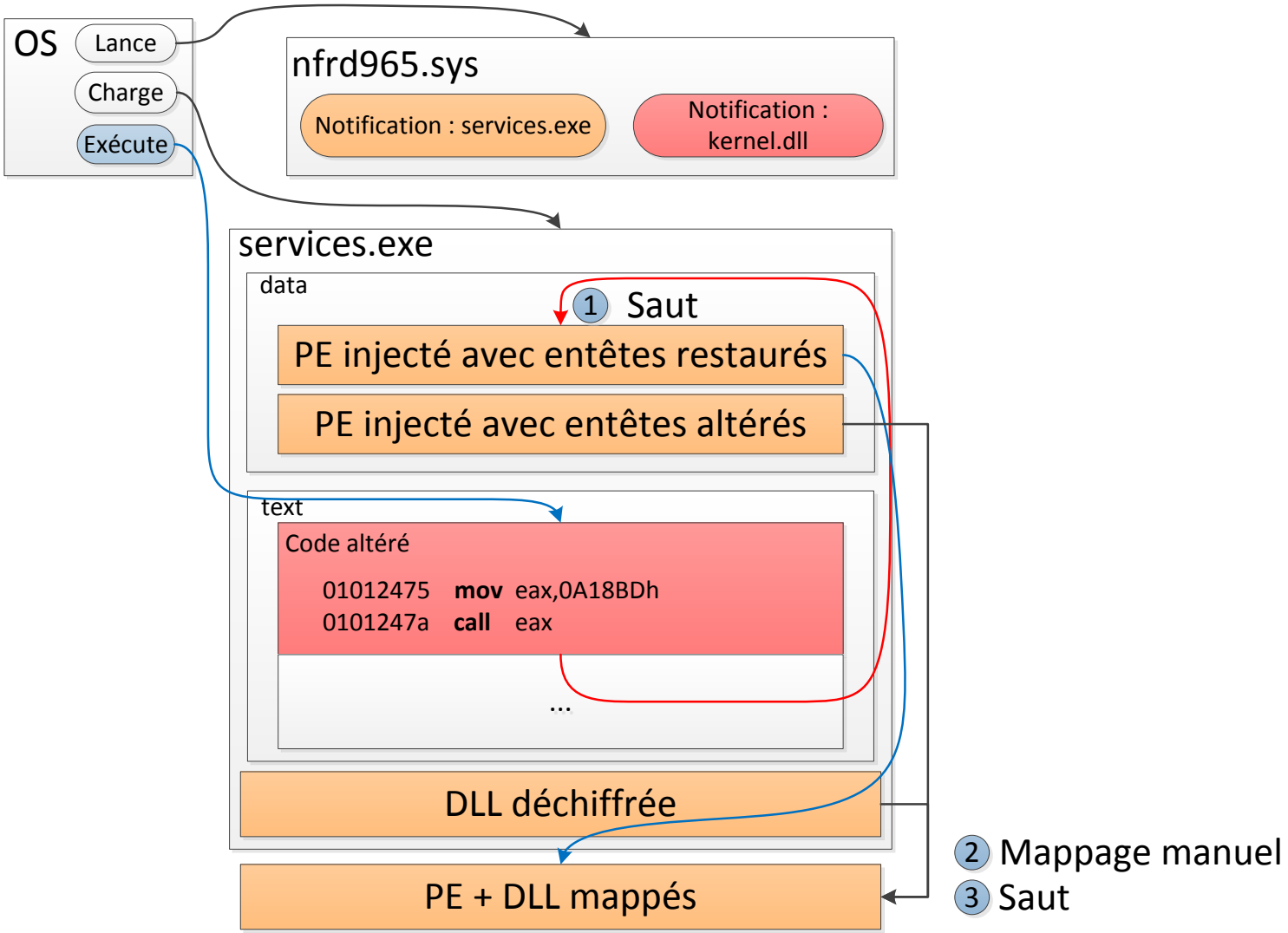
Injection au démarrage



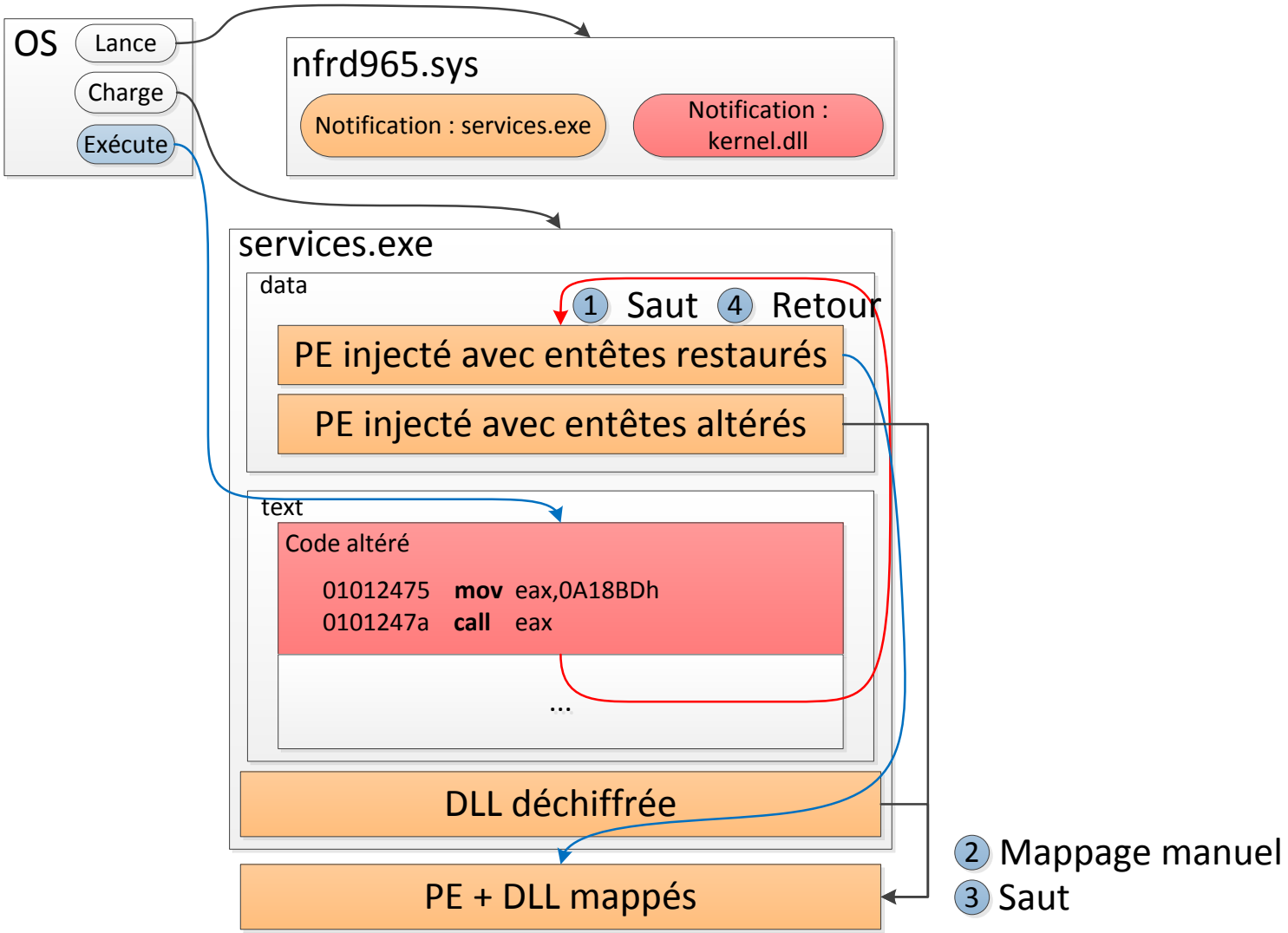
Injection au démarrage



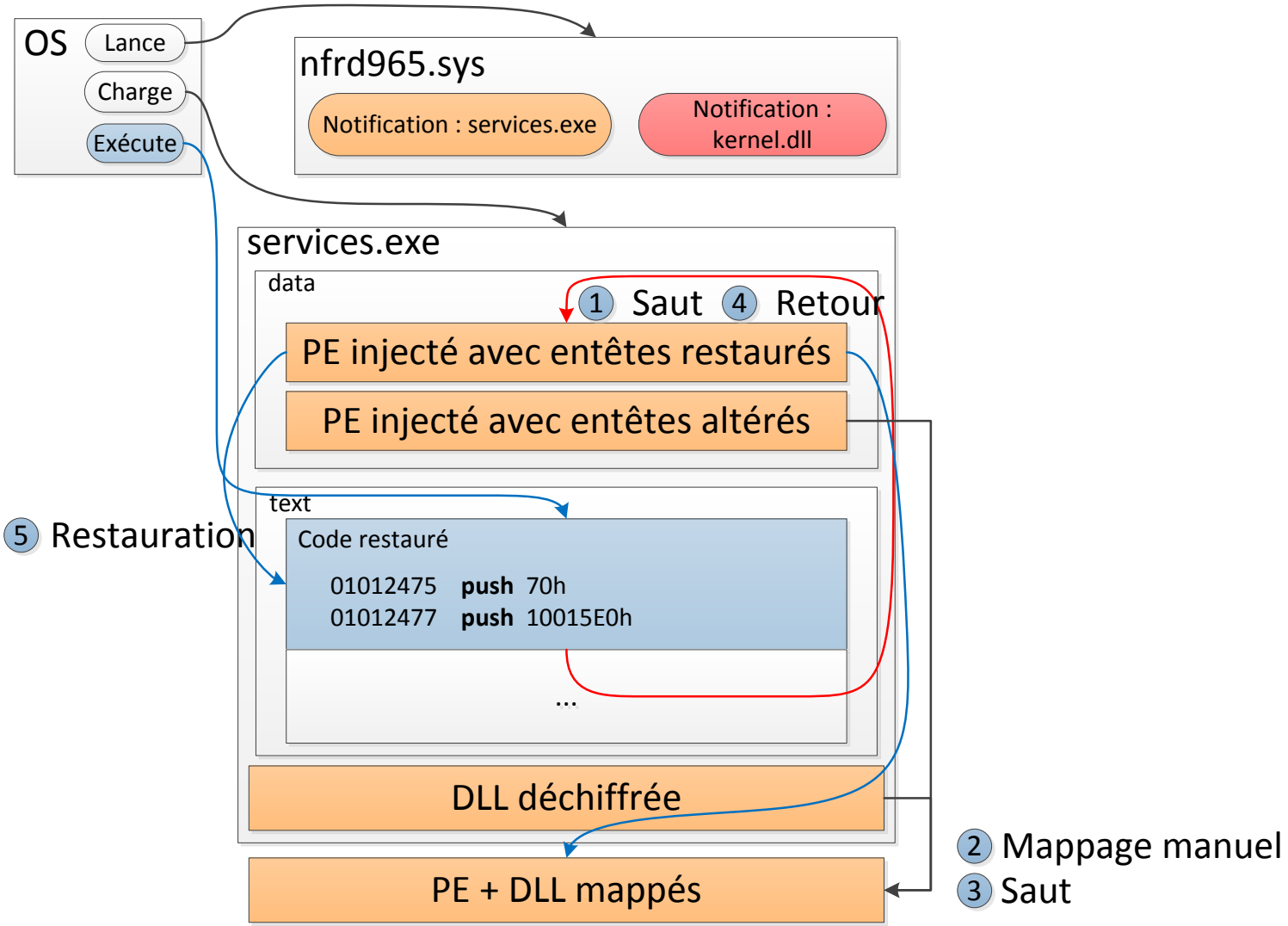
Injection au démarrage



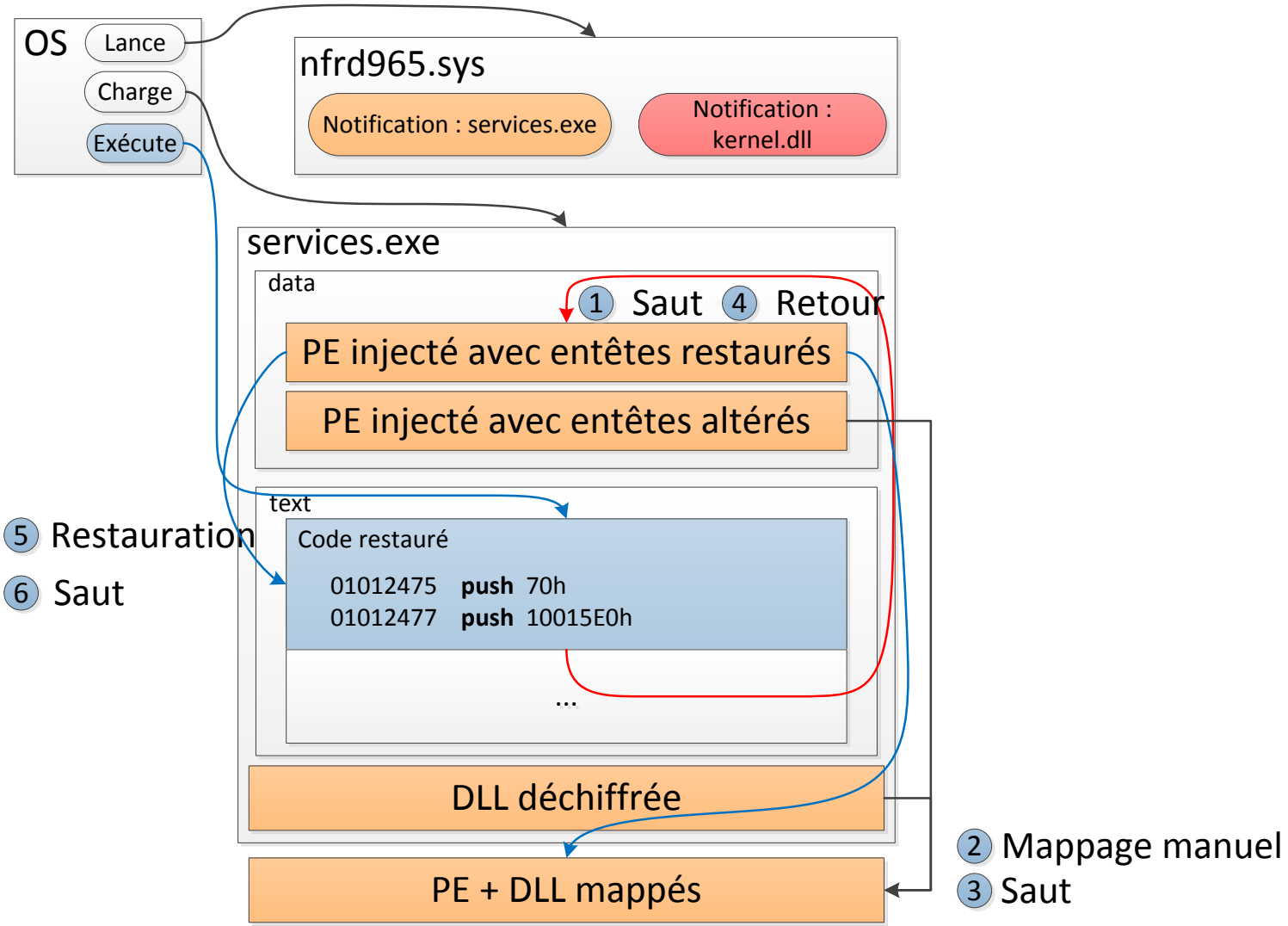
Injection au démarrage



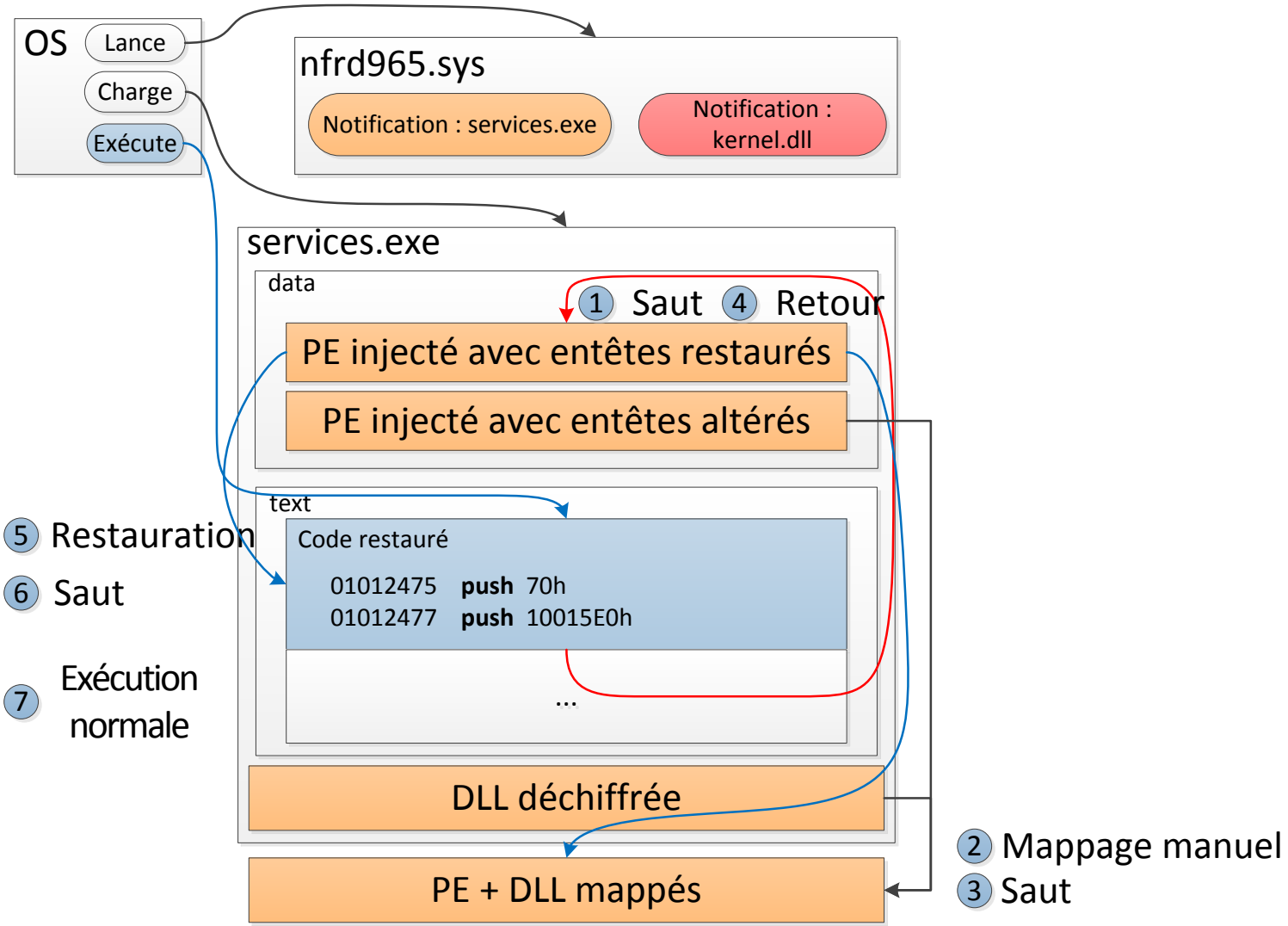
Injection au démarrage



Injection au démarrage



Injection au démarrage



Fonctionnement de Duqu

- Exploitation d'une faille pour installer un driver
- Mise en place d'une notification à chaque chargement de module
- Injection de la charge finale dans `services.exe`

Fonctionnement de Duqu

- Exploitation d'une faille pour installer un driver
- Mise en place d'une notification à chaque chargement de module
- Injection de la charge finale dans `services.exe`

On veut être capable de détecter Duqu au moment de l'injection, avant que la charge ne soit exécutée.

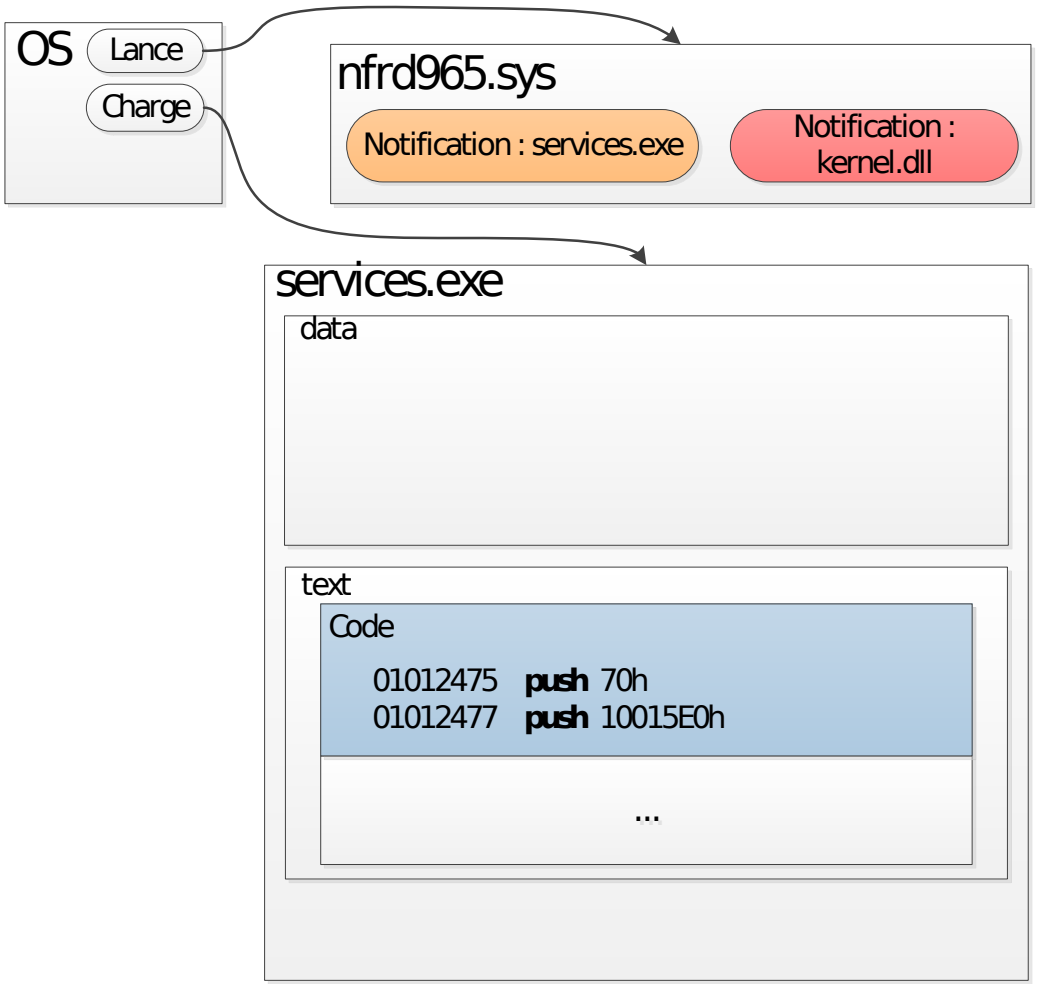
Initialisation et mémorisation

- À la première notification le point d'entrée est intact
- Duqu écrase le point d'entrée de `services.exe` au chargement de `kernel.dll`

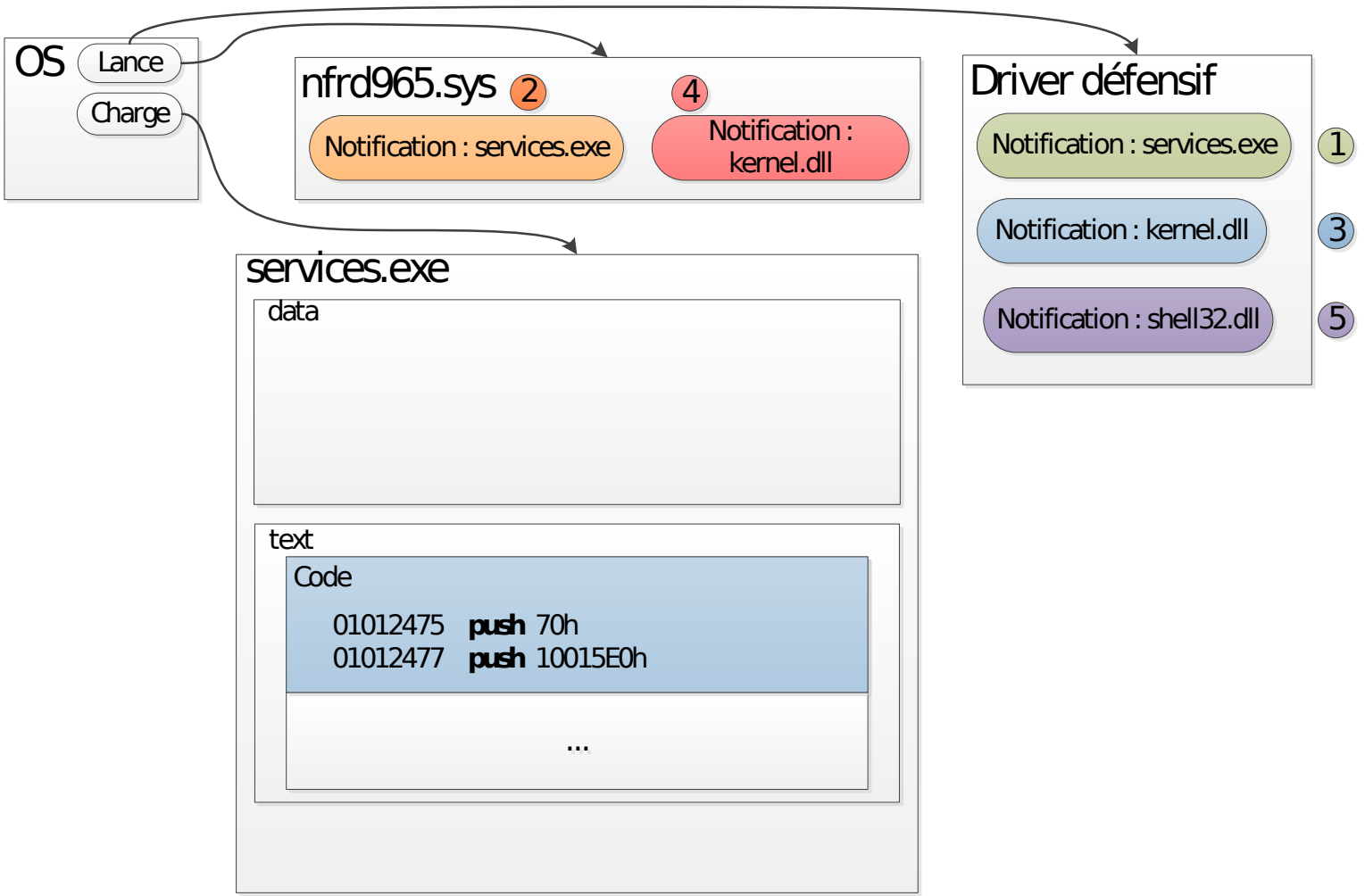
Initialisation et mémorisation

- À la première notification le point d'entrée est intact
- Duqu écrase le point d'entrée de `services.exe` au chargement de `kernel.dll`
- Sauvegarde d'un hash sur le point d'entrée des processus notifiés
- Lors des notifications suivantes, émission d'une alerte (et arrêt du processus) si le point d'entrée a été modifié

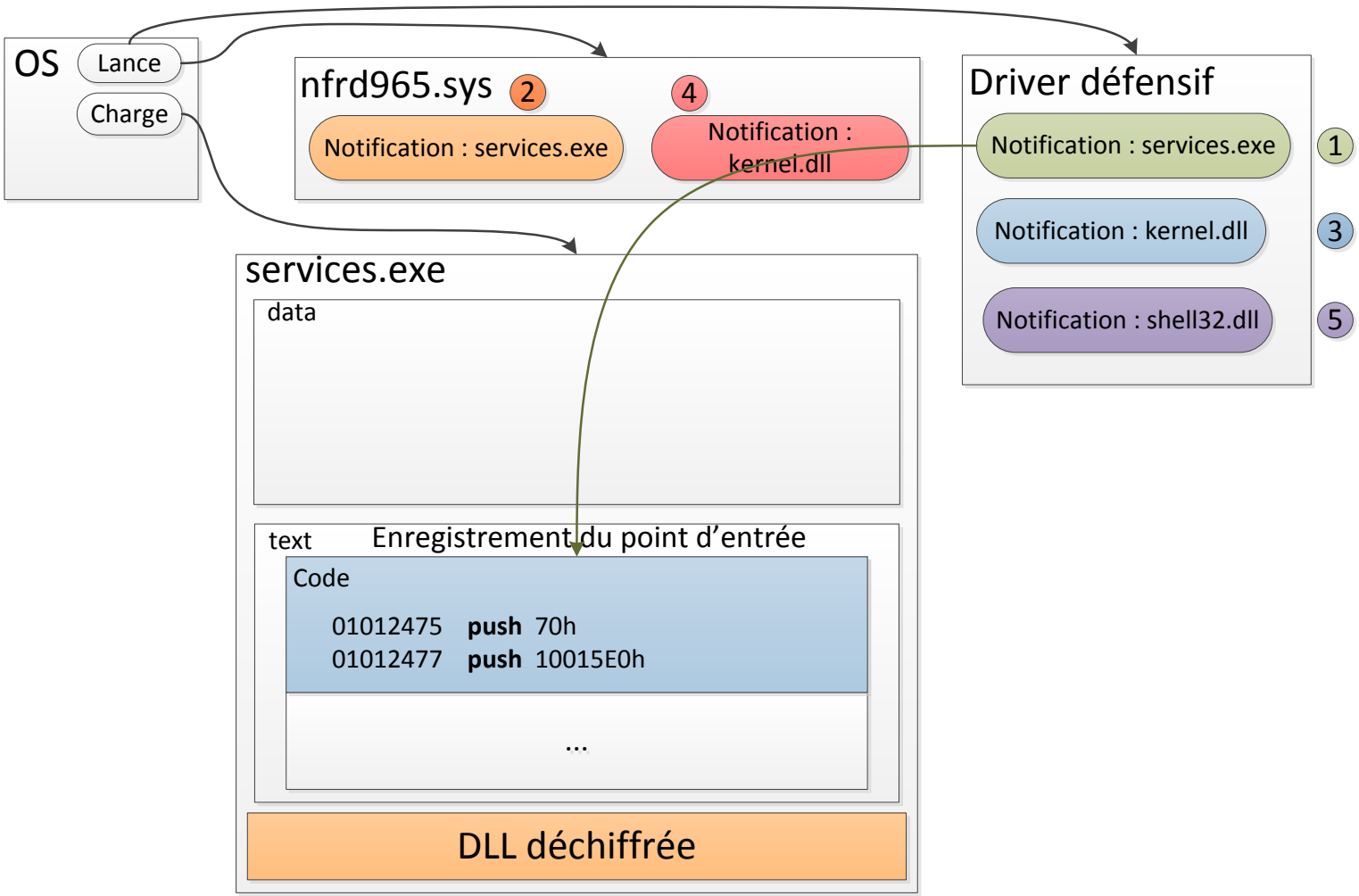
Scénario de défense : tuer `services.exe`



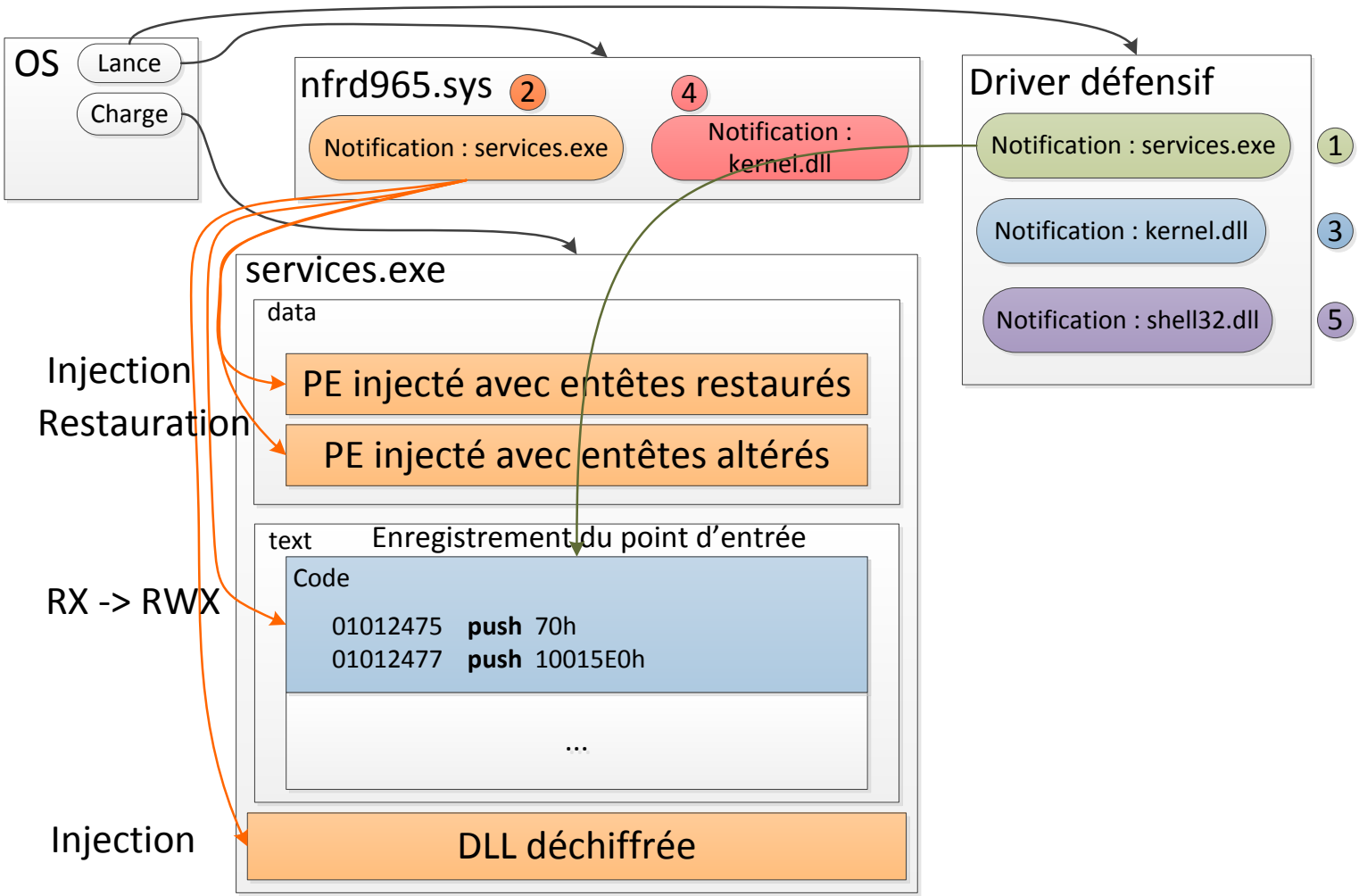
Scénario de défense : tuer `services.exe`



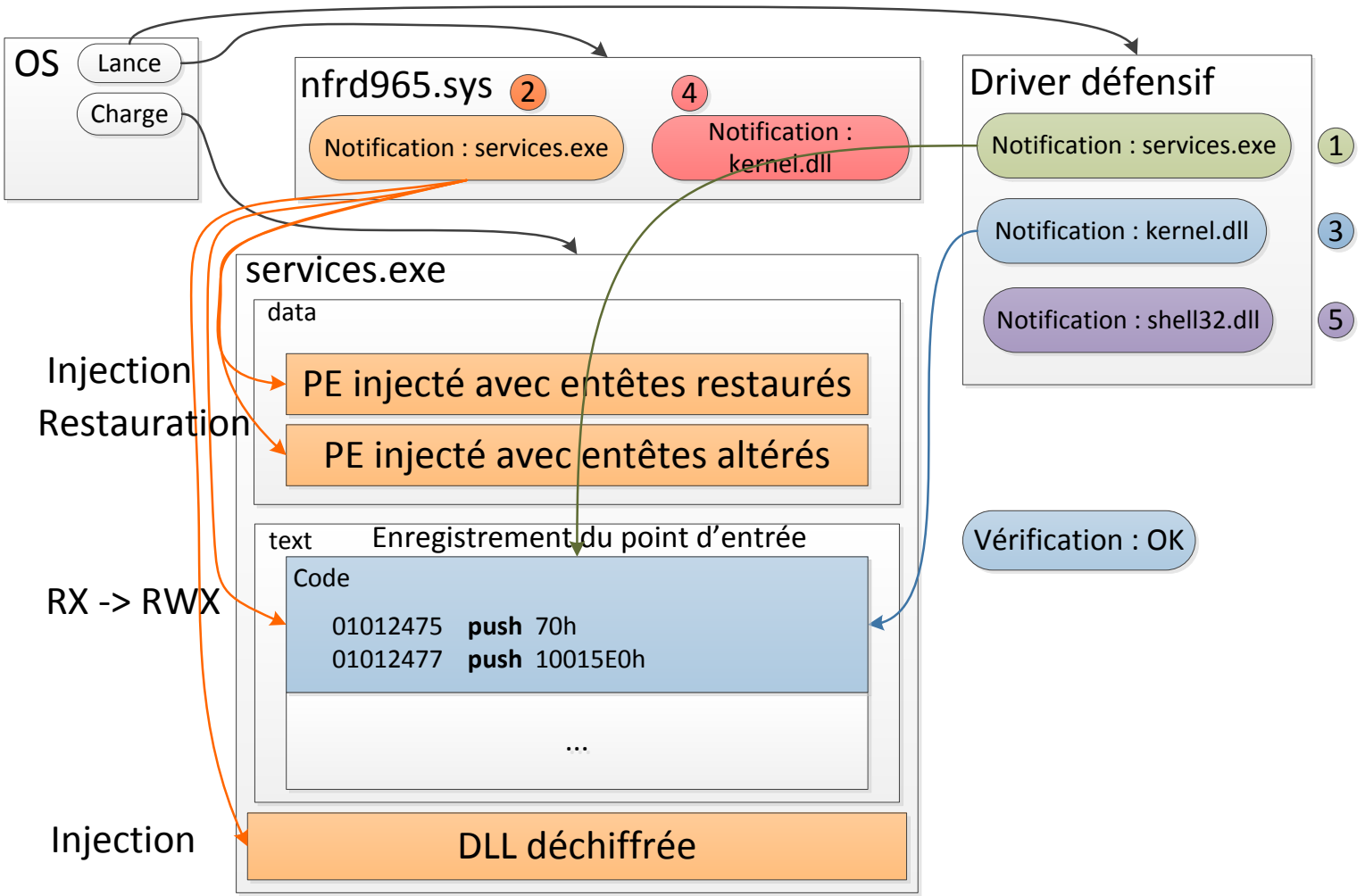
Scénario de défense : tuer `services.exe`



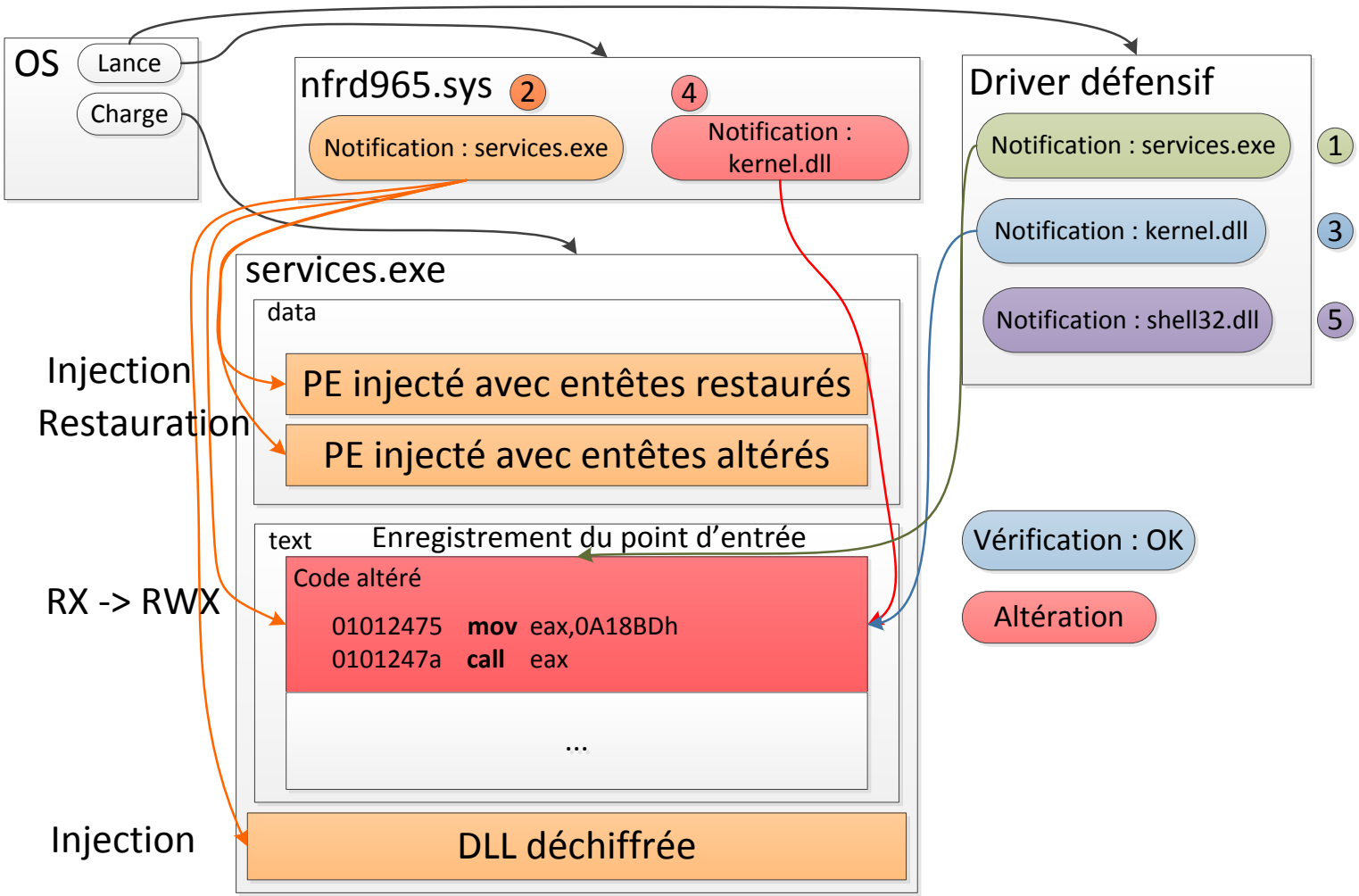
Scénario de défense : tuer `services.exe`



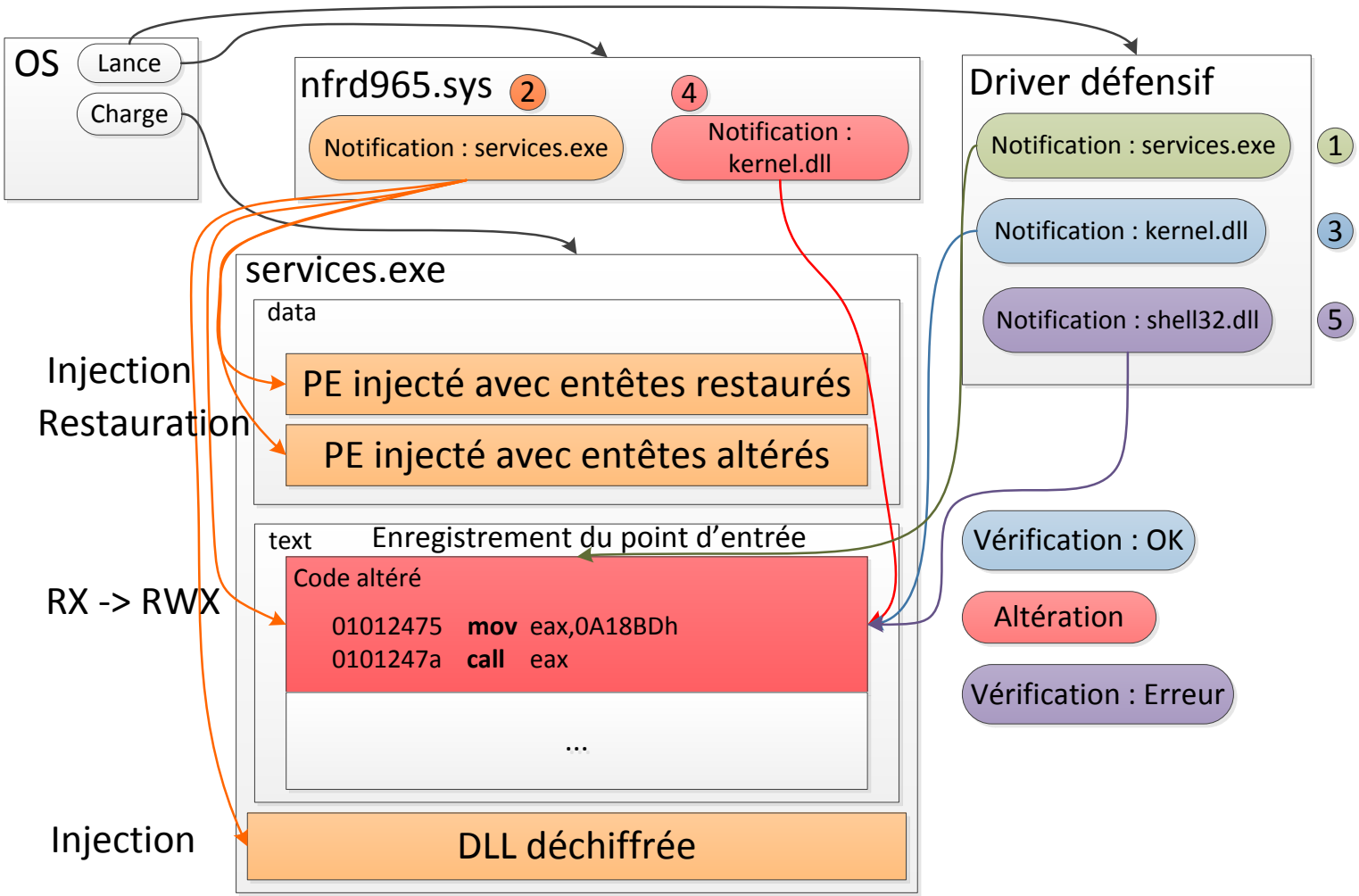
Scénario de défense : tuer `services.exe`



Scénario de défense : tuer `services.exe`



Scénario de défense : tuer `services.exe`



Code réutilisé

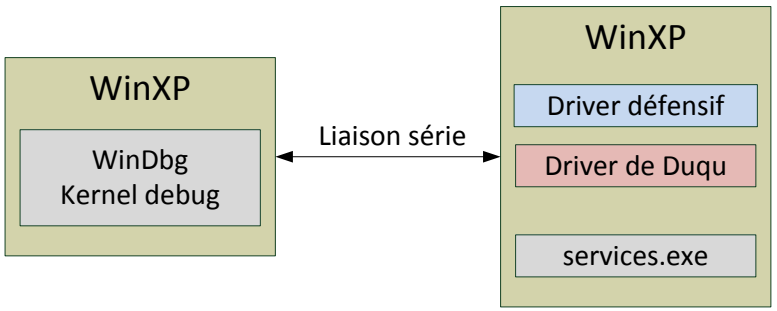
- Parser des binaires (fonction ParsePE)
- Calculer des hash (à l'origine pour obfusquer les noms de fonctions)
- Enregistrer et gérer les notifications de chargement de processus

Démonstration

- On lance les drivers (Duqu et driver défensif) à la demande (clé de registre) et on renomme calc.exe en services.exe

Démonstration

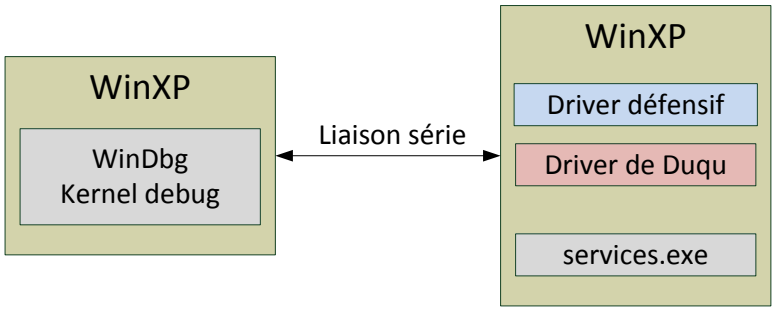
- On lance les drivers (Duqu et driver défensif) à la demande (clé de registre) et on renomme calc.exe en services.exe



Pour la démo on patch le driver de Duqu pour qu'il accepte de se lancer en mode debug.

Démonstration

- On lance les drivers (Duqu et driver défensif) à la demande (clé de registre) et on renomme calc.exe en services.exe

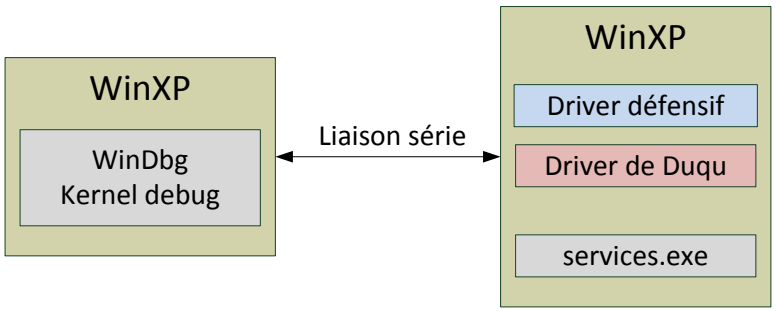


Pour la démo on patch le driver de Duqu pour qu'il accepte de se lancer en mode debug.

- Le driver défensif détecte l'altération du point d'entrée et termine le processus `services.exe`
- Détection lors de la notification de `Shell32.dll`, parce que le driver défensif intervient avant Duqu

Démonstration

- On lance les drivers (Duqu et driver défensif) à la demande (clé de registre) et on renomme calc.exe en services.exe



Pour la démo on patch le driver de Duqu pour qu'il accepte de se lancer en mode debug.

- Le driver défensif détecte l'altération du point d'entrée et termine le processus `services.exe`
- Détection lors de la notification de `Shell32.dll`, parce que le driver défensif intervient avant Duqu
- Si on démarre le driver de Duqu en premier, on détecte l'altération à la notification de `kernel.dll`

Démonstration

```
----------++++++****** Create process 0xaa4******++++++----------
ProcessImageInformation: PEB=0x7ffdf000 ImageBaseAddress=0x01000000 ProcessId=0xaa4
ParsePEModule: PE
Entrypoint bytes at 0x01012475: 0x6a 0x70 0x68 0xe0 0x15 0x00 0x01 0xe8
ProcessImageInformation: Entrypoint=0x01012475 DataDirectory=0x01000168
ProcessImageName: \Device\HDV1\Documents and Settings\fabrice\Bureau\services.exe
ProcessImageName: save processID=0xaa4
CreateProcessNotify: Create=1 0x748=>0xaa4
CreateProcessNotify: EntrypointChecksum=0x49af1bf2

----------****** A PID Load module \WINDOWS\system32\kernel32.dll *****----------
LoadImageNotifyRoutine: ImageBaseAddress=0x7c800000 ProcessId=0xaa4
ParsePEModule: PE
Entrypoint bytes at 0x7c80b64e: 0x8b 0xff 0x55 0x8b 0xec 0x83 0x7d 0x0c
LoadImageNotifyRoutine: Entrypoint=0x7c80b64e DataDirectory=0x7c800168
-> Verify services.exe :Entrypoint at 0x01012475:
0x6a 0x70 0x68 0xe0 0x15 0x00 0x01 0xe8
OK!

----------****** A PID Load module \WINDOWS\system32\shell32.dll *****----------
LoadImageNotifyRoutine: ImageBaseAddress=0x7c9d0000 ProcessId=0xaa4
ParsePEModule: PE
Entrypoint bytes at 0x7c9f7496: 0x8b 0xff 0x55 0x8b 0xec 0x53 0x8b 0x5d
LoadImageNotifyRoutine: Entrypoint=0x7c9f7496 DataDirectory=0x7c9d0158
-> Verify services.exe :Entrypoint at 0x01012475:
0xb8 0xbd 0x18 0x0a 0x00 0xff 0xd0 0xe8

!!!!!>>> Checksum error !!!!!!!!!
Terminating services.exe
drvTerminateProcess( 2724 )
```

Perspectives

On détecte l'altération du point d'entrée et on empêche l'exécution de la charge finale.

Perspectives

On détecte l'altération du point d'entrée et on empêche l'exécution de la charge finale.

Et maintenant ? On voudrait :

- Scanner la mémoire à la recherche d'images exécutables
- Les analyser avec le détecteur
- S'en servir pour détecter d'autres attaques

Perspectives

On détecte l'altération du point d'entrée et on empêche l'exécution de la charge finale.

Et maintenant ? On voudrait :

- Scanner la mémoire à la recherche d'images exécutables
- Les analyser avec le détecteur
- S'en servir pour détecter d'autres attaques

Et puis on termine `services.exe`...

- Restaurer son point d'entrée, permettre à Windows de se lancer normalement ?

Conclusion

- Décompilation du driver de Duqu et analyse de son fonctionnement : injection discrète dans `services.exe`
- Construction d'un driver dérivé défensif capable de détecter une attaque par Duqu

Conclusion

- Décompilation du driver de Duqu et analyse de son fonctionnement : injection discrète dans `services.exe`
- Construction d'un driver dérivé défensif capable de détecter une attaque par Duqu

- Merci
- Des questions ? (aurelien.thierry@inria.fr)