Hacking Windows CE

san@nsfocus.com
san@xfocus.org

XFocus Team
www.xfocus.org
www.xfocus.net

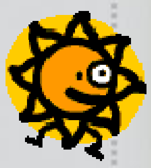
X'con 2005

Structure Overview

- Windows CE Overview
- Windows CE Memory Management
- Windows CE Processes and Threads
- Windows CE API Address Search Technology
- The Shellcode for Windows CE
- System Call
- Windows CE Buffer Overflow Demonstration
- About Decoding Shellcode
- Conclusion
- Reference

Windows CE Overview(1)

- Windows CE is a very popular embedded operating system for PDAs and mobiles
- Windows developers can easily develop applications for Windows CE
- Windows CE 5.0 is the latest version
- This presentation is based on Windows CE.net(4.2)
- Windows Mobile Software for Pocket PC and Smartphone are also based on the core of Windows CE
- By default Windows CE is in little-endian mode



Windows CE Overview(2)

- **ARM Architecture**
  - **RISC**
  - **ARMv1 - v6**

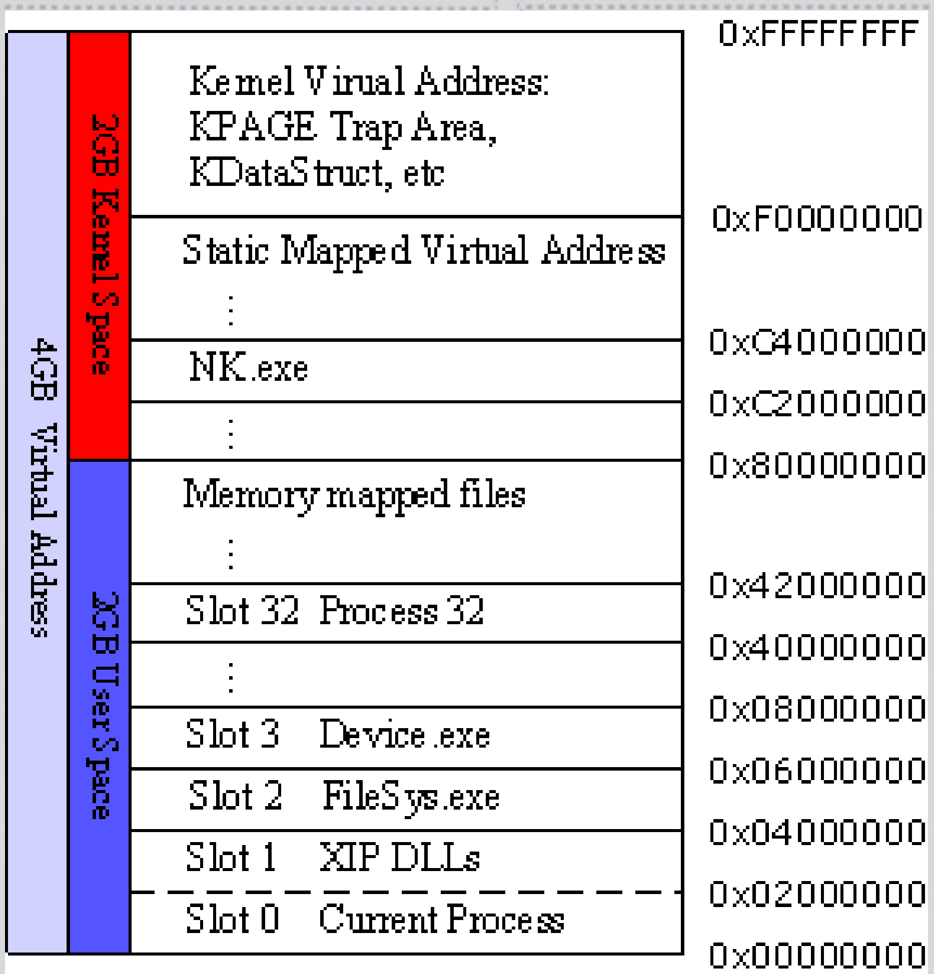
All Modes (User – Fast Interrupt)
Privelege Mode (System – Fast Interrupt)
Exception Mode (Supervisor – Fast Interrupt)

User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Memory Management(1)

- **Windows CE uses ROM (read only memory), RAM (random access memory)**
  - **The ROM in a Windows CE system is like a small read-only hard disk**
  - **The RAM in a Windows CE system is divided into two areas: program memory and object store**
- **Windows CE is a 32-bit operating system, so it supports 4GB virtual address space**
- **Upper 2GB is kernel space, used by the system for its own data**

Memory Management(2)

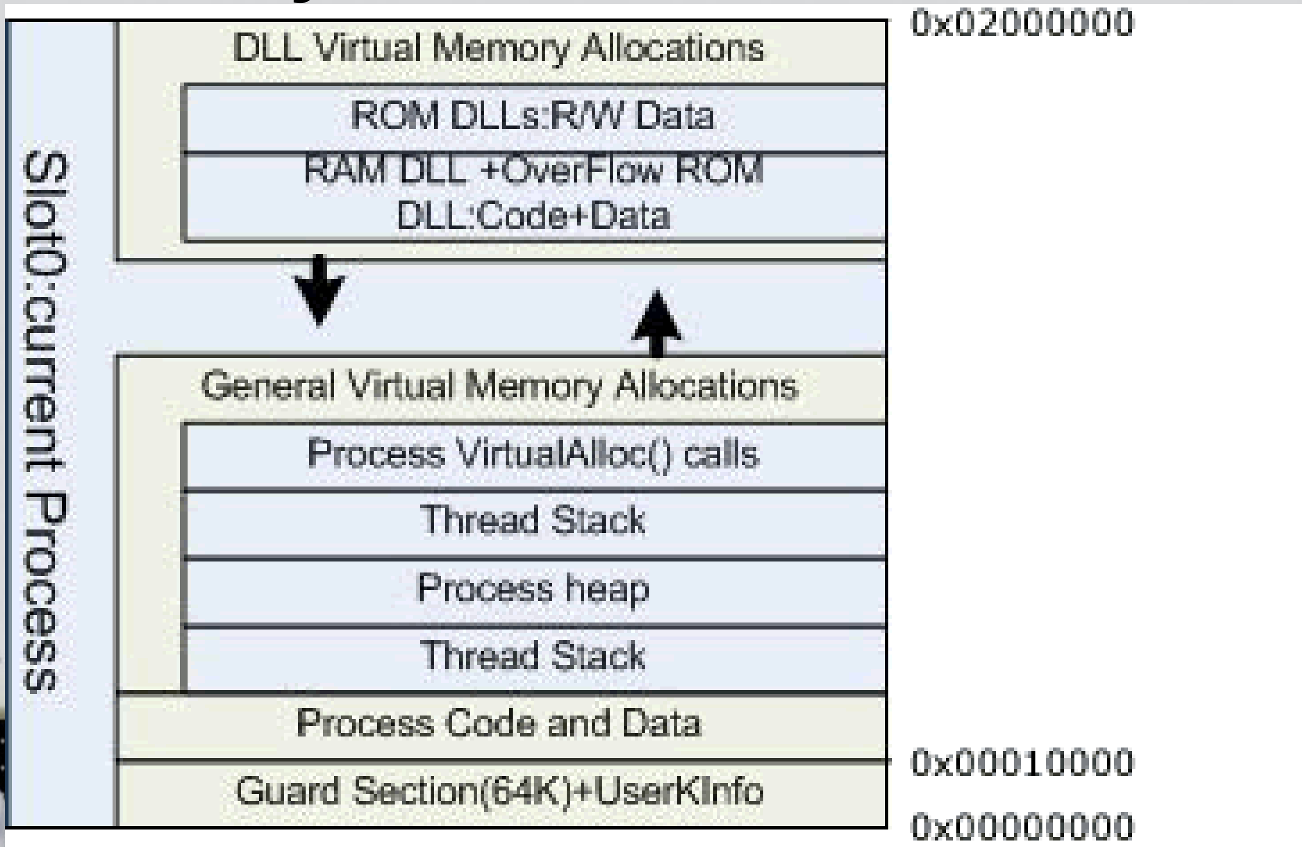


Memory Management(3)

- **Lower 2GB is user space**
  - **0x42000000-0x7FFFFFFF memory is used for large memory allocations, such as memory-mapped files**
  - **0x0-0x41FFFFFF memory is divided into 33 slots, each of which is 32MB**

Memory Management(4)

- **Slot 0 layout**



Processes and Threads(1)

- **Windows CE limits 32 processes being run at any one time**
- **Every process at least has a primary thread associated with it upon starting (even if it never explicitly created one)**
- **A process can created any number of additional threads (only limited by available memory)**
- **Each thread belongs to a particular process (and shares the same memory space)**
- **SetProcPermissions API will give the current thread access to any process**
- **Each thread has an ID, a private stack and a set of registers**

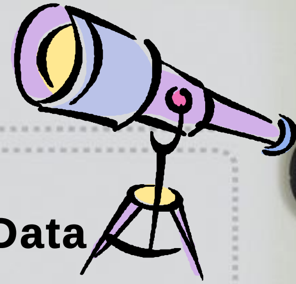
Processes and Threads(2)

- When a process is loaded
  - Assigned to next available slot
  - DLLs loaded into the slot
  - Followed by the stack and default process heap
  - After this, then executed
- When a process' thread is scheduled
  - Copied from its slot into slot 0
- This is mapped back to the original slot allocated to the process if the process becomes inactive

Processes and Threads(3)

- Processes allocate stack for each thread, the default size is 64KB, depending on the link parameter when the program is compiled
  - Top 2KB used to guard against stack overflow
  - Remained available for use
- Variables declared inside functions are allocated in the stack
- Thread's stack memory is reclaimed when it terminates

API Address Search(1)



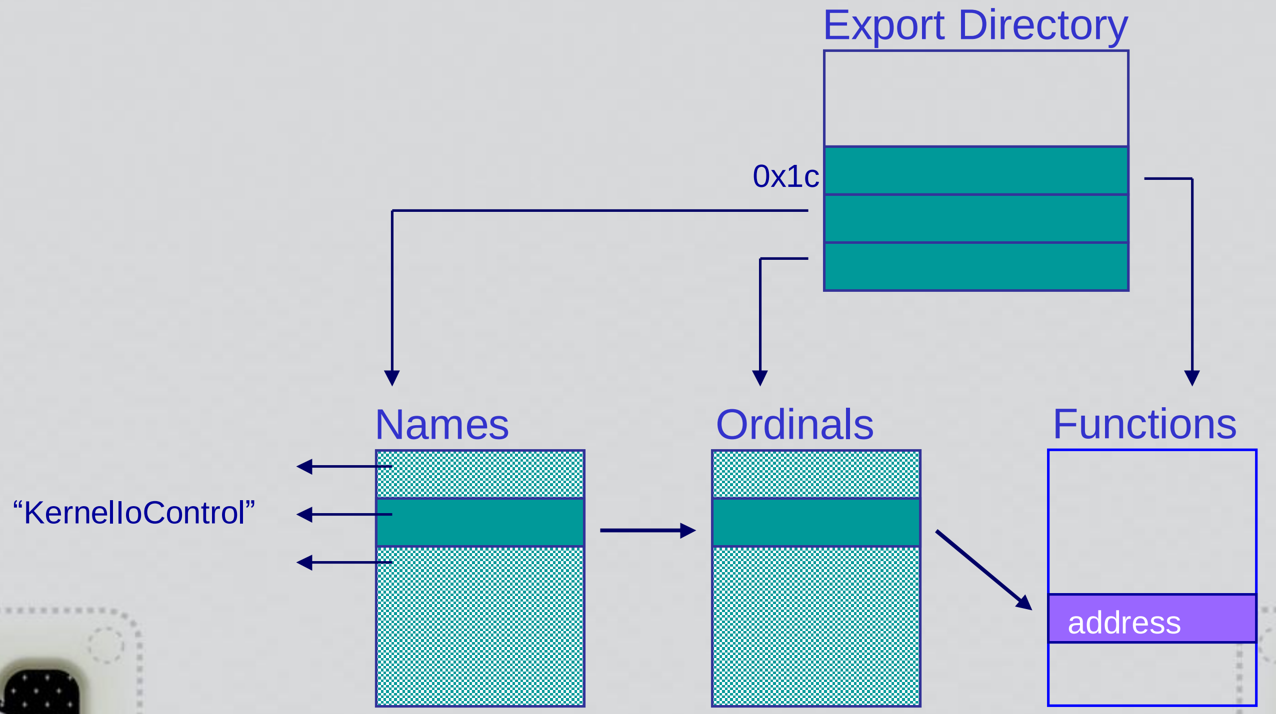
- Locate the loaded address of the coredll.dll
  - struct KDataStruct kdata;   // 0xFFFFC800: PUserKData
  - 0x324 KINX_MODULES   ptr to module list
  - LPWSTR   lpszModName;   /* 0x08 Module name */
  - PMODULE   pMod;   /* 0x04 Next module in chain */
  - unsigned long   e32_vbase;   /* 0x7c Virtual base address of module   */
  - struct info e32_unit[LITE_EXTRA]; /* 0x8c Array of extra info units   */
    - 0x8c   EXP Export table position
- PocketPC ROMs were builded with Enable Full Kernel Mode option
- We got the loaded address of the coredll.dll and its export table position.

API Address Search(2)

- **Find API address via IMAGE_EXPORT_DIRECTORY structure like Win32.**

```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
    ......
    DWORD   AddressOfFunctions;        // +0x1c RVA
   from base of image
    DWORD   AddressOfNames;            // +0x20 RVA
   from base of image
    DWORD   AddressOfNameOrdinals;  // +0x24 RVA
   from base of image
                                              // +0x28
} IMAGE_EXPORT_DIRECTORY,
   *PIMAGE_EXPORT_DIRECTORY;
```

API Address Search(3)



Shellcode(1)

- **test.asm - the final shellcode**
  - **get_export_section**
  - **find_func**
  - **function implement of the shellcode**

- **It will soft reset the PDA and open its bluetooth for some IPAQs(For example, HP1940)**

Shellcode(2)

- **Something to attention while writing shellcode**
  - **LDR pseudo-instruction**
    - **"ldr r4, =0xffffc800" => "ldr r4, [pc, #0x108]"**
    - **"ldr r5, =0x324" => "mov r5, #0xC9, 30"**
  - **r0-r3 used as 1st-4th parameters of API, the other stored in the stack**

Shellcode(3)

- **EVC has several bugs that makes debug difficult**
  - **EVC will change the stack contents when the stack reclaimed in the end of function**
  - **The instruction of breakpoint maybe change to 0xE6000010 in EVC sometimes**
  - **EVC allows code modify .text segment without error while using breakpoint. (sometimes it's useful)**

System Call

- **Windows CE APIs implement by system call**
- **There is a formula to calculate the system call address**
  - **0xf0010000-(256*apiset+apinr)*4**
- **The shellcode is more simple and it can used by user mode**

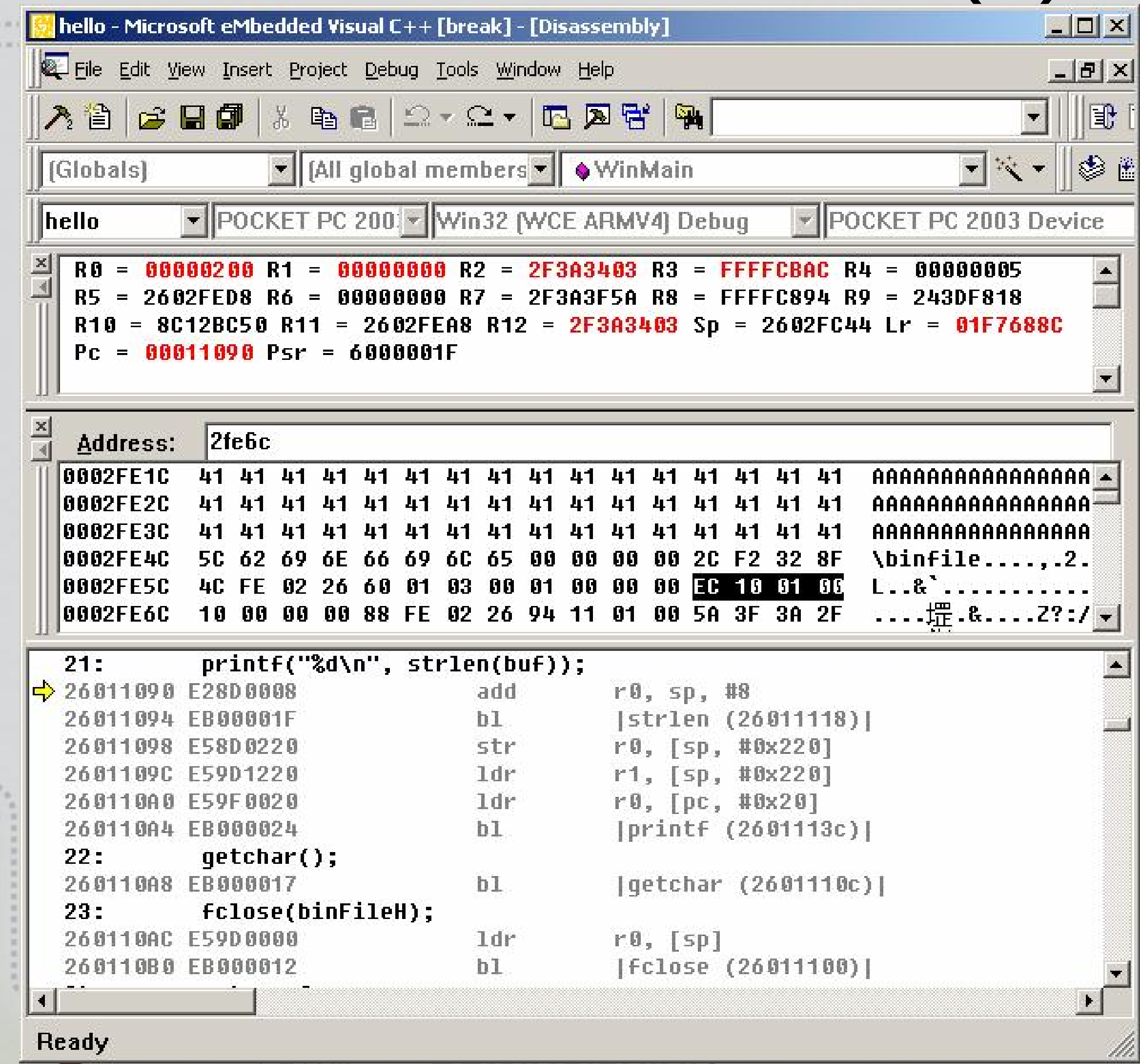
Buffer Overflow Demo(1)

- **hello.cpp - the vulnerable program**
  - **Reading data from the "binfile" of the root directory to stack variable "buf" by fread()**
  - **Then the stack variable "buf" will be overflowed**
- **ARM assembly language uses bl instruction to call function**
  - **"str lr, [sp, #-4]! " - the first instruction of the hello() function**
  - **"ldmia sp!, {pc} " - the last instruction of the hello() function**
  - **Overwriting lr register that is stored in the stack will obtain control when the function returned**

Buffer Overflow Demo(2)

- **The variable's memory address allocated by program is corresponding to the loaded Slot, both stack and heap**
- **The process maybe loaded into the difference Slot at each start time, so the base address always alters**
- **Slot 0 is mapped from the current process' Slot, so its stack address is stable**

Buffer Overflow Demo(3)



Buffer Overflow Demo(4)

- **A failed exploit**

PAD……	Return Address	……NOP……	Shellcode

–The PDA is frozen when the hello program is executed

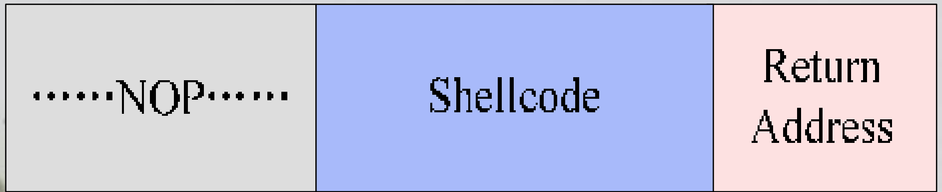
–Why?

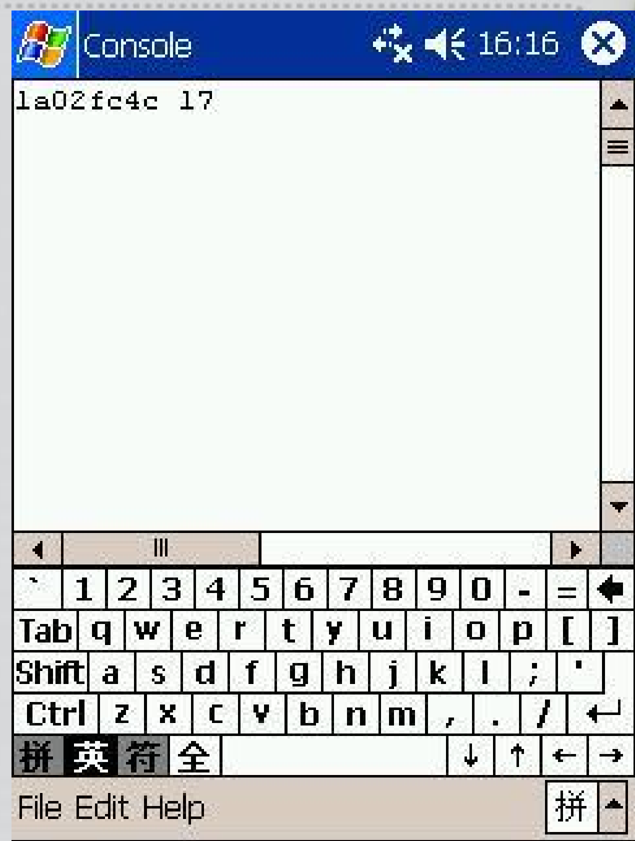
•The stack of Windows CE is small

•Buffer overflow destroyed the 2KB guard on the top of stack boundary

Buffer Overflow Demo(5)

- **A successful exploit - exp.c**
  - **The PDA restarts when the hello program is executed**
    - **The program flows to our shellcode**





About Decoding Shellcode(1)

- **Why need to decode shellcode?**
  - **The other programs maybe filter the special characters before string buffer overflow in some situations**
  - **It is difficult and inconvenient to write a shellcode without special characters by API address search method in Windows CE**

About Decoding Shellcode(2)

- **The newer ARM processor has Harvard Architecture**
  - **ARM9 core has 5 pipelines and ARM10 core has 6 pipelines**
  - **It separates instruction cache and data cache**
  - **Self-modifying code is not easy to implement**

About Decoding Shellcode(3)

- **A successful example**
  - **only use store(without load) to modify self-code**
  - **you'll get what you want after padding enough nop instructions**
  - **ARM10 core processor need more pad instructions**
  - **Seth Fogie's shellcode use this method**

About Decoding Shellcode(4)

- **A puzzled example**
  - **load a encoded byte and store it after decoded**
  - **pad instructions have no effect**
  - **SWI does nothing except 'movs pc,lr' under Windows CE**
  - **On PocketPC, applications run in kernel mode. So we can use mcr instruction to control coprocessor to manage cache system, but it hasn't been successful yet**

Conclusion

- The codes talked above are the real-life buffer overflow example in Windows CE
- Because of instruction cache, the decoding shellcode is not good enough
- Internet and handset devices are growing quickly, so threats to the PDAs and mobiles become more and more serious
- The patch of Windows CE is more difficult and dangerous

Reference

- [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
  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-PLaNET

Thank You!

san@nsfocus.com
san@xfocus.org