# Table of Contents

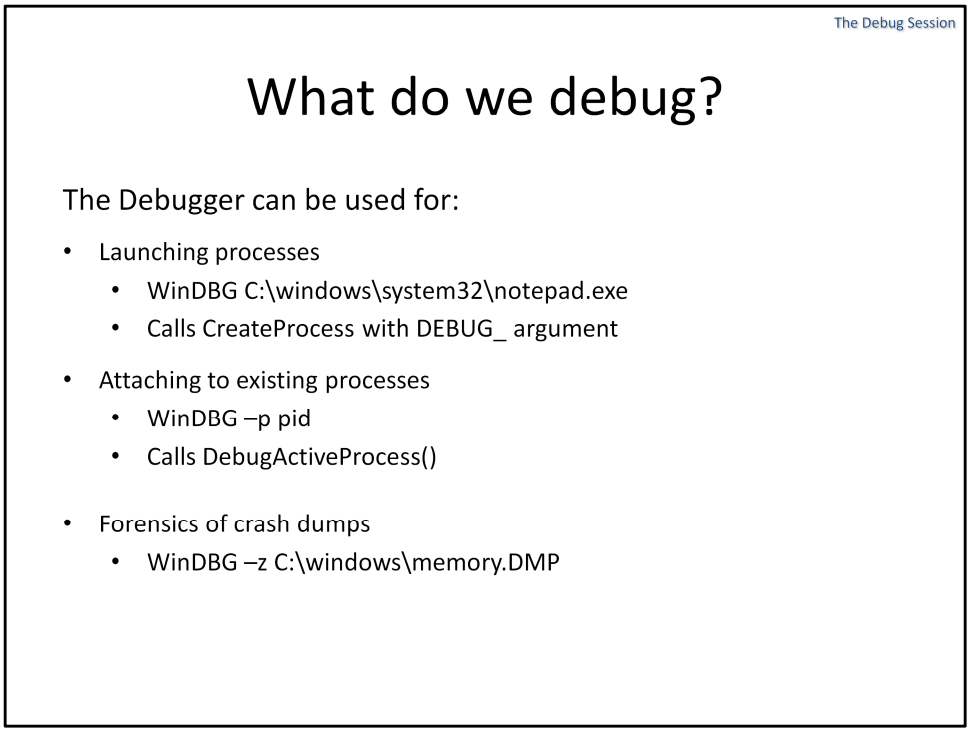


Jonathan Levin specializes in training and consulting services. This, and many other training materials, are created and constantly updated to reflect the ever changing environment of the IT industry.

To report errata, provide feedback, or for more details, feel free to email JL@HisOwn.com



Printed on 100% recycled paper. I hope you like the course and keep the handout.. Else – Recycle!




Normally, we distinguish between three usage scenarios of debuggers:

- **Launching Processes:** When you want to debug a program from its very first stages, the debugger accepts the program name as an argument, and starts the program as its own subprocess. When doing so, the debugger immediately has full control of the program, and all operations on it are allowed. The Debugger starts the process by using the Win32 API call **CreateProcess()**:

```
BOOL WINAPI CreateProcess(__in_opt LPCTSTR lpApplicationName,
                        __inout_opt LPTSTR lpCommandLine,
                           __in_opt LPSECURITY_ATTRIBUTES lpProcAttr,
                           __in_opt LPSECURITY_ATTRIBUTES lpThrAttr,
                               __in BOOL            bInheritHandles,
                               __in DWORD dwCreationFlags,
                          __in_opt LPVOID lpEnvironment,
                          __in_opt LPCTSTR lpCurrentDirectory,
                          __in     LPSTARTUPINFO lpStartupInfo,
                             __out LPPROCESS_INFORMATION lpProcInfo);
```

And specifying as an argument to “dwCreationFlags” either DEBUG_PROCESS (0x1) to debug the process and any children, or DEBUG_ONLY_THIS_PROCESS (0x2) to debug the process but no follow forks..

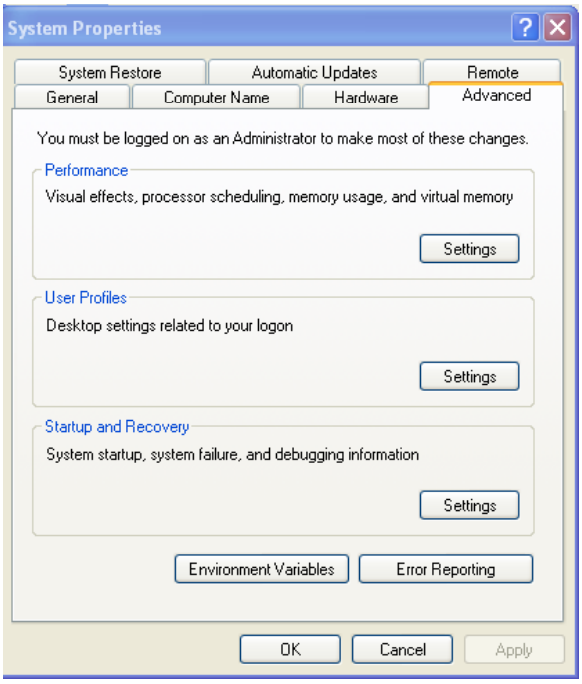
- **Attaching to Processes:** When the debugged process is already running, the debugger has to *attach* to it. Attaching involves a call to the Win32 **DebugActiveProcess()** and setting the debugger to receive notifications or debugging events from the process. Detaching involves a call to **DebugActiveProcessStop()** and optionally **DebugSetProcessKillOnExit().**

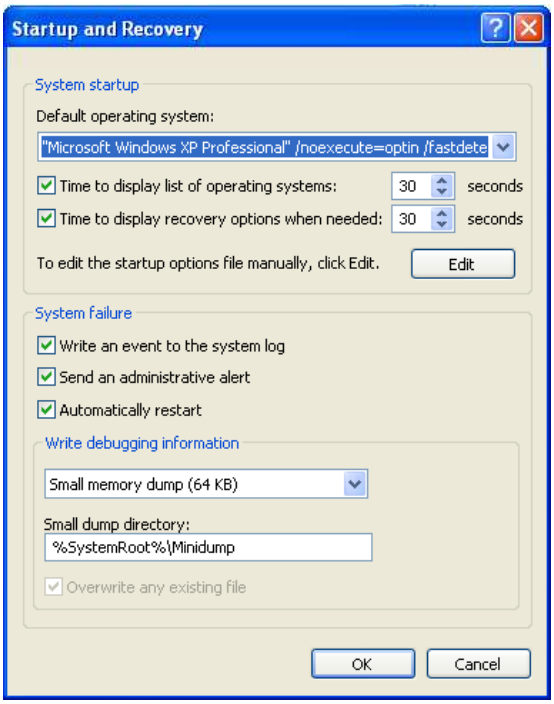
The debugger can attach to a process by one of two ways:

- Specifying the PID or process name on the command line

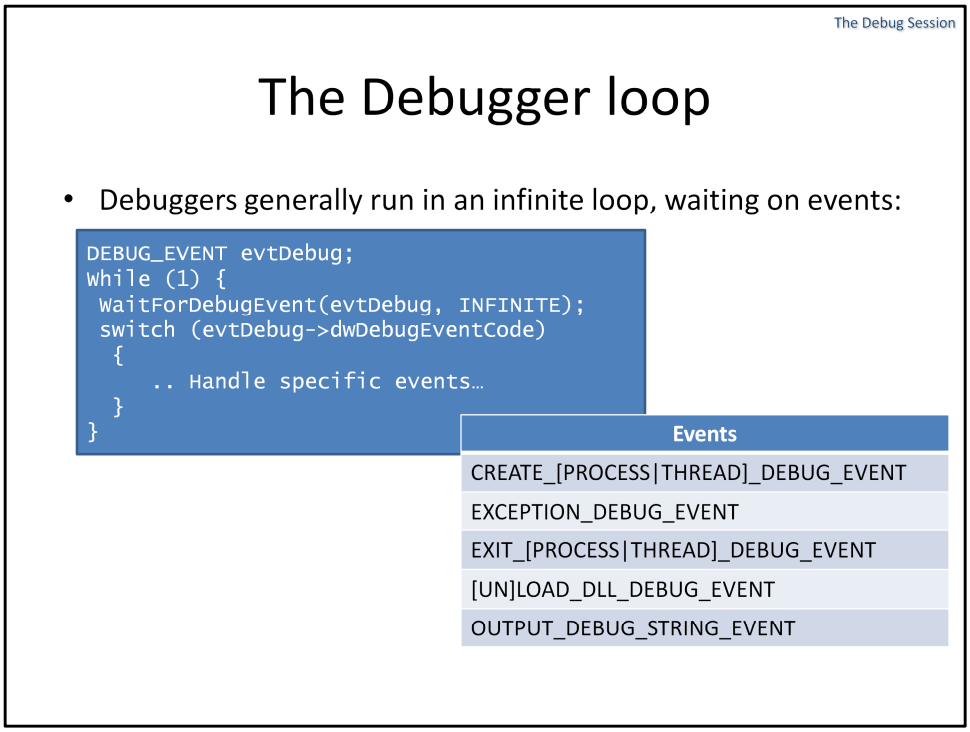
- Using the built-in ".attach" command with the PID.

- **Crash Dump Debugging:** One of the Debugger's most useful features is debugging from a crash dump. A dump is much like a UNIX core file, often with a .dmp extension. Dumps are created when the application crashes via "dumprep.exe" (which also suggests sending the dump to Microsoft), or when the system crashes on a blue screen of death, if the option is selected. In Windows XP, this is set under "Advanced" in the computer properties:




The "-z" switch is used to open the dump, and inspect. Useful in this context is the debugger extension of "!analyze" which automatically diagnoses the dump and produces a detailed report of what happened, and who the culprit was.




Writing a basic debugger in Windows isn't as hard as it might seem. Windows has built-in debugging support, and a debugger need only open or create a target process, and enter an infinite loop to wait for its debug events.

The Debug events are defined in the above table, and they are effectively an enum, so the main loop can switch() on them:

| Events | Event Code | Occurs when |
|---|---|---|
| CREATE_PROCESS_DEBUG_EVENT<br>CREATE_THREAD_DEBUG_EVENT | lbp cpr<br>ct | Main process and/or subprocess created<br>CreateThread() |
| EXCEPTION_DEBUG_EVENT | | Breakpoint (INT 3), or general exception |
| EXIT_PROCESS_DEBUG_EVENT<br>EXIT_THREAD_DEBUG_EVENT | epr<br>et | Last thread of a process exits<br>A thread of a process exits |
| LOAD_DLL_DEBUG_EVENT<br>UNLOAD_DLL_DEBUG_EVENT | ld<br>ud | LoadLibrary() is called<br>FreeLibrary() is called |
| OUTPUT_DEBUG_STRING_EVENT | out | OutputDebugString() is called |

The Debug Session

## Setting Breakpoints

b – set breakpoint

| switch | |
|---|---|
| l | List all breakpoints |
| c | Clear (remove) all breakpoints |
| e/d | Enable/disable all or selected breakpoints |
| m | Set symbolic breakpoint; optionally run command |
| r | Renumber breakpoints |
| a | Break on **r**ead, **w**rite or e**x**ecute access |

- Programmatic breakpoints may be set by "int 3" or DebugBreak()
- Conditional breakpoints can be set with "j" or with ".if":
  - **bp _____ "g;.if (*condition*) {g} .else {gc}"**
  - **bp _____*"j (Condition) 'OptionalCommands'; 'gc' "***

Breakpoints are undoubtedly one of the most powerful features of a debugger – the ability to pause program execution indefinitely, and use it to inspect the program's state, registers and memory, as well as potentially modify it. Even more useful is the ability to automatically run a debugger command on a breakpoint.

The "b" command can be used from inside the debugger to set and/or manage breakpoints. A breakpoint can be set at a specific address, and is set by replacing the opcode in that address by "INT 3" (0xCC). If a debugger is attached, this opcode triggers a DebugEvent, (if it isn't, an unhandled exception occurs). Any debugger waiting on the event can intercept it (as an EXCEPTION_DEBUG_EVENT), and handle it.

A common technique is to embed a selective breakpoint in a program. This can be done by emitting an "int 3" directly in the code (in an __asm block):

```
__asm { int 3 }
```

or by calling the **DebugBreak()** API (which essentially does the same thing). Usually, this is coupled with a call to the Kernel32 export **IsDebuggerPresent()**:

```
if (IsDebuggerPresent())
  {
      DebugBreak();
  }
```

**DebugBreak**() calls **ntdll!DbgBreakPoint**, which you can also call directly.

**Some examples:**

Break only once:

- bp *addr* /1

Break only on 5th or later time:

- bp *addr 5*

Break on write to var, a DWORD (4-byte) value:

- ba w4 var

Breakpoints can be set as "conditional": not really conditional per se, but rather breakpoints that do break, then evaluate an expression and – if false, simply resume automatically by executing "g".

For example:

```
- bu func "j (dwo(@esp+8) == 1)
     '.echo Second argument is 1 - breaking; kn' ; 'g' "
```

Will break only if the second argument to "func" is "1".

**Other examples:**

Break on a call to CreateFileW. Dump as unicode the first argument:

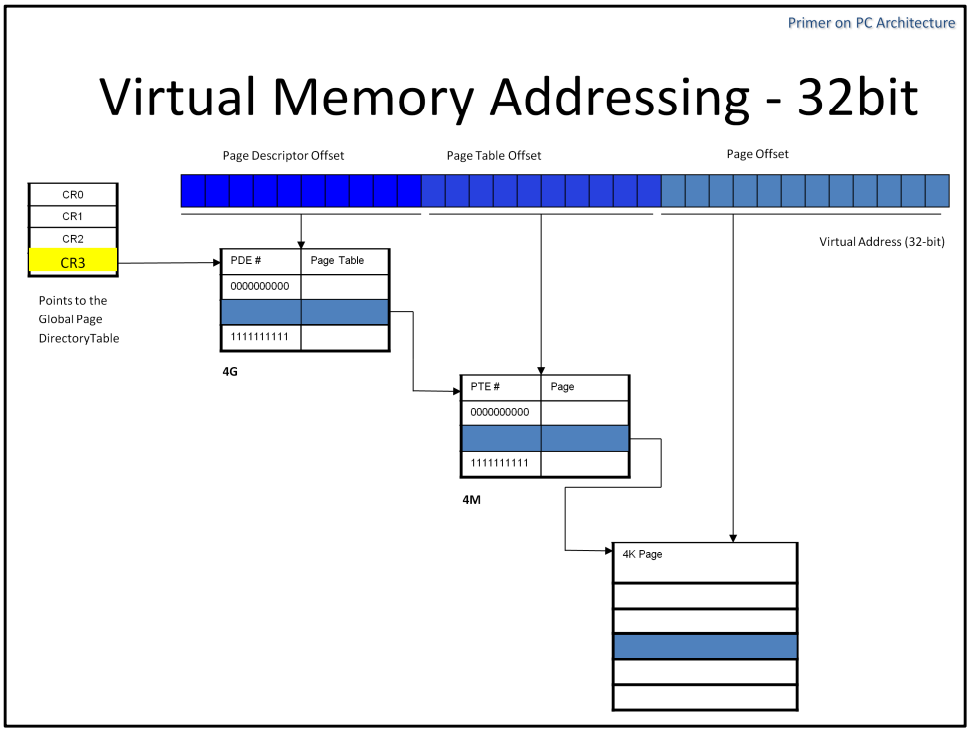
- bu kernel32!CreateFileW ".echo Got file; du dwo(@esp+4) ; gc"

This will print the filename (= last on the stack, @ESP+4) to CreateFileW.

Similarly, break on a call to LoadLibraryW. Dump as unicode the first argument:

- bu kernel32!LoadLibraryW ".echo Loaded:; du dwo(@esp+4); g"

This will print the filename (= last on the stack, @ESP+4) to LoadLibraryW.




The translation of Virtual Addresses into physical ones is a three staged process. Given a 32-bit address, The CPU segments the address into three separate parts:

**The first 10 bits** – point to one of $2^{10}$ entries in a global ***Page Directory Table***. This table is, in effect, a table of page tables, and the 10 bits select a specific page table index by a ***Page Directory Entry*** or ***PDE***. This table is defined per process, and maintained in a Page Descriptor Base Register, which on the Intel architectures is Control Register #3 (CR3). This register is reloaded on each process context switch from the KPROCESS object, since each process has a different virtual memory image.

**The next 10 bits** – point to a specific page (a.k.a ***Page Table Entry*** - PTE) in the ***Page Table*** that was selected by the previous 10 bits. 10 bits again mean $2^{10}$ – so each page table maintains the addresses of 4 MB (=$2^{10}$ * 4KB) of memory.

**The last 12 bits** – are the specific offset in the page itself. Since the page itself is 4KB (=4096 bytes) this works out perfectly with 4096 being $2^{12}$. However, most addresses are aligned on a DWORD boundary, which allows the system to reserve the last two bits for its own internal use.

Each page table maintains 4MB, and there are $2^{10}$ tables in the Page Descriptor Table – so $2^{10}$ * 4MB = 4GB, which is the size of the virtual address space of the process. Things look somewhat different when Physical Address Extensions* (PAE) are employed, but are sufficiently similar – as is shown next.

Debugging Processes

# Calling Conventions

- Functions can be called according to one of the following:

| Convention | Switch | Stack Cleanup | Arguments | Order | Decoration |
|---|---|---|---|---|---|
| __cdecl | /Gd | Caller | Stack, RTL | RTL | _name |
| WINAPI __stdcall | /Gz | Callee | Stack, RTL | RTL | _name@## |
| __fastcall | /Gr | Callee | ECX, EDX, stack | LTR/RTL | @name@## |

- Default is __cdecl, may be changed

- C++ member functions are called using “thiscall”:
  - “this” is passed in ECX. Rest as in __cdecl
  - Varargs force _cdecl

When calling functions, one of several “calling conventions” can be used. The common ones are listed here.

The default calling convention, **__cdecl**, is the “classic” method of calling functions in C and C++: The caller pushes the arguments in reverse order, right-to-left, on to the stack, and is also responsible for clearing the stack space upon function return – usually by adding the stack space value to ESP using an assembly “add” instruction.

A _cdecl call, e.g.

char *c = (char *) c malloc(240);
memset ((void *), ‘x’, 240) would look like this:

Would emit opcodes like this:

```
00401052 68f0000000      push    0F0h            // 0xF0 = 240
0040105f e85e000000      call    Test!malloc (004010c2)
// Malloc returns: Stack NOT cleared (optimization).malloc’ed ptr in EAX
00401064 68f0000000      push    0F0h            // 0xF0 = 240
00401069 8bf0            mov     esi,eax         // ESI = addr of ptr
0040106b 6a78            push    78h             // 0x78 = ‘x’
0040106d 56              push    esi             // addr of ptr
0040106e e869080000      call    Test!memset (004018dc)
00401073 83c410          add     esp,10h         // NOW we clear stack
```

The code is pretty simple: Arguments are pushed on the stack in reverse order. Note, though, that even this simple code has an optimization. The return from malloc doesn't clean up the stack – rather, the operation is deferred until we return from the memset, where we clear 0x10 bytes: 0x0C of the call to memset (3 DWORD arguments), and 0x04 of the call to malloc(1 DWORD argument).

The _cdecl'ed function's prolog normally begins:

```
00401000 55              push    ebp       // Save current frame pointer
00401001 8bec            mov     ebp,esp   // Set new frame pointer
```

And ends:

```
0040102d 8be5            mov     esp,ebp  // Restore stack pointer
0040102f 5d              pop     ebp      // from base of frame
00401030 c3              ret              // Pop RA off stack and jump
```

The **__stdcall** calling convention is a variant of the PASCAL calling convention (__pascal). It is a common convention for all the Win32 API functions – the compiler #define of "WINAPI" in fact resolves to it. In this calling convention, the callee is responsible for clearing up the stack space, and – as a result – variable argument functions are not allowed. The code generated, however, is smaller (though often marginally) since the cleanup code is inside the function, meaning only one instance thereof regardless of number of function invocations. The callee (called function) clears up its own stack space, usually by specifying the space as an argument to the assembly "ret" instruction. The prolog is thus largely the same:

```
:000> u Kernel32!CreateFileA
kernel32!CreateFileA:
7c801a24 8bff            mov     edi,edi // ??? - Stay Tuned ☺
7c801a26 55              push    ebp
7c801a27 8bec            mov     ebp,esp
```

While the epilog specifies a value to pop the stack by:

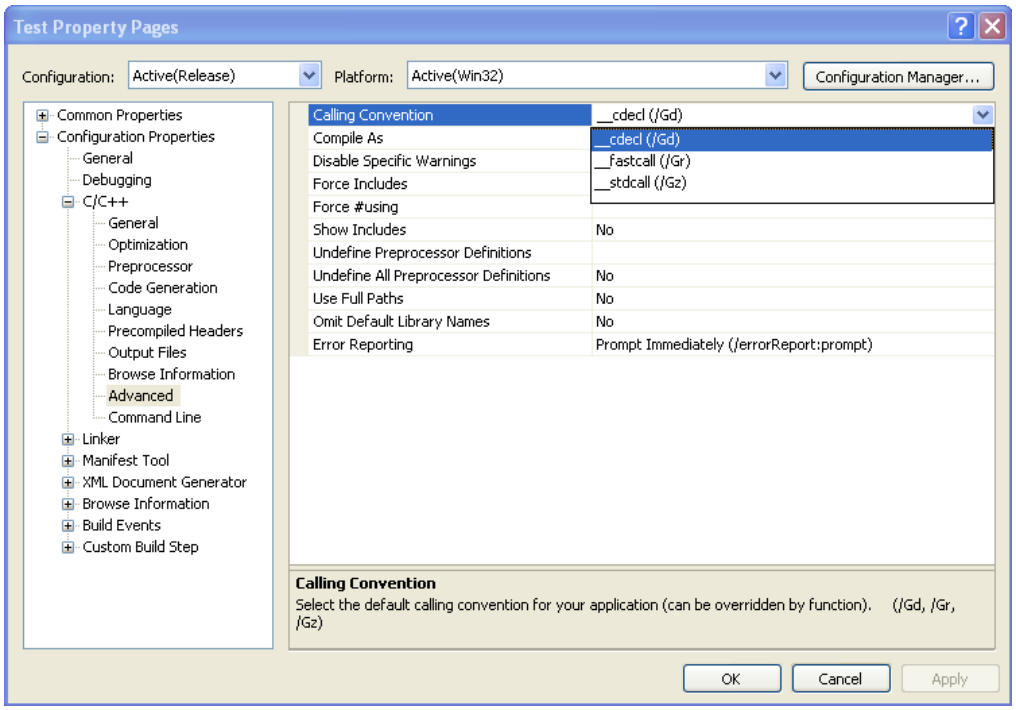
```
:000> u Kernel32!CreateFileA + 0x26
kernel32!CreateFileA+0x26:
7c801a4a e827ef0000      call    kernel32!CreateFileW (7c810976)
7c801a4f 5d              pop     ebp
7c801a50 c21c00          ret     1Ch // Pop RA, AND inc ESP by 0x1C bytes
```

Though __cdecl is the default calling convention in C++ programs as well as in C, it behaves a bit differently for member functions of objects. In those cases, if the function is of fixed arguments the “this” parameter (address of the calling object) is passed in ECX. If the member function supports variable arguments, the call becomes a standard __cdecl, with the “this” argument passed on the stack as the very last argument (so that it is popped first).

__fastcall allows for a simple optimization on __stdcall, by passing the first two arguments in the otherwise unused registers of ECX and EDX. This is faster because for functions of one or two arguments, there is no need to lay anything on the stack.

To avoid confusion and to make it easier during both linking and debugging, the function names are “decorated” by the compiler. This “decoration” is effectively a form of name-mangling, in which an underscore (_) or at sign (@) is prepended to the function, to distinguish its type, and for __stdcall and __fastcall, which allow only fixed arguments, the expected size of the stack is specified after the function name.

The default convention in a project may be changed by overriding the switch directly (in the “Command Line” setting, or as a direct argument in the Makefile, or – from the project properties, in “Advanced”:



*…If you liked this course, consider…*

**Networking:**

Networking Protocols – OSI Layers 2-4:

Focusing on - Ethernet, Wi-Fi, IPv4, IPv6, TCP, UDP and SCTP

Application Protocols – OSI Layers 5-7:

Including - DNS, FTP, SMTP, IMAP/POP3, HTTP and SSL

VoIP:

In depth discussion of H.323, SCCP, SIP and RTP/RTCP, down to the packet level.

Windows Networking Internals:

NetBIOS/SMB, CIFS, DCE/RPC, Kerberos, NTLM, and networking architecture

---

**Linux:**

Linux Survival and Basic Skills:

Graceful introduction into the wonderful world of Linux for the non-command line oriented user. Basic skills and commands, work in shells, redirection, pipes, filters and scripting

Linux Administration:

Follow up to the Basic course, focusing on advanced subjects such as user administration, software management, network service control, performance monitoring and tuning.

Linux User Mode Programming:

Programming POSIX and UNIX APIs in Linux, including processes, threads, IPC mechanisms and networking. Linux User experience required.

Linux Kernel Programming:

Guided tour of the Linux Kernel, 2.4 and 2.6, focusing on design, architecture, writing device drivers (character, block), performance and network devices. The counterpart of this course, for Linux.

Embedded Linux Kernel Programming:

Similar to the Linux Kernel programming course, but with a strong emphasis on development on non-intel and/or tightly constrained embedded platforms

---

**Windows:**

Windows Programming:

Windows Application Development, focusing on Processes, Threads, DLLs, Memory Management, and Winsock

Windows Kernel Programming

Windows Kernel Architecture and Device Driver development – focusing on Network Device Drivers (in particular, NDIS) and the Windows Driver Model. Updated to include NDIS 6 and Winsock Kernel

---

**Security:**

Cryptography:

From Basics to implementations in 5 days: foundations, Symmetric Algorithms, Asymmetric Algorithms, Hashes, and protocols. Design, Logic and implementation

Application Security

Writing secure code – Dealing with Buffer Overflows, Code, SQL and command Injection, and other bugs… before they become vulnerabilities that hackers can exploit.