UNDERSTANDING WINDOWS 2K SOURCES (part 1)

Written By: **AndreaGeddon**
[*www.andreageddon.com*] [*www.reteam.org*] [*www.quequero.org*]
{**andreageddon@gmail.com**}

:: INTRO ::

This is the first of a series of articles in which I will deal a little bit in detail with the windows 2000 kernel. In particular I will refer to the stolen sources that have been published. For obvious causes I shall not write the code directly in this article, but I will make precise references to the files I will describe, so if you have the sources you will find easy it easy to understand this text.

:: REQUIREMENTS ::

Well, first of all it would be good if you have the sources, if you don’t have them you can read the article the same as it will have a quite generic stamp. Second, you have to know hardware x86 architecture basics, infact I will not deal with things like IDT etc, so get your Intel manuals and study them! Last, I will assume you have some basic knowledge about an operating system, that is you know what is a file system, what is a scheduler and so on. Now we can begin.

:: BIBLIOGRAPHY ::

Here are some books on the argument that I advise you to read:

- The Windows 2000 Device Driver Book - Art Baker, Jerry Lozano
- Inside Windows 2000 - Russinovich, Solomon (sysinternals)
- Windows driver model - Oney
- Windows NT Native Api - Gary Nebbett
- Undocumented Windows NT - Dabak, Phadke, Borate
- Windows NT File SYstem Internals - Nagar
- Windows NT Device Driver Development - Viscarola

:: THE BEGINNING ::

The source leak is dated on the first ten days of february, with the direct responsibility for it, being Mainsoft; an old partner of Microsoft. However, the leak contest is still not very clear.

Let’s start from the beginning, that is*:*

***Where can I find these sources*?** Well you should resolve this by yourself! You can search on filesharing networks, or private ftps. I do not advise you to use public networks, better if you use some crypto p2p network, or if you can find a friend that cand send you it.

***How many versions are there*?** Well there are lots of fakes, however the versions are two: one is for windows nt4 sources and the other is for windows 2000 sp1 sources. Here we will mainly refer to win2k ones, but it would be good for you to have the nt4 as well. They are quite different and both have some unique things to them. In fact, the first part of the boot process will be described using nt4 sources. Clearly the base is the same, so the things we will say here are good also on win xp and 2k3.

***Are they complete sources*?** Not in a strict sense. There is the kernel, the userspace dlls (see later), there is even the source for the solitaire game! The real bad lack is the ntfs source, it’s managed by the relative driver, and its source is not present in either of the the sources. However there is still something useful, which we will see later. The whole GDI is missing too. However, In general we can say that the interesting part (the kernel) is complete.

***So can I recompile it*?** Well the sources are missing some definitions and other files, the kernel *could* be recompiled with the help of the ifs kit, but all the usermode code is not recompilable. At the time of writing this article, I have no knowledge of recompiled windows kernels or similiar things. If you know of something like this, or you did it yourself please let me know.

***What are these sources useful for*?** Well not all that much I would say. They are useful mainly as a documentation for driver developers or windows emulators. In effect, this source package makes part of the Microsoft WISE program (Windows Interface Source Environment), that is a program that aimes to help developers to integrate Windows Based solutions on Unix and Macintosh systems.

***Is it true that the source leak can be a threat for security*?** Absolutely false, although advisors have been quite paranoid on this topic. At the moment I write there is no notice of bad bugs derived from kernel source analysis. The only bug reported is on IE 5, that is the overflow in the bitmaps handling.

***How is the code written, and how is it compiled*?** The kernel is written all in C (not C++) and some parts in asm, the parts strictly related to hardware. The applicative usermode code instead is mainly written in C++. Obviously the kernel design was made with an object oriented mentality. The code is compiled with Visual Studio, clearly not the commercial version but a proper version that is only for internal use.

***Will the diffusion of these sources change something at hacking level, programmers, final users etc*?** No, nothing will change. Driver developers already have excellent knolwedge of the windows kernel (for example: System Internals, OSR, NTDEV etc etc). Certainly the sources will enrich the already existing kernel documentation, but whoever works in this branch is used to reverse engineering, so probably whatever they needed they already learned it from kernel reversing ☺. Not many people know in fact that Microsft offers the debug symbols of every windows module, so in the disassebly of a piece of code like this:

```
mov  [0x11223344], eax
push  0x22334455
call 0x77889900
```

with debug symbols would be resolved in something like:

```
mov  [_TickCount], eax
push _dwSeconds
call _GetTime
```

so once you solve the names, then the translation from asm to C is really easy. There is not much difference in reading the code above or the relative C code:

```
TickCount = ...blahblah...;
GetTime(dwSeconds);
```

What to say? Windows IS opensource, you just have to know how to read assembler. Matt Pietrek for example rewrote a lot of the windows 9x kernel in pseudo code. The only real progress we will see will be in windows emulators, for the rest the windows source leak echo is already dead.
Now that we have a general overview of the argument we can run into code details!

:: SOURCE ORGANIZATION ::

If you don’t want to get lost in the sea of code lines you will have to make a map of the principal components. First of all we see three main directories:

```
\bsc
\private
\public
```

the first contains the glimpse data for the search engine, the last contains sdk and oak, both of scarce interest for us here. What we want is the \private, base of all the code. In nt4 sources only \private is present. Now this directory as you can see is really big! So we are not going to comment it all. Let’s see a map of the components:

module: ntoskrnl.exe
location: \private\ntos
description: windows kernel, the equivalente of bzImage in Linux

module: ntdll.dll
location: \private\ntos\dll
description: gateway for um->km transitions (syscalls)

module: kernel32.dll
location: \private\windows\base\client
description: usermode part of windows kernel

module: user32.dll
location: \private\ntos\w32\ntuser\client
description: various utilities, such as windows creation and text manipulation etc

module: advapi32.dll
location: \private\windows\screg\winreg
description: registry apis

These are the main components, but we will concentrate 90% on the kernel portion. In \private\windows\shell\ you can find the sources for regedit, taskmanager, games and other applications. There are also other component sources, such as comdlg32 etc. The real lack, as previously mentioned, are

ntfs.sys – driver for ntfs
gdi32.dll – base graphic functions library

In the win2k sources the bootloader is not present, but it is present in the sources for nt4, precisely in the directory

\private\ntos\boot

There you can find the bootsector, ntloader, ntdetect and setup loader, each in its own directory. There is the bootsector relative to each fs, in particular for ntfs, so here you can find good data about an ntfs partition (same data you can find on Nagar book). If you are interested in ntfs there are the logfile management functions too, which we will see later. Back to the win2k code, we find part of the code relative to the net in

\private\inet

that is part of the IE (mshtml), urlmon, wininet.
Ok, now we have a general idea of the source structure, if you are searching for things I did not mention use a good grep tool.

:: WE START FROM BOOT ::

Time to begin touching the code! Uhm where are we starting from? We start from boot? Ok! As seen above we have to dig into the \private\ntos\boot directory. The bootsector itself is located in the \bootcode subdirectory, in which every file system has its own subdirectory. We can see that the starting point is the \mbr subdirectory, that is the physical code that will reside in the master boot record. Its the very first piece of operating system that is executed at the boot immediately after the bios code (the file is x86mboot.asm). As you can see from the comments, it is the STANDARD code for every pc. This code reads the partition table at the end of the master boot record, finds the partition marked as bootable, copies its bootsector in memory and executes it. The master boot record infact has this structure:

```
   +------------+  -- MBR --
   | BootCode   |  Executable Code
   | Partition1 |  PartitionTable
   | Partition2 |
   | Partition3 |
   | Partition4 |
   +------------+  -- END MBR --
   | Partition1 |  Partitions
   |            |
   .            .
   .            .
```

So the bootcode will just relocate to the address 0000:0600, jump to relocated code, read the bootable entry from the partition table, copy its bootsector in memory at standard boot address (0000:7C00), and finally execute it. So it is as if the bios itself had booted the partition directly. Uhm, note that in x86mboot.asm at line 48, that after that, the code auto relocates at address 0000:0600, to jump to it there is used a far jmp which is hand encoded as you can see here:

```
 db OEAh
 db ...blabla...
```

Those are the bytes relative for the opcode of the JMP 0000:0600 instruction, whose address is resolved at compile time. You will find again this hand coded opcode later, when the bootable partition is found. The code will jump again to 0000:7C00 to boot the new bootsector. At this point the code changes, as we have a different piece of code for every supported file system:

```
 \etfs  Electronic Tariff Filing System
 \fat   fat32
 \hpfs  Pinball File System (high performance file syste, os2)
 \ntfs  nt native file system
 \ofs   surprise! Void directory!
```

Well just fat32 and ntfs are really supported, and it is really not good that you install windows nt on a fat32 partition. Now we can concentrate on the ntfs bootsector, as other bootsectors work the same way. The role of this piece of code (ntfsboot.asm) is simply to read the ntldr file, map it to the address 2000:0000 and execute it. Note that we are still in real mode, so all the code is still 16 bit. As we can see this code is a bit too large and cannot stay in the 512 bytes of the first sector: in fact the bios maps the first physical sector of the bootdisk (track 0, head 0, sector 1) in memory at physical address 7C00. Well the bios now did not really map in memory the first sector of the bootable partition, rather the mbr code did it. Why does this code not map all necessary code for ntfsboot which is bigger than 512 bytes? Obviously for compatibility with other systems. So the just mapped ntfsboot now has the immediate task of mapping all it’s other code. In fact, the code begins reading from the first sector, through all of the bootsector and relocates it to address 0D00:0000, so we have in memory at that address the bootsector and following sectors that contain needed code. Once the code is mapped, it jumps to 0D00:0200, that is to the second sector that has been read from the disk (the first one was already executed so it is no longer useful). This code is at physical address D200h, and is far below the address 20000h where the ntldr will be mapped, so there is no interference problems. Again we see that the jump to the newly relocated code is made (at line 165) with this code:

```
 push seg
 push offset
 ret
```

Hehe, it seems they had some problem in writing far jmps! Ok nothing important, just a curiosity. So now the execution moves to the second sector at the mainboot procedure. Before this procedure in the code we can see some data and the “55AA” signature that is at the end of the first sector. The mainboot procedure reads the ntldr file (you can see several functions to read ntfs to find that file), and at the end it returns the execution to the ntldr memory image, that as we said above is located at 2000:0000. The bootsectors relative to other file systems act the same way, only the code that reads ntldr data from the disk changes from system to system. Now we have to move to the ntldr code. Note that until now there has been no initialization, like paging abilitation, switching to protected mode, etc. We are still in real mode, so ntldr starts its executions at 16bit, and then will make the passage to protected mode and so to 32bit. Ok now the file we are examinating is

```
 \ntos\boot\startup\i386\su.asm
```

Do remember that we still are in nt4 sources. We see that the first line is a jmp RealStart. Among this jmp and the routine itself we can see some code concerning FAT. If the system was booted from FAT32, the code would still have to handle the ntldr reading and mapping before executing it. In this case we are considering ntfs, so we dont care of other FAT32 problems. The RealStart routine just prepares stack and segments to pass the execution to the procedure SuMain that is located in the file

```
 \ntos\boot\startup\i386\main.c
```

Finally we come to some C code! However, keep the file su.asm in mind, because it exports the protected mode enabling function that we will see in a while. This procedure initializes the video subsystem (InitializeVideoSubSystem in display.c), turns off floppy motor in case the system was booted from floppy (TurnMotorOff in su.asm), makes other initialization works, such as calculating the size of necessary memory for os loader, then after this stuff we get to the main point: the switching to 32bit.

Infact we see that the code enables the A20 line (EnableA20 in a20.asm) and relocates the IDT and GDT structures that will be used in protected mode. So now it’s time to switch to protected mode (EnableProtectPaging in su.asm). Note that the first time the code is executed, the paging will not be enabled, in fact the startup loader still has not determined a valid descriptor for PDE and PTE. The paging enabling will be done at the beginning of the os loader code that we will see soon. In particular we see that the code sets the selector for the PCR in the segment FS, that is the Process Control Region, a fundamental structure for the kernel, so we expect that soon the code will pass execution to the ntoskrnl.exe module. Besides, the memory areas for IDT and GDT have been set, while the LDT is zeroed. Windows NT infact, does not use LDT, opposed to consumer windows. So the code for the protected mode is:

```
  mov     eax, c30
  ...     (first time the paging enabling is avoided)
  or      eax,PROT_MODE
  mov     cr0,eax
```

But it still does not have all the switching to full 32bit finished, as now the code is setting segments, structures and the TSS descriptor. The execution comes back to the SuMain after the protected mode switching. The SuMain calls the RelocateLoaderSections functions to calculate the correct address where the entry point of the os loader is. This is infact a valid coff PE, and it is embedded inside the ntldr, so we can consider it the first real process that windows executes. Once its entry point is found the execution passes to it with the function TransferToLoader using as an entry point the just computed address. So now we move to the directory:

```
 \ntos\boot\lib\i386
```

where there are the files we are going to execute. In particular, the file:

```
 entry.c
```

This is the entry point of the just mentioned PE, and it is identified by the function NtProcessStartup. Let’s analize it, and we will see that the first called function is DoGlobalInitialization. Also here we can see that there is a function call: InitializeMemorySubsystem. Sounds interesting! Parenthesis: many functions use as a parameter the BootContextRecord, a structure whose declaration is made in bootx86.h (_BOOT_CONTEXT structure). Let’s go back to the InitializeMemorySubsystem in the file memory.c. In this file we also find a memory map (components images and related stacks / heaps) that can be useful. This function has immediatly a while that cycles for all MemoryDescriptor that are located in the BootContextRecord. Each memory descriptor, infact, is a structure with two fields: BlockBase & BlockSize. They describe the starting address of a memory area and its size. So

```
BootContext->MemoryDescriptorList
```

is an array of memory descriptors that are used to describe all the memory blocks that are needed. Remember that now we are in protected mode but with no paging, so at this moment every address we use corresponds to a physical address. So this “while” prepares memory addresses (respecting the page boundary) for all known memory blocks. The loader does not use memory that is above 16mb (to avoid interference with isa bus data transfers), so all the memory above 16mb is marked as MemoryFirmwareTemporary. Once the code exits the while, all physical memory has been described (with MempAllocDescriptor & MempSetDescriptorRegion functions), the descriptor array is maintained in the variable MDArray[] defined in arcemul.c. These are the “macro” descriptors that makes an approssimative description of physical memory, so after the while there is portion of code that handles the description of the first megabyte of memory. In fact, here there are all the memory components useful to the loader, such as the interrupt vector area, system heaps etc. Note that the first virtual memory megabyte will coincide with the first physical memory megabyte, to permit the os loader to continue the execution below first mega and map the kernel dedicated memory. Again with MempAllocDescriptor we can see that from the initial MDArray, some subdescriptors are obtained for the memory areas below a mega. When all the creation of this descriptor is finished we finally reach the MempTurnOnPaging. This function just makes a walk of the MDArray so it can call MempSetupPaging function, with which the PDE\PTE entries are created for all the necessary memory that was just calculated. The global variables are PDE for the PDE, and HalPT for the PTE. Once the memory descriptor walk is made, the PDE\PTE is correctly set, so the MempTurnOnPaging is called:

```
        mov eax,PDE
        mov cr3,eax

        mov eax,cr0
        or  eax,CR0_PG
        mov cr0,eax
```

and the paging is enabled. This is the first time paging is enabled since the system was booted. As you can see, the ptr to the PDE array is placed in the page directory base register (CR3), son in CR0 is enabled the flag relative to the paging. After the paging is enabled we come back to the InitializeMemorySubsystem function, that calls MempCopyGdt to move GDT and IDT in a new memory area. Ok now the function is finished and we can go back to DoGlobalInitialization. We see that there is other stuff and finally the call to InitializeMemoryDescriptors function, that as we can see from the comments is the second step of the InitializeMemorySubsystem. First the PDE\PTE were created to turn on paging, now this function comes back to the MDArray and allocates the memory for all descriptors that marked the memory as “reserved”. Now we’ve finished the DoGlobalInitialization. We head back to the NtProcessStartup. We have some other initialization functions, that find the partition from where we booted, initialize the system memory and I/O system, then we arrive at the call to BlStartup. Immediatly after there is this code:

```
    // we should never get here!
    do {
        GET_KEY();
    } while ( 1 );
```

So this means that ntldr work ends inside the BlStartup function, that is located in the file initx86.c in the directory

```
 \ntos\boot\bldr\i386
```

What does this function do? It takes care of opening the drive and reading the boot.ini file, where all bootable entries are defined. Such entries are shown with the classic choose menu, so once the boot entry is chosen the os determines disk/partition/path to boot, and then arrives at the function BlOsLoader which is located in the file

```
 \ntos\boot\bldr\osloader.c
```

We can see just before this function the definition of the names "ntoskrnl.exe" and "hal.dll", these will be the components that will be loaded. As you can see the code is well commented, so it’s easy to understand what happens: it opens the boot and system partitions, it opens the input/output console, it initializes the memory with BlMemoryInitialize, present in the directory

\ntos\boot\lib

in the file blmemory.c, where we find intialized, the stack, the heap and the memory allocation list. In this case the unique memory descriptor that has been allocated is the one relative to the os loader, in fact no other programs have been loaded. So the function will search the first memory descriptor below the os loader, and will allocate the heap at the highest possible address, allocate the space for the loader parameter block, then the loader stack, and finally the loader heap. After the memory init we see other initializations (i/o and resource section). So we see that there is the handling of the boot parameters that were specified in the boot.ini, in fact the parameters /KERNEL= /HAL= are handled, that permit to load a kernel/hal different from the default ones. So now the paths of the kernel and hal components and of the system hive are generated. The first one to be loaded is ntoskernel.exe with the function BlLoadImage that maps its image in memory. So the loader determines the type of fs used and keeps eventual arguments to pass to the kernel. Now it is the turn of hal.dll, which is loaded with BlLoadImage as well. Remeber that these two modules are PE coffs, in fact the loader function is in

\ntos\boot\lib\peldr.c

It is not the complete pe loader, in fact immediately after in the code is used the function BlScanImportDescriptorTable: with this the imported dll are loaded and the import/export are bound. So now the kernel, hal and all relative modules are loaded. It’s now time to load the drivers. To load the drivers the osloader must consult the system hive. What is this hive? Hives are the file that contains the information stored in the registry. In particular now the

\windows\config\system

hive is used. It contains all hardware settings and realtive drivers. Once the drivers are loaded, the BlSetupForNt is called: It makes some hardware initialization, such as abios relocation, tss relocation, etc. Finally the loader has finished its role, and we arrive at the line:

```
(SystemEntry)(BlLoaderBlock);
```

This line calls the entry point of the ntoskrnl module. From now on we can pass to the win2k sources (we still were in the nt4 ones!).

The kernel entry point is located in the file:

```
 \win2k\private\ntos\ke\i386\newsysbg.asm
```

and the function is KiSystemStartup. It takes as a argument the loader block mentioned some lines above.

This first part ends here. We have seen the initial portion now, the boot process. We arrived at the kernel, so in the next part we will describe kernel initialization and other more interesting stuff.

See ya in next chapter!

AndreaGeddon