J Comput Virol (2009) 5:105–117
DOI 10.1007/s11416-008-0112-2

ORIGINAL PAPER

A little journey inside Windows memory

Damien Aumaitre

Received: 3 September 2008 / Accepted: 6 December 2008 / Published online: 10 January 2009

Abstract In 2005 and 2006, two security researchers, Maximilian Dornseif and Adam Boileau, showed an offensive use of the FireWire bus. They demonstrated how to take control of a computer equipped with a FireWire port. This work has been continued. After a brief summary of how memory works on modern OS, we will explain how the FireWire bus works, and it can be used to access physical memory. Since modern operating system and processors use virtual addresses (and not physical ones), we rebuild the virtual space of each process in order to retrieve and understand kernel structures. Thus, we now have an instant view of the operating system without being submitted to the security protections provided by the processor or the kernel. We will demonstrate several uses for this. First we will show what can be done only with an interpretation of kernel structures (read access). For example, we can have the list of all processes, access to the registry with no control even for protected keys like the SAM ones. This is used to dump credentials. Then, we see what can be done when one modifies the memory (write access). As an example, we show a 2 byte patch to unlock a workstation without knowing the password. Last but not least, code execution is not supposed to happen through FireWire since it is only a bus providing read/write access to the memory. However, slightly modifying the running kernel lets us do whatever we want. We will explain how to have a shell with SYSTEM privileges before any authentication.

1 Memory basics

Virtual memory is a key concept for modern operating systems. Instead of letting software deal with physical memory, the processor and the operating system create an invisible layer between the software and the physical memory. Whenever the processor need to address memory, it consults a table called "page table" that tells the processor which physical memory address to actually use. The memory is divided into pages which are fixed-size chunks of memory. The size of a page of memory differs between processor architectures. On x86 processors, 4K pages are generally used but these processors also support 2 and 4 MB pages. The mechanism to convert a virtual address is slightly different for each page size but it always uses several tables as we can see on the Fig. 1.

Using page tables has several advantages. The first and most important one is that it enables the creation of multiple address spaces. An address space is an isolated page table that only allows access to memory useful for a process. It ensures that every process is completely isolated from each other. Another advantage is that it is very easy to tell the processor which rules are to be enforced on memory access. For example, page-table entries have a set of flags that determine several properties regarding this specific entry (read/write access, supervisor/user access, etc.). Thus a simple flag ensures that userland processes cannot access kernel data. That's how kernelspace is separated from userspace.

As we can see on the Fig. 2, physical memory is like a jigsaw puzzle. We can convert a virtual address into a physical one but there is no straightforward way to convert a physical address into a virtual one. Furthermore several adjoining virtual pages are not necessarily adjoining in physical memory. However two things are to be noted:

– kernel space is identically mapped for each process since it is shared by each process;
– different virtual pages can be mapped to the same physical page.

D. Aumaitre (✉)
Sogeti/ESEC, Paris, France
e-mail: damien.aumaitre@sogeti.com; damien@security-labs.org

Fig. 1 Linear translation

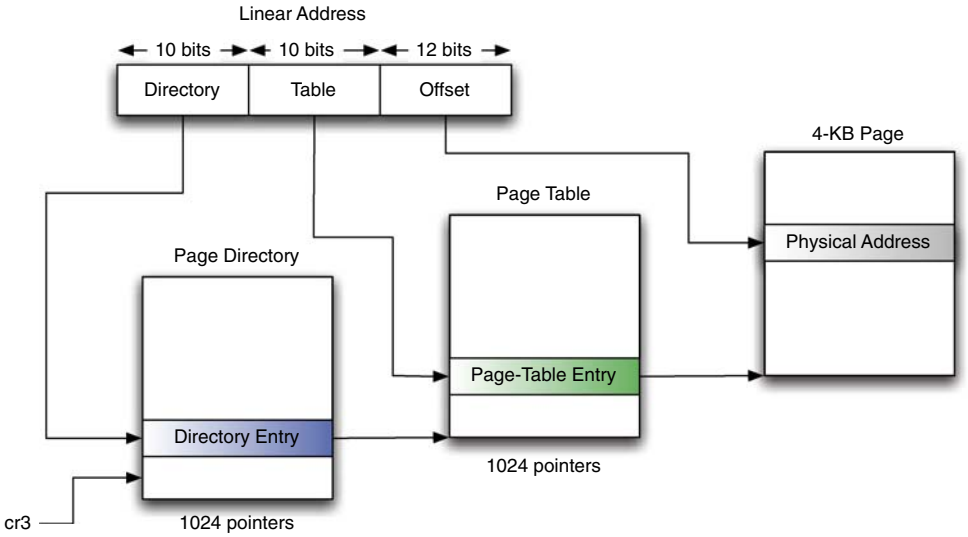
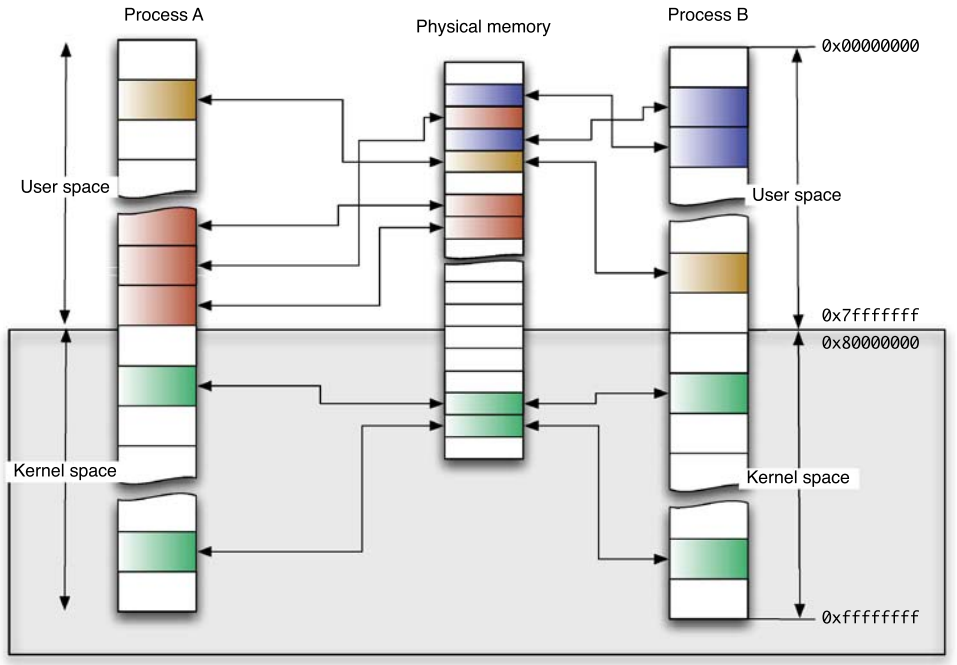


Fig. 2 Processes isolation



2 Accessing the physical memory

There are several methods for accessing physical memory. We could use memory dumps obtained with forensics tools, the FireWire bus, VMware files, hibernate files, even memory dumps obtained with cold boot attacks.

The main advantage is that we have an unbiased view of the data structures of the operating system. Because we only interpret data, we also short-circuit all security measures enforced by the processor. Thus we have a privileged position to access and modify key structures of the operating system.

Before we go deep inside the kernel, we will view some details about the use of the FireWire bus to read/write arbitrary locations in memory.

2.1 FireWire basics

Apple has developed FireWire in the late 80s. It was standardized by the IEEE in 1995. In 2000, Sun, Apple, Microsoft, Compaq, Intel, National Semiconductors and Texas Instruments wrote the OHCI 1394 (Open Host Controller Interface) specification.

A FireWire device can read (and write) to a computer's main memory by accessing a system's DMA controller, while the operating system, be it Windows, Mac OS, Linux, etc., is oblivious to the event. By pulling a copy of memory through FireWire, the target CPU and operating system are bypassed as are any infections, triggers or traps. This is not a bug but exactly how DMA and PCI devices, like FireWire, were designed to operate.

DMA allows memory transfers between devices and processes to take place while a computer's CPU performs other tasks.

1. the CPU/operating system programs the DMA controller to instruct a FireWire device to read a portion of memory; the CPU/operating system is now free to work on other tasks;
2. the DMA controller sends a message to the FireWire controller, informing it of the read request and the location and length in memory;
3. the FireWire device negotiates control of the PCI bus and reads the memory location specified…;
4. …and once completed, it informs the DMA controller;
5. finally, the DMA controller triggers an interrupt, informing the CPU the read operation is complete.

It should be noted that devices are not limited only to reading/writing to the memory address specified by the operating system. FireWire and other DMA bus master devices act independently of the CPU; the CPU need not initiate the transaction. A FireWire device can program the DMA controller and set up its own reads and writes, as per the PCI and IEEE 1394 specifications.

2.2 Abusing FireWire

In 2005, Maximilian Dornseif [5] explained how we can use the FireWire bus to read or write to arbitrary locations in physical memory. He uses an iPod to subvert a laptop running Mac OSX. In 2006, Adam Boileau continues Dornseif's work and explains how we can trick Windows in order to access physical memory.

OHCI specifies "AsynchronousRequestFilter" and "PhysicalRequestFilter" CSRs (CSR stands for Control and Status Register); if these CSRs are set to zero, the FireWire chipset will reject requests to access host physical memory. According to the specification, they default to zero. Windows does not set them. So, by default, Windows disallows FireWire DMA.

Since FireWire bus is supported by PCI bus, CSRs are mapped in memory. On our laptop, they are mapped from 0xf0500000. We can see on the Fig. 3 the offsets for the registers responsible for gaining access to physical memory.

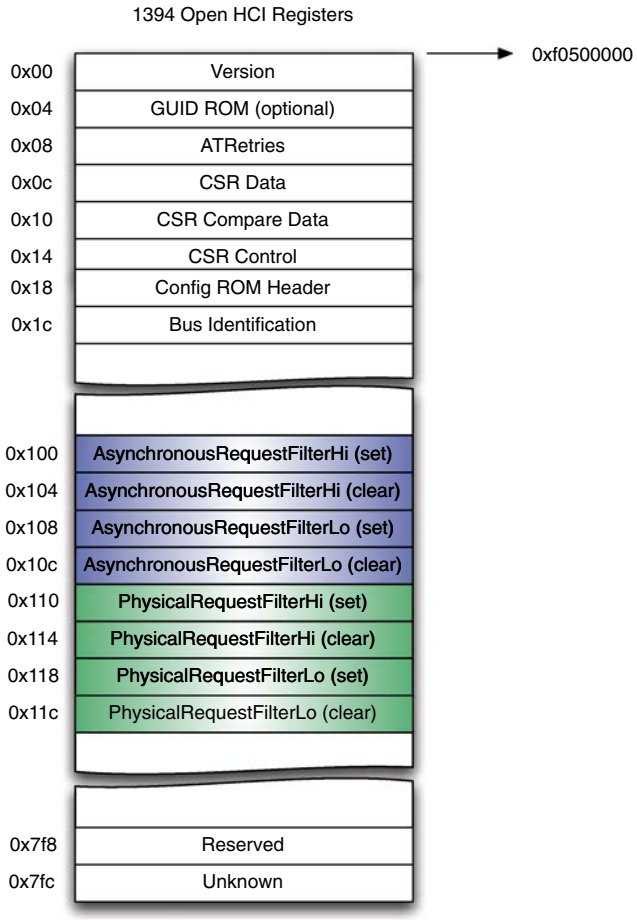


Fig. 3 OHCI registers

Adam Boileau has shown how we can create a phony config ROM that, when read by the Windows host, will open up the filter CSRs. This attack basically advertises his machine as requiring DMA access, and Windows complies.

For more information, the reader can go on Boileau's website which explains in detail how everything works [2].

3 Reconstruction of virtual space

We have seen previously the mechanisms deployed by the processor and the kernel to translate virtual addresses. But no mechanism directly exists to translate physical addresses into virtual addresses. Thus we have a find a way to do so.

As we have seen, the main advantage of pagination is the isolation of the processes. To address its virtual space, we need the value of the page table. Luckily the operating system keeps a backup value inside the kernel structures responsible for handling processes.

Andreas Schuster has proposed a method for searching for processes and threads in memory dumps [8].

Listing. 1.1 struct _DISPATCHER_HEADER

```
typedef struct _DISPATCHER_HEADER    // 6 elements, 0x10 bytes (sizeof)
{
/*0x000*/     UINT8         Type;
/*0x001*/     UINT8         Absolute;
/*0x002*/     UINT8         Size;
/*0x003*/     UINT8         Inserted;
/*0x004*/     LONG32        SignalState;
/*0x008*/     struct _LIST_ENTRY WaitListHead;
}DISPATCHER_HEADER, *PDISPATCHER_HEADER;
```

```
typedef struct _KPROCESS // 29 elements, 0x6C bytes (sizeof)
{
/*0x000*/     struct _DISPATCHER_HEADER Header;
/*0x010*/     struct _LIST_ENTRY ProfileListHead;
/*0x018*/     ULONG32       DirectoryTableBase[2];
/*0x020*/     struct _KGDTENTRY LdtDescriptor;
/*0x028*/     struct _KIDTENTRY Int21Descriptor;
/*0x030*/     UINT16        IopmOffset;
/*0x032*/     UINT8         Iopl;
/*0x033*/     UINT8         Unused;
/*0x034*/     ULONG32       ActiveProcessors;
/*0x038*/     ULONG32       KernelTime;
/*0x03C*/     ULONG32       UserTime;
/*0x040*/     struct _LIST_ENTRY ReadyListHead;
/*0x048*/     struct _SINGLE_LIST_ENTRY SwapListEntry;
/*0x04C*/     VOID*         VdmTrapcHandler;
/*0x050*/     struct _LIST_ENTRY ThreadListHead;
/*0x058*/     ULONG32       ProcessLock;
/*0x05C*/     ULONG32       Affinity;
/*0x060*/     UINT16        StackCount;
/*0x062*/     CHAR          BasePriority;
/*0x063*/     CHAR          ThreadQuantum;
/*0x064*/     UINT8         AutoAlignment;
/*0x065*/     UINT8         State;
/*0x066*/     UINT8         ThreadSeed;
/*0x067*/     UINT8         DisableBoost;
/*0x068*/     UINT8         PowerState;
/*0x069*/     UINT8         DisableQuantum;
/*0x06A*/     UINT8         IdealNode;
            union
            {
/*0x06B*/         struct _KEXECUTE_OPTIONS Flags;
/*0x06B*/         UINT8         ExecuteOptions;
            };
}KPROCESS, *PKPROCESS;
```

Listing. 1.2 struct _KPROCESS

```
typedef struct _EPROCESS  // 107 elements, 0x260 bytes (sizeof)
{
/*0x000*/     struct _KPROCESS Pcb;   // 29 elements, 0x6C bytes
/*0x06C*/     struct _EX_PUSH_LOCK ProcessLock;
/*0x070*/     union _LARGE_INTEGER CreateTime;
/*0x078*/     union _LARGE_INTEGER ExitTime;
/*0x080*/     struct _EX_RUNDOWN_REF RundownProtect;
/*0x084*/     VOID*         UniqueProcessId;
/*0x088*/     struct _LIST_ENTRY ActiveProcessLinks;

[...]

/*0x0C8*/     struct _EX_FAST_REF Token;
/*0x0CC*/     struct _FAST_MUTEX WorkingSetLock;
/*0x0EC*/     ULONG32       WorkingSetPage;
/*0x0F0*/     struct _FAST_MUTEX AddressCreationLock;
/*0x110*/     ULONG32       HyperSpaceLock;
/*0x114*/     struct _ETHREAD* ForkInProgress;
/*0x118*/     ULONG32       HardwareTrigger;
/*0x11C*/     VOID*         VadRoot;
/*0x120*/     VOID*         VadHint;

[...]

/*0x174*/     UINT8         ImageFileName[16];
/*0x184*/     struct _LIST_ENTRY JobLinks;
/*0x18C*/     VOID*         LockedPagesList;
/*0x190*/     struct _LIST_ENTRY ThreadListHead;

[.. .]

/*0x25C*/     UINT8         _PADDING1_[0x4];
}EPROCESS, *PEPROCESS;
```

Listing. 1.3 struct _EPROCESS

Processes are synchronizable objects, therefore they share a common substructure, the _DISPATCHER_HEADER structure (see Listing 1.1).

This header contains some constants which will help to find it. It contains a `Type` field and a `Size` field which have hard-coded values for a given version of Windows. For example, on Windows XP the `Type` field is `0x03` and the `Size` field is `0x1b`.

This is combined with additional tests in order to validate the candidate _EPROCESS structure. The first field of the _EPROCESS structure is a _KPROCESS structure that contains a _DISPATCHER_HEADER (see Listing 1.2).

The `cr3` value is found in the field `DirectoryTableBase`. Furthermore the `ThreadListHead` field must be located in kernel space. As we can see on the Listing 1.3, the _EPROCESS structure has several fields also situated in kernel space. Combining these tests allow to find all _EPROCESS structures residing in physical memory.

So we only need to find one of the _DISPATCHER_HEADER structures by pattern-matching. Then we check that it is the beginning of a _EPROCESS structure. Once we have it, we have the page table value that permits the mapping of this particular process virtual space.

Now we want to map the other processes virtual space. The kernel virtual space is identically mapped for each process. All _EPROCESS structures belong to a doubly linked list. This list is accessible with the `ActiveProcessLinks` field. By following this list, we find the other _EPROCESS structures and we can map their virtual space too.

The Fig. 4 summarizes all these steps.

4 Examples

Firstly we accessed memory through the FireWire bus. Secondly this memory was reconstructed in order to get a full view of the operating system. Now we are able to show several examples:

1. Read access to clone Process Explorer and Regedit with no restriction.
2. Write access to login without password.
3. Tricks with the kernel structures in order to execute arbitrary code.

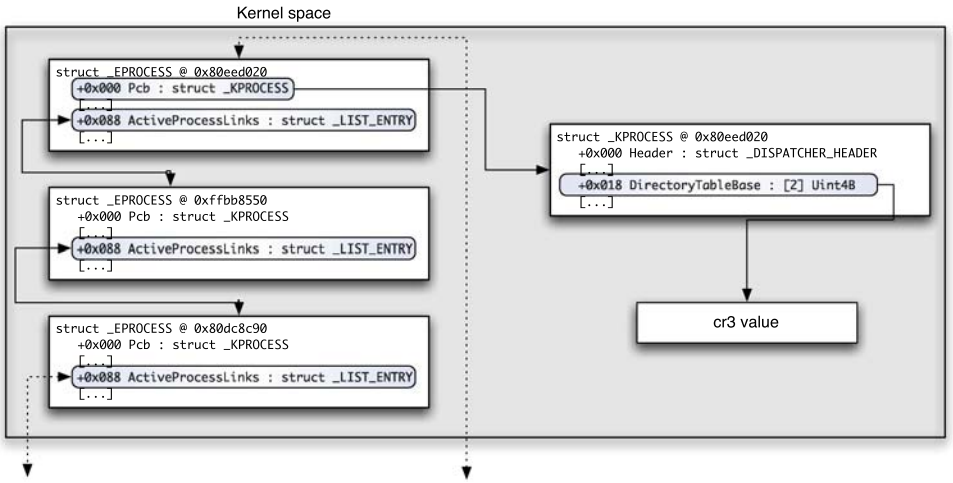


Fig. 4 Following the `_EPROCESS` list

4.1 Processes

In this section we will show how we can find all structures involved in a process. We will rebuild a kind of Process Explorer based on kernel structures. We can use this during a forensic analysis of a memory dump or during a malware analysis. Since we only interpret data, we short-circuit all security measures.

On Windows, processes are just a container for the information needed by the executive and the dispatcher. Thus we find all necessary information in the `_EPROCESS` structure. The `ThreadListHead` field points to the doubly linked list of `_ETHREAD` structures. We can enumerate all the threads by following this list.

The `_EPROCESS` structure contains also an interesting structure named `_PEB`. All userland information resides in this structure. In particular, we can find the list of the DLLs loaded by the process. First we follow the field `Ldr` in the `_PEB` structure (Listing 1.4).

This field is a `_PEB_LDR_DATA` structure (Listing 1.5).

In this structure, three doubly linked lists (`InLoadOrderLinks`, `InMemoryOrderLinks`, `InInitializationOrderLinks`) point to the DLLs list (Listing 1.6).

Thus we have all necessary information to examine and dump any processes. In order to improve our Process Explorer we need now to find how handles are stored. The `_EPROCESS` structure has a field named `HandleTable` which points to a `_HANDLE_TABLE` structure (Listing 1.7).

The `TableCode` field points to a table of `_HANDLE_TABLE_ENTRY` structures (Listing 1.8).

Depending on the number of handles, the `TableCode` field can point on a table of pointers instead of `_HANDLE_TABLE_ENTRY` structures.

In order to retrieve the object hidden behind the handle, we need to apply a mask on the `Object` field because the kernel uses some bits to store additional information about the object. So the real object address is obtained with the following formula: `real_addr = (object_addr |0x80000000) & 0xfffffff8`.

```
typedef struct _PEB     // 65 elements, 0x210 bytes (sizeof)
{
/*0x000*/     UINT8          InheritedAddressSpace;
/*0x001*/     UINT8          ReadImageFileExecOptions;
/*0x002*/     UINT8          BeingDebugged;
/*0x003*/     UINT8          SpareBool;
/*0x004*/     VOID*          Mutant;
/*0x008*/     VOID*          ImageBaseAddress;
/*0x00C*/     struct _PEB_LDR_DATA* Ldr;

[...]

}PEB, *PPEB;
```

Listing. 1.4 struct `_PEB`

```
typedef struct _PEB_LDR_DATA    // 7 elements, 0x28 bytes (sizeof)
{
/*0x000*/     ULONG32        Length;
/*0x004*/     UINT8          Initialized;
/*0x005*/     UINT8          _PADDING0_[0x3];
/*0x008*/     VOID*          SsHandle;
/*0x00C*/     struct _LIST_ENTRY InLoadOrderModuleList;
/*0x014*/     struct _LIST_ENTRY InMemoryOrderModuleList;
/*0x01C*/     struct _LIST_ENTRY InInitializationOrderModuleList;
/*0x024*/     VOID*          EntryInProgress;
}PEB_LDR_DATA, *PPEB_LDR_DATA;
```

Listing. 1.5 struct `_PEB_LDR_DATA`

On the Fig. 5, we can see all the interactions between these structures.

In summary we have all the necessary information contained in the `_EPROCESS` structure as we can see on the Fig. 6.

4.2 GDT, IDT and SSDT

The `PCR` (Processor Control Region) is used by the kernel and the HAL in order to contain specific hardware data. There is one PCR per processor. The kernel uses a structure called `_KPCR` (Listing 1.9) to store these pieces of information.

It contains several interesting fields and in particular a pointer (`PrcbData`) to a structure named `_KPCRB` which contains the processor control block. With this structure we

Listing. 1.6 struct _LDR_DATA_TABLE_ENTRY

```
typedef struct _LDR_DATA_TABLE_ENTRY  // 18 elements, 0x50 bytes (sizeof)
{
/*0x000*/     struct _LIST_ENTRY InLoadOrderLinks;
/*0x008*/     struct _LIST_ENTRY InMemoryOrderLinks;
/*0x010*/     struct _LIST_ENTRY InInitializationOrderLinks;
/*0x018*/     VOID*        DllBase;
/*0x01C*/     VOID*        EntryPoint;
/*0x020*/     ULONG32      SizeOfImage;
/*0x024*/     struct _UNICODE_STRING FullDllName;
/*0x02C*/     struct _UNICODE_STRING BaseDllName;

[...]

}LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

Listing. 1.7 struct _HANDLE_TABLE

```
typedef struct _HANDLE_TABLE   // 14 elements, 0x44 bytes (sizeof)
{
/*0x000*/     ULONG32      TableCode;
/*0x004*/     struct _EPROCESS* QuotaProcess;
/*0x008*/     VOID*        UniqueProcessId;
/*0x00C*/     struct _EX_PUSH_LOCK HandleTableLock[4];
/*0x01C*/     struct _LIST_ENTRY HandleTableList;
/*0x024*/     struct _EX_PUSH_LOCK HandleContentionEvent;
/*0x028*/     struct _HANDLE_TRACE_DEBUG_INFO* DebugInfo;
/*0x02C*/     LONG32       ExtraInfoPages;
/*0x030*/     ULONG32      FirstFree;
/*0x034*/     ULONG32      LastFree;
/*0x038*/     ULONG32      NextHandleNeedingPool;
/*0x03C*/     LONG32       HandleCount;
              union
              {
/*0x040*/         ULONG32      Flags;
/*0x040*/         UINT8        StrictFIFO : 1; // 0 BitPosition
              };
}HANDLE_TABLE, *PHANDLE_TABLE;
```

Listing. 1.8 struct _HANDLE_TABLE_ENTRY

```
typedef struct _HANDLE_TABLE_ENTRY // 8 elements, 0x8 bytes (sizeof)
{
   union
   {
/*0x000*/         VOID*        Object;
/*0x000*/         ULONG32      ObAttributes;
/*0x000*/         struct _HANDLE_TABLE_ENTRY_INFO* InfoTable;
/*0x000*/         ULONG32      Value;
   };
   union
   {
/*0x004*/         ULONG32      GrantedAccess;
                  struct
                  {
/*0x004*/             UINT16       GrantedAccessIndex;
/*0x006*/             UINT16       CreatorBackTraceIndex;
                  };
/*0x004*/         LONG32       NextFreeTableEntry;
  };
}HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

find the value of the IDT and the GDT by following the `ProcessorState` field.

However, the most interesting thing is that we have a pointer to an undocumented structure which contains various non-exported kernel variables: the `KdVersionBlock` field. Two articles [1,6] on http://rootkit.com describe what kind of structures are involved.

`KdVersionBlock` points to a `_DBGKD_GET_VERSION64` structure (Listing 1.10).

With this structure we find the virtual address of the kernel and the head of the doubly linked list of the kernel modules (`PsLoadedModuleList`) which is the same kind of list as the lists used for the DLLs. The `DebuggerDataList` field points to a `_KDDEBUGGER_DATA64` structure. For the sake of brevity we only show its beginning.

These fields are used by the debugger but they are always filled. Thus we have a long list of non-exported kernel variables which give us precious information about the kernel.

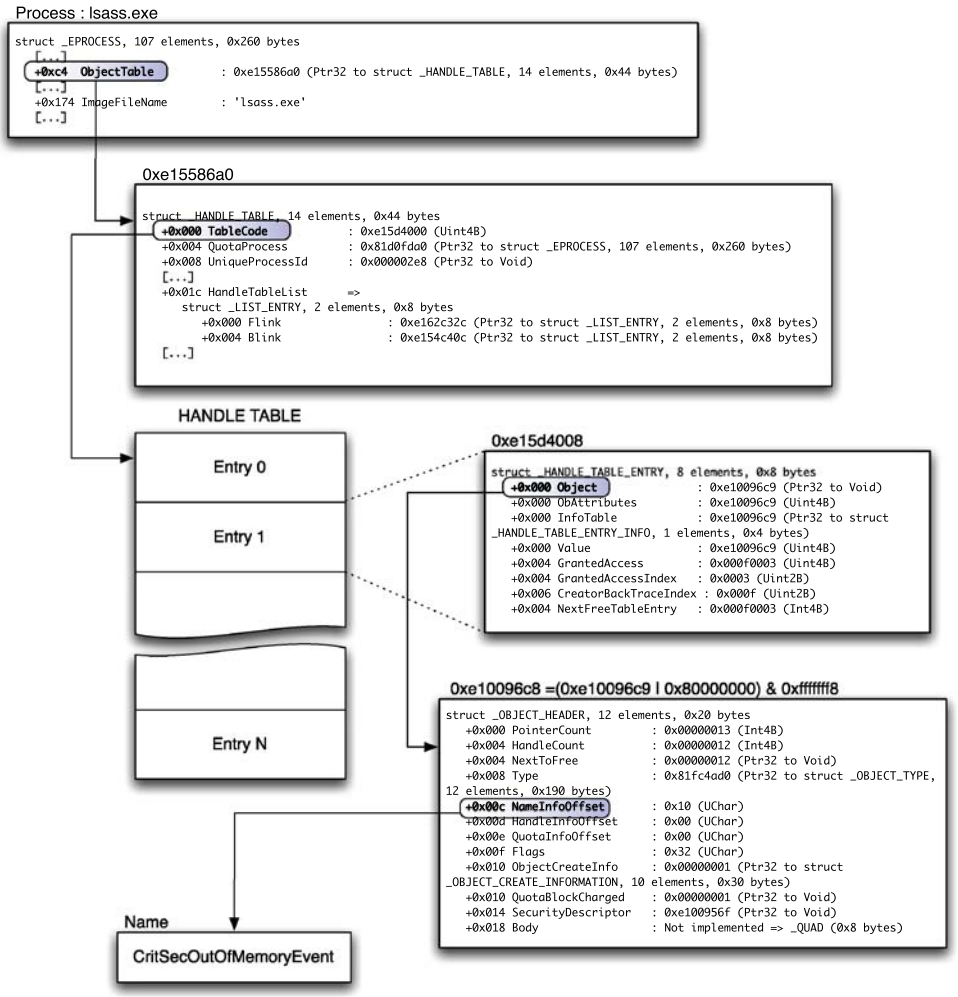


Fig. 5 Handle of lsass.exe

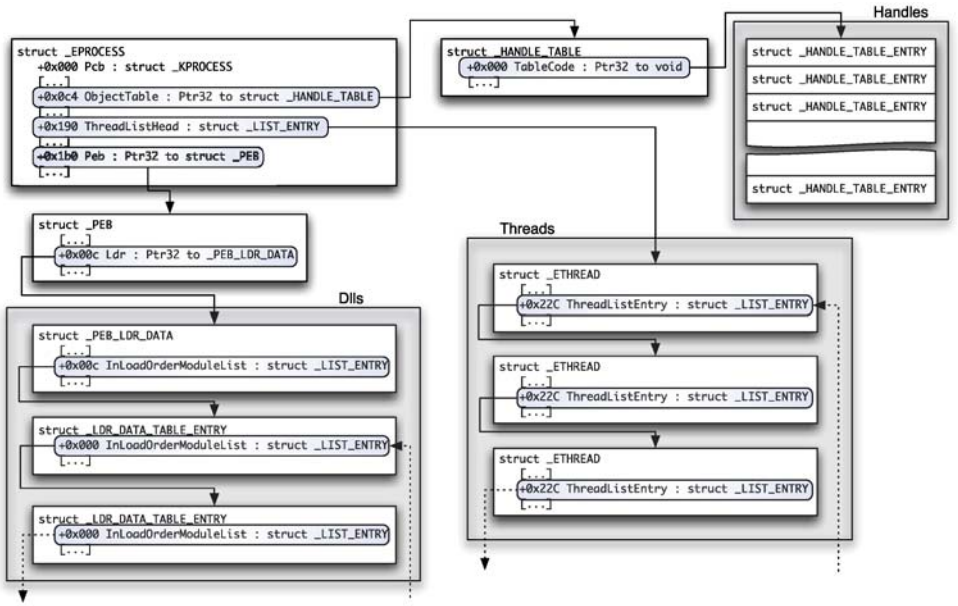


Fig. 6 Structures involved in a process

```
typedef struct _KPCR // 27 elements, 0xD70 bytes (sizeof)
{
/*0x000*/     struct _NT_TIB NtTib;
/*0x01C*/     struct _KPCR* SelfPcr;
/*0x020*/     struct _KPRCB* Prcb;
/*0x024*/     UINT8        Irql;
/*0x025*/     UINT8        _PADDING0_[0x3];
/*0x028*/     ULONG32      IRR;
/*0x02C*/     ULONG32      IrrActive;
/*0x030*/     ULONG32      IDR;
/*0x034*/     VOID*        KdVersionBlock;
/*0x038*/     struct _KIDTENTRY* IDT;
/*0x03C*/     struct _KGDTENTRY* GDT;
/*0x040*/     struct _KTSS* TSS;
/*0x044*/     UINT16       MajorVersion;
/*0x046*/     UINT16       MinorVersion;
/*0x048*/     ULONG32      SetMember;
/*0x04C*/     ULONG32      StallScaleFactor;
/*0x050*/     UINT8        DebugActive;
/*0x051*/     UINT8        Number;
/*0x052*/     UINT8        Spare0;
/*0x053*/     UINT8        SecondLevelCacheAssociativity;
/*0x054*/     ULONG32      VdmAlert;
/*0x058*/     ULONG32      KernelReserved[14];
/*0x090*/     ULONG32      SecondLevelCacheSize;
/*0x094*/     ULONG32      HalReserved[16];
/*0x0D4*/     ULONG32      InterruptMode;
/*0x0D8*/     UINT8        Spare1;
/*0x0D9*/     UINT8        _PADDING1_[0x3];
/*0x0DC*/     ULONG32      KernelReserved2[17];
/*0x120*/     struct _KPRCB PrcbData;
}KPCR, *PKPCR;
```

Listing. 1.9 struct _KPCR

So in order to find these, we have to find the _KPCR structure. Luckily for us, on Windows XP, it is always mapped to the 0xffdff000 virtual address. However, on Vista this is not true. Due to the particular form of this structure, we can use a signature plus some tests in order to find it by scanning the physical memory.

The _KPCR structure is self-referencing since the SelfPcr field (offset 0x1c contains its virtual address. Furthermore the Prcb field which contains the _KPCRB structure is located 0x120 bytes further. So we can find it with a simple algorithm.

Another interesting piece of information to gather is the system calls table. A system call takes place when user-mode code needs to call a kernel-mode function. Since the arrival of the Pentium II processors, Windows use the SYSTENTER instruction to manage the system calls. This instruction uses specific registers of the processor called MSR (Model Specific Register).

SYSENTER is essentially a high-performance kernel -mode switch instruction that calls into a predetermined function whose address is stored in a special MSR called SYSENTER_EIP_MSR. The implementation by the kernel is quite simple. If we examine what happen when we use CreateFile we observe the following behaviour.

```
ntdll!NtCreateFile:
7c91d682 b825000000  mov    eax,25h
7c91d687 ba0003fe7f      mov     edx,offset
  SharedUserData!SystemCallStub (7ffe0300)
7c91d68c ff12        call    dword ptr [edx]
7c91d68e c22c00      ret     2Ch
```

Listing. 1.10 struct _DBGKD_GET_VERSION64

```
typedef struct _DBGKD_GET_VERSION64 {
    UINT16  MajorVersion;
    UINT16  MinorVersion;
    UCHAR   ProtocolVersion;
    UCHAR   KdSecondaryVersion; // Cannot be 'A' for compat with dump header
    UINT16  Flags;
    UINT16  MachineType;

    // Protocol command support descriptions.
    // These allow the debugger to automatically
    // adapt to different levels of command support
    // in different kernels.

    // One beyond highest packet type understood, zero based.
    UCHAR   MaxPacketType;
    // One beyond highest state change understood, zero based.
    UCHAR   MaxStateChange;
    // One beyond highest state manipulate message understood, zero based.
    UCHAR   MaxManipulate;

    // Kind of execution environment the kernel is running in,
    // such as a real machine or a simulator.  Written back
    // by the simulation if one exists.
    UCHAR   Simulation;

    UINT16  Unused[1];

    ULONG64 KernBase;
    ULONG64 PsLoadedModuleList;

    // Components may register a debug data block for use by
    // debugger extensions.  This is the address of the list head.
    //
    // There will always be an entry for the debugger.

    ULONG64 DebuggerDataList;

} DBGKD_GET_VERSION64, *PDBGKD_GET_VERSION64;
```

Listing. 1.11 struct _KDDEBUGGER_DATA64

```
typedef struct _KDDEBUGGER_DATA64 {
    struct _DBGKD_DEBUG_DATA_HEADER64 Header;

    // Base address of kernel image
    ULONG64   KernBase;

    // DbgBreakPointWithStatus is a function which takes an argument
    // and hits a breakpoint.  This field contains the address of the
    // breakpoint instruction.  When the debugger sees a breakpoint
    // at this address, it may retrieve the argument from the first
    // argument register, or on x86 the eax register.
    ULONG64   BreakpointWithStatus;       // address of breakpoint

    // Address of the saved context record during a bugcheck
    //
    // N.B. This is an automatic in KeBugcheckEx's frame, and
    // is only valid after a bugcheck.
    ULONG64   SavedContext;

    // help for walking stacks with user callbacks:
    //
    // The address of the thread structure is provided in the
    // WAIT_STATE_CHANGE packet.  This is the offset from the base of
    // the thread structure to the pointer to the kernel stack frame
    // for the currently active usermode callback.
    UINT16  ThCallbackStack;            // offset in thread data

    // these values are offsets into that frame:
    UINT16  NextCallback;               // saved pointer to next callback
         frame
    UINT16  FramePointer;               // saved frame pointer

    // pad to a quad boundary
    UINT16  PaeEnabled;

    // Address of the kernel callout routine.
    ULONG64   KiCallUserMode;             // kernel routine

    // Address of the usermode entry point for callbacks.
    ULONG64   KeUserCallbackDispatcher;   // address in ntdll

[...]

} KDDEBUGGER_DATA64, *PKDDEBUGGER_DATA64;
```

Listing. 1.12 struct _SYSTEM_SERVICE_TABLE

```
typedef struct _SYSTEM_SERVICE_TABLE
{
     PNTPROC ServiceTable;              // array of entry points
     PDWORD CounterTable;               //  array of usage counters
     DWORD     ServiceLimit ;           //  number of table entries
     PBYTE     ArgumentTable;           //  array of byte counts
)
SYSTEM_SERVICE_TABLE
```

The number of the system call is stored in the `eax` register. Then a stub (which is common for every system call) is executed. This stub has the following instructions.

```
ntdll!KiFastSystemCall:
7c91eb8b 8bd4     mov     edx,esp
7c91eb8d 0f34      sysenter
```

The arguments list is stored in the `edx` register, then the `SYSENTER` instruction is executed.

Each thread has an array of system call tables. Its address is stored inside the `_KTHREAD` structure in the field `ServiceTable`. A thread can have up to 4 system call tables (Listing 1.12). Generally only 2 are used, one for the kernel and one for kernel-mode graphics subsystem (`win32k.sys`).

```
typedef struct _CM_KEY_BODY  // 7 elements, 0x44 bytes (sizeof)
{
/*0x000*/     ULONG32      Type;
/*0x004*/     struct _CM_KEY_CONTROL_BLOCK* KeyControlBlock;
/*0x008*/     struct _CM_NOTIFY_BLOCK* NotifyBlock;
/*0x00C*/     VOID*        ProcessID;
/*0x010*/     ULONG32      Callers;
/*0x014*/     VOID*        CallerAddress[10];
/*0x03C*/     struct _LIST_ENTRY KeyBodyList;
}CM_KEY_BODY, *PCM_KEY_BODY;
```

Listing. 1.13 struct _CM_KEY_BODY

4.3 Registry

Most parts of the registry reside in memory. Thus it will be very interesting to be able to reconstruct a sort of Regedit. We can bypass all security measures and manipulate some keys that we cannot under normal circumstances. For example we can use this to implement a Pwdump-like program.

Listing. 1.14 struct _CM_KEY_CONTROL_BLOCK

```
typedef struct _CM_KEY_CONTROL_BLOCK    // 25 elements, 0x48 bytes (sizeof)
{
/*0x000*/     UINT16       RefCount;
/*0x002*/     UINT16       Flags;
              struct
              {
/*0x004*/         ULONG32      ExtFlags : 8;
/*0x004*/         ULONG32      PrivateAlloc : 1;
/*0x004*/         ULONG32      Delete : 1;
/*0x004*/         ULONG32      DelayedCloseIndex : 12;
/*0x004*/         ULONG32      TotalLevels : 10;
              };
              union
              {
/*0x008*/         struct _CM_KEY_HASH KeyHash;
                  struct
                  {
/*0x008*/             ULONG32      ConvKey;
/*0x00C*/             struct _CM_KEY_HASH* NextHash;
/*0x010*/             struct _HHIVE* KeyHive;
/*0x014*/             ULONG32      KeyCell;
                  };
              };
/*0x018*/     struct _CM_KEY_CONTROL_BLOCK* ParentKcb;
/*0x01C*/     struct _CM_NAME_CONTROL_BLOCK* NameBlock;
/*0x020*/     struct _CM_KEY_SECURITY_CACHE* CachedSecurity;
/*0x024*/     struct _CACHED_CHILD_LIST ValueCache;
              union
              {
/*0x02C*/         struct _CM_INDEX_HINT_BLOCK* IndexHint;
/*0x02C*/         ULONG32      HashKey;
/*0x02C*/         ULONG32      SubKeyCount;
              };
              union
              {
/*0x030*/         struct _LIST_ENTRY KeyBodyListHead;
/*0x030*/         struct _LIST_ENTRY FreeListEntry;
              };
/*0x038*/     union _LARGE_INTEGER KcbLastWriteTime;
/*0x040*/     UINT16       KcbMaxNameLen;
/*0x042*/     UINT16       KcbMaxValueNameLen;
/*0x044*/     ULONG32      KcbMaxValueDataLen;
}CM_KEY_CONTROL_BLOCK, *PCM_KEY_CONTROL_BLOCK;
```

Before this, we will describe some structures manipulated by the configuration manager. The type manipulated by the object manager is `Key`. It is the type shown in the handles. Behind this type it is a `_CM_KEY_BODY` structure (Listing 1.13).

The `_CM_KEY_CONTROL_BLOCK` (Listing 1.14) structure contains much information.

Several fields are important. For example the fields `KeyHive` and `KeyCell` point to a `_HHIVE` structure and an index inside this hive. We will explain later how indexes are converted into addresses. The `NameControlBlock` field points to a structure that contains the name of the cell.

As we said before, cells are stored with indexes. The configuration manager converts them into addresses. The method is quite similar to the one used to convert virtual addresses.

The index is broken into several parts:

1. the 12 least significant bits are an offset in the page that contains the data;
2. the 9 following bits are an index inside a `_HMAP_TABLE` structure;
3. the 10 following bits are an index inside a `_HMAP_DIRECTORY` structure;

```
typedef struct _HHIVE // 24 elements, 0x210 bytes (sizeof)
{
/*0x000*/     ULONG32      Signature;

[.. .]

/*0x058*/     struct _DUAL Storage[2];
}HHIVE, *PHHIVE;
```

Listing. 1.15 struct _HHIVE

4. the most significant bit indicates if the cell is volatile or resident.

How can we find the appropriate `_HMAP_TABLE` and `_HMAP_DIRECTORY` structures?

We need to examine the `_HHIVE` structure (Listing 1.15).

This structure has a field named `Storage` which is a table of two `_DUAL` structures (Listing 1.16). One is for volatile storage, one is for resident storage. Volatile storage means that the keys are only present in memory.

The `Map` field points to a table of `_HMAP_DIRECTORY` which also points to a table of `_HMAP_TABLE` which points to a table of `_HMAP_ENTRY` structures (Listing 1.17).

The field `BlockAddress` contains the address of the memory area that contains the cell. By adding the offset found

Listing. 1.16 struct _DUAL

```
typedef struct _DUAL     // 7 elements, 0xDC bytes (sizeof)
        {
/*0x000*/      ULONG32       Length;
/*0x004*/      struct _HMAP_DIRECTORY* Map;
/*0x008*/      struct _HMAP_TABLE* SmallDir;
/*0x00C*/      ULONG32       Guard;
/*0x010*/      struct _RTL_BITMAP FreeDisplay[24];
/*0x0D0*/      ULONG32       FreeSummary;
/*0x0D4*/      struct _LIST_ENTRY FreeBins;
        }DUAL, *PDUAL;
```

Listing. 1.17 struct _HMAP_ENTRY

```
typedef struct _HMAP_ENTRY  // 4 elements, 0x10 bytes (sizeof)
{
/*0x000*/      ULONG32       BlockAddress;
/*0x004*/      ULONG32       BinAddress;
/*0x008*/      struct _CM_VIEW_OF_FILE* CmView;
/*0x00C*/      ULONG32       MemAlloc;
}HMAP_ENTRY, *PHMAP_ENTRY;
```

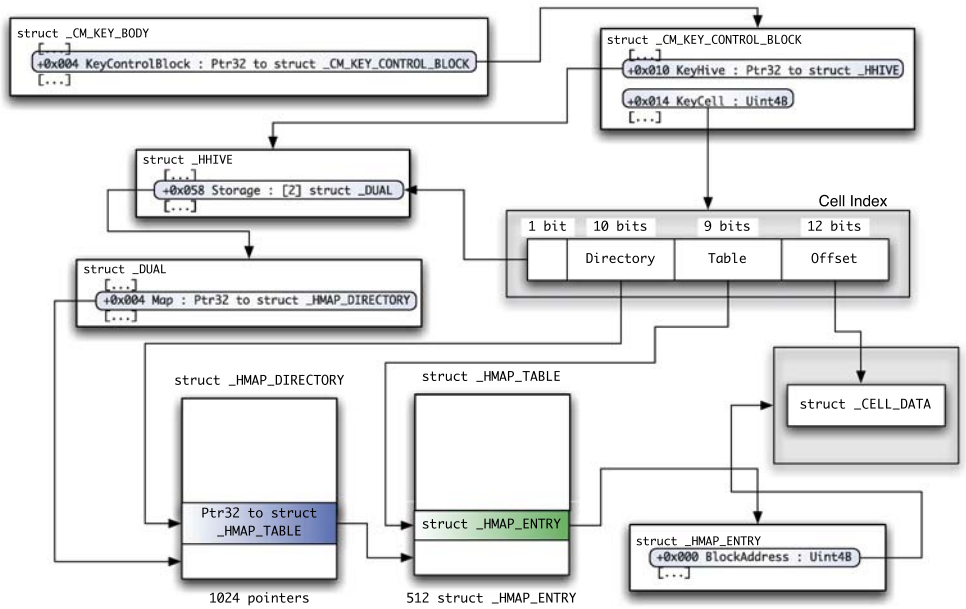


Fig. 7 Interactions between cell-related structures

with the index, we find the address of the cell. The Fig. 7 shows all the interactions between these structures.

Since we now have a clone of Regedit, we can begin to do interesting things! We can dump SAM hashes, LSA secrets, domains credentials, etc. Everything you can do with an unlimited access to the registry.

Since we could also write to the registry we could, for example, login without knowing the password.

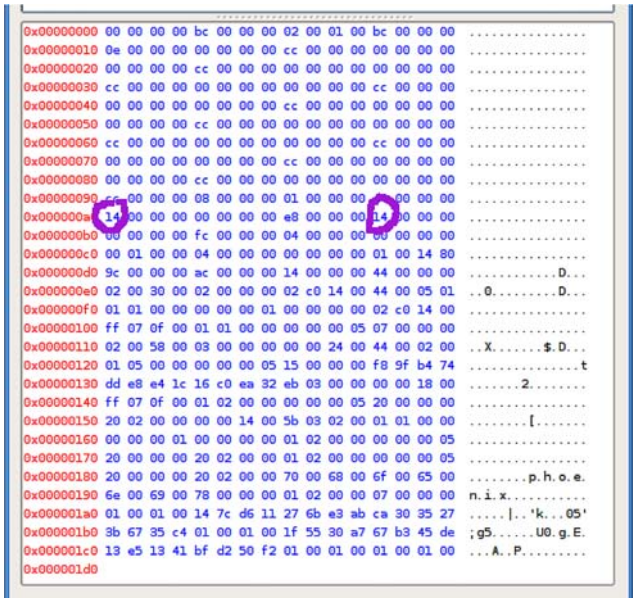
Adam Boileau released a tool, *winlockpwn*, that patches the function responsible for checking the credentials used during interactive login.

However, we can do that in a different way. The registry key holding details of the user account (`HKLM\SAM\Domains\Account\Users\[SID]\V`) contains 2 bytes at the offsets `0xa0` and `0xac` indicating whether passwords hashes are needed for this account [4]. For a password-protected account, these bytes are equal to `0x14` (Fig. 8). If we replace their values by `0x04` then no credentials are checked before log in since we request that no hashes are needed! As a result, we could log in with a simple keystroke.

4.4 Arbitrary code execution

It's clear now that we can read or write anything in physical memory. But how can we execute arbitrary code? The answer is simple, we just need to overwrite some pointers. We will show one way to do it but there are many ways.

We want to divert the flow of execution as quickly as we can. So we choose to hijack the system calls. A particular structure, called `_KUSER_SHARED_DATA` (Listing 1.18),



has been used extensively in Windows exploitation [3,7]. This structure has the particularity of being mapped in user-space and in kernelspace and at a fixed address. On top of that it contains various useful data.

In particular we have the `SystemCall` field which contains the address of the stub executed before any system calls. By hooking this address, we gain control before every system call.

By combining this with a few tests to determine if our payload is already executing and under which process we want to execute we have a reliable way to execute any userland code.

For a proof-of-concept we decided to spawn an admin shell before any authentication. To do so we just used a payload that uses `CreateProcess` with the `winlogon` desktop.

As we can see on the Fig. 9, the result is self-explanatory!

Fig. 8 Bytes involved when unlocking workstation

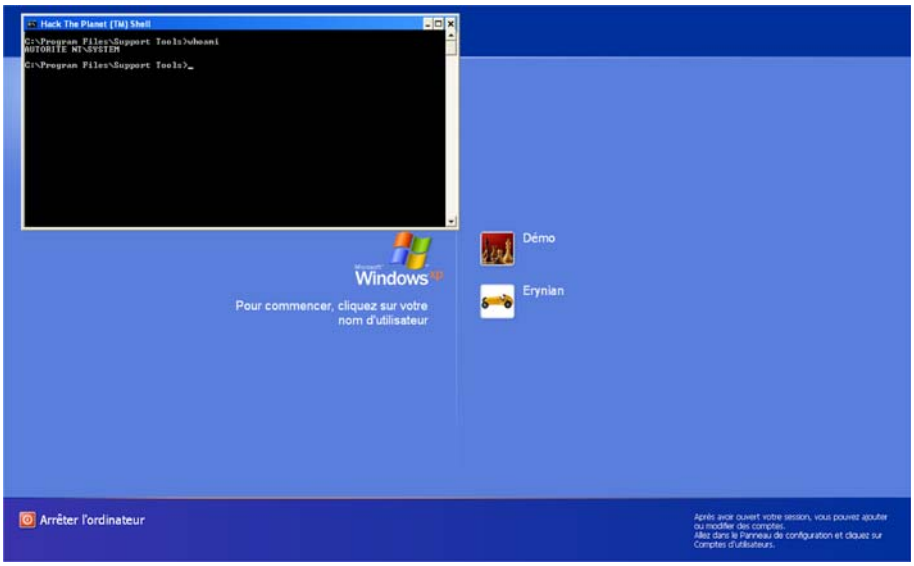
Listing. 1.18 struct _KUSER_SHARED_DATA

```
typedef struct _KUSER_SHARED_DATA  // 39 elements, 0x338 bytes (sizeof)
{
/*0x000*/     ULONG32       TickCountLow;
/*0x004*/     ULONG32       TickCountMultiplier;
/*0x008*/     struct _KSYSTEM_TIME InterruptTime;
/*0x014*/     struct _KSYSTEM_TIME SystemTime;
/*0x020*/     struct _KSYSTEM_TIME TimeZoneBias;

[...]

/*0x300*/     ULONG32       SystemCall;
/*0x304*/     ULONG32       SystemCallReturn;
/*0x308*/     UINT64        SystemCallPad[3];
              union
              {
/*0x320*/         struct _KSYSTEM_TIME TickCount;
/*0x320*/         UINT64       TickCountQuad;
              };
/*0x330*/     ULONG32       Cookie;
/*0x334*/     UINT8         _PADDING5_[0x4];
}KUSER_SHARED_DATA, *PKUSER_SHARED_DATA;
```

Fig. 9 Hack the planet: D



5 Conclusion

We have seen how memory works on modern operating systems. We also explained how FireWire can be used to access physical memory. We have combined these to show several examples both for defensive and offensive purposes. Techniques to reconstruct the virtual memory are mature enough to allow the implementation of user-friendly tools.

It is always been said that physical access to a machine is equivalent to owning it. This is even more true with a machine equipped with a FireWire or a PCMCIA port. For older machines, few things can be done. We can remove the support of the FireWire with the aid of the operating system. On the other hand with newer machines we could use the IOMMU (Input/Output Memory Management Unit). The IOMMU takes care of mapping device-visible virtual addresses (also called I/O addresses) to physical ones. Thus we can enforce some restrictions regarding the access of memory with a DMA-capable device.

References

1. Barbosa, E.: Find some Non-Exported Kernel Variables in Windows xp. http://www.rootkit.com/newsread.php?newsid=101
2. Boileau, A.: Hit by a Bus: Physical Access Attacks with Firewire (2006). http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf
3. Bugcheck, Skape: Kernel-Mode Payloads on Windows (2005). http://www.uninformed.org/?v=3&a=4&t=pdf
4. clark@hushmail.com.: Security Accounts Manager (2005). http://www.beginningtoseethelight.org/ntsecurity/index.php
5. Dornseif, M.: All Your Memory are Belong to Us (2005). http://md.hudora.de/presentations/firewire/2005-firewire-cansecwest.pdf
6. ionescu007.: Getting Kernel Variables from KdVersionBlock, Part2. http://www.rootkit.com/newsread.php?newsid=153
7. Jack, B.: Remote windows Kernel Exploitation Step into the Ring 0 (2005). http://research.eeye.com/html/papers/download/StepIntoTheRing.pdf
8. Schuster, A.: Searching for Processes and Threads in Microsoft Windows Memory Dumps (2006). http://dfrws.org/2006/proceedings/2-Schuster-pres.pdf