YOUR CLOUD IS IN MY POCKET

Matthieu Suiche, MoonSols SARL – http://www.moonsols.com
msuicheàmoonsols.com

INTRODUCTION

Physical memory is one of the key elements of any computer. As a volatile memory container, we can retrieve everything we are looking for. But physical memory snapshot are not new, especially on Windows. Microsoft introduced more than 10 years ago with the Blue Screen of the death – which was one of the most "stable" way to get a physical memory snapshot on Windows. Moreover, the Microsoft crash dump file format was designed to work with Microsoft Windows Debugger which is probably the most advanced analyze utility for memory dump.

Because of the several advantages this format provides, I decided to create a Toolkit able to convert any Windows memory dump into a Microsoft crash dump. Moreover, I also created a live acquisition utility which is able to produce a physical memory snapshot in two different formats.

- Linear memory mump
- Microsoft crash dump

All these utilities are in one toolkit called: MoonSols Windows Memory Toolkit.

MoonSols Windows Memory Toolkit is the ultimate toolkit for memory dump conversion and acquisition on Windows. This toolkit had been designed to deal with various types of memory dumps such as VMWare memory snapshot, Microsoft crash dump and even Windows hibernation file.

In addition to this, I later release "**MoonSols LiveCloudKd**" which aims at using directly Kd or WinDbg on the physical memory of a Virtual Machine. Firstly, LiveCloudKd was supporting only Microsoft Hyper-V – but now LiveCloudKd also supports VMWare Workstation.

This initiative encouraged Microsoft Technical Fellow, Mark Russinovich, to add Hyper-V support to Hyper-V. (http://blogs.technet.com/b/markrussinovich/archive/2010/10/14/3360991.aspx)

LIVECLOUDKD

Unlike LiveKd, MoonSols LiveKd also provides a write access. In other words, you can modify memory of the target virtual machine. One more important thing is that you don't need the debug mode to use LiveCloudKd to do bad things to the Virtual Machine Memory (VMM) – without have to pause or to stop the Virtual Machine (VM).

This is really interesting for various purposes, from Administration to Rootkit writing. This can be used for monitoring process and also for exchanging information between the host and the virtual machine without having a dedicated protocol.

We can imagine application for a new generation of Task Manager, or even a non-killable anti-virus running from out-side the virtual machine – but we can also imagine a new generation of Low Level Rootkit applied for commercialized Rootkits, a new kind of Rootkit able to modify the memory of its target and able to inject third party code without caring about code signing etc.

LiveCloudKd loads a "fake" Microsoft crash dump in Windows Debugger (Windbg) and Kernel Debugger (Kd).

MICROSOFT HYPER-V

User-land applications, mainly Virtual Machine worker processes, are using the VM Infrastructure Driver *(vid.sys)* to communicate with the Hypervisor to access to the memory devices like the physical memory. So by using the VID interface we can have the full control of any virtual machine.

You can reference to the header file *hvgdk.h* from the WDK to have a look at possible hypercalls. Here is the list.

```
//
// Declare the various hypercall operations.
//
typedef enum _HV_CALL_CODE
{
    //
    // Reserved Feature Code
    //

    HvCallReserved0000                  = 0x0000,

    //
    // V1 Address space enlightment IDs
    //

    HvCallSwitchVirtualAddressSpace     = 0x0001,
    HvCallFlushVirtualAddressSpace      = 0x0002,
    HvCallFlushVirtualAddressList       = 0x0003,

    //
    // V1 Power Management and Run time metrics IDs
    //

    HvCallGetLogicalProcessorRunTime    = 0x0004,
    HvCallDeprecated0005                = 0x0005,
    HvCallDeprecated0006                = 0x0006,
    HvCallDeprecated0007                = 0x0007,

    //
    // V1 Spinwait enlightenment IDs
    //

    HvCallNotifyLongSpinWait            = 0x0008,

    //
```

```
// V2 Core parking IDs
//

HvCallParkLogicalProcessors         = 0x0009,

//
// V2 Invoke Hypervisor debugger
//

HvCallInvokeHypervisorDebugger      = 0x000a,

//
// V1 enlightenment name space reservation.
//

HvCallReserved000b                  = 0x000b,
HvCallReserved000c                  = 0x000c,
HvCallReserved000d                  = 0x000d,
HvCallReserved000e                  = 0x000e,
HvCallReserved000f                  = 0x000f,
HvCallReserved0010                  = 0x0010,
HvCallReserved0011                  = 0x0011,
HvCallReserved0012                  = 0x0012,
HvCallReserved0013                  = 0x0013,
HvCallReserved0014                  = 0x0014,
HvCallReserved0015                  = 0x0015,
HvCallReserved0016                  = 0x0016,
HvCallReserved0017                  = 0x0017,
HvCallReserved0018                  = 0x0018,
HvCallReserved0019                  = 0x0019,
HvCallReserved001a                  = 0x001a,
HvCallReserved001b                  = 0x001b,
HvCallReserved001c                  = 0x001c,
HvCallReserved001d                  = 0x001d,
HvCallReserved001e                  = 0x001e,
HvCallReserved001f                  = 0x001f,
HvCallReserved0020                  = 0x0020,
HvCallReserved0021                  = 0x0021,
HvCallReserved0022                  = 0x0022,
HvCallReserved0023                  = 0x0023,
HvCallReserved0024                  = 0x0024,
HvCallReserved0025                  = 0x0025,
HvCallReserved0026                  = 0x0026,
HvCallReserved0027                  = 0x0027,
HvCallReserved0028                  = 0x0028,
HvCallReserved0029                  = 0x0029,
HvCallReserved002a                  = 0x002a,
HvCallReserved002b                  = 0x002b,
HvCallReserved002c                  = 0x002c,
HvCallReserved002d                  = 0x002d,
HvCallReserved002e                  = 0x002e,
HvCallReserved002f                  = 0x002f,
HvCallReserved0030                  = 0x0030,
HvCallReserved0031                  = 0x0031,
HvCallReserved0032                  = 0x0032,
HvCallReserved0033                  = 0x0033,
HvCallReserved0034                  = 0x0034,
HvCallReserved0035                  = 0x0035,
```

```
    HvCallReserved0036                  = 0x0036,
    HvCallReserved0037                  = 0x0037,
    HvCallReserved0038                  = 0x0038,
    HvCallReserved0039                  = 0x0039,
    HvCallReserved003a                  = 0x003a,
    HvCallReserved003b                  = 0x003b,
    HvCallReserved003c                  = 0x003c,
    HvCallReserved003d                  = 0x003d,
    HvCallReserved003e                  = 0x003e,
    HvCallReserved003f                  = 0x003f,

    //
    // V1 Partition Management IDs
    //

    HvCallCreatePartition               = 0x0040,
    HvCallInitializePartition           = 0x0041,
    HvCallFinalizePartition             = 0x0042,
    HvCallDeletePartition               = 0x0043,
    HvCallGetPartitionProperty          = 0x0044,
    HvCallSetPartitionProperty          = 0x0045,
    HvCallGetPartitionId                = 0x0046,
    HvCallGetNextChildPartition         = 0x0047,

    //
    // V1 Resource Management IDs
    //

    HvCallDepositMemory                 = 0x0048,
    HvCallWithdrawMemory                = 0x0049,
    HvCallGetMemoryBalance              = 0x004a,

    //
    // V1 Guest Physical Address Space Management IDs
    //

    HvCallMapGpaPages                   = 0x004b,
    HvCallUnmapGpaPages                 = 0x004c,

    //
    // V1 Intercept Management IDs
    //

    HvCallInstallIntercept              = 0x004d,

    //
    // V1 Virtual Processor Management IDs
    //

    HvCallCreateVp                      = 0x004e,
    HvCallDeleteVp                      = 0x004f,
    HvCallGetVpRegisters                = 0x0050,
    HvCallSetVpRegisters                = 0x0051,

    //
    // V1 Virtual TLB IDs
    //
```

```
HvCallTranslateVirtualAddress       = 0x0052,
HvCallReadGpa                       = 0x0053,
HvCallWriteGpa                      = 0x0054,

//
// V1 Interrupt Management IDs
//

HvCallAssertVirtualInterrupt        = 0x0055,
HvCallClearVirtualInterrupt         = 0x0056,

//
// V1 Port IDs
//

HvCallCreatePort                    = 0x0057,
HvCallDeletePort                    = 0x0058,
HvCallConnectPort                   = 0x0059,
HvCallGetPortProperty               = 0x005a,
HvCallDisconnectPort                = 0x005b,
HvCallPostMessage                   = 0x005c,
HvCallSignalEvent                   = 0x005d,

//
// V1 Partition State IDs
//

HvCallSavePartitionState            = 0x005e,
HvCallRestorePartitionState         = 0x005f,

//
// V1 Trace IDs
//

HvCallInitializeEventLogBufferGroup = 0x0060,
HvCallFinalizeEventLogBufferGroup   = 0x0061,
HvCallCreateEventLogBuffer          = 0x0062,
HvCallDeleteEventLogBuffer          = 0x0063,
HvCallMapEventLogBuffer             = 0x0064,
HvCallUnmapEventLogBuffer           = 0x0065,
HvCallSetEventLogGroupSources       = 0x0066,
HvCallReleaseEventLogBuffer         = 0x0067,
HvCallFlushEventLogBuffer           = 0x0068,

//
// V1 Dbg Call IDs
//

HvCallPostDebugData                 = 0x0069,
HvCallRetrieveDebugData             = 0x006a,
HvCallResetDebugSession             = 0x006b,

//
// V1 Stats IDs
//

HvCallMapStatsPage                  = 0x006c,
HvCallUnmapStatsPage                = 0x006d,
```

```
    //
    // V2 Guest Physical Address Space Management IDs
    //

    HvCallMapSparseGpaPages             = 0x006e,

    //
    // V2 Set System Property
    //

    HvCallSetSystemProperty             = 0x006f,

    //
    // V2 Port Ids.
    //

    HvCallSetPortProperty               = 0x0070,

    //
    // V2 Test IDs
    //

    HvCallOutputDebugCharacter,
    HvCallEchoIncrement,
    HvCallPerfNop,
    HvCallPerfNopInput,
    HvCallPerfNopOutput,

    //
    // Total of all hypercalls
    //
    HvCallCount

} HV_CALL_CODE, *PHV_CALL_CODE;
```

This is how we can gain access to the physical memory of a virtual machine.

VMWare Workstation

Regarding VMWare this is way easier, all we need is to use the .vmem file to have access to the physical memory, and the file name is the GUID of the running virtual machine in the same directory of the Virtual Machine.

LiveCloudKd

LiveCloudKd takes advantage of the Microsoft crash dump file format to be able to use Kd.exe (Kernel Debugger) or WinDbg (Windows Debugger). LiveCloudKd is a 100% pure user-land application, it's done by hooking the Debugger Engine Library (dbgeng.dll)'s Import Address Table (IAT) to avoid to directly create a mapping of a virtual file or hooking syscalls.

List of hooked functions are:

- **CreateFile**

- **CreateFileMapping**
- **MapViewOfFile**
- **UnMapViewOfFile**
- **GetFileSize**
- **VirtualProtect**

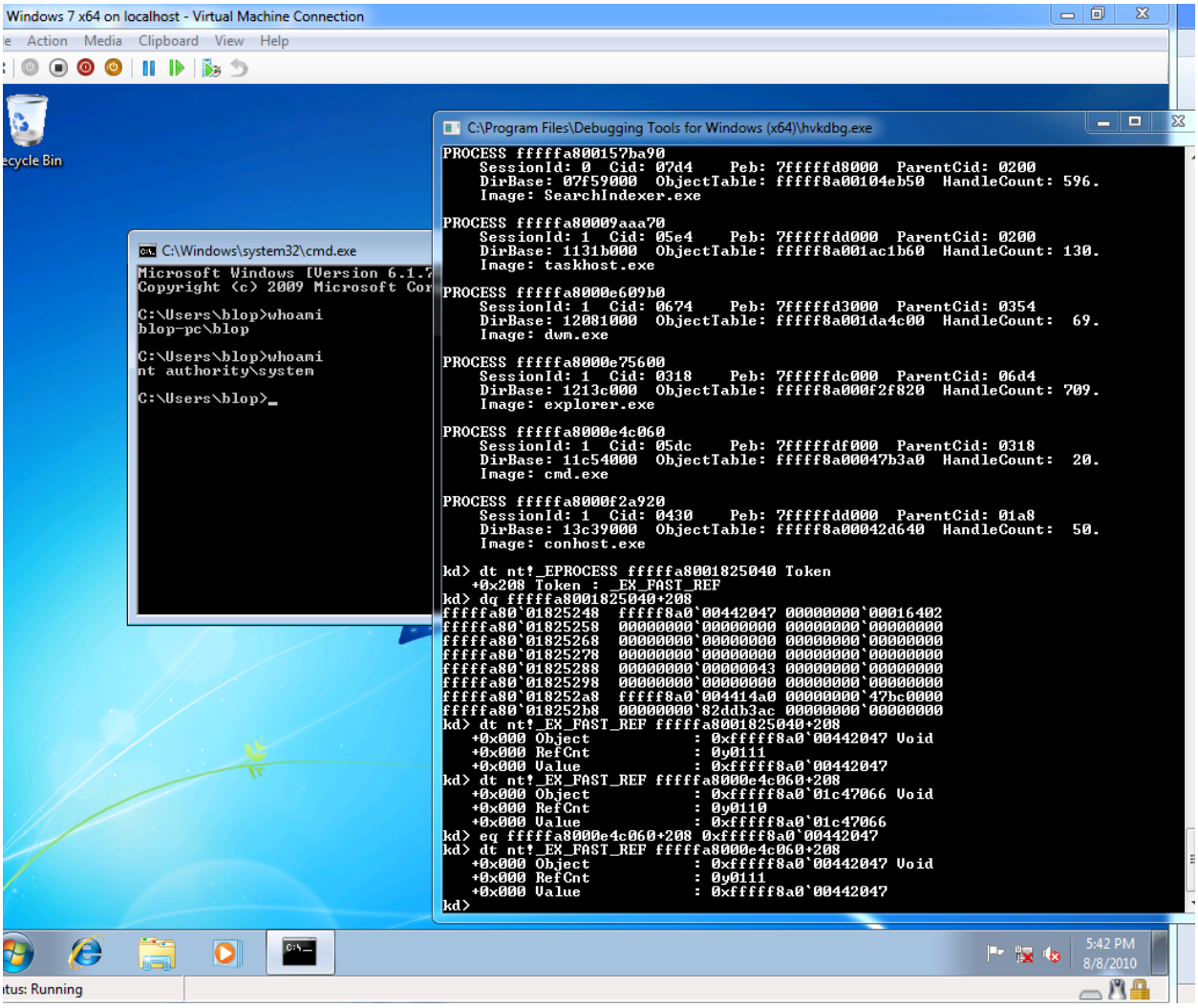
```
PatchIAT(Page, "kernel32.dll", "CreateFileW", (ULONG64)CREATEFILE_OFFSET);
PatchIAT(Page, "kernel32.dll", "CreateFileMappingA", (ULONG64)CREATEFILEMAPPINGA_OFFSET);
PatchIAT(Page, "kernel32.dll", "CreateFileMappingW", (ULONG64)CREATEFILEMAPPINGW_OFFSET);
PatchIAT(Page, "kernel32.dll", "MapViewOfFile", (ULONG64)MAPVIEWOFFILE_OFFSET);
PatchIAT(Page, "kernel32.dll", "UnmapViewOfFile", (ULONG64)UNMAPVIEWOFFILE_OFFSET);
PatchIAT(Page, "kernel32.dll", "GetFileSize", (ULONG64)GETFILESIZE_OFFSET);
PatchIAT(Page, "kernel32.dll", "VirtualProtect", (ULONG64)VIRTUALPROTECT_OFFSET);
```

Read operations by the Debugger Engine Library are done by reading cached memory pages by using *MapViewOfFile*.

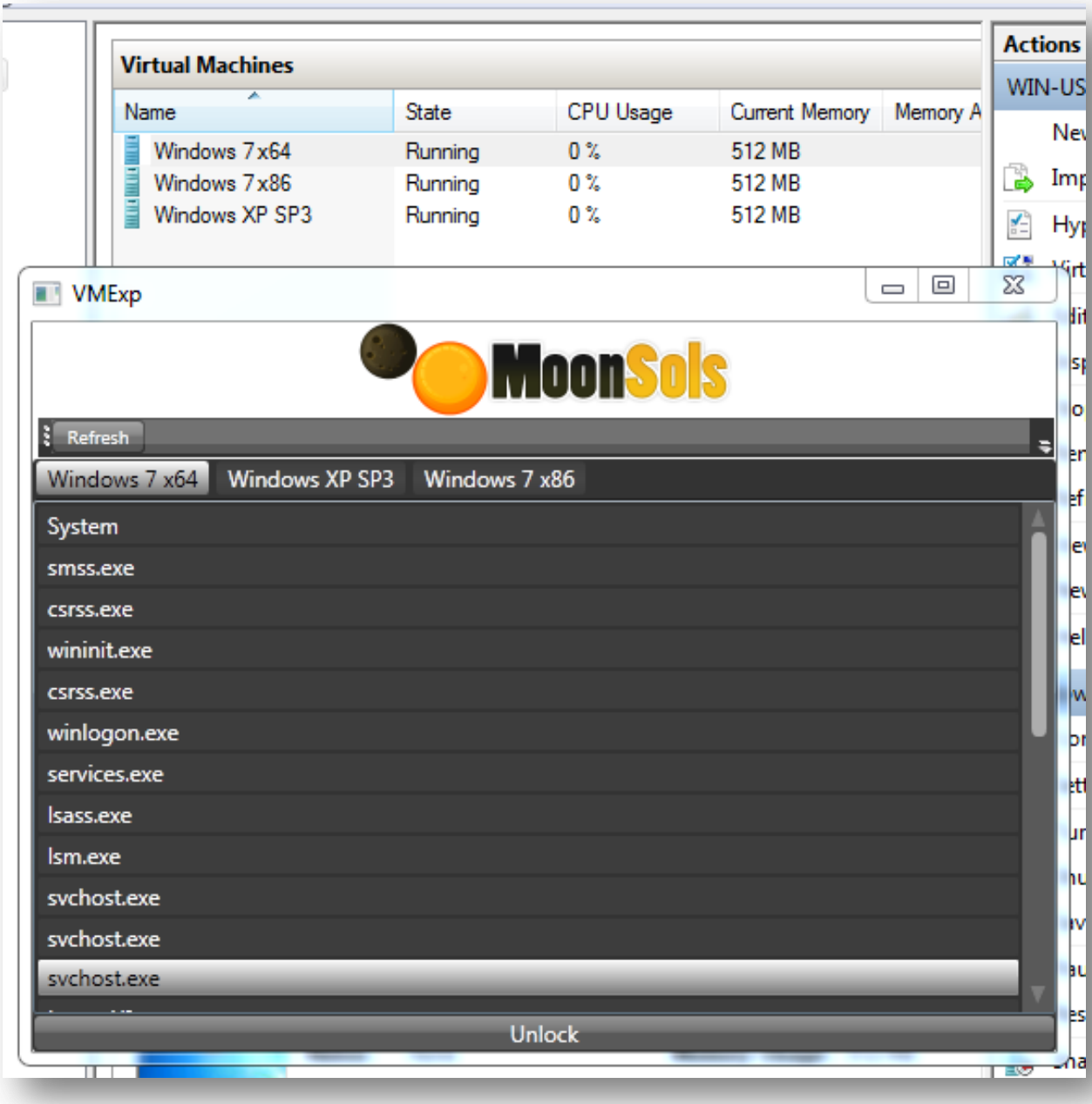
And write operations are done by an internal and exported function which is calling *VirtualProtect*, at the very beginning so the address of the caller can be retrieved by reading the stack like the following :

```
BOOL WINAPI MyVirtualProtect(
  __in   LPVOID lpAddress,
  __in   SIZE_T dwSize,
  __in   DWORD flNewProtect,
  __out  PDWORD lpflOldProtect
)
{
PFUNCTION_TABLE FT;
ULONG i, j;
ULONG Offset;
PUCHAR Page, Source;
BOOL Status;

PULONGLONG RegEsp;

    FT = (PFUNCTION_TABLE)NULL;

    //
    // Even if __fastcall is the 64-bits convention, arguments are saved in the stack.
    // Using this method we can retrieve the initial RSP address in a generic way.
    //
    RegEsp = (PULONGLONG)((PUCHAR)&lpAddress - sizeof(PVOID) + 0x80);
    // rsp + 0x80 = dbgeng!DbsDemandMappedFile::Write RSP

    Status = FALSE;
    Page = NULL;
[…]
```

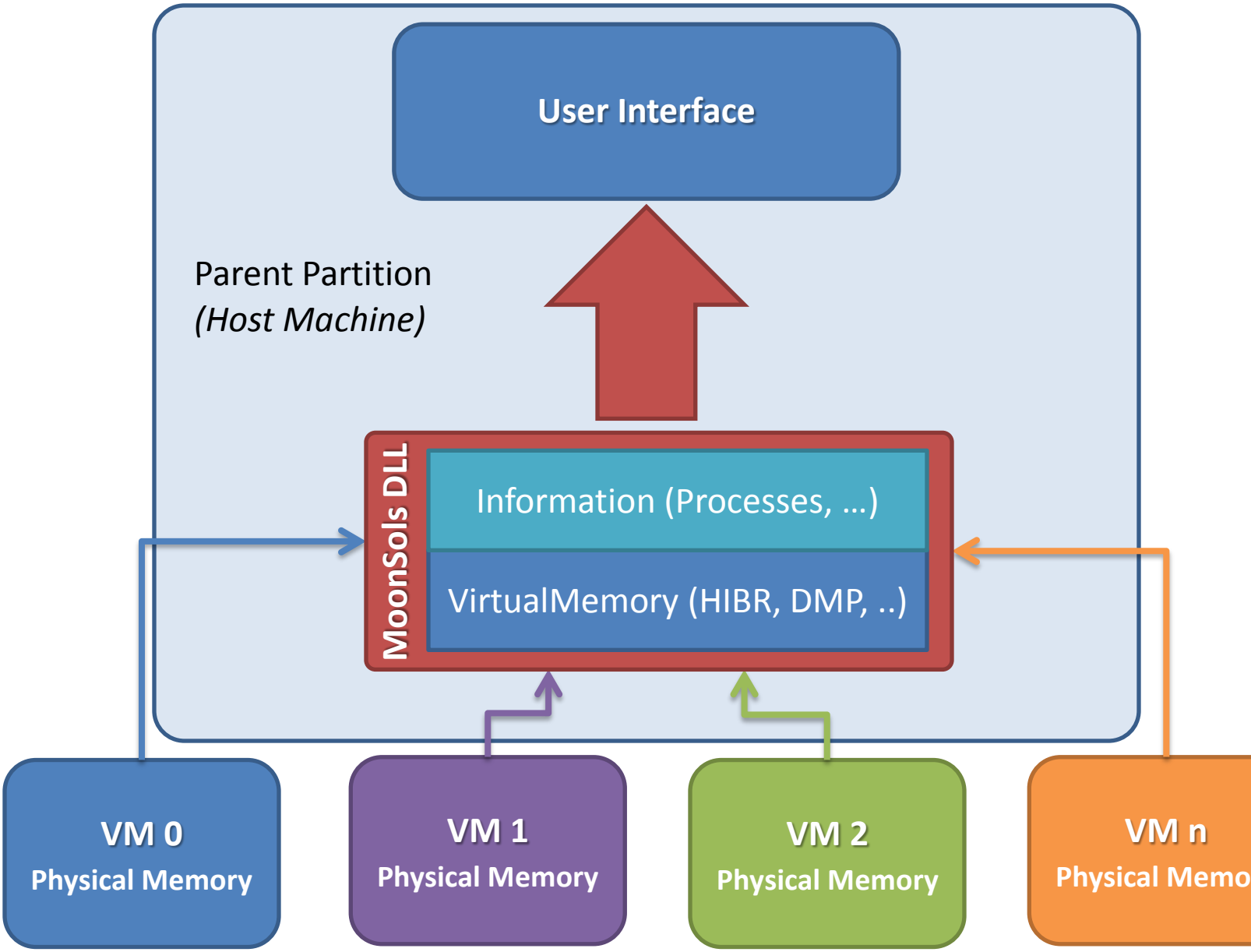
From now, we can use the Windows Debugger to browse the physical memory, and to modify it.



A complete interface for standalone utilities can also be created from scratch like the following:



In the screenshot above the architecture of the software is different; the debugger engine is not used at all everything had been rewritten from scratch for “various purposes”.


User Interface
Parent Partition
(Host Machine)
MoonSols DLL
Information (Processes, …)
VirtualMemory (HIBR, DMP, ..)
VM 0
Physical Memory
VM 1
Physical Memory
VM 2
Physical Memory
VM n
Physical Memo