Hibernation File Format

Peter Kleissner

I want to discuss the Hibernation File Format. It is part of my presentation "Hibernation File Attack" at Black Hat Europe 2009.

The hibernation file is a binary file storing the physical memory and processor state for restoring a hibernated PC. I specifically discuss the Windows Hibernation File Format here.

The file name is "hiberfil.sys" and stored in the root directory of the booted drive. Its size is the size of physical memory installed, though the data is stored compressed and so there is much preserved space in the hibernation file left unused.

This is the format of the hibernation file:

Page	Offset in File	
0	0	Hibernation File Header
1	4096	Processor State
2	8192	Reserved Memory Map
		Memory Table 1
		Xpress Image
		Xpress Image
		…
		Memory Table 2
		Xpress Image
		…
		Memory Table 3
		Xpress Image
		…
		Memory Table N
		Xpress Image
		…

In Windows XP and lower the last two pages (1 and 2) are reversed.

Hibernation File Header

The first page (4096 bytes) contains the hibernation header:

```
typedef struct
{
     ULONG Signature;
     ULONG Version;
     ULONG CheckSum;
     ULONG LengthSelf;
     ULONG PageSelf;
     UINT32 PageSize;
     ULONG64 SystemTime;
     ULONG64 InterruptTime;
     DWORD FeatureFlags;
     DWORD HiberFlags;
     ULONG NoHiberPtes;
     ULONG HiberVa;
     ULONG64 HiberPte;
     ULONG NoFreePages;
     ULONG FreeMapCheck;
     ULONG WakeCheck;
     UINT32 TotalPages;
     ULONG FirstTablePage;
     ULONG LastFilePage;
     PO_HIBER_PREF PerfInfo;
     ULONG NoBootLoaderLogPages;
     ULONG BootLoaderLogPages[8];
     ULONG TotalPhysicalMemoryCount;
} PO_MEMORY_IMAGE, *PPO_MEMORY_IMAGE;
```

The HIBER_PER structure was introduced in Windows XP and the BootLoaderLog fields were introduced in Windows Vista. There was a second ImageType field after PageSize, removed with Windows Vista.

```
typedef struct _PO_HIBER_PERF
{
     UINT64 IoTicks;
     UINT64 InitTicks;
     UINT64 CopyTicks;
     UINT64 StartCount;
     ULONG ElapsedTime;
     ULONG IoTime;
     ULONG CopyTime;
     ULONG InitTime;
     ULONG PagesWritten;
     ULONG PagesProcessed;
     ULONG BytesCopied;
     ULONG DumpCount;
     ULONG FileRuns;
     UINT64 ResumeAppStartTime;
     UINT64 ResumeAppEndTime;
     UINT64 HiberFileResumeTime;
} PO_HIBER_PERF, *PPO_HIBER_PERF;
```

The signature is the most important part:

‘hibr’	Valid Windows XP or lower hibernation file, `ntldr` shall call `osloader.exe` to load hibernation file and process hibernation resume
‘wake’	Invalid Windows XP or lower hibernation file, system shall start normally
‘HIBR’	Valid Windows Vista or higher hibernation file, `winload.exe` shall call `winresume.exe` to process hibernation resume
‘WAKE’	Invalid Windows Vista or higher hibernation file, system shall start normally
‘RSTR’	During restoration (resume) the state is RSTR (will only occur if hibernation file is erroneous)
‘RSTR’	Also used in NTFS for restoration area; Windows Vista and above (currently unknown)

Processor State

The second page contains the processor state:

```
typedef struct _KPROCESSOR_STATE
{
     CONTEXT ContextFrame;
     KSPECIAL_REGISTERS SpecialRegisters;
} KPROCESSOR_STATE, *PKPROCESSOR_STATE;
```

“The context structure is platform specific” a Microsoft employee would say now. But I’m not and I’m working with the Intel Architecture 32 bit. So here’s the processors context structure, also appearing in task switching and exception handling:

```
typedef struct _CONTEXT
{
     ULONG ContextFlags;
     ULONG Dr0;
     ULONG Dr1;
     ULONG Dr2;
     ULONG Dr3;
     ULONG Dr6;
     ULONG Dr7;
     FLOATING_SAVE_AREA FloatSave;
     ULONG SegGs;
     ULONG SegFs;
     ULONG SegEs;
     ULONG SegDs;
     ULONG Edi;
     ULONG Esi;
     ULONG Ebx;
     ULONG Edx;
     ULONG Ecx;
     ULONG Eax;
     ULONG Ebp;
     ULONG Eip;
     ULONG SegCs;
     ULONG EFlags;
     ULONG Esp;
     ULONG SegSs;
     UCHAR ExtendedRegisters[512];
} CONTEXT, *PCONTEXT;
```

```
typedef struct _FLOATING_SAVE_AREA
{
     ULONG ControlWord;
     ULONG StatusWord;
     ULONG TagWord;
     ULONG ErrorOffset;
     ULONG ErrorSelector;
     ULONG DataOffset;
     ULONG DataSelector;
     UCHAR RegisterArea[80];
     ULONG Cr0NpxState;
} FLOATING_SAVE_AREA, *PFLOATING_SAVE_AREA;
```

Beside the application registers the system registers are also dumped:

```
typedef struct _KSPECIAL_REGISTERS
{
     ULONG Cr0;
     ULONG Cr2;
     ULONG Cr3;
     ULONG Cr4;
     ULONG KernelDr0;
     ULONG KernelDr1;
     ULONG KernelDr2;
     ULONG KernelDr3;
     ULONG KernelDr6;
     ULONG KernelDr7;
     DESCRIPTOR Gdtr;
     DESCRIPTOR Idtr;
     WORD Tr;
     WORD Ldtr;
     ULONG Reserved[6];
} KSPECIAL_REGISTERS, *PKSPECIAL_REGISTERS;
```

```
typedef struct _DESCRIPTOR
{
     WORD Pad;
     WORD Limit;
     ULONG Base;
} DESCRIPTOR, *PDESCRIPTOR;
```

Reserved Memory Map

At the third page the free / reserved memory map starts. It is used to initialize and reload spaces in memory.

Memory Map

The physical memory is stored compressed as multiple parts in a list. Every entry in that list is called a memory map. The FirstTablePage variable in the hibernation file header points to the first table. A memory table is always followed by one or more xpress images which contain the compressed data:

Page	Offset	
+ 0	+ 00000000h	Memory Table
+ 1	+ 00001000h	Xpress Image 0
	dynamic	Xpress Image 1
	dynamic	…
	dynamic	Xpress Image N

The structure of a memory table:

```
struct MEMORY_TABLE
{
     DWORD PointerSystemTable;
     UINT32 NextTablePage;
     DWORD CheckSum;
     UINT32 EntryCount;
     MEMORY_TABLE_ENTRY MemoryTableEntries[EntryCount];
};
```

NextTablePage points to the next memory table. CheckSum is unused and always zero. EntryCount is the count of memory table entries which is always 255 except for the last table. The size of a memory table is exactly one page, calculated by 16 bytes header + 255 entries * 16 bytes per entry = 4096.

An entry in the memory table describes a physical memory range in an Xpress Image:

```
struct MEMORY_TABLE_ENTRY
{
     UINT32 PageCompressedData;
     UINT32 PhysicalStartPage;
     UINT32 PhysicalEndPage;
     DWORD CheckSum;
};
```

PageCompressedData should point to the xpress image but is invalid (except for the first entry). CheckSum is (again) unused and always set to zero.

Following the memory table are xpress images, always one immediately after the other. An xpress image consists of an IMAGE_XPRESS_HEADER and its following compressed data:

```
struct IMAGE_XPRESS_HEADER
{
     CHAR Signature[8] = 81h, 81h, "xpress";
     BYTE UncompressedPages = 15;
     UINT32 CompressedSize;
     BYTE Reserved[19] = 0;
};
```

To get the real size of compressed data calculate CompressedSize / 4 + 1 and round it up to 8.
To get the real amount of uncompressed pages calculate UncompressedPages + 1.
Directly after the header follows the compressed data.

Note that the physical pages described via the memory map table entries are stored as compressed pages one after the other in xpress images one after the other.

Compression Algorithm

The algorithm used for compression is LZ77 + DIRECT2. MS-OXCRPC explains how these two algorithms work and how they are used.

See the example for LZ77 of MS-OXCRPC:

Input Stream:

Position	1	2	3	4	5	6	7	8	9
Byte	A	A	B	C	B	B	A	B	C

Step	Position	Match	Byte	Output
1.	1	-	A	(0, 0) A
2.	2	A	B	(1, 1) B
3.	4	-	C	(0, 0) C
4.	5	B	B	(2, 1) B
5.	7	A B	C	(5, 2) C

The data is compressed as (Position Backward, Copy Bytes) New Byte.

DIRECT2 algorithm is used for the decoder to distinguish data (the byte) from metadata (the position/size). It defines a bit-wise indicator declaring a byte sequence as metadata or data. Further DIRECT2 defines how the metadata (position + length) is stored.

This algorithm is used in the Microsoft Exchange Protocol.
Also note that the MS-OXCRPC is proprietary and not covered by the Microsoft's Open Specification Promise.

References

- Microsoft Wire Format Specification Protocol (MS-OXCRPC)
  http://msdn.microsoft.com/en-us/library/cc425493.aspx
- SandMan project
  http://sandman.msuiche.net/docs/SandMan_Project.pdf
  http://msuiche.net/con/bhusa2008/Windows_hibernation_file_for_fun_%27n%27_profit-0.6.pdf
  http://www.msuiche.net/pres/PacSec07-slides-0.4.pdf
  http://www.msuiche.net/con/euro2008/Exploiting_Windows_Hibernation_File.pdf
  http://sandman.msuiche.net/
- NirSoft Windows Vista Kernel Structures
  http://www.nirsoft.net/kernel_struct/vista/
- Custom hibernation file reverse engineering