WHITE PAPER

Extracting Compressed Pages from the Windows 10 Virtual Store

Abstract

Windows 8.1 introduced memory compression in August 2013. By the end of 2013 Linux 3.11 and OS X Mavericks leveraged compressed memory as well. Disk I/O continues to be orders of magnitude slower than RAM, whereas reading and decompressing data in RAM is fast and highly parallelizable across the system's CPU cores, yielding a significant performance increase. However, this came at the cost of increased complexity of process memory reconstruction and thus reduced the power of popular tools such as Volatility, Rekall, and Redline.

In this document we introduce a method to retrieve compressed pages from the Windows 10 Memory Manager Virtual Store, thus providing forensics and auditing tools with a way to retrieve, examine, and reconstruct memory artifacts regardless of their storage location.

Introduction

Windows 10 moves pages between physical memory and the hard disk or the Store Manager's virtual store when memory is constrained. Universal Windows Platform (UWP) applications leverage the Virtual Store any time they are suspended (as is the case when minimized). When a given page is no longer in the process's working set, the corresponding Page Table Entry (PTE) is used by the OS to specify the storage location as well as additional data that allows it to start the retrieval process. In the case of a page file, the retrieval is straightforward because both the page file index and the location of the page within the page file can be directly retrieved. For pages compressed in the Virtual Store, the retrieval involves deriving a Store Manager key from the PTE, then using it to traverse cascaded structures, finally generating the virtual address of the compressed page inside a dedicated process used by the Memory Manager.

Background

The Windows 10 Memory Manager is responsible for providing each process on a 64-bit Windows machine with a 16 TB virtual address range, and a 4 GB range on 32-bit builds. The memory manager creates the illusion of having the entire system's memory space available for the process by transferring pages between:

- Physical memory (RAM)
- Hard disk (pagefile.sys and swapfile.sys)
- Virtual Store (MemoryCompression process)

We will only cover the case of translating small pages (4KB) using either x86 or x64 paging mode, since large (2MB) and huge (1GB) pages reside in non-pageable memory and do not use the hard disk or the Virtual Store.

The page correlating to a virtual memory address can be located by traversing the process's page tables. In Intel x86 and x64 paging modes, the CR3 register is process-specific and provides the physical base address of the Page Map Level 4 (PML4) table on x64 (Figure 1) or the Page Directory Pointer Table (PDPT) on x86 (Figure 2). The PML4 table in x64 was introduced to support the additional memory space requirements.

Figure 1:

Intel 64-bit page tables.

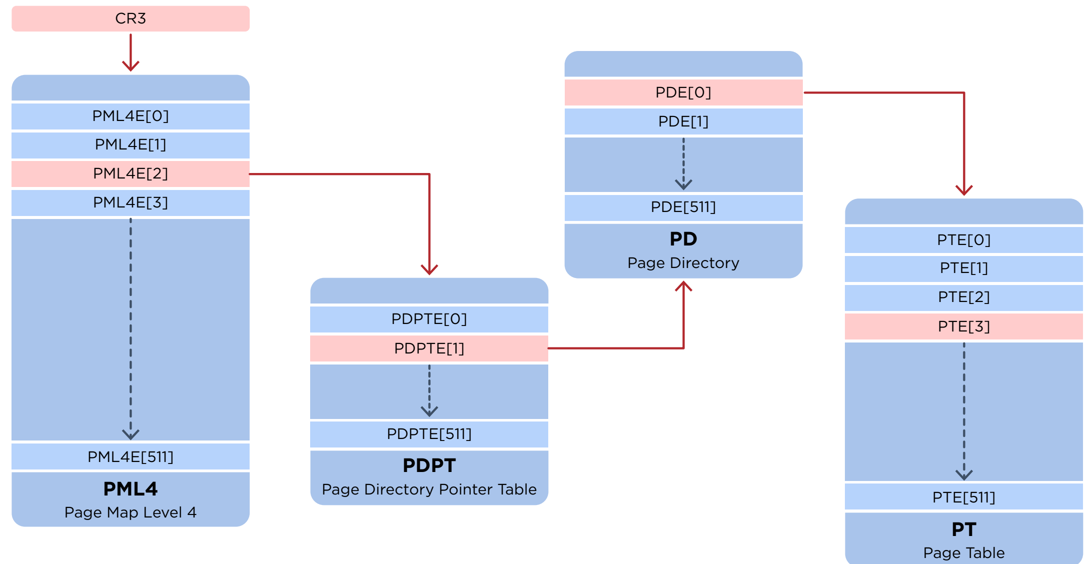
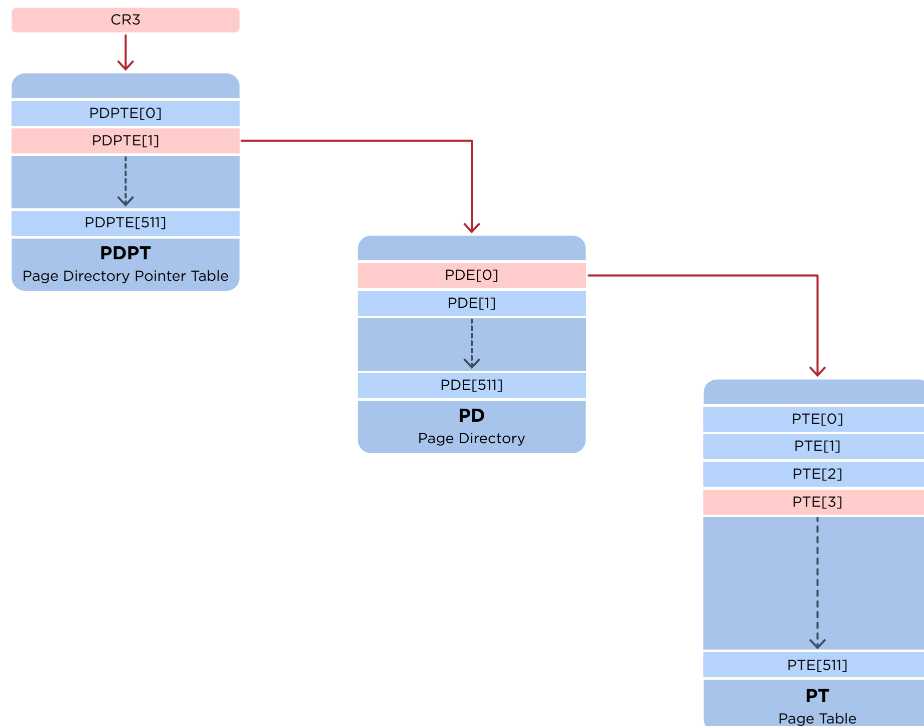


Figure 2:

Intel 32-Bit page tables with Physical Address Extensions (PAE) enabled.



Each of the indices inside the page tables is calculated by splitting the virtual address in chunks of nine bits starting from bit 12. The 12 least-significant bits of a virtual address (from 0 to 11) represent the offset within the page and are not used to traverse the page tables. The last table contains the Page Table Entry (PTE), a 64-bit structure that contains information about the current location and state of the corresponding page. An example of 64-bit virtual address parsing is shown in Figure 3 followed by a 32-bit example in Figure 4.

Figure 3: Intel 64-bit Virtual Address Translation.

Figure 4: Intel 32-Bit Virtual Address Translation.

From a forensic point of view, the data in the Virtual Store is most interesting because it has been inaccessible during investigations of memory snapshots until now. In the next section, we will identify if a PTE corresponds to a page located in the Virtual Store and then focus on a method to traverse the Store Manager's structures in order to reliably retrieve the compressed pages.

Virtual Store Page Retrieval

Overview

In the event that a page has been moved to the Virtual Store or a page file, its PTE's validity (bit 0) will be cleared. The Memory Manager then interprets the nt!_MMPTE structure as an nt!_MMPTE_SOFTWARE structure. The software PTE contains the PageFileLow and PageFileHigh fields which identify the page file (pagefile.sys, swapfile.sys or Virtual Store) and offset the data can be located within. PageFileLow is an index into the nt!_MMPAGING_FILE pointer array located at nt!MmPagingFile . The nt!MmPagingFile address contains a series of up to 15 pointers to nt!_MMPAGING_FILE structures, each of which represents a page file on the system. In Windows 10, the Virtual Store is represented as another page file on the system and described by the same nt!_MMPAGING_FILE structure as page files on disk.

On a Windows machine with default configuration, the software PTE's PageFileLow field refers to pagefile.sys if 0, and swapfile.sys if 1. A value of 2 indicates that the page is located in the Virtual Store. It is possible to configure Windows to have up to 15 physical page files on the system, so in such cases the Virtual Store index will be the index of the last entry in the array of nt!_MMPAGING_FILE structures pointed by the nt!MmPagingFile global variable. The PTE provides an index without distinction between a page file or Virtual Store, so it is important to enumerate the entries in the array in order to ensure that the PTE is describing a page in the Virtual Store before embarking on the store manager structure traversal.

The exact location of the nt!MmPagingFile global variable is another challenge to overcome because Microsoft no longer exports the variable in the kernel's PDB file. One way to find the nt!MmPagingFile involves the disassembling of a specific kernel function and is described further in the Appendix A.

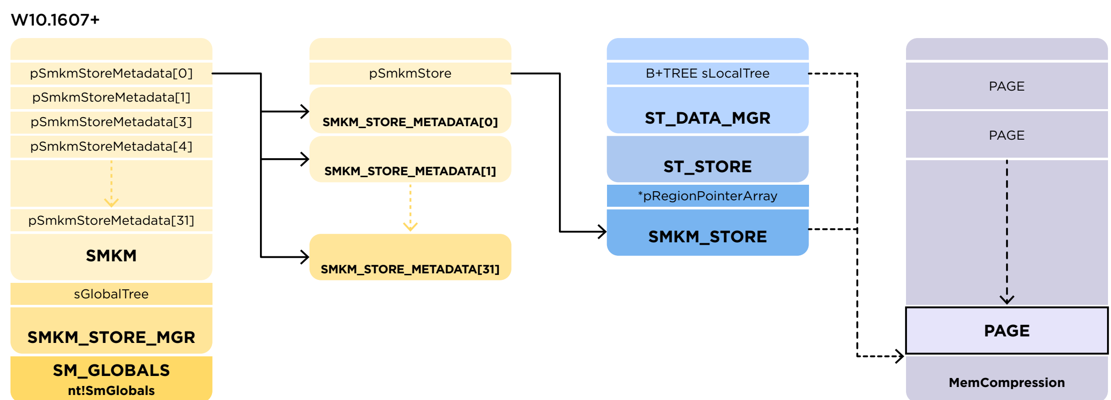
General Algorithm

The process of locating a compressed page in memory begins with calculating the value of the store manager key (SM_PAGE_KEY) using the nt!_MMPTE_SOFTWARE's PageFileLow, PageFileHigh and SwizzleBit fields from the PTE. The Virtual Store's top level structure is called SM_GLOBALS and is pointed to by the exported nt!SmGlobals symbol. The SM_GLOBALS structure has several nested structures which store information about all the SM_PAGE_KEYs used by the Store Manager along with a two-dimensional array that contains information for up to 1024 possible stores. Using a global B+Tree, each of the SM_PAGE_KEYs on the system can be associated with a store index (as well as creation flags) that is used to select the particular store from the two-dimensional array. Once the store is known, another B+Tree container provides the relationship between the particular SM_PAGE_KEY key and another value, called a chunk key. After the chunk key is known, it is used to derive the base address and the offset of the compressed page inside the Memory Compression process. A high-level graphical representation of the page retrieval process is given in Figure 5 below.

Note that the traversal from a given PTE to the corresponding compressed page is possible only in one direction. Although it's possible to dump and decompress all the pages inside the Memory Compression process, there is no way to associate the pages with their corresponding processes from the pages alone.

Figure 5:

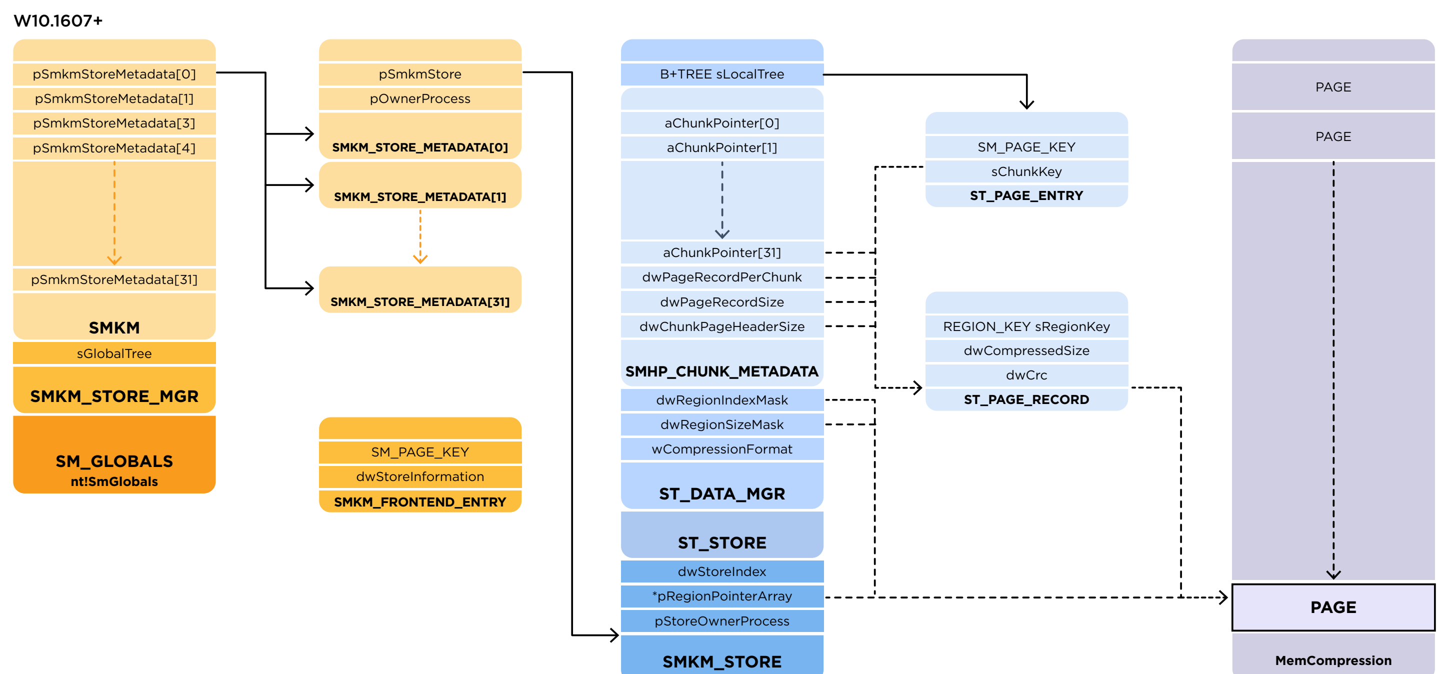
General overview of structures involved in page retrieval from the Virtual Store.



Detailed Algorithm

In this section we will describe the retrieval process in detail, as shown in the Figure 6, below:

Figure 6: Detailed Structure View.



We will start from a PTE that specifies a page in the Virtual Store, as shown in Figure 7.

Figure 7: Using the WinDbg's !pte to view the PTE of a virtual address which has been paged out to the Virtual Store.

```
kd> !pte 0x227d1a60000
                                           VA 00000227d1a60000
PXE at FFFFBEDF6FB7D020    PPE at FFFFBEDF6FA044F8    PDE at FFFFBEDF4089F468    PTE at FFFFBE8113E8D300
contains 0A00000061E95867  contains 0A0000001C996867  contains 0A0000001B717867  contains 001BBEDA00002084
pfn 61e95     ---DA--UWEV  pfn 1c996     ---DA--UWEV  pfn 1b717     ---DA--UWEV  not valid
                                                                                  PageFile:  2
                                                                                  Offset: 1bbeda
                                                                                  Protect: 4 - ReadWrite
```

In this example, the software PTE was parsed and the PageFile index was 2. On a default Windows configuration, that will be the Virtual Store. To verify this, we can use WinDbg's !vm extension as shown in Figure 8:

Figure 8:

Using the !vm extension to view pagefiles.

```
kd> !vm
Page File: \??\C:\pagefile.sys
  Current:   1769472 Kb  Free Space:   1769080 Kb
  Minimum:   1769472 Kb  Maximum:      4194304 Kb
Page File: \??\C:\swapfile.sys
  Current:     16384 Kb  Free Space:     16376 Kb
  Minimum:     16384 Kb  Maximum:      1572076 Kb
No Name for Paging File
  Current:   5242356 Kb  Free Space:   5239660 Kb
  Minimum:   5242356 Kb  Maximum:      5242356 Kb
```

To do this programmatically, the location of nt!MmPagingFile will be needed. The variable can be found by disassembling the nt!MmStoreRegister kernel function, as shown in Appendix A.

The PTE in Figure 7, (parsed as nt!_MMPTE_SOFTWARE) contains the two pieces of data that combined together, comprise the SM_PAGE_KEY, as shown in Figure 9 and Figure 10.

Figure 9:

Deriving the SM_PAGE_KEY from the PTE (pre-1803).

```
SM_PAGE_KEY = (PTE.PageFileLow << 0x1c) | PTE.PageFileHigh
```

Figure 10:

Figure 10: Deriving the SM_PAGE_KEY from the PTE (1803+)

```
SM_PAGE_KEY = (PTE.PageFileLow << 1c) | ((PageFileHigh) & ~InvalidPteMask)
```

For example, the SM_PAGE_KEY associated with the PTE on Figure 7 is calculated as 0x201bbeda. Beginning in 1803, a new algorithm was introduced to calculate the SM_PAGE_KEY. The modification depends on the value of nt!_MMPTE_SOFTWARE.SwizzleBit and if not set the PTE must be bit-flipped with a mask named InvalidPteMask that belongs inside the nt!_MI_HARDWARE_STATE structure (see Figure 8) . The InvalidPteMask can be retrieved by following the global variable nt!MiState into the nt!_MI_SYSTEM_INFORMATION structure.

Once the SM_PAGE_KEY is calculated, the next task is to find its associated store. The relationship between the store index and the SM_PAGE_KEY is established in a B+ tree container pointed by the SMKM_STORE_MGR.sGlobalTree field. The entries inside the B+ tree are SMKM_FRONTEND_ENTRY structures that associate a store index to each of the SM_PAGE_KEYs on the system. The Windows 10 versions after 1607 use a two-dimensional array (32x32) to describe each of the SMKM_STORE stores. The array indices derive from the retrieved store index, as shown in Figure 11:

Figure 11:

Calculating the store metadata array indices by the store index.

```
/* dwStoreIndex divided by 32: i = quotient, j = remainder */
i = dwStoreIndex >> 5, j = dwStoreIndex & 1f
```

Each SMKM_STORE structure represents a single store and each store describes multiple pages within it along with all the necessary information for their retrieval. The SMKM_STORE.pRegionPointerArray field points to an array of pointers to regions of compressed pages inside the Memory Compression host process. The nested SMHP_CHUNK_METADATA structure contains a two-dimensional array of chunks (32 rows x Variable number of columns, depending on the memory pressure), with each of them containing a vector of ST_PAGE_RECORD entries preceded by a header with size given in the SMHP_CHUNK_METADATA.dwChunkPageHeaderSize field. The chunk key is associated with the SM_PAGE_KEY by an one-to-one relationship, given in the ST_DATA_MGR.sLocalTree B+ tree. The calculations leading to the location of the ST_PAGE_RECORD structure of interest are shown in Figure 12.

Figure 12:

Deriving the ST_PAGE_RECORD of interest from the associated with the SM_PAGE_KEY chunk key.

```
/* dwChunkKey is retrieved from the ST_DATA_MGR.sLocalTree using the SM_PAGE_KEY as an index */
DWORD i, j, k, m;
k = dwChunkKey >> pSmhpChunkMetadata->dwVectorSize; /* k = array indices */
i = BitScanReverse(k);
j = BitTestAndComplement(k, i) * 2;
m = (dwChunkKey & pSmhpChunkKeyMetadata->dwPageRecordsPerChunk) * pSmhpChunkMetadata-
>dwPageRecordSize;
PVOID pChunk = pSmhpChunkMetadata->aChunkPointer[i][j];
ST_PAGE_RECORD* pPageRecord = pChunk + pSmhpChunkMetadata->dwChunkPageHeaderSize + m;
```

The ST_PAGE_RECORD structure contains the region key sRegionKey, which encodes the index and the offset inside the regions of compressed pages pointed by the SMKM_STORE.pRegionPointerArray. The region key is decoded with the help of the ST_DATA_MGR fields dwRegionIndexMask and dwRegionSizeMask and the result of those calculations leads to the virtual address of the compressed page of interest inside the Memory Compression process, as shown in Figure 13:

Figure 13:

Deriving the compressed page's virtual address by the associated ST_PAGE_RECORD.

```
/* pPageRecord points to the found ST_PAGE_RECORD structure in Figure 9 */
DWORD dwRegionIndex, dwRegionOffset;
PVOID pCompressedPage;
dwRegionIndex = pPageRecord->sRegionKey >> (ST_DATA_MGR.dwRegionIndexMask & 0xFF);
dwRegionOffset = (pPageRecord->sRegionKey & ST_DATA_MGR.dwRegionSizeMask) << 4
pCompressedPage = SMKM_STORE.pCompressedRegionPtrArray[dwRegionIndex] +
dwRegionOffset;
```

At this point we can locate the compressed page's PTE from its virtual address within the Memory Compression process and find out if it resides in RAM or it has been evicted to a page file, in case of a high memory pressure in the Virtual Store. In both cases, the page retrieval follows the demand-paging model at this point and simply involves reading from the physical memory or from the specified page file and offset within it. Once the compressed page is retrieved, it needs to be decompressed using the algorithm, specified in the ST_DATA_MGR.wCompressionFormat field. On the observed systems the value is consistently set as COMPRESSION_FORMAT_XPRESS (0x3), which specifies Microsoft's Xpress LZ77 compression engine. The Memory Manager currently uses the RtlDecompressBufferEx API to decompress the page before returning it to the process's working set.

Conclusion

The Windows 10 memory compression presents a significant challenge for existing forensics and auditing tools. Critical data inside the Virtual Store is often not retrieved, which diminishes the effectiveness of the analysis. The presented method of retrieval from the Virtual Store allows developers to upgrade the existing tools or enables analysts to analyze Windows 10 processes. Often, Microsoft has been changing parts of the structures that implement the Virtual Store. Those changes (although small in the recent Windows releases) show that the retrieval algorithm is still evolving and it is quite likely that the future Windows 10 versions will have upgrades of the Virtual Store design and implementation.

Appendix A

Locating the nt!MmPagingFile Global Variable

Following Windows 7, Microsoft discontinued exporting the nt!MmPagingFile global variable. The variable points to an array of pointers to nt!_MMPAGING_FILE structures, each of which describes a page file or the Virtual Store. The nt!_MMPAGING_FILE structures describe important properties of the paging files on the system, such as their name and location on the disk. Another field, VirtualStorePageFile, can be used to confirm if the structure correlates to the Virtual Store.

One way to discover the address of the array of nt!_MMPAGING_FILE structures involves disassembling the nt!MmStoreRegister kernel function and either emulating or parsing the code in order to find out the location of the array. The nt!MmStoreRegister function in both 32-bit and 64-bit ntoskrnl.exe enumerates the elements of the paging files array with code similar to the one shown in Figure 14. It is important to keep in mind that the exact registers might change across different instances of ntoskrnl.exe. In this case the array address in the kernel is provided as the second parameter of the LEA instruction and the EAX register will hold the number of the elements inside the array.

Figure 14:

Portion of the nt!MmStoreRegister code that enumerates the array of nt!_MMPAGING_FILE elements.

```
   …
   lea r8, <array_addr> ; an array of nt!_MMPAGING_FILE structures
   mov r9d, eax ; the number of elements
loop:
   … ; instructions that enumerate the elements of the array
   jnz loop
```

Discovering the nt!SmGlobals global variable

The nt!SmGlobals can be located by parsing the ntoskrnl.exe PDB file or disassembling the kernel functions that reference it. One such a function is nt!SmInitSystem which provides the address of the variable as a parameter to calls to the nt!SmGlobalsInitialize and nt!SmQueryRegistry kernel functions (Figure 15):

Figure 15:

Portion of the nt!SmInitSystem code that shows the nt!SmGlobals variable used as a parameter.

```
…
lea rcx, <nt!SmGlobals>
call nt!SmGlobalsInitialize
lea rcx, <nt!SmGlobals>
call nt!SmQueryRegistry
```

Appendix B

The structures involved in the traversal of the Virtual Store from the calculated SM_PAGE_KEY to the compressed page address are given below (Windows 10 Version 1709 x64). Note that many actual structures contain more fields, but we list only the ones that are directly involved in the page retrieval process described in this document.

Global Structures

These structures implement the association between all SM_PAGE_KEYs on the system and their respective store (Figure 16).

Figure 16:

Global structures, involved in the compressed pages retrieval.

```
typedef struct _B_TREE_PAGE_HEADER {
    WORD wEntryCount;
    BYTE bLevel;
    BYTE bIsLeaf;
    DWORD dwPadding;
    struct _B_TREE_PAGE_HEADER* pNextNode;
} B_TREE_PAGE_HEADER;

typedef struct _B_TREE {
    B_TREE_PAGE_HEADER* pRoot;
    QWORD dwEntries;
    …
} B_TREE;

typedef struct _SMKM_STORE_METADATA {
    SMKM_STORE* pSmkmStore;
    EX_RUNDOWN_REF sRundownRef;
    PKPROCESS pOwnerProcess;
    DWORD dwFlags;
    …
} SMKM_STORE_METADATA;

typedef struct _SMKM {
    SMKM_STORE_METADATA* pSmkmStoreMetadata[32];
} SMKM;

typedef struct _SMKM_STORE_MGR {
    SMKM sSmkm;
    EX_PUSH_LOCK vLock;
    B_TREE sGlobalTree;    /* SM_PAGE_KEY to Store Index association */
} SMKM_STORE_MGR;

typedef struct _SM_GLOBALS {
    SMKM_STORE_MGR sSmkmStoreMgr;
} SM_GLOBALS;
```

Store Structures

The structures listed below implement the store defined by the particular SM_PAGE_KEY (Figure 17).

Figure 17:

Local structures, involved in the compressed pages retrieval.

```
typedef struct _ST_PAGE_RECORD {
    DWORD sRegionKey;
    DWORD dwCompressedSize;
    DWORD dwCRC;
    …
} ST_PAGE_RECORD;

typedef struct _ST_CHUNK_PAGE_RECORD {
    PVOID pPageRecords;
    …
} ST_CHUNK_PAGE_RECORD;

typedef struct _SMHP_CHUNK_METADATA {
    ST_CHUNK_PAGE_RECORD aChunkPointer[32];
    DWORD dw1[3];
    DWORD dwPageRecordsPerChunk;
    DWORD dwPageRecordSize;
    DWORD dw2;
    DWORD dwChunkPageHeaderSize;
} SMHP_CHUNK_METADATA;

typedef struct _ST_DATA_MGR {
    B_TREE sLocalTree;
    DWORD dw1[42];
    SMHP_CHUNK_METADATA sChunkMetadata;
    DWORD dw2[75];
    DWORD dwStoreFlags;
    DWORD dw3[5];
    struct SMKM_STORE* pSmkmStore;
    DWORD dwRegionSizeMask;
    DWORD dwRegionIndexMask;
    WORD wCompressionFormat;
    …
} ST_DATA_MGR;


typedef struct _ST_STORE {
    DWORD dw[20];
    ST_DATA_MGR sStDataMgr;
} ST_STORE;

typedef struct _SMKM_STORE {
    ST_STORE sStStore;
    DWORD dwStoreIndex;
    union {
        struct {
            DWORD bVal: 8;
            DWORD bMappingFlags: 8;
            DWORD wPadVal: 16;
        };
        DWORD dwPadding;
    };
    EX_PUSH_LOCK vLock;
    DWORD dw1[46];
    PVOID** pRegionPointerArray;
    DWORD dw2[86];
    PVOID pStoreOwnerProcess;
} SMKM_STORE;
```

Acknowledgements

Omar Sardar is a Staff Reverse Engineer on the FireEye Labs Advanced Reverse Engineering (FLARE) team where he analyzes a wide variety of malware to support incident response and intelligence analysis. He contributes to keeping FireEye products up to date by analyzing the Windows 10 kernel. Omar is involved with FLARE Education and previously wrote the WinDbg training module, available in the Malware Analysis Masters Course. Prior to the FLARE team, Omar specialized in developing and reverse engineering embedded systems with a focus on the USB protocol. Omar is based out of Alexandria, VA and enjoys road biking, making pizza and reading science fiction.

Dimiter Andonov is a Senior Staff Reverse Engineer on FireEye's FLARE team. He has specialized on low level malware, including bootkits and rootkits. Dimiter has over 12 years of experience as a reverse engineer and another 20 as an Assembly/C/C++ programmer. Prior to joining FLARE, Dimiter has worked in the Antivirus industry, leading the AV labs for Sunbelt Software, GFI, and ThreatTrack Security. In addition to the daily malware reversing, he currently works on reversing parts of the Windows 10 OS to provide support for the FireEye products.

The **FireEye Labs Advanced Reverse Engineering (FLARE) Team** is an internationally recognized center of excellence for reverse engineers skilled in malware, exploit, and vulnerability analysis. FLARE research and development leverages custom automatic malware triage, network traffic decryption, zero day discovery, and host artifact recovery to quickly get results out to those in the field. FLARE shares with the security community by releasing free tools and putting on the annual FLARE On Challenge.

We would like to thank the following contributors:

- **Claudiu Teodorescu:** Initial research & proof-of-concept
- **Blaine Stancill:** Volatility research integration
- **Sebastian Vogl:** Rekall research integration

To learn more about FireEye, visit: **www.FireEye.com**

FireEye, Inc.
601 McCarthy Blvd. Milpitas, CA 95035
408.321.6300/877.FIREEYE (347.3393)
info@FireEye.com

©2019 FireEye, Inc. All rights reserved. FireEye is a registered trademark of FireEye, Inc. All other brands, products, or service names are or may be trademarks or service marks of their respective owners.

About FireEye, Inc.
FireEye is the intelligence-led security company. Working as a seamless, scalable extension of customer security operations, FireEye offers a single platform that blends innovative security technologies, nation-state grade threat intelligence, and world-renowned Mandiant® consulting. With this approach, FireEye eliminates the complexity and burden of cyber security for organizations struggling to prepare for, prevent and respond to cyber attacks.