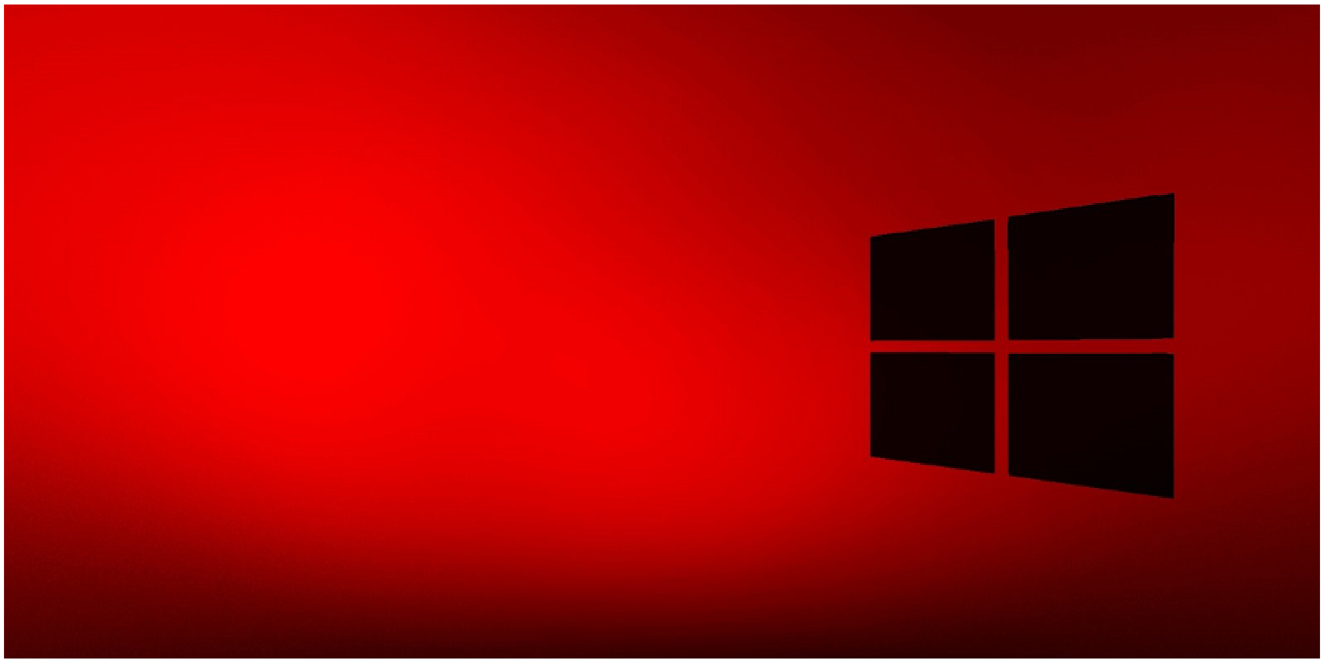
Inside the Windows Cache Manager

linkedin.com/pulse/inside-windows-cache-manager-artem-baranov



Artem Baranov

Artem Baranov

Security Researcher at Kaspersky

Published Sep 5, 2022

+ Follow

Introduction

The cache is an integral part of the operating system and its hybrid kernel. Roughly speaking, it's just a virtual memory region in the kernel address space, on which the Cache Manager maps file data to provide quick access to them in the future. This access is frequently used by the File System Driver (FSD) or the Windows Memory Manager (VMM). Instead of reading file data from disk every time a user or system needs to access to it, the OS kernel calls the Cache Manager in an attempt to get this data from memory. In turn, the Cache Manager is a set of function in the kernel executable file ntoskrnl.exe, which starts with a prefix *Cc*. These functions are private, so to get to their names, you need to configure the symbol server settings in WinDbg or IDA.

Learning the Windows Cache Manager is quite a difficult task for beginners. This Windows kernel subsystem is closely related to the VMM, so if you don't have enough knowledge in it, try to understand the basic concepts without going into complicated technical aspects. In addition, you should have some knowledge in the field of file system drivers (FSD), because they are the most frequent clients of the Cache Manager. It's worth to note that the cache concept exists only at the level of file system, lower drivers on the device stack like the volume manager, partition manager, disk driver, and disk port driver don't use it.

This blog post is dedicated to the technical aspects of the Windows Cache Manager and designed for the skilled Windows Internals readers. If you lack knowledge on this topic, read the corresponding chapter in the Windows Internals book and then get back to this post. I would say that this blog post is some kind of technical addition to the chapter about the cache in the book (or I hope it claims..).

Let's take a look at some terms for newbie.

Working Set (WS) - the set of pages in the user mode or kernel mode address space that are currently resident in physical memory. The kernel mode working set called System Working Set.

PTE (Page Table Entry) - a structure that is used by the CPU and VMM to translate virtual addresses to physical ones.

Proto-PTE (Prototype PTE, PPTE) - a special type of so called Software PTE that is used only by the VMM (not CPU) to work with section objects (memory-mapped files) and serves as an intermediate level for the translation mapped section pages to the real hardware PTE. PPTE is a key structure for understanding the section objects.

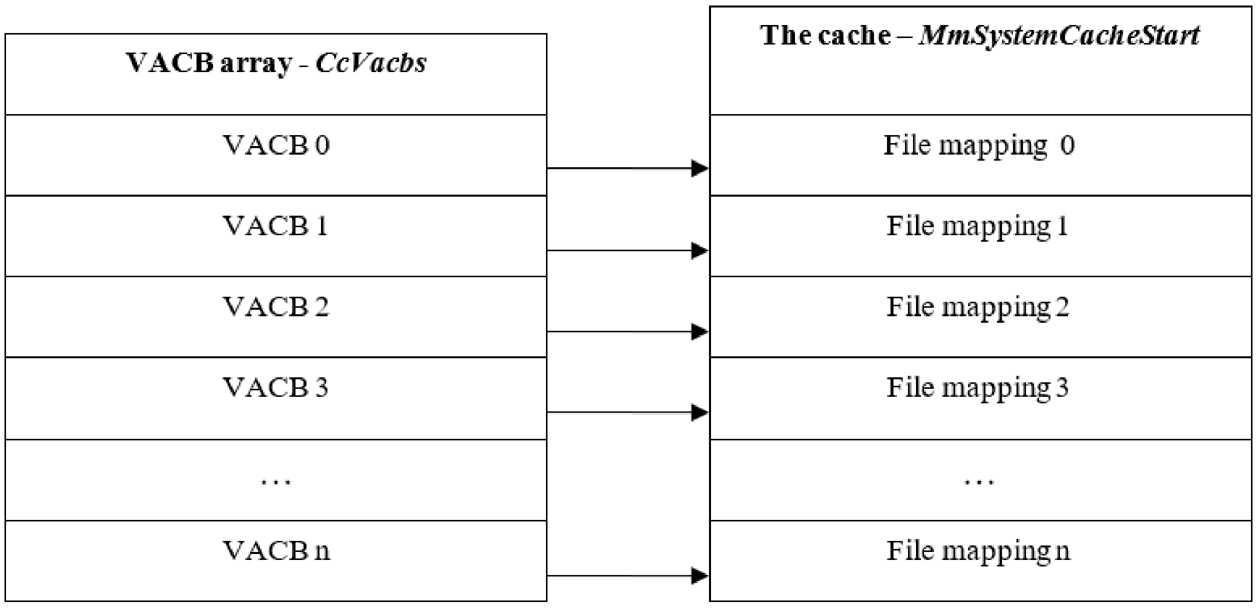
Segment Control Area (or just Control Area, CA) - a structure that contains information required for performing I/O operations with file data in or from the mapped file. It's stored in the non-paged pool. With the help of CA the VMM can address the same file as binary and as executable.

The basic concepts

The memory region in the kernel mode address space occupied by the cache starts with the value of the VMM variable *MmSystemCacheStart* and ends with the value of *MmSystemCacheEnd*. Thus, if X - is a pointer to the memory region that belongs to the cache, then *MmSystemCacheStart*<=X<=*MmSystemCacheEnd*. File data in this region are mapped into slots, 256MB blocks of data. The cache has two features, which are a consequence of the fact that the VMM is responsible for its internal implementation.

- The section objects maintained by the VMM are used to map file data into slots. Thus, the VMM is responsible for paging file data.
- The cache is a part of the system working set. This means that its pages can be unloaded to the page file.

These features emphasize the fact that the Cache Manager doesn't know for sure whether the file data is in physical memory or not. Undocumented structure called Virtual Address Control Block (VACB) is used to describe the cache slots, which are reserved in the paged pool. The control blocks are addressed from *CcVacbs* variable. Each of these blocks controls a specific slot. The variable *CcNumberVacbs* stores the number of slots.



VACB has the following format.

```
typedef struct _VACB
{
    PVOID BaseAddress; //pointer to the slot
    PSHARED_CACHE_MAP SharedCacheMap; //pointer to the shared cache map
    union
    {
        LARGE_INTEGER FileOffset; //file offset
        USHORT ActiveCount; //reference count to the image
    } Overlay;
LIST_ENTRY LruList; //VACB are linked in the list using this field
} VACB;
```

There are two VACBs lists.

- *CcVacbFreeList*. It's a list of free VACBs, i e those VACBs that are ready for use.
- *CcVacbLru*. A list of all other structures. A VACB has free status if its .ActiveCount field is zero. When reused, the slot address is re-mapped. The following WinDbg command confirms these facts.

r eax=0; !list "-t ntdll!_LIST_ENTRY.Flink -x \"r eax=@eax+1;? @eax;? @$extret-10; dt nt!_VACB @$extret-10\" nt!CcVacbFreeList "

We can use it to print free VACBs and their numbers, for example.

```
Evaluate expression: 1714 = 000006b2
Evaluate expression: -2120921208 = 81954f88
   +0x000 BaseAddress      : (null)
   +0x004 SharedCacheMap   : (null)
   +0x008 Overlay          : __unnamed
   +0x010 LruList          : _LIST_ENTRY [ 0x81954fb0 - 0x81954f80 ]

Evaluate expression: 1715 = 000006b3
Evaluate expression: -2120921184 = 81954fa0
   +0x000 BaseAddress      : (null)
   +0x004 SharedCacheMap   : (null)
   +0x008 Overlay          : __unnamed
   +0x010 LruList          : _LIST_ENTRY [ 0x81954fc8 - 0x81954f98 ]

```

Next from the first - 0x81954fb0=81954fa0 + 10.

We can do the same for the remaining (*CcVacbLru*).

r eax=0; !list "-t ntdll!_LIST_ENTRY.Flink -x \"r eax=@eax+1;? @eax;? @$extret-10; dt nt!_VACB @$extret-10\" nt!CcVacbLru"

```
Evaluate expression: 330 = 0000014a
Evaluate expression: -2120961816 = 8194b0e8
   +0x000 BaseAddress      : 0xc6000000
   +0x004 SharedCacheMap   : 0x817f61a0 _SHARED_CACHE_MAP
   +0x008 Overlay          : __unnamed
   +0x010 LruList          : _LIST_ENTRY [ 0x819491d8 - 0x81949178 ]

Evaluate expression: 331 = 0000014b
Evaluate expression: -2120969784 = 819491c8
   +0x000 BaseAddress      : 0xc2040000
   +0x004 SharedCacheMap   : 0x818c7b08 _SHARED_CACHE_MAP
   +0x008 Overlay          : __unnamed
   +0x010 LruList          : _LIST_ENTRY [ 0x8194ad38 - 0x8194b0f8 ]

```

Most of these structures have initialized shared maps and are mapped to the cache. If sum up the last VACB numbers from both lists, u get something like this.

14b+6b3 = 7fe

dd CcNumberVacbs l1

8055f670 000007fe

The virtual address of a specific slot will refer to the PTE pointing to the PPTE, the latter is linked to the subsection that describes the file (usually there's a one subsection that linked to the shared map and maps the file as binary, look at *MmMapViewInSystemCache*). You can learn more about PPTEs from my blog post here.

The cached file is described by two important structures called a *shared cache map* and a *private cache map*. Unlike the shared cache map, the private cache map isn't so interesting for exploring, because it's used for so-called intelligence ahead-read. Let's take a look at the shared cache map. It represents a structure that the Cache Manager maintains for caching a specific file. As in the case of control areas, which are unique for disk files (one is used to map the file as binary and another one to map it as an executable), the shared cache maps are unique as well and are addressed with SECTION_OBJECT_POINTERS structure, the latter is held by the FSD in the FCB structure of a specific file. Thus the Cache Manager knows what exactly slot describes a specific file via VACB, which stores a pointer to the shared cache map.

```
typedef struct _SECTION_OBJECT_POINTERS
{
    VOID*         DataSectionObject;
    VOID*         SharedCacheMap; //pointer to the shared map
    VOID*         ImageSectionObject;
}SECTION_OBJECT_POINTERS, *PSECTION_OBJECT_POINTERS;

```

The cache manager can find this structure for each opened FileObject, because it points to SECTION_OBJECT_POINTERS (FileObject->SectionObjectPointer). The shared cache map is described by the following structure.

```
typedef struct _SHARED_CACHE_MAP
{
…
/*0x008*/     union _LARGE_INTEGER FileSize;
…
/*0x018*/     union _LARGE_INTEGER SectionSize;
/*0x020*/     union _LARGE_INTEGER ValidDataLength;
…
/*0x030*/     struct _VACB* InitialVacbs[4]; //VACB index array
/*0x040*/     struct _VACB** Vacbs; // refers to the previous field if file_size <= 1MB
/*0x044*/     struct _FILE_OBJECT* FileObject; //the first file object linked to a shared map
/*0x048*/     struct _VACB* ActiveVacb;
/*0x04C*/     VOID*        NeedToZero;
…
/*0x058*/     ULONG32      ActiveVacbSpinLock;
/*0x05C*/     ULONG32      VacbActiveCount;
/*0x060*/     ULONG32      DirtyPages;
/*0x064*/     struct _LIST_ENTRY SharedCacheMapLinks;
/*0x06C*/     ULONG32      Flags;
…
/*0x074*/     struct _MBCB* Mbcb;
/*0x078*/     VOID*        Section;    //section object to map a file
…
/*0x090*/     struct _CACHE_MANAGER_CALLBACKS* Callbacks;
…
/*0x098*/     struct _LIST_ENTRY PrivateList;
…
/*0x0B4*/     struct _VACB* NeedToZeroVacb;
…
/*0x0D8*/     struct _PRIVATE_CACHE_MAP PrivateCacheMap; //describes a private map
}SHARED_CACHE_MAP, *PSHARED_CACHE_MAP;

```

The Cache Manager can find out quickly which of the specific files are already mapped (i e have used slots), the shared cache map points to the VACB index array. The first element of the array points to the first 256KB of the file, the second to the next 256KB and so on. In case if the file has size not more than 1MB, i e can fit in four slots, the array InitialVacbs from the shared cache map acts as an index array, otherwise the array is allocated from the paged pool. In any case, the pointer to it is stored in the Vacbs field. All shared cache maps linked into a list with the head in PrivateList (&SharedCacheMap->PrivateList, &PrivateCacheMap->PrivateLinks). Moreover, all shared cache maps are also linked into lists with SharedCacheMapLinks. There's a special function of the Cache Manager *CcInitializeCacheMap*, which is called by the FSD, and is responsible for initializing a shared cache map (if it hasn't been created yet), creating a section object and creating a private cache map.

VOID CcInitializeCacheMap (__in PFILE_OBJECT FileObject, __in PCC_FILE_SIZES FileSizes, __in BOOLEAN PinAccess, __in PCACHE_MANAGER_CALLBACKS Callbacks, __in PVOID LazyWriteContext)

This function is responsible for.

- It creates and initializes the shared cache map if it doesn't exist yet (FileObject->SectionObjectPointer->SharedCacheMap is zeroed), SharedCacheMap->FileObject is initialized by the first file object for which the map is created.
- It creates the section object with *MmCreateSection*. Further, this section will be used to map file data into cache slots.

- Creates a VACB index array with *CcCreateVacbArray*. This function initializes fields .Vacbs and .SectionSize.

If the FSD needs to read data from the cache, it calls *CcCopyRead*.

```
BOOLEAN
CcCopyRead (
    __in PFILE_OBJECT FileObject, //file object that was initialized using CcInitializeCacheMap
    __in PLARGE_INTEGER FileOffset, //file data offset
    __in ULONG Length, //size of read data
    __in BOOLEAN Wait, //if true, than the caller is ok with paging, otherwise file data should be already be in the cache
    __out_bcount(Length) PVOID Buffer, //copy buffer
    __out PIO_STATUS_BLOCK IoStatus
    )

```

Internally, the Cache Manager maps parts of file data with help of *CcGetVirtualAddress,* this function returns the base address of the data in memory. The function operates only with one VACB and one slot.

```
PVOID
CcGetVirtualAddress (
    IN PSHARED_CACHE_MAP SharedCacheMap, //ptr to a shared map
    IN LARGE_INTEGER FileOffset, //file offset for mapping
    OUT PVACB *Vacb, //returns a pointer to VACB that will be used for describing 256KB slot mapping
    IN OUT PULONG ReceivedLength //the number of adjacent bytes from the returned address
    )
```

The Cache Manager uses the following function to map file data.

PVACB CcGetVacbMiss (IN PSHARED_CACHE_MAP SharedCacheMap, IN LARGE_INTEGER FileOffset, IN OUT PKLOCK_QUEUE_HANDLE LockHandle, IN LOGICAL HasBcbListHeads)

The function searches for a VACB to map file data into cache slots and maps it using *MmMapViewInSystemCache* (the value for the file mapping is taken from &Vacb->BaseAddress).

The following WinDbg script explores the cache.

```
.expr /s masm;
.for(r eax=0; @eax < poi(CcNumberVacbs); r eax=@eax+1)
{
    r ecx=poi(CcVacbs) + @eax * 0x18;
    r ebx=poi(@ecx + 4);
    .printf "Vacb #%d    0x%p -> 0x%p\n", @eax, @ecx, poi(@ecx);
    .if( @ebx != 0 )
    {
        r ebx = poi( @ebx + 0x44 );
        .if( @ebx != 0 )
        {
            r ebx = @ebx + 0x30;
            .if( poi(@ebx+0x4) != 0 )
            {
                .printf "\tFile: 0x%p\n\tOffset: 0x%p\n%msu\n\n",  @ebx-0x30, poi(@ecx+8)&ffff0000, @ebx
            }
            .else
            {
            }
        }
        .else
        {
        }
    }
    .else
    {
    }
}
```

Take a look at some printed data from my system.

```
Vacb #282    0x8194aa70 -> 0xd90c0000
       File: 0x818ed338
       Offset: 0x00ac0000
\$Mft

```

Therefore, the $Mft file is cached at 0xd90c0000 with an offset 0x00ac0000 from its beginning. Take a look at it.

```
!pte 0xd90c0000
               VA d90c0000
PDE at   C0300D90         PTE at C0364300
contains 01D55963       contains 0123EC80
pfn 1d55       -G-DA--KWEV     not valid
                       Proto: FFFFFFFFE148FB00

This PTE refers to the PPTE at E148FB00. Calculate its address manually.
0x123EC80 = 10010001111101 1 0 0 1000000 0
                      |
                      |->PPTE
Index=1001000111110110000000=123EC0 << 2=48FB00; MmPagedPoolStart = e1000000;
48FB00+ e1000000 = e148FB00.

dd e148FB00 l1
e148fb00  87944cd6

PPTE is
0x87944cd6 = 1 0000111100101000l001 1 00110 1011 0
                                    |            |->PTE refers to a subsection
                                    |->describes mapped file
Calculate the address of the subsection.
Index = 000011110010100010011011 = F289B << 3 = 7944D8; MmNonPagedPoolStart = 81181000;
7944D8 + 81181000 = 819154D8 - the address of the section.

dt _subsection 819154D8
nt!_SUBSECTION
   +0x000 ControlArea      : 0x819154a8 _CONTROL_AREA
   +0x004 u                : __unnamed
   +0x008 StartingSector   : 0
   +0x00c NumberOfFullSectors : 0x1000
   +0x010 SubsectionBase   : 0xe148d000 _MMPTE
   +0x014 UnusedPtes       : 0
   +0x018 PtesInSubsection : 0x1000
   +0x01c NextSubsection   : 0x81913660 _SUBSECTION
```

```
!ca 0x819154a8

ControlArea  @ 819154a8
  Segment      e13d66c8  Flink      00000000  Blink        00000000
  Section Ref         1  Pfn Ref         2b6  Mapped Views       3c
  User Ref            0  WaitForDel        0  Flush Count         0
  File Object  818ed338  ModWriteCount     0  System Views       3c

  Flags (8088) NoModifiedWriting File WasPurged

      File: \$Mft

The segment looks like.
dt _SEGMENT e13d66c8
nt!_SEGMENT
   +0x000 ControlArea      : 0x819154a8 _CONTROL_AREA
   +0x004 TotalNumberOfPtes : 0x1b00
   +0x008 NonExtendedPtes  : 0x1000
   +0x00c WritableUserReferences : 0
   +0x010 SizeOfSegment    : 0x1b00000
   +0x018 SegmentPteTemplate : _MMPTE
   +0x01c NumberOfCommittedPages : 0
   +0x020 ExtendInfo       : (null)
   +0x024 SystemImageBase  : (null)
   +0x028 BasedAddress     : (null)
   +0x02c u1               : __unnamed
   +0x030 u2               : __unnamed
   +0x034 PrototypePte     : 0x61564d43 _MMPTE
   +0x038 ThePtes          : [1] _MMPTE

Retrive these value using the shared cache map.

dt _vacb SharedCacheMap 0x8194aa70
nt!_VACB
   +0x004 SharedCacheMap : 0x818c7b08 _SHARED_CACHE_MAP
```

```
Selective output of the structure.

dt _SHARED_CACHE_MAP 0x818c7b08
nt!_SHARED_CACHE_MAP
   +0x008 FileSize         : _LARGE_INTEGER 0x1ae8000
   +0x010 BcbList          : _LIST_ENTRY [ 0x81913a60 - 0x819138b8 ]
   +0x018 SectionSize      : _LARGE_INTEGER 0x1b00000
   +0x044 FileObject       : 0x818ed338 _FILE_OBJECT    //matches the address specified in
//the control_area (!ca output).
   +0x078 Section          : 0xe13d6698 //corresponding section

dt _SECTION_OBJECT Segment 0xe13d6698
nt!_SECTION_OBJECT
   +0x014 Segment : 0xe13d66c8 _SEGMENT_OBJECT   //segment for mapping a file as binary
```

Let's ask the question how does the kernel maps sections into the cache. The answer is located in the *MmMapViewInSystemCache function*. Before analyzing it, point out some facts.

- The cache PTEs start from address that stores in *MmSystemCachePteBase* (usually it matches the address of the beginning of the page table, 0xC0000000).
- Free cache slots are linked to MMPTE_LIST list to provide quick access to them (see WRK for more info about this structure). The pointer to the head of the list is stored in *MmFirstFreeSystemCache*. The field .NextEntry in MMPTE_LIST stores a value that points to the next field (next block of PTEs). This value is relative to *MmSystemCachePteBase*. The *MiInitializeSystemCache* function is responsible for initializing of the PTEs cache list. The PTEs for the cache are reserved by adjacent blocks, i e to cover 256KB, the block is included 64 PTEs, see *MiInitializeSystemCache*.

MmMapViewInSystemCache maps only one cache slot, i e *CapturedViewSize* argument can contain a value-size of no more than 256KB. Below you can see is a pseudocode for the typical behavior of *MmMapViewInSystemCache*. Take a look at the comments, they explain the operations to be performed.

```
#define GetVirtualAddressByPte(PTE) ((PVOID)((ULONG)(PTE) << 10))
```

```
NTSTATUS
MmMapViewInSystemCache (
    IN PVOID SectionToMap, //ptr to a section
    OUT PVOID *CapturedBase, //this variable gets the base address of the mapping
    IN OUT PLARGE_INTEGER SectionOffset, //section offset in a file
    IN OUT PULONG CapturedViewSize //gets the mapping size in bytes
    )
{
    PSECTION Section;
    UINT64 PteOffset;
    UINT64 LastPteOffset;
    PMMPTE PointerPte;
    PMMPTE LastPte;
    PMMPTE ProtoPte;
    PMMPTE LastProto;
    PSUBSECTION Subsection;
    PCONTROL_AREA ControlArea;
    NTSTATUS Status;
    ULONG Waited;
    MMPTE PteContents;
    PFN_NUMBER NumberOfPages;

    Section = SectionToMap;

    // The check verifies that the section was mapped as binary

    if (Section->u.Flags.Image) {
        return STATUS_NOT_MAPPED_DATA;
    }

    ControlArea = Section->Segment->ControlArea;

    //The number of pages needed for mapping
    NumberOfPages = BYTES_TO_PAGES (*CapturedViewSize);

    Subsection = (PSUBSECTION)(ControlArea + 1);

    //Calculate the offset to the first PPTE
    PteOffset = (UINT64)(SectionOffset->QuadPart >> PAGE_SHIFT);
    //The offset to the last PPTE in the subsection
    LastPteOffset = PteOffset + NumberOfPages;

    //Select the appropriate subsection describing the range of the section that
    //was requested for file mapping and fix the indexes of the initial and final
    //PPTE (in the found subsection)

    while (PteOffset >= (UINT64) Subsection->PtesInSubsection)
    {
        PteOffset -= Subsection->PtesInSubsection;
        LastPteOffset -= Subsection->PtesInSubsection;
        Subsection = Subsection->NextSubsection;
    }

    //After that we get the following values, PteOffset - the index of the first PPTE relative
    //to the appropriate subsection, LastPteOffset is the index of the last and Subsection
    //points to the subsection

    //Get a pointer to the first free slot
    PointerPte = MmFirstFreeSystemCache;

    //Remove it from the list by changing the "pointer" to the next
    MmFirstFreeSystemCache = MmSystemCachePteBase + PointerPte->u.List.NextEntry;

    //Increment the number of mapped views for the section
    ControlArea->NumberOfMappedViews += 1;
    ControlArea->NumberOfSystemCacheViews += 1;

    //If needed, create more PPTEs for the section (according to this call,
    //even PPTEs are creted on demand)

    MiAddViewsForSection ((PMSUBSECTION)Subsection,
                          LastPteOffset,
                          OldIrql,
                          &Waited);

    //Get a virtual address by the first PTE (i e it's a reverse conversation from
    //the PTE address to the virtual address, we need to take the PTE address and shift
    //it on the left by 10). Thus we get the base address.

    *CapturedBase = MiGetVirtualAddressByPte (PointerPte);

    //Retrieve a pointer to the first PPTE for the required section offset
    ProtoPte = &Subsection->SubsectionBase[PteOffset];

    //Last PPTE
    LastProto = &Subsection->SubsectionBase[Subsection->PtesInSubsection];

    //The address of the last PTE cache for this mapping
    LastPte = PointerPte + NumberOfPages;
```

```
    //Next, starts a loop to fill cache PTEs so they point to the prototype ones.
    while (PointerPte < LastPte)
   {
        //By this address to the PPTE the function returns PTE content that refers to it.
        //i e we're dealing with a reverse conversion when we need to get an index for the
        //reverse translation of PPTE to PTE.

        PteContents.u.Long = MiProtoAddressForKernelPte (ProtoPte);
        MI_WRITE_INVALID_PTE (PointerPte, PteContents);

        PointerPte += 1;
        ProtoPte += 1;
    }

    return STATUS_SUCCESS;
}
```