在上一篇文章《内核中实现Dump进程转储》中我们实现了ARK工具的转存功能，本篇文章继续以内存为出发点介绍VAD结构，该结构的全程是Virtual Address Descriptor即虚拟地址描述符，VAD是一个AVL自平衡二叉树，树的每一个节点代表一段虚拟地址空间。程序中的代码段，数据段，堆段都会各种占用一个或多个VAD节点，由一个MMVAD结构完整描述。

VAD结构的遍历效果如下:

VAD是Windows操作系统中用于管理进程虚拟地址空间的数据结构之一，全称为Virtual Address Descriptor，即虚拟地址描述符。VAD是一个基于AVL自平衡二叉树的数据结构，它用于维护一段连续的虚拟地址空间。每个VAD节点都描述了一段连续的虚拟地址空间，并包含了该空间的属性信息，如该空间是否可读、可写、可执行等等。

在Windows操作系统中，每个进程都有自己的虚拟地址空间，用于存储该进程的代码、数据和堆栈等信息。这个虚拟地址空间被分为许多段，每个段都由一个或多个VAD节点表示。这些VAD节点构成了一个树形结构，树的根节点表示整个虚拟地址空间，而每个节点表示一段连续的虚拟地址空间。

每个VAD节点都是由一个MMVAD结构体来表示，MMVAD结构体中包含了该节点的各种属性信息，如虚拟地址的起始地址、结束地址、访问权限、保护属性等等。此外，MMVAD结构体还包含了指向下一个和上一个VAD节点的指针，以及指向该节点子节点的指针。这些指针使得VAD节点可以组成一个树形结构，并且可以方便地进行遍历和访问。

总之，VAD结构是Windows操作系统中管理进程虚拟地址空间的重要数据结构之一，它通过构建一个树形结构来管理进程的虚拟地址空间，并提供了丰富的属性信息，使得操作系统可以对虚拟地址空间进行有效的管理和保护。

那么这个VAD结构体在哪里呢?

每一个进程都有自己单独的VAD结构树，这个结构通常在EPROCESS结构里面里面，在内核调试模式下使用dt _EPROCESS可得到如下信息。

EPROCESS 结构体是用于表示操作系统中的一个进程的数据结构，其中包含了许多与该进程相关的信息，包括了该进程的虚拟地址空间描述符树（VAD 结构树）。

在内核调试模式下，使用 dt _EPROCESS 命令可以显示出该结构体的定义和各个字段的信息。其中与 VAD 结构树相关的字段为 VadRoot 和 VadHint。

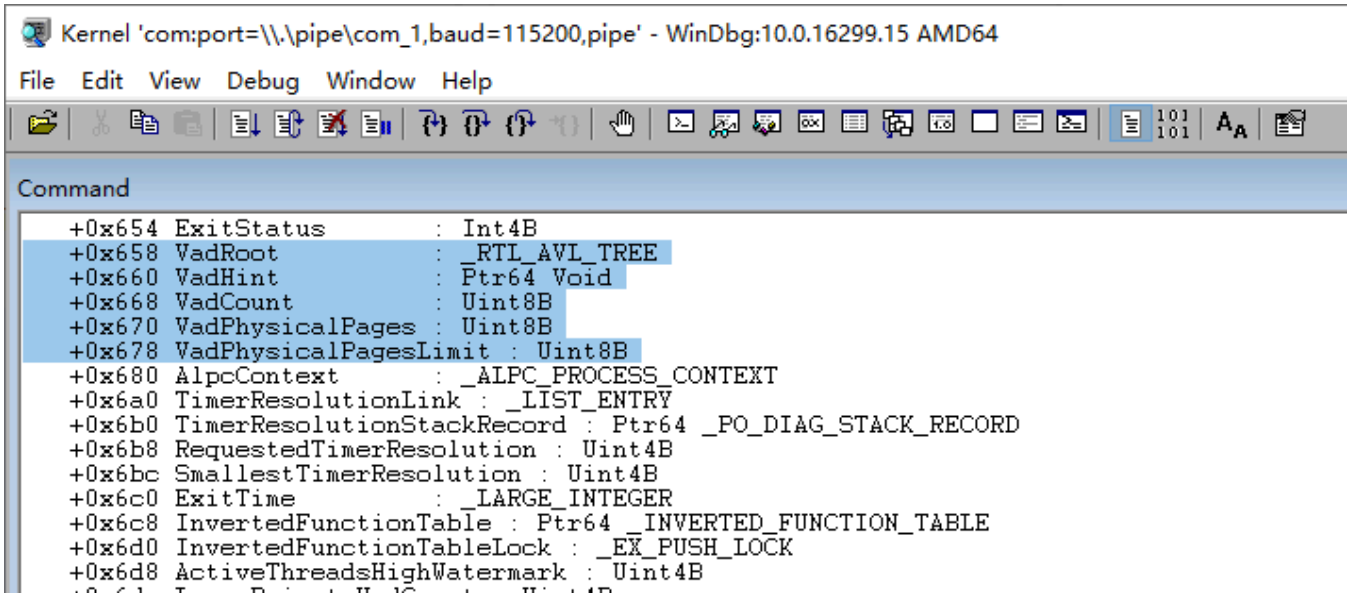
- VadRoot 字段：表示该进程的虚拟地址空间描述符树的根节点，类型为 PMMVAD_SHORT。
- VadHint 字段：表示该进程上一次访问的虚拟地址空间描述符节点，类型为 PMMVAD.

VadRoot 字段指向一个 `MM_AVL_TABLE` 结构体，该结构体包含了一个平衡二叉树，用于存储该进程的虚拟地址空间描述符节点。每个节点都包含了一个虚拟地址空间的起始地址、结束地址，以及一些其他描述符信息，如该区域是否是可读、可写、可执行等。

VadHint 字段则指向该进程最近访问的虚拟地址空间描述符节点，这个字段可以被用来优化访问虚拟地址空间描述符树的性能。

```
lyshark.com 1: kd> dt _EPROCESS
ntdll!_EPROCESS
   +0x500 Vm               : _MMSUPPORT_FULL
   +0x640 MmProcessLinks   : _LIST_ENTRY
   +0x650 ModifiedPageCount : Uint4B
   +0x654 ExitStatus       : Int4B
   +0x658 VadRoot          : _RTL_AVL_TREE
   +0x660 VadHint          : Ptr64 Void
   +0x668 VadCount         : Uint8B
   +0x670 VadPhysicalPages : Uint8B
   +0x678 VadPhysicalPagesLimit : Uint8B
```

可以看到在本系统中VAD的偏移是 `+0x658` 紧跟其后的还有 `vadCount` 的计数等。



VAD结构是如何被添加的?

通常情况下系统调用 `VirtualAllocate` 等申请一段堆内存时，则会在VAD树上增加一个结点 `_MMVAD` 结构体，需要说明的是栈并不受VAD的管理。由系统直接分配空间，并把地址记录在了TEB中。

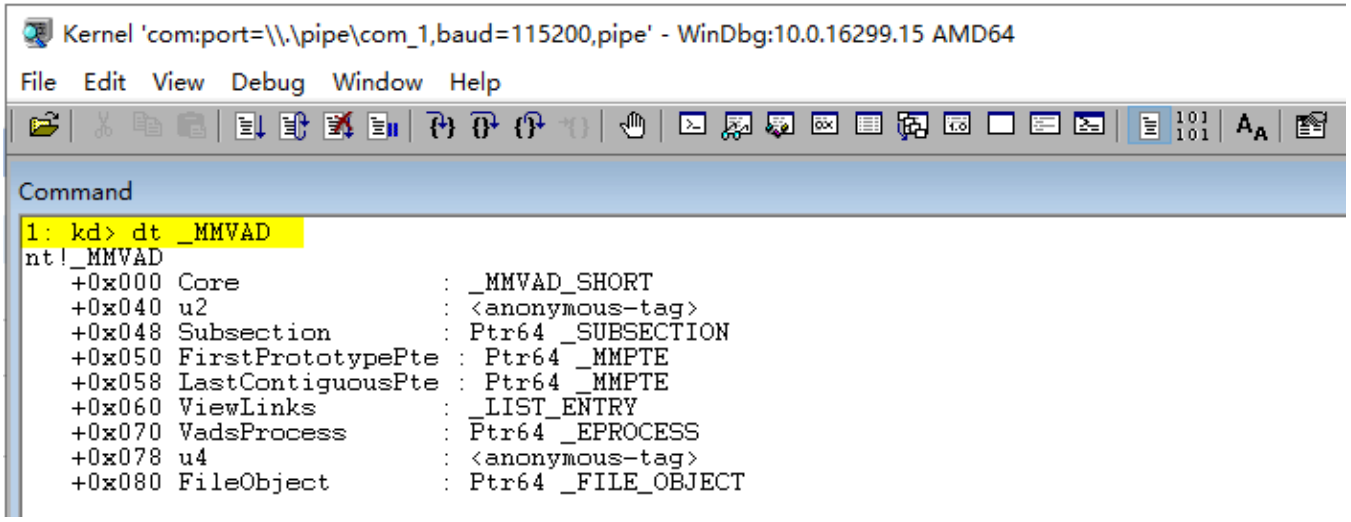
在 Windows 操作系统中，申请堆内存时，系统调用 `VirtualAlloc` 或 `HeapAlloc` 等函数会向操作系统请求一段连续的虚拟地址空间，然后内核会分配一些物理内存页并映射到该虚拟地址空间上，从而完成了内存的分配和管理。

在这个过程中，内核会在当前进程的 `VAD` 树中创建一个新的 `MMVAD` 结构体，用于描述这个新分配的虚拟地址空间的起始地址、大小、保护属性等信息。同时，内核会将这个 `MMVAD` 结构体插入到当前进程的 `VAD` 树中，并通过平衡二叉树的方式来维护这个树的结构，使得树的查询和插入操作都能够以 `O(log n)` 的时间复杂度完成。

但需要注意的是，栈并不受 `VAD` 树的管理，因为栈空间的分配和管理是由系统直接实现的。每个线程都拥有自己的 `TEB（Thread Environment Block）` 结构体，其中包含了该线程的栈空间的起始地址、大小等信息。系统在创建线程时，会为该线程分配一段物理内存页，并映射到该线程的栈空间中，然后将栈空间的起始地址记录在该线程的 `TEB` 中。因此，栈空间的分配和管理是由系统直接实现的，不需要通过 `VAD` 树来管理。

```
lyshark.com 0: kd> dt _MMVAD
nt!_MMVAD
   +0x000 Core             : _MMVAD_SHORT
   +0x040 u2               : <anonymous-tag>
   +0x048 Subsection       : Ptr64 _SUBSECTION
   +0x050 FirstPrototypePte : Ptr64 _MMPTE
   +0x058 LastContiguousPte : Ptr64 _MMPTE
   +0x060 ViewLinks        : _LIST_ENTRY
   +0x070 VadsProcess      : Ptr64 _EPROCESS
   +0x078 u4               : <anonymous-tag>
   +0x080 FileObject       : Ptr64 _FILE_OBJECT
```

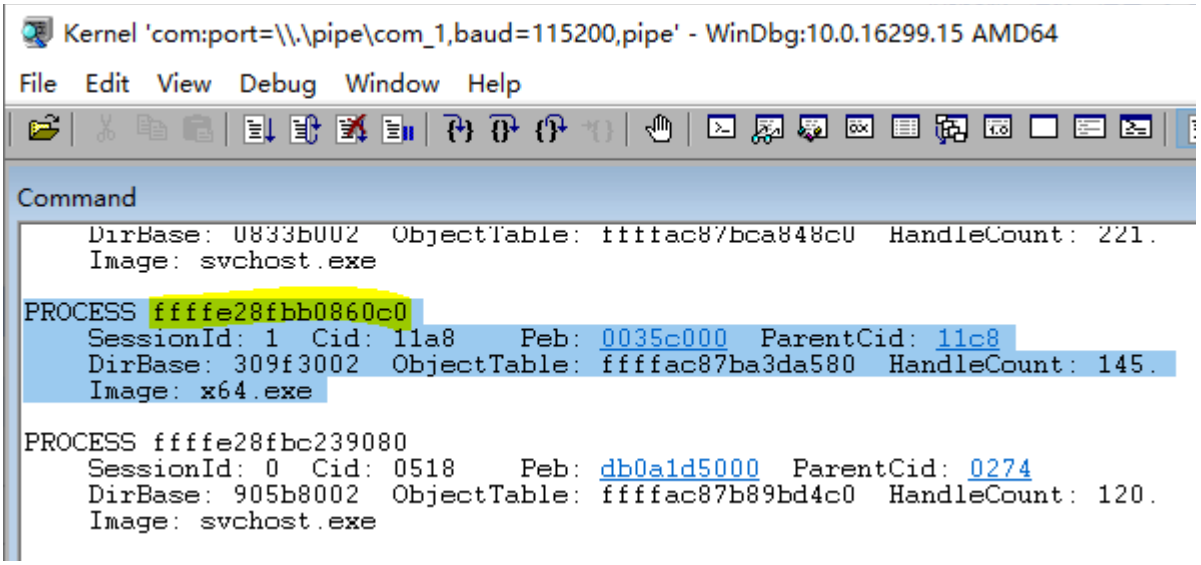
结构体 `MMVAD` 则是每一个 `VAD` 内存块的属性，这个内存结构定义在WinDBG中可看到。



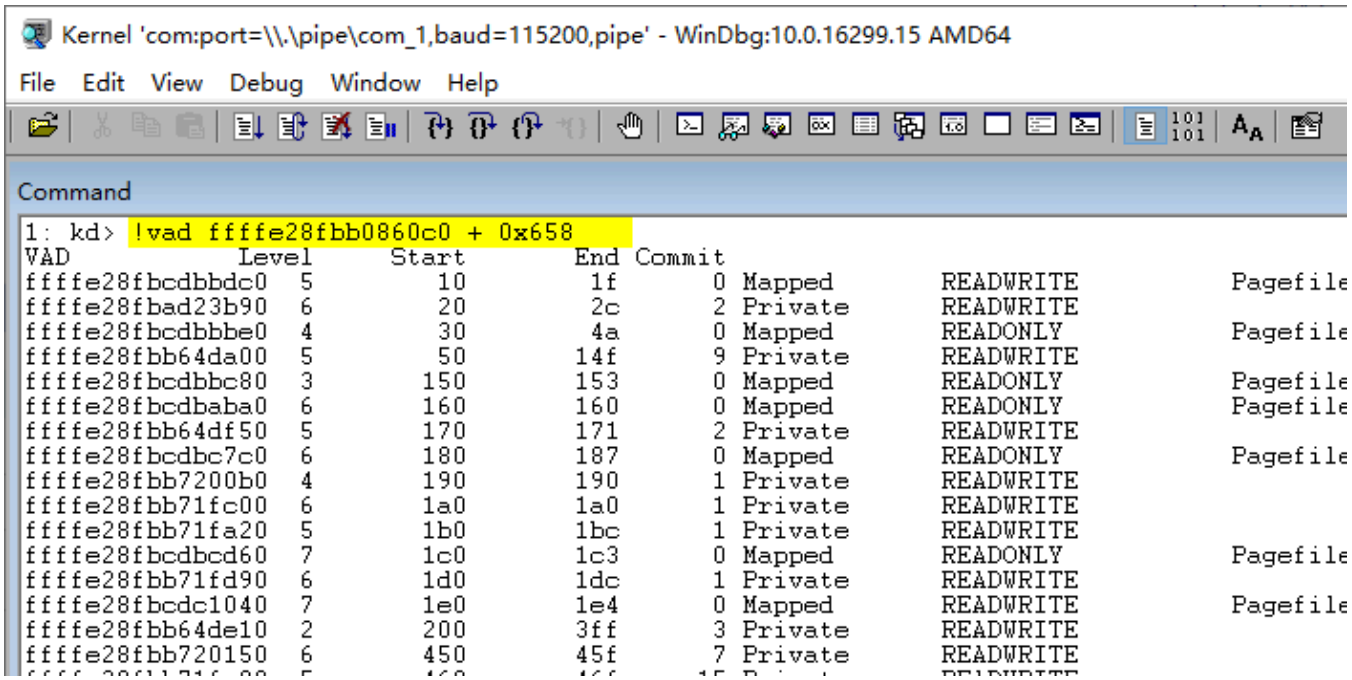
如上在 `EPROCESS` 结构中可以找到VAD结构的相对偏移 `+0x658` 以及进程VAD计数偏移 `+0x668`，我们首先通过 `!process 0 0` 指令得到当前所有进程的 `EPROCESS` 结构，并选中进程。

```
lyshark.com 0: kd> !process 0 0
PROCESS ffffe28fbb0860c0
    SessionId: 1  Cid: 11a8    Peb: 0035c000  ParentCid: 11c8
    DirBase: 309f3002  ObjectTable: ffffac87ba3da580  HandleCount: 145.
    Image: x64.exe
```

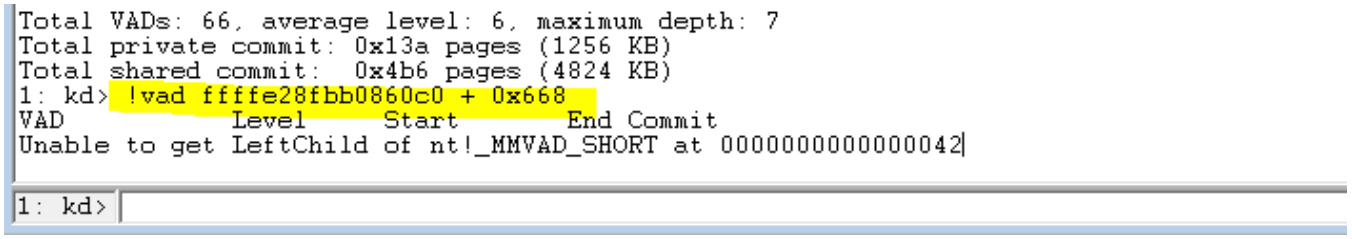
此处的 `ffffe28fbb0860c0` 正是我们所需要的 `EPROCESS` 结构。



当需要得到该进程的VAD结构时，只需要使用`!vad ffffe28fbb0860c0 + 0x658`来显示该进程的VAD树。



至于获取VAD有多少条，则可以直接使用`!vad ffffe28fbb0860c0 + 0x668`来获取到。



既然手动可以遍历出来，那么自动化也并不难，首先定义头文件`vad.h`同样这是微软定义，如果想要的到最新的，自己下载WinDBG调试内核输入命令。

```
#pragma once
#include <ntifs.h>

typedef struct _MM_GRAPHICS_VAD_FLAGS        // 15 elements, 0x4 bytes (sizeof)
{
```

```
    /*0x000*/     ULONG32      Lock : 1;                  // 0 BitPosition
    /*0x000*/     ULONG32      LockContended : 1;         // 1 BitPosition
    /*0x000*/     ULONG32      DeleteInProgress : 1;      // 2 BitPosition
    /*0x000*/     ULONG32      NoChange : 1;              // 3 BitPosition
    /*0x000*/     ULONG32      VadType : 3;               // 4 BitPosition
    /*0x000*/     ULONG32      Protection : 5;            // 7 BitPosition
    /*0x000*/     ULONG32      PreferredNode : 6;         // 12 BitPosition
    /*0x000*/     ULONG32      PageSize : 2;              // 18 BitPosition
    /*0x000*/     ULONG32      PrivateMemoryAlwaysSet : 1; // 20 BitPosition
    /*0x000*/     ULONG32      WriteWatch : 1;            // 21 BitPosition
    /*0x000*/     ULONG32      FixedLargePageSize : 1;    // 22 BitPosition
    /*0x000*/     ULONG32      ZeroFillPagesOptional : 1; // 23 BitPosition
    /*0x000*/     ULONG32      GraphicsAlwaysSet : 1;     // 24 BitPosition
    /*0x000*/     ULONG32      GraphicsUseCoherentBus : 1; // 25 BitPosition
    /*0x000*/     ULONG32      GraphicsPageProtection : 3; // 26 BitPosition

}MM_GRAPHICS_VAD_FLAGS, *PMM_GRAPHICS_VAD_FLAGS;
typedef struct _MM_PRIVATE_VAD_FLAGS         // 15 elements, 0x4 bytes (sizeof)
{
    /*0x000*/     ULONG32      Lock : 1;                  // 0 BitPosition
    /*0x000*/     ULONG32      LockContended : 1;         // 1 BitPosition
    /*0x000*/     ULONG32      DeleteInProgress : 1;      // 2 BitPosition
    /*0x000*/     ULONG32      NoChange : 1;              // 3 BitPosition
    /*0x000*/     ULONG32      VadType : 3;               // 4 BitPosition
    /*0x000*/     ULONG32      Protection : 5;            // 7 BitPosition
    /*0x000*/     ULONG32      PreferredNode : 6;         // 12 BitPosition
    /*0x000*/     ULONG32      PageSize : 2;              // 18 BitPosition
    /*0x000*/     ULONG32      PrivateMemoryAlwaysSet : 1; // 20 BitPosition
    /*0x000*/     ULONG32      WriteWatch : 1;            // 21 BitPosition
```

```
    /*0x000*/     ULONG32      FixedLargePageSize : 1;     // 22 BitPosition
    /*0x000*/     ULONG32      ZeroFillPagesOptional : 1;  // 23 BitPosition
    /*0x000*/     ULONG32      Graphics : 1;               // 24 BitPosition
    /*0x000*/     ULONG32      Enclave : 1;                // 25 BitPosition
    /*0x000*/     ULONG32      ShadowStack : 1;            // 26 BitPosition
}MM_PRIVATE_VAD_FLAGS, *PMM_PRIVATE_VAD_FLAGS;


typedef struct _MMVAD_FLAGS            // 9 elements, 0x4 bytes (sizeof)
{
    /*0x000*/     ULONG32      Lock : 1;             // 0 BitPosition
    /*0x000*/     ULONG32      LockContended : 1;    // 1 BitPosition
    /*0x000*/     ULONG32      DeleteInProgress : 1; // 2 BitPosition
    /*0x000*/     ULONG32      NoChange : 1;         // 3 BitPosition
    /*0x000*/     ULONG32      VadType : 3;          // 4 BitPosition
    /*0x000*/     ULONG32      Protection : 5;       // 7 BitPosition
    /*0x000*/     ULONG32      PreferredNode : 6;    // 12 BitPosition
    /*0x000*/     ULONG32      PageSize : 2;         // 18 BitPosition
    /*0x000*/     ULONG32      PrivateMemory : 1;    // 20 BitPosition
}MMVAD_FLAGS, *PMMVAD_FLAGS;

typedef struct _MM_SHARED_VAD_FLAGS            // 11 elements, 0x4 bytes (sizeof)
{
    /*0x000*/     ULONG32      Lock : 1;                     // 0 BitPosition
    /*0x000*/     ULONG32      LockContended : 1;            // 1 BitPosition
    /*0x000*/     ULONG32      DeleteInProgress : 1;         // 2 BitPosition
    /*0x000*/     ULONG32      NoChange : 1;                 // 3 BitPosition
    /*0x000*/     ULONG32      VadType : 3;                  // 4 BitPosition
    /*0x000*/     ULONG32      Protection : 5;               // 7 BitPosition
    /*0x000*/     ULONG32      PreferredNode : 6;            // 12 BitPosition
    /*0x000*/     ULONG32      PageSize : 2;                 // 18 BitPosition
    /*0x000*/     ULONG32      PrivateMemoryAlwaysClear : 1; // 20 BitPosition
    /*0x000*/     ULONG32      PrivateFixup : 1;             // 21 BitPosition
    /*0x000*/     ULONG32      HotPatchAllowed : 1;          // 22 BitPosition
}MM_SHARED_VAD_FLAGS, *PMM_SHARED_VAD_FLAGS;

typedef struct _MMVAD_FLAGS2            // 7 elements, 0x4 bytes (sizeof)
```

```
{
    /*0x000*/     ULONG32      FileOffset : 24;        // 0 BitPosition
    /*0x000*/     ULONG32      Large : 1;              // 24 BitPosition
    /*0x000*/     ULONG32      TrimBehind : 1;         // 25 BitPosition
    /*0x000*/     ULONG32      Inherit : 1;            // 26 BitPosition
    /*0x000*/     ULONG32      NoValidationNeeded : 1; // 27 BitPosition
    /*0x000*/     ULONG32      PrivateDemandZero : 1;  // 28 BitPosition
    /*0x000*/     ULONG32      Spare : 3;              // 29 BitPosition
}MMVAD_FLAGS2, *PMMVAD_FLAGS2;

typedef struct _MMVAD_SHORT
{
    RTL_BALANCED_NODE VadNode;
    UINT32 StartingVpn;                /*0x18*/
    UINT32 EndingVpn;                  /*0x01C*/
    UCHAR StartingVpnHigh;
    UCHAR EndingVpnHigh;
    UCHAR CommitChargeHigh;
    UCHAR SpareNT64VadUChar;
    INT32 ReferenceCount;
    EX_PUSH_LOCK PushLock;             /*0x028*/
    struct
    {
        union
        {
            ULONG_PTR flag;
            MM_PRIVATE_VAD_FLAGS PrivateVadFlags;                        /*0x030*/
            MMVAD_FLAGS  VadFlags;
            MM_GRAPHICS_VAD_FLAGS GraphicsVadFlags;
            MM_SHARED_VAD_FLAGS   SharedVadFlags;
        }Flags;

    }u1;

    PVOID EventList;                         /*0x038*/

}MMVAD_SHORT, *PMMVAD_SHORT;

typedef struct _MMADDRESS_NODE
{
    ULONG64 u1;
    struct _MMADDRESS_NODE* LeftChild;
    struct _MMADDRESS_NODE* RightChild;
    ULONG64 StartingVpn;
    ULONG64 EndingVpn;
}MMADDRESS_NODE, *PMMADDRESS_NODE;

typedef struct _MMEXTEND_INFO     // 2 elements, 0x10 bytes (sizeof)
{
    /*0x000*/     UINT64       CommittedSize;
    /*0x008*/     ULONG32      ReferenceCount;
    /*0x00C*/     UINT8        _PADDING0_[0x4];
}MMEXTEND_INFO, *PMMEXTEND_INFO;
```

```
struct _SEGMENT
{
    struct _CONTROL_AREA* ControlArea;
    ULONG TotalNumberOfPtes;
    ULONG SegmentFlags;
    ULONG64 NumberOfCommittedPages;
    ULONG64 SizeOfSegment;
    union
    {
        struct _MMEXTEND_INFO* ExtendInfo;
        void* BasedAddress;
    }u;
    ULONG64 SegmentLock;
    ULONG64 u1;
    ULONG64 u2;
    PVOID* PrototypePte;
    ULONGLONG ThePtes[0x1];
};

typedef struct _EX_FAST_REF
{
    union
    {
        PVOID Object;
        ULONG_PTR RefCnt : 3;
        ULONG_PTR Value;
    };
} EX_FAST_REF, *PEX_FAST_REF;

typedef struct _CONTROL_AREA                    // 17 elements, 0x80 bytes (sizeof)
{
    /*0x000*/     struct _SEGMENT* Segment;
    union                                       // 2 elements, 0x10 bytes (sizeof)
    {
        /*0x008*/         struct _LIST_ENTRY ListHead;             // 2 elements, 0x10
bytes (sizeof)
        /*0x008*/         VOID*        AweContext;
    };
    /*0x018*/     UINT64       NumberOfSectionReferences;
    /*0x020*/     UINT64       NumberOfPfnReferences;
    /*0x028*/     UINT64       NumberOfMappedViews;
    /*0x030*/     UINT64       NumberOfUserReferences;
    /*0x038*/     ULONG32 u;                  // 2 elements, 0x4 bytes (sizeof)
    /*0x03C*/     ULONG32 u1;                 // 2 elements, 0x4 bytes (sizeof)
    /*0x040*/     struct _EX_FAST_REF FilePointer;             // 3 elements, 0x8 bytes
(sizeof)
    // 4 elements, 0x8 bytes (sizeof)
}CONTROL_AREA, *PCONTROL_AREA;

typedef struct _SUBSECTION_
{
    struct _CONTROL_AREA* ControlArea;
```

```
}SUBSECTION, *PSUBSECTION;

typedef struct _MMVAD
{
    MMVAD_SHORT Core;
    union                   /*0x040*/
    {
        UINT32 LongFlags2;
        //现在用不到省略
        MMVAD_FLAGS2 VadFlags2;

    }u2;
    PSUBSECTION Subsection;              /*0x048*/
    PVOID FirstPrototypePte;        /*0x050*/
    PVOID LastContiguousPte;        /*0x058*/
    LIST_ENTRY ViewLinks;           /*0x060*/
    PEPROCESS VadsProcess;          /*0x070*/
    PVOID u4;                       /*0x078*/
    PVOID FileObject;               /*0x080*/
}MMVAD, *PMMVAD;

typedef struct _RTL_AVL_TREE         // 1 elements, 0x8 bytes (sizeof)
{
    /*0x000*/     struct _RTL_BALANCED_NODE* Root;
}RTL_AVL_TREE, *PRTL_AVL_TREE;

typedef struct _VAD_INFO_
{
    ULONG_PTR pVad;
    ULONG_PTR startVpn;
    ULONG_PTR endVpn;
    ULONG_PTR pFileObject;
    ULONG_PTR flags;
}VAD_INFO, *PVAD_INFO;

typedef struct _ALL_VADS_
{
    ULONG nCnt;
    VAD_INFO VadInfos[1];
}ALL_VADS, *PALL_VADS;

typedef struct _MMSECTION_FLAGS                        // 27 elements, 0x4 bytes (sizeof)
{
    /*0x000*/     UINT32       BeingDeleted : 1;                    // 0 BitPosition

    /*0x000*/     UINT32       BeingCreated : 1;                    // 1 BitPosition

    /*0x000*/     UINT32       BeingPurged : 1;                     // 2 BitPosition

    /*0x000*/     UINT32       NoModifiedWriting : 1;               // 3 BitPosition

    /*0x000*/     UINT32       FailAllIo : 1;                       // 4 BitPosition
```

```
    /*0x000*/     UINT32       Image : 1;                          // 5 BitPosition
    /*0x000*/     UINT32       Based : 1;                          // 6 BitPosition
    /*0x000*/     UINT32       File : 1;                           // 7 BitPosition
    /*0x000*/     UINT32       AttemptingDelete : 1;               // 8 BitPosition
    /*0x000*/     UINT32       PrefetchCreated : 1;                // 9 BitPosition
    /*0x000*/     UINT32       PhysicalMemory : 1;                 // 10 BitPosition
    /*0x000*/     UINT32       ImageControlAreaOnRemovableMedia : 1; // 11 BitPosition
    /*0x000*/     UINT32       Reserve : 1;                        // 12 BitPosition
    /*0x000*/     UINT32       Commit : 1;                         // 13 BitPosition
    /*0x000*/     UINT32       NoChange : 1;                       // 14 BitPosition
    /*0x000*/     UINT32       WasPurged : 1;                      // 15 BitPosition
    /*0x000*/     UINT32       UserReference : 1;                  // 16 BitPosition
    /*0x000*/     UINT32       GlobalMemory : 1;                   // 17 BitPosition
    /*0x000*/     UINT32       DeleteOnClose : 1;                  // 18 BitPosition
    /*0x000*/     UINT32       FilePointerNull : 1;                // 19 BitPosition
    /*0x000*/     ULONG32      PreferredNode : 6;                  // 20 BitPosition
    /*0x000*/     UINT32       GlobalOnlyPerSession : 1;           // 26 BitPosition
    /*0x000*/     UINT32       UserWritable : 1;                   // 27 BitPosition
    /*0x000*/     UINT32       SystemVaAllocated : 1;              // 28 BitPosition
    /*0x000*/     UINT32       PreferredFsCompressionBoundary : 1;  // 29 BitPosition
    /*0x000*/     UINT32       UsingFileExtents : 1;               // 30 BitPosition
    /*0x000*/     UINT32       PageSize64K : 1;                    // 31 BitPosition

}MMSECTION_FLAGS, *PMMSECTION_FLAGS;

typedef struct _SECTION                          // 9 elements, 0x40 bytes (sizeof)
{
    /*0x000*/     struct _RTL_BALANCED_NODE SectionNode;       // 6 elements, 0x18 bytes
(sizeof)
    /*0x018*/     UINT64       StartingVpn;
    /*0x020*/     UINT64       EndingVpn;
    /*0x028*/     union {
```

```
        PCONTROL_AREA   ControlArea;
        PVOID   FileObject;

    }u1;                    // 4 elements, 0x8 bytes (sizeof)
    /*0x030*/     UINT64       SizeOfSection;
    /*0x038*/     union {
        ULONG32 LongFlags;
        MMSECTION_FLAGS Flags;
    }u;                     // 2 elements, 0x4 bytes (sizeof)
    struct                                      // 3 elements, 0x4 bytes (sizeof)
    {
        /*0x03C*/         ULONG32      InitialPageProtection : 12; // 0 BitPosition

        /*0x03C*/         ULONG32      SessionId : 19;             // 12 BitPosition

        /*0x03C*/         ULONG32      NoValidationNeeded : 1;     // 31 BitPosition

    };
}SECTION, *PSECTION;
```

引入`vad.h`头文件，并写入如下代码，此处的`eprocess_offset_VadRoot`以及`eprocess_offset_VadCount`则是上方得出的相对于`EPROCESS`结构的偏移值，每个系统都不一样，版本不同偏移值会不同。

```
#include "vad.h"
#include <ntifs.h>

// 定义VAD相对于EProcess头部偏移值
#define eprocess_offset_VadRoot 0x658
#define eprocess_offset_VadCount 0x668

VOID EnumVad(PMMVAD Root, PALL_VADS pBuffer, ULONG nCnt)
{
    if (!Root || !pBuffer || !nCnt)
    {
        return;
    }

    __try
    {
        if (nCnt > pBuffer->nCnt)
        {
            // 得到起始页与结束页
            ULONG64 endptr = (ULONG64)Root->Core.EndingVpnHigh;
            endptr = endptr << 32;

            ULONG64 startptr = (ULONG64)Root->Core.StartingVpnHigh;
            startptr = startptr << 32;

            // 得到根节点
            pBuffer->VadInfos[pBuffer->nCnt].pVad = (ULONG_PTR)Root;

            // 起始页: startingVpn * 0x1000
```

```
            pBuffer->VadInfos[pBuffer->nCnt].startVpn = (startptr | Root->Core.StartingVpn)
<< PAGE_SHIFT;

            // 结束页: EndVpn * 0x1000 + 0xfff
            pBuffer->VadInfos[pBuffer->nCnt].endVpn = ((endptr | Root->Core.EndingVpn) <<
PAGE_SHIFT) + 0xfff;

            // VAD标志 928 = Mapped    1049088 = Private    ....
            pBuffer->VadInfos[pBuffer->nCnt].flags = Root->Core.u1.Flags.flag;

            // 验证节点可读性
            if (MmIsAddressValid(Root->Subsection) && MmIsAddressValid(Root->Subsection-
>ControlArea))
            {
                if (MmIsAddressValid((PVOID)((Root->Subsection->ControlArea-
>FilePointer.Value >> 4) << 4)))
                {
                    pBuffer->VadInfos[pBuffer->nCnt].pFileObject = ((Root->Subsection-
>ControlArea->FilePointer.Value >> 4) << 4);
                }
            }
            pBuffer->nCnt++;
        }

        if (MmIsAddressValid(Root->Core.VadNode.Left))
        {
            // 递归枚举左子树
            EnumVad((PMMVAD)Root->Core.VadNode.Left, pBuffer, nCnt);
        }

        if (MmIsAddressValid(Root->Core.VadNode.Right))
        {
            // 递归枚举右子树
            EnumVad((PMMVAD)Root->Core.VadNode.Right, pBuffer, nCnt);
        }
    }
    __except (1)
    {
    }
}

BOOLEAN EnumProcessVad(ULONG Pid, PALL_VADS pBuffer, ULONG nCnt)
{
    PEPROCESS Peprocess = 0;
    PRTL_AVL_TREE Table = NULL;
    PMMVAD Root = NULL;

    // 通过进程PID得到进程EProcess
    if (NT_SUCCESS(PsLookupProcessByProcessId((HANDLE)Pid, &Peprocess)))
    {
        // 与偏移相加得到VAD头节点
        Table = (PRTL_AVL_TREE)((UCHAR*)Peprocess + eprocess_offset_VadRoot);
        if (!MmIsAddressValid(Table) || !eprocess_offset_VadRoot)
```

```
        {
            return FALSE;
        }

        __try
        {
            // 取出头节点
            Root = (PMMVAD)Table->Root;

            if (nCnt > pBuffer->nCnt)
            {
                // 得到起始页与结束页
                ULONG64 endptr = (ULONG64)Root->Core.EndingVpnHigh;
                endptr = endptr << 32;

                ULONG64 startptr = (ULONG64)Root->Core.StartingVpnHigh;
                startptr = startptr << 32;

                pBuffer->VadInfos[pBuffer->nCnt].pVad = (ULONG_PTR)Root;

                // 起始页: startingVpn * 0x1000
                pBuffer->VadInfos[pBuffer->nCnt].startVpn = (startptr | Root-
>Core.StartingVpn) << PAGE_SHIFT;

                // 结束页: EndVpn * 0x1000 + 0xfff
                pBuffer->VadInfos[pBuffer->nCnt].endVpn = (endptr | Root->Core.EndingVpn) <<
PAGE_SHIFT;
                pBuffer->VadInfos[pBuffer->nCnt].flags = Root->Core.u1.Flags.flag;

                if (MmIsAddressValid(Root->Subsection) && MmIsAddressValid(Root->Subsection-
>ControlArea))
                {
                    if (MmIsAddressValid((PVOID)((Root->Subsection->ControlArea-
>FilePointer.Value >> 4) << 4)))
                    {
                        pBuffer->VadInfos[pBuffer->nCnt].pFileObject = ((Root->Subsection-
>ControlArea->FilePointer.Value >> 4) << 4);
                    }
                }
                pBuffer->nCnt++;
            }

            // 枚举左子树
            if (Table->Root->Left)
            {
                EnumVad((MMVAD*)Table->Root->Left, pBuffer, nCnt);
            }

            // 枚举右子树
            if (Table->Root->Right)
            {
                EnumVad((MMVAD*)Table->Root->Right, pBuffer, nCnt);
            }
```

```
        }
        __finally
        {
            ObDereferenceObject(Peprocess);
        }
    }
    else
    {
        return FALSE;
    }

    return TRUE;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint(("Uninstall Driver Is OK \n"));
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint(("hello lyshark \n"));

    typedef struct
    {
        ULONG nPid;
        ULONG nSize;
        PALL_VADS pBuffer;
    }VADProcess;

    __try
    {
        VADProcess vad = { 0 };

        vad.nPid = 4520;

        // 默认有1000个线程
        vad.nSize = sizeof(VAD_INFO) * 0x5000 + sizeof(ULONG);

        // 分配临时空间
        vad.pBuffer = (PALL_VADS)ExAllocatePool(PagedPool, vad.nSize);

        // 根据传入长度得到枚举数量
        ULONG nCount = (vad.nSize - sizeof(ULONG)) / sizeof(VAD_INFO);

        // 枚举VAD
        EnumProcessVad(vad.nPid, vad.pBuffer, nCount);


        // 输出VAD
        for (size_t i = 0; i < vad.pBuffer->nCnt; i++)
        {
            DbgPrint("StartVPN = %p | ", vad.pBuffer->VadInfos[i].startVpn);
```

```
            DbgPrint("EndVPN = %p | ", vad.pBuffer->VadInfos[i].endVpn);
            DbgPrint("PVAD = %p | ", vad.pBuffer->VadInfos[i].pVad);
            DbgPrint("Flags = %d | ", vad.pBuffer->VadInfos[i].flags);
            DbgPrint("pFileObject = %p \n", vad.pBuffer->VadInfos[i].pFileObject);
        }
    }
    __except (1)
    {
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

程序运行后输出效果如下图所示;