<http://www.cppblog.com/ay19880703/archive/2011/10/30/159364.html>

WINDBG的堆调试--了解HEAP组织

HEAP的概念

堆栈堆栈，在操作系统内存中有两种存储空间，一个是堆，一个是栈。堆主要用于存储用户动态分配的变量，而栈呢，则是存储我们程序过程中的临时变量。当然栈的作用远不止用作存储变量，但这不是我们这篇文章的讨论内容。

堆（HEAP）的分配，使用，回收都是通过微软的API来管理的，最常见的API是malloc和new。在往底层走一点呢，这两个函数都会调用HeapAlloc（RtlAllocateHeap）。同样的相关函数还有HeapFree用来释放堆，HeapCreate用来创建自己的私有堆。下面是这些函数的调用链：

HeapCreate->RtlCreateHeap->ZwAllocateVirtualMemory (这里会直接申请一大片内存,至于申请多大内存,由进程PEB结构中的字段觉得，HeapSegmentReserve字段指出要申请多大的虚拟内存，HeapSegmentCommit指明要提交多大内存，对虚拟内存的申请和提交概念不清楚的童鞋，请参见windows核心编程相关内容~)

HeapAlloc->RtlAllocateHeap（至于这里申请的内存，由于HeapCreate已经申请了一大片内存，堆管理器这片内存中划分一块出来以满足申请的需要。这一步申请操作是堆管理器自己维护的，仅当申请内存不够的时候才会再次调用ZwAllocateVirtualMemory ）

HeapFree->RtlFreeHeap （对于释放的内存，堆管理器只是简单的把这块内存标志位已释放让后加入到空闲列表中，仅当空闲的内存达到一定阀值的时候会调用ZwFreeVirtualMeMory ）

HeapDestroy->RtlDestroyHeap->ZwFreeVirtualMeMory （销毁我们申请的堆）

如何找到我们的HEAP信息？

WINDBG观察堆

源码：

#include "windows.h"

int main()

{

HANDLE heap\_handle = HeapCreate( NULL , 0x1000 , 0x2000 ) ;

char \*buffer = (char\*)HeapAlloc(heap\_handle , NULL , 128) ;

char \*buffer1 = (char\*)HeapAlloc(heap\_handle , NULL , 121) ;

HeapFree(heap\_handle, 0 , buffer ) ;

HeapFree(heap\_handle, 0 , buffer1 ) ;

HeapDestroy( heap\_handle) ;

return 0 ;

}

该源码生成编译生成heap.exe，然后用windbg调试这个程序，在main函数下断，紧接着执行第五行语句，执行结果如下

0:000> p

eax=002e1ca0 ebx=00000000 ecx=6d29b6f0 edx=00000000 esi=00000001 edi=01033374

eip=01031012 esp=0022fe8c ebp=0022feac iopl=0 nv up ei pl nz na po nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202

heap!main+0x12:

01031012 ff150c200301 call dword ptr [heap!\_imp\_\_HeapCreate (0103200c)] ds:0023:0103200c={kernel32!HeapCreateStub (769a29d7)}

0:000> p

eax=002c0000 ebx=00000000 ecx=77429897 edx=77498500 esi=00000001 edi=01033374

eip=01031018 esp=0022fe98 ebp=0022feac iopl=0 nv up ei pl nz na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206

heap!main+0x18:

01031018 8945fc mov dword ptr [ebp-4],eax ss:0023:0022fea8=6d222201

0:000> !heap

Index Address Name Debugging options enabled

1: 00300000

2: 00010000

3: 00020000

4: 002e0000

5: 002c0000

HeapCreate执行的返回值存放在eax处，这个函数返回了一个堆句柄：0x002c0000。用!heap命令查看可以看到第五个堆就是我们创建的堆句柄了。

每个进程都存在多个堆，我们也可以通过PEB结构来得到进程中存在的堆，结果和!heap命令显示的内容是一样的。

heap!\_PEB

+0x018 ProcessHeap : 0x00300000 Void ; 进程的默认堆

+0x068 NtGlobalFlag : 0 ; 这个标志位记录了当前堆调试模式,0为普通调试模式

+0x078 HeapSegmentReserve : 0x100000 ; 进程在新建堆的时候默认申请的虚拟内存大小

+0x07c HeapSegmentCommit : 0x2000 ; 进程在每次申请提交的虚拟内存大小，在提交的内存用完后，进程会又在一次提交HeapSegmentCommit中指定的内存大小

+0x080 HeapDeCommitTotalFreeThreshold : 0x10000 ; 当释放的内存大小大于这个阀值，就进行内存解除提交操作

+0x084 HeapDeCommitFreeBlockThreshold : 0x1000 ; 当一次性释放的块大小超过这个阀值，就进行内存解除提交操作，只有当满足这两个条件时才会调用ZwFreeVirtualMeMory 释放物理内存

+0x088 NumberOfHeaps : 5 ; 当前进程的堆数目,这个数目对应着!heap命令的堆显示个数

+0x08c MaximumNumberOfHeaps : 0x10 ; 进程所能运行的最大堆数目,若堆数目超过这个值估计HeapCreate就失败了吧

+0x090 ProcessHeaps : 0x77498500 -> 0x00300000 Void ;存储堆句柄的数组,这里我们可以得到进程的所有堆句柄

我们可以输入如下命令来查看现有的堆句柄

0:000> dd 0x77498500

77498500 00300000 00010000 00020000 002e0000

77498510 002c0000 00000000 00000000 00000000

77498520 00000000 00000000 00000000 00000000

77498530 00000000 00000000 00000000 00000000

77498540 00000000 77498340 7749bb08 77498220

77498550 00000000 00000000 00000000 00000000

77498560 77498220 00317bd0 00000000 00000000

77498570 00000000 00000000 00000000 00000000

可以看得到这里面的内容和!heap命令的输出结果是一样的

而堆句柄的存放范围,从MaximumNumberOfHeaps 上来看,就是77498500-77498540这0x40个字节，因为每个堆句柄占4个字节，0x10个堆句柄的存放空间就是0x40。

HEAP的组织结构

堆的管理，我们可以理解为一个内存池，它申请一大块空间，然后负责接管应用程序的申请释放等请求。只有在创建堆，释放堆（注意！是释放堆，不是堆中的空间！）在这之前，我们需要对堆有关的数据结构做一些解释

我这里观察到的HEAP结构，HEAP\_SEGMENT结构和HEAP\_ENTRY结构都和软件调试里面描述的不一样，当年奎哥写软件调试的时候估计还没用上WIN7吧。。。我的演示系统是WIN7

HeapCreate函数返回的堆句柄其实就是一个指向堆管理结构的指针，每个堆都会涉及到这样三个结构：HEAP,HEAP\_SEGMENT,HEAP\_ENTRY

HEAP\_ENTRY结构：

在堆管理中，每一块申请下来的内存都会有下面所示的固定模式：

HEAP\_ENTRY（8 bytes）

我们new或malloc分配的空间

固定填充空间

这个结构用来记录所分配的空间的信息，包括用户申请的空间，填充的空间，所在的段号等等信息。所以我们new或者malloc的地址减去8就指向该结构。第三部分的固定填充空间是为了内存对齐而生成的，当然这部分空间还有一部分是用来额外记录这块内存的其它信息，这里就不详细做介绍了。

HEAP\_SEGMENT结构：

我们可以这么认为，堆申请内存的大小是以段为单位的，当新建一个堆的时候，系统会默认为这个堆分配一个段叫0号段，通过刚开始的new和malloc分配的空间都是在这个段上分配的，当这个段用完的时候，如果当初创建堆的时候指明了HEAP\_GROWABLE这个标志，那么系统会为这个堆在再分配一个段，这个时候新分配的段就称为1号段了，以下以此类推。每个段的开始初便是HEAP\_SEGMENT结构的首地址，由于这个结构也是申请的一块内存，所以它前面也会有个HEAP\_ENTRY结构：

HEAP\_ENTRY（8 bytes）

HEAP\_SEGMENT

HEAP\_ENTRY（8 bytes）

我们new或malloc分配的空间

固定填充空间

HEAP\_SEGMENT结构会记录段的一些基本信息，该段申请的大小，已经提交内存的大小，第一个HEAP\_ENTRY结构的入口点。（我观察看貌似段申请的内存并不会一次性全部提交，而是每次提交一个页的大小，比如一个段大小2个页，那么它会先提交一个页内存，若用完了再提交一个页的内存，若内存还用完了那就新建一个段，这个新建的段也会是先提交一个页内存。）但是0号段很特别，这个段的起始地址就是堆句柄指针指向的值，也就是说，HeapCreate返回的堆句柄总是指向0号段，为什么呢？因为HEAP结构是HEAP\_ENTRY,HEAP\_SEGMENT的合体加长版~

HEAP结构：

HEAP结构则是记录了这个堆的信息，这个结构可以找到HEAP\_SEGMENT链表入口，空闲内存链表的入口，内存分配粒度等等信息。HEAP的首地址便是堆句柄的值，但是堆句柄的值又是0号段的首地址也是堆句柄，何解？其实很简单，0号段的HEAP\_SEGMENT就在HEAP结构里面，HEAP结构类定义如这样：

struct \_HEAP

{

\_HEAP\_ENTRY Entry ; //HEAP\_ENTRY结构，用来描述存储HEAP内存块大小等信息的

\_HEAP\_SEGMENT Segment ; //0号段的首地址

…… //对于该HEAP的描述信息

} ;

在我们看来，内存组织结构应该如下所示：

HEAP\_ENTRY（8 bytes）

HEAP\_SEGMENT

HEAP

更确切的说，HEAP结构中本身就包含了HEAP\_ENTRY和HEAP\_SEGMENT，HEAP\_ENTRY结构是HEAP的第一个数据成员，HEAP\_SEGMENT是它第二个数据成员。而对于HEAP\_SEGMENT,它的第一个数据成员便是HEAP\_ENTRY。这里为了方便理解，才在内存组织结构中把它们拆开展示。（注：这里是win7的情况，和软件调试这本书中所描述的有一些差异，也属正常现象，毕竟这部分结构微软并未公开）

用WINDBG观察HEAP结构

在之前已经演示了如何从PEB结构中找到所有的堆句柄，可以看到002c0000便是我们创建的句柄。然后我们执示例程序的第7行代码。执行完后结果如下：

0:000> p

eax=002c0000 ebx=00000000 ecx=77429897 edx=77498500 esi=00000001 edi=01033374

eip=01031026 esp=0022fe8c ebp=0022feac iopl=0 nv up ei pl nz na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206

heap!main+0x26:

01031026 ff1500200301 call dword ptr [heap!\_imp\_\_HeapAlloc (01032000)] ds:0023:01032000={ntdll!RtlAllocateHeap (774120b5)}

0:000> p

eax=002c0590 ebx=00000000 ecx=774134b4 edx=002c0180 esi=00000001 edi=01033374

eip=0103102c esp=0022fe98 ebp=0022feac iopl=0 nv up ei pl zr na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246

heap!main+0x2c:

0103102c 8945f0 mov dword ptr [ebp-10h],eax ss:0023:0022fe9c={heap!envp (0103301c)}

可以看到EAX保存的返回值为002c0590。我们通过两种途径来观察我们申请的内存，通过!heap命令观察和通过dt命令观察

通过!heap命令观察

输入命令!heap –a 2c0590得到的结果如下：

0:000> !heap -a 2c0000

Index Address Name Debugging options enabled

5: 002c0000

Segment at 002c0000 to 002c2000 (00001000 bytes committed)

Flags: 00001000

ForceFlags: 00000000

Granularity: 8 bytes

Segment Reserve: 00100000

Segment Commit: 00002000

DeCommit Block Thres: 00000200

DeCommit Total Thres: 00002000

Total Free Size: 0000013a

Max. Allocation Size: 7ffdefff

Lock Variable at: 002c0138

Next TagIndex: 0000

Maximum TagIndex: 0000

Tag Entries: 00000000

PsuedoTag Entries: 00000000

Virtual Alloc List: 002c00a0

Uncommitted ranges: 002c0090

002c1000: 00001000 (4096 bytes)

FreeList[ 00 ] at 002c00c4: 002c0618 . 002c0618

002c0610: 00088 . 009d0 [100] - free

Segment00 at 002c0000:

Flags: 00000000

Base: 002c0000

First Entry: 002c0588

Last Entry: 002c2000

Total Pages: 00000002

Total UnCommit: 00000001

Largest UnCommit:00000000

UnCommitted Ranges: (1)

Heap entries for Segment00 in Heap 002c0000

002c0000: 00000 . 00588 [101] - busy (587)

002c0588: 00588 . 00088 [101] - busy (80)

002c0610: 00088 . 009d0 [100]

002c0fe0: 009d0 . 00020 [111] - busy (1d)

002c1000: 00001000 - uncommitted bytes.

这个命令分别提炼出了HEAP（绿色区域）,HEAP\_SEGMENT（红色区域）和HEAP\_ENTRY（灰色区域）结构中的信息。虽然在灰色区域中，我们找不到2c0590，但是找到了一个2c0588，这个正是2c0590-8的结果，也就是说最右边的地址是每个HEAP\_ENTRY的首地址，接着00588这个字段表示了前面一个HEAP\_ENTRY所占用的大小，后面的0088表示这个内存块的总大小，即我们申请的内存+HEAP\_ENTRY（128+8=0x80+0x8=0x88），[101]是这块内存的标志位，最右边一位为1表示该内存块被占用。然后busy（80）就是解释说这块内存是被占用的（非空闲的），它申请的内存为0x80，转化成十进制正好就是我们申请的128字节大小。

但是这里用dt \_HEAP\_ENTRY 2c0588命令却没办法查看对应的结构信息，真是怪哉，有篇博文也提到win2008中HEAP相关结构也有变，看来到NT6后，HEAP结构变得不小，起码windbg中直接dt HEAP\_ENTRY是无法原始数据的了，貌似对HEAP\_ENTRY做了编码。

通过dt命令观察

同样的，已知HEAP的首地址，那么先从HEAP下手好了，dt \_HEAP 002c0000可以显示HEAP的数据结构

ntdll!\_HEAP

+0x000 Entry : \_HEAP\_ENTRY

+0x008 SegmentSignature : 0xffeeffee

+0x00c SegmentFlags : 0

+0x010 SegmentListEntry : \_LIST\_ENTRY [ 0x2c00a8 - 0x2c00a8 ]

+0x018 Heap : 0x002c0000 \_HEAP

+0x01c BaseAddress : 0x002c0000 Void

+0x020 NumberOfPages : 2

+0x024 FirstEntry : 0x002c0588 \_HEAP\_ENTRY

+0x028 LastValidEntry : 0x002c2000 \_HEAP\_ENTRY

+0x02c NumberOfUnCommittedPages : 1

+0x030 NumberOfUnCommittedRanges : 1

+0x034 SegmentAllocatorBackTraceIndex : 0

+0x036 Reserved : 0

+0x038 UCRSegmentList : \_LIST\_ENTRY [ 0x2c0ff0 - 0x2c0ff0 ]

+0x040 Flags : 0x1000

+0x044 ForceFlags : 0

+0x048 CompatibilityFlags : 0

+0x04c EncodeFlagMask : 0x100000

+0x050 Encoding : \_HEAP\_ENTRY

+0x058 PointerKey : 0x17c06e63

+0x05c Interceptor : 0

+0x060 VirtualMemoryThreshold : 0xfe00

+0x064 Signature : 0xeeffeeff

+0x068 SegmentReserve : 0x100000

+0x06c SegmentCommit : 0x2000

+0x070 DeCommitFreeBlockThreshold : 0x200

+0x074 DeCommitTotalFreeThreshold : 0x2000

+0x078 TotalFreeSize : 0x13a

+0x07c MaximumAllocationSize : 0x7ffdefff

+0x080 ProcessHeapsListIndex : 5

+0x082 HeaderValidateLength : 0x138

+0x084 HeaderValidateCopy : (null)

+0x088 NextAvailableTagIndex : 0

+0x08a MaximumTagIndex : 0

+0x08c TagEntries : (null)

+0x090 UCRList : \_LIST\_ENTRY [ 0x2c0fe8 - 0x2c0fe8 ]

+0x098 AlignRound : 0xf

+0x09c AlignMask : 0xfffffff8

+0x0a0 VirtualAllocdBlocks : \_LIST\_ENTRY [ 0x2c00a0 - 0x2c00a0 ]

+0x0a8 SegmentList : \_LIST\_ENTRY [ 0x2c0010 - 0x2c0010 ]

+0x0b0 AllocatorBackTraceIndex : 0

+0x0b4 NonDedicatedListLength : 0

+0x0b8 BlocksIndex : 0x002c0150 Void

+0x0bc UCRIndex : (null)

+0x0c0 PseudoTagEntries : (null)

+0x0c4 FreeLists : \_LIST\_ENTRY [ 0x2c0618 - 0x2c0618 ]

+0x0cc LockVariable : 0x002c0138 \_HEAP\_LOCK

+0x0d0 CommitRoutine : 0x17c06e63 long +17c06e63

+0x0d4 FrontEndHeap : (null)

+0x0d8 FrontHeapLockCount : 0

+0x0da FrontEndHeapType : 0 ''

+0x0dc Counters : \_HEAP\_COUNTERS

+0x130 TuningParameters : \_HEAP\_TUNING\_PARAMETERS

就如本文前面所述的，第一个字段是HEAP\_ENTRY结构，接着应该是HEAP\_SEGMENT，这里只不过把HEAP\_SEGMENT结构的字段展开了，可以dt \_HEAP\_SEGMENT来观察下这个结构的字段

0:000> dt \_heap\_segment

ntdll!\_HEAP\_SEGMENT

+0x000 Entry : \_HEAP\_ENTRY

+0x008 SegmentSignature : Uint4B

+0x00c SegmentFlags : Uint4B

+0x010 SegmentListEntry : \_LIST\_ENTRY

+0x018 Heap : Ptr32 \_HEAP

+0x01c BaseAddress : Ptr32 Void

+0x020 NumberOfPages : Uint4B

+0x024 FirstEntry : Ptr32 \_HEAP\_ENTRY

+0x028 LastValidEntry : Ptr32 \_HEAP\_ENTRY

+0x02c NumberOfUnCommittedPages : Uint4B

+0x030 NumberOfUnCommittedRanges : Uint4B

+0x034 SegmentAllocatorBackTraceIndex : Uint2B

+0x036 Reserved : Uint2B

+0x038 UCRSegmentList : \_LIST\_ENTRY

可以看到HEAP结构中灰色部分是和HEAP\_SEGMENT结构中的字段是重复的，也就是说灰色部分字段便是HEAP\_SEGMENT结构。在HEAP\_SEGMENT结构中，我们可以找到FirstEntry字段，这里指的便是我们的分配的内存，不过HEAP\_ENTRY结构无法观察，这里便没办法枚举出所有的HEAP\_ENTRY结构了，但是说一下思路：

每个HEAP\_ENTRY和它对应的内存我们可以称为一个内存块，计算下一个内存块需要用到现有内存块中的2个字段，Size和UnsedBytes，Size的值乘上粒度（就是0:000> !heap -a 2c0000命令显示的信息中的Granularity: 8 bytes字段，这里是8字节），下一个内存块地址就是 本内存块地址+Size\*8+UnsedBytes。当然这里的粒度可以通过HEAP字段中的AlignMask 字段算出来。

HEAP的分配粒度

在HEAP结构中指明了分配粒度，这个分配粒度是说每次堆分配的时候，都以这个粒度为最小单位，这里看到粒度为8字节。所以这里就有了第二次分配内存的实验，我们让程序执行第9行，然后用!heap -a 002c0000观察分配情况

Heap entries for Segment00 in Heap 002c0000

002c0000: 00000 . 00588 [101] - busy (587)

002c0588: 00588 . 00088 [101] - busy (80)

002c0610: 00088 . 00088 [101] - busy (79)

002c0698: 00088 . 00948 [100]

002c0fe0: 00948 . 00020 [111] - busy (1d)

002c1000: 00001000 - uncommitted bytes.

这里可以看出多出了一个占用块，大小是0x79（121） bytes，但是实际分配的大小还是0x 88 （128）bytes，这是因为系统是以8 bytes为粒度分配的，所以为这块121 bytes的内存自动填充了7个字节，可见申请121 bytes和申请128 bytes所使用的空间是一样的。

HEAP的释放和销毁

执行了11行和12行的代码后，堆中的内容分别如下：

执行11行代码的堆情况

FreeList[ 00 ] at 002c00c4: 002c06a0 . 002c0590

002c0588: 00588 . 00088 [100] – free ；空闲列表中多出了一块内存

002c0698: 00088 . 00948 [100] – free ；空闲内存，空闲空间为948

Heap entries for Segment00 in Heap 002c0000

002c0000: 00000 . 00588 [101] - busy (587)

002c0588: 00588 . 00088 [100] ；原先的这块内存释放掉了

002c0610: 00088 . 00088 [101] - busy (79)

002c0698: 00088 . 00948 [100] ; 空闲内存

002c0fe0: 00948 . 00020 [111] - busy (1d)

002c1000: 00001000 - uncommitted bytes.

执行12行代码的堆情况

FreeList[ 00 ] at 005c00c4: 005c0590 . 005c0590

005c0588: 00588 . 00a58 [100] – free ；回收了buffer1的内存后，由于由于空闲内存是连续的，所以直接合并成一块内存。可以看到之前内存free空间是948，现在合并了以后便是948+88+88=a58,也就是当前内存大小

Heap entries for Segment00 in Heap 005c0000

005c0000: 00000 . 00588 [101] - busy (587)

005c0588: 00588 . 00a58 [100]

005c0fe0: 00a58 . 00020 [111] - busy (1d)

005c1000: 00001000 - uncommitted bytes.

最后执行14行代码,对堆进行释放,释放后我们通过!heap也可以看到只有4个堆了,我们申请的堆被释放了.

0:000> !heap

Index Address Name Debugging options enabled

1: 00300000

2: 00010000

3: 00020000

4: 002e0000

至于HEAP\_ENTRY结构的问题,有时间在调试看看是怎么回事吧~另外，这里说明下，new和malloc内部都会调用HeapAlloc来申请内存，但是堆句柄从哪来呢？它会检测\_crtheap变量是否为空，若不为空则拿\_crtheap变量来作为自己的堆句柄去调用HeapAlloc

参考：

软件调试 张奎银

MSDN

React OS

<http://www.cppblog.com/ay19880703/archive/2012/01/05/163604.html>

<http://debug-sai.blogbus.com/logs/145383417.html>

<http://www.cnblogs.com/gussing/archive/2009/09/26/1574551.html>

<http://debug-sai.blogbus.com/logs/50262846.html>

<http://www.mouseos.com/windows/kernel/lookaside.html>

<http://www.2cto.com/Article/200901/31191.html>