Microsoft Edge MemGC Internals

Henry Li,TrendMicro

2015/08/29

Agenda

- Background
- MemGC Internals
- Prevent the UAF'S exploit
- Weaknesses of MemGC

Notes

- Research is based on Windows 10 10041(edgehtml.dll, chakra.dll)
- The latest windows versions(windows 10 10240) data structure there are some small changes

Who am i?

- A security research in TrendMicro CDC zero day discovery team.
- Four years of experience in vulnerability & exploit research.
- Research interests are browser 0day vulnerability analysis, discovery and exploit.
- Twitter/Weibo:zenhumany

Background

- June 2014 IE introduce ISOLATE HEAP
- July 2014 IE introduce DELAY FREE

Background

- Isolated Heap can bypass
- Delay Free
  - Pointer to the free block remains on the stack for the entire period of time from the free until the reuse, can prevent UAF EXPLOIT
  - Other situation, can bypass

What’s MemGC

- Chakra GC use Concurrent Mark-Sweep (CMS) Managing Memory
- Edge use the same data structures to mange DOM and DOM’S supporting objects, called MemGC

MemGC Internals

- Data Structures
- Algorithms

MemGC Data Structures

MemProtectHeap		
0x000	m_tlsIndex	:int
0x108	m_recycler	:Recycler

Recycler		
0x026c	m_HeapBlock32Map	HeapBlock32Map
0x42bc	m_HeapInfo	:HeapInfo

HeapInfo		
0x4400	m_HeapBucketGroup[0x40]	:HeapBucketGroup array
0x5544	m_LargeHeapBucket[0x20]	:LargeHeapBucket array
0x5b44	m_lastLargeHeapBucket	:LargeHeapBucket

HeapBucketGroup

HeapBucketGroup	0x154
0x000	m_HeapBucketT<SmallNormalHeapBlock>
0x044	m_HeapBucketT<SmallLeafHeapBlock>
0x080	m_HeapBucketT<SmallFinalizableHeapBlock>
0x0c8	m_HeapBucketT<SmallNormalWithBarrierHeapBlock>
0x10c	m_HeapBucketT<SmallFinalizableWithBarrierHeapBlock>

HeapBucketT<SmallNormalHeapBlock>

HeapBucketT<SmallNormalHeapBlock>	
0x04	size :int
0x0c	m_SmallHeapBlockAllocator
0x20	pPartialReuseHeapBlockList
0x24	pEmptyHeapBlockList
0x28	pFullMarkedHeapBlockList
0x2c	pPendingNewHeapBlockList

SmallHeapBlockAllocator<SmallNormalHeapBlock>	
0x00	endadderss
0x04	startaddress
0x08	pSmallNormalHeapblock

LargeHeapBucket

LargeHeapBucket	
0x04	size
0x0c	pSweepLargeHeapBlockList
0x10	pNewLargeHeapBlockList
0x18	pDisposeLargeHeapBlockList
0x1c	pPendingLargeHeapBlockList
0x28	pFreeListHead
0x2c	pExplicitFreeList

SmallNormalHeapBlock

Attribute Array															
SmallNormalHeapBlock															
0x04: StartAddress															
0x20:pNextSmallHeapblock															
0x24: pFreeHeapObject															
0x2c: pValidPointersBuffer															
0x34: blockSize															
0x36: objectCapacity															
0x44: pMarkBitMapTable															
0x48: freeBitVector															
0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f

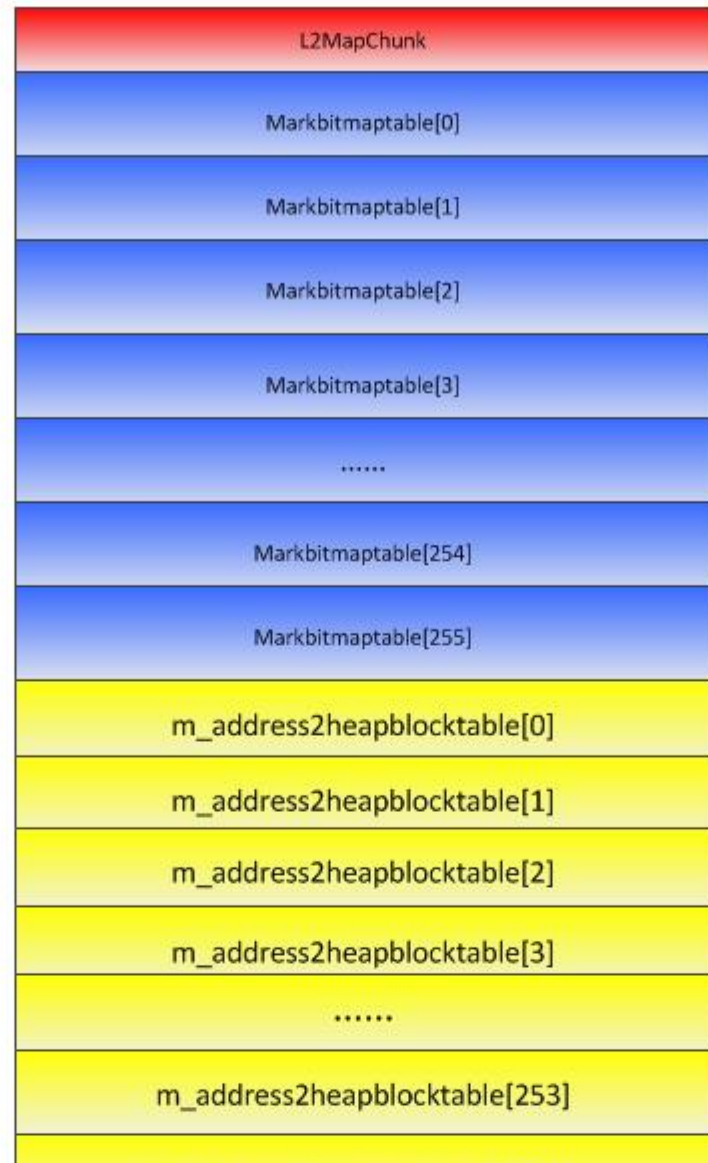
LargeHeapBlock

LargeHeapBlock				
0x04 pageAddress				
0x28 allocblockcount				
0x2c blockCapacity				
0x30 allocAddress				
0x34 endAddress				
0x38 pNextLargeHeapBlock				
0x44 pPrevFreeList				
0x48 pNextFreeList				
0x4c pFreeHeapObject				
0x64 allocBlockAddressArray[]				
0	1	2	…	blockCapacity-1

HeapBlock32Map

HeapBlock32Map	
0x00	count
0x04	m_pL2MapChunkArray[4096]

L2MapChunk	
0x0000	markbitmaptable[256]
0x2000	m_address2heapblocktable[256]



OVERVIEW

MemProtectHeap
0x108 m_recycler :Recycler

Recycler
0x026c m_HeapBlock32Map : HeapBlock32Map
0x42bc m_HeapInfo :HeapInfo

HeapBlock32Map
0x00 m_pL2MapChunkArray[4096]
:L2MapChunkArray[]

L2MapChunk
0x0000 m_markbitmaptable[256]
0x2000 m_address2heapblocktable[256]

markbitmaptable

address2heapblocktable

HeapInfo
0x0044 m_HeapBucketGroup[0x40] :HeapBucketGroup[]
0x5544 m_LargeHeapBucket[0x20] :LargeHeapBucket[ ] (each element is 0x30)
0x5b44 m_LastLargeHeapBucket :LargeHeapBucket[ ]

HeapBucketGroup
0x00m_HeapBucketT<SmallNormalHeapBlock>

HeapBucketT

SmallHeapBlock
0x04 startAddress
0x08 pPagesegment
0x20 pNextSmallHeapblock
0x24 pFreeHeapObjec
0x44 pMarkBitMapTable
0x48 freeBitVector

LargeHeapBucket

LargeHeapBlock
0x30 allocAddress
0x34 endAddress
0x38 pNextLargeBlock
0x44 pPrevFreeList
0x48 pNextFreeList
0x4c pFreeHeapObject
0x64 allocBlockAddressArray[]

LargeHeapBucket

LargeHeapBlock
0x30 allocAddress
0x34 endAddress
0x38 pNextLargeBlock
0x44 pPrevFreeList
0x48 pNextFreeList
0x4c pFreeHeapObject
0x64 allocBlockAddressArray[]

Algorithms

- Alloc
- Free
- Mark
- Sweep

MemGC Alloc

- edgehtml!MemoryProtection::HeapAlloc<1>
  - edgehtml!MemoryProtection::CMemoryGC::ProtectedAlloc<3>
    - chakra! MemProtectHeapRootAlloc
      - **chakra!Recycler::NoThrowAllocImplicitRoot**

Alloc

- (0x00-0x400]—HeapBucketGroup
  - array size: 0x400/0x10 = 0x40
- (0x400-0x2400]—LargeHeapBucket
  - array size: 0x2000/0x100 = 0x20
- (0x2400-)—LargeHeapBucket
  - size: 0x01

MemGC Alloc

size	HeapBucket address
0x10	m_HeapBucketGroup[0x00]
0x20	m_HeapBucketGroup[0x10]
0x30	m_HeapBucketGroup[0x20]
……	……
0x390	m_HeapBucketGroup[0x38]
0x400	m_HeapBucketGroup[0x39]
0x500	m_LargeHeapBucket[0x00]
0x600	m_LargeHeapBucket[0x01]
……	……
0x2300	m_LargeHeapBucket[0x18]
0x2400	m_LargeHeapBucket[0x19]
>0x2400	m_LastLargeHeapBucket

HeapBucketT<SmallNormalHeapBlock>

HeapBucketT<SmallNormalHeapBlock>	
0x00	pHeapInfo
0x04	size
0x0c	m_SmallHeapBlockAllocator 0x00 endaddress 0x04 startaddress 0x08 pSmallNormalHeapBlock
0x20	pPartialReuseHeapBlockList

1、 startaddress + blocksize <= endaddress
   return startaddress
2、 HeapBucketT::SnailAlloc
   pPartialReuseHeapBlockList !=null;
   alloc from pPartialReuseHeapBlockList
3、 New SmallNormalHeapBlock

LargeHeapBucket

LargeHeapBucket	
0x00	pHeapInfo
0x04	size
0x10	pNewLargeHeapBlockList
0x28	pFreeListHead
0x2c	pExplicitFreeList

1、pNewLargeHeapBlockList
2、pExplicitFreeList, pFreeListHead
3、New LargeHeapBlock

Alloction
Recycler::NoThrowAllocImplicitRoot(int size)

NoThrowAllocImplicitRoot:Part I

Align the size to 0x10

Size <= 0x400, go into small heap object alloc

```
    HeapInfo* pHeapInfo = &(this->m_HeapInfo);
    Recycler* pRecycler = this->pRecycler;
    //Adjust the size to 0x10 bytes align.
    int align_size = (size + 15) & 0xfffffff0;
     //size <= 0x400,go into small heap

    if( size <= 0x400)
    {
        //Get the HeapBucketGroup index in the HeapInfo->m_HeapBucketGroup
        int index  = align_size / 16;
        //Get the SmallNormalHeapBlock type HeapBucketT
        HeapBucketT<SmallNormalHeapBlock>* pHeapBucketT = &pHeapInfo->m_HeapBucketGroup[ index
].m_HeapBucketT<SmallNormalHeapBlock>;

        //Get SmallHeapBlockAllocator
        SmallHeapBlockAllocator* pSmallHeapBlockAllocator = pHeapBucketT->pSmallHeapBlockAllocatorT;
        //Get pSmallNormalHeapBlock
        SmallNormalHeapBlock* pSmallNormalHeapBlock = pSmallHeapBlockAllocator->pSmallHeapBlock ;
        //if startAddress + align_size > endAddress,go into SnailAlloc or
        if (pSmallHeapBlockAllocator->startAddress + align_size <= pSmallHeapBlockAllocator->endAddress)
        {
            //startAddress + align_size <= endAddress, return the startAddress
            allocAddress =   pHeapBucketT->startAddress;
            //update the startAddress of pSmallHeapBlockAllocator equal startAddress + align_size
            pSmallHeapBlockAllocator->startAddress = pHeapBucketT->startAddress + align_size;
            if( pSmallHeapBlockAllocator->NeedSetAttributes( 8 ))
            {
                pSmallNormalHeapBlock->SetAttribute(pHeapBucketT, 8);
            }
            return allocAddress;
        }
```

NoThrowAllocImplicitRoot:Part I

Goto snailalloc or reuse freeobject

```
//if startAddress + align_size > endAddress,go into SnailAlloc or
else
{
    //startAddress==0 or endAddress!=0 ,gointo SnailAlloc
    if( pSmallHeapBlockAllocator->startAddress ==0 || pSmallHeapBlockAllocator->endAddress!=0 )
    {
        allocAddress = pHeapBucketT->SnailAlloc(pRecycler, pSmallHeapBlockAllocator, align_size, 8, 1);
        if( allocAddress == 0)
            return 0;
        else
            *allocAddress = 0;
            return allocAddress
    }
    if( pSmallHeapBlockAllocator->NeedSetAttributes( 8 ))
    {
        pSmallNormalHeapBlock->SetAttribute(pHeapBucketT, 8);
    }
    //startAddress !=0 &&endAddress==0, we can reuse the free heap object
    //free heap object first dword is a pointer which pointer to the next heap object.
    allocAddress =  pSmallHeapBlockAllocator->startAddress;
    //startAddress to the next heap object
    pSmallHeapBlockAllocator->startAddress = (*pSmallHeapBlockAllocator->startAddress) & 0xfffffe;
    return allocAddress;

}
```

NoThrowAllocImplicitRoot:Part II

Alloc middle object

```
else if( size <= 0x2400 ) //size in (0x400,0x2400],进入LargeHeapBucket进行分配
    {
        //Calculate largeheapbucket index in HeapInfo.m_LargeHeapBucket
        int largebucketIndex = ( align_size - 1025 ) / 256 ;
        //Get LargeHeapBucket
        LargeHeapBucket* pLargeHeapBucket = &pHeapInfo->m_LargeHeapBucket[ largebucketIndex];
        //Get pLargeHeapBlockList
        pLargeHeapBlock = pLargeHeapBucket->pLargeHeapBlockList;
        //pLargeHeapBlockList not zero, go into LargeHeapBlock::Alloc
        if(pLargeHeapBlock)
        {
            allocAddress = pLargeHeapBlock ->Alloc( align_size, 8)
            if( allocAddress)
            {
                *allocAddress = 0;
                return allocAddress;
            }
        }
        //pLargeHeapBlockList is zero, check freelistflag,if true, to alloc from freelsit
        else if( pLargeHeapBucket-> freelistflag)
        {
            allocAddress = pLargeHeapBucket->TryAllocFromExplicitFreeList(pLargeHeapBucket,
(int)v2, v12, 8u);
            if( allocAddress || allocAddress = pLargeHeapBucket->TryAllocFromFreeList( )!=0 )
                return allocAddress;
        }
        //if above two step alloc fail, go into LargeHeapBucket::SnailAlloc
        pLargeHeapBucket->SnailAlloc(pRecycler, align_size, 8, 1);
    }
```

NoThrowAllocImplicitRoot:Part III

Alloc large object

```
else //size > 0x2400,go into Recycler::LargeAlloc
 {
     allocAddress = Recycler::LargeAlloc( pHeapInfo, size, 8 )
     *allocAddress = 0;
     return allocAddress;
 }
```

HeapBucketT<SmallNormalHeapBlock>::SnailAlloc Part I

```
//SmallHeapBlockAllocator<SmallNormalHeapBlock>::Clear( )
pSmallHeapBlockAllocator->clear( );
//get reuse smallheapblock
SmallHeapBlock* pFreeListHeapBlock = this->pFreeListHeapBlock;
//pFreeListHeapBlock not zero, go into reuse the heapblock
if(pFreeListHeapBlock )
{
    //set the pFreeListHeapBlock to the NextSmallHeapBlock
    this->pFreeListHeapBlock = pFreeListHeapBlock->pNextSmallHeapBlock;
    pFreeListHeapBlock->markFlag = 1;
    //set SmallHeapBlockAllocator::pSmallHeapblock pointer pFreeListHeapBlock
    pSmallHeapBlockAllocator->pSmallHeapBlock = pFreeListHeapBlock;
    //beginAddress point to the reuse heapblock pFreeHeapObject
    pSmallHeapBlockAllocator->beginAddress = pFreeListHeapBlock->pFreeHeapObject;
}
```

HeapBucketT<SmallNormalHeapBlock>::SnailAlloc Part II

```
   //Get HeapBlockMap32
HeapBlockMap32* pHeapBlockMap32 = &pRecycler->m_HeapBlockMap32;

//initial SmallNormalHeapBlock
pSmallNormalHeap-> pPageSegment = pageSegment
pSmallNormalHeap-> startAddress = pageaddress;
int first_index = (pageaddress /2^20)
int second_index = (pageaddress / 2^12) & 0xff;
pL2MapChunk = pHeapBlockMap32->m_pL2MapChunkArray[ first_index ];
//map the pageaddress and SmallHeapBlock relation
pL2MapChunk->Set(second_index, 1, pSmallHeapBlock);

//Get markbitmaptable to initial SmallNormalHeapBlock::pMarkBitMap
markbitmaptable = pL2MapChunk->m_markbitmaptable[ second_index ];
pSmallHeapBlock->pMarkBitMap  = markbitmaptable;
//add the new SmallNormalHeapBlock into Recycelr-> pSmallNormalHeapBlockList list
pSmallHeapBlock-> pNextSmallHeapblock = pRecycler-> pSmallNormalHeapBlockList;
pRecycler-> pSmallNormalHeapBlockList = pSmallHeapBlock;
pSmallHeapBlock-> markflag = 1;
//Initial pSmallHeapBlockAllocator
pSmallHeapBlockAllocator->pSmallHeapBlock = pSmallHeapBlock;
pSmallHeapBlockAllocator->startAddress = pSmallHeapBlock->startAddress;
pSmallHeapBlockAllocator->endAddress = pSmallHeapBlock->startAddress + 0x1000;
……
return pageaddress;
```

LargeHeapBucket::AddLargeHeapBlock Part I

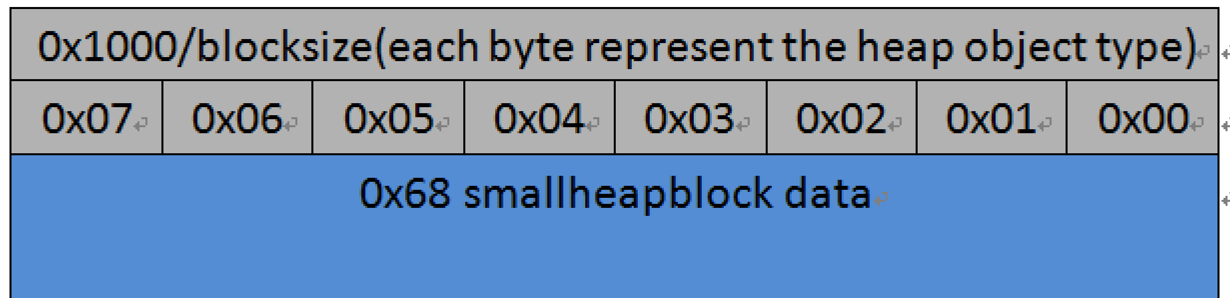
```
LargeHeapBlock* LargeHeapBucket::AddLargeHeapBlock(int blockSize, bool param4)
{
    int freelistflag = this->freelistflag;
    int memoryCapacity =0;
    struct Segment *pSegment = null;
    HeapInfo* pHeapInfo = this->pHeapInfo;
    Recycler* pRecycler = pHeapInfo->pRecycler;

    if( freelistflag == 0)
        memoryCapacity = 4* blockSize;
    //memoryCapacity + 16 >= memoryCapacity check int overflow
    if( memoryCapacity + 16 >= memoryCapacity )
    {
        //culation requires allocation of pages
        int pagenum =( memoryCapacity + 0xfff)>>12;

        //select pageallocator
        RecyclerPageAllocator* pRecyclerPageAllocator = &pRecycler-
>m_RecyclerPageAllocator[2];
        //alloc pages
        int pageaddress = pRecyclerPageAllocator->AllocInternal( pagenums,
&pSegment);

        //Calculate the pages contains how may blocks
        int blocksNum =(((pagenums<<12) - blockSize - 16 ) >>10 ) + 1;
        //largeheapblock size is 0x64 add blocksNum*4
        int largeheapblockSize = 0x64 + blocksNum*4;
        //alloc LargeHeapBlock
        LargeHeapBlock* pLargeHeapblock =
(LargeHeapBlock*)HeapAllocator::NoThrowAllocZero(largeheapblockSize );
```

LargeHeapBucket::AddLargeHeapBlock Part II

```
if( pLargeHeapBlock)
        {
            pLargeHeapBlock->pLargeHeapBucket = this;
            pLargeHeapBlock->pagenum = pagenum;
            pLargeHeapBlock->blockCapacity = blocksNum;
            pLargeHeapBlock->allocAddress = pageaddress;
            pLargeHeapBlock->largeHeapType = 0x05;
            pLargeHeapBlock->pNextLargeHeapBlock = pLargeHeapBlock;
            pLargeHeapblock->pPageSegment = pSegment;
            pLargeHeapblock->pageAddress = pageaddress;
            pLargeHeapblock->unknown = 0;
            pLargeHeapblock->endaddress = pageaddress + pagenum<<12;
            pRecycler->pageCount += pagenum;
            pLargeHeapblock ->pHeapInfo = pHeapInfo;
            *(pLargeHeapBlock+0x54) = 0;
            bool result = pRecycler->m_HeapBlockMap32.SetHeapBlock(
pageaddress,pagenum,pLargeHeapBlock );
            if(result)
            {
                //link the new LargeHeapBlock to the pLargeHeapBucket-
>pLargeHeapBlockList list
                pLargeHeapblock->pNextLargeBlock = pLargeHeapBucket-
>pLargeHeapBlockList;
                pLargeHeapBucket->pLargeHeapBlockList = pLargeHeapblock;
                return pLargeHeapblock;
            }
        }
```

SmallNormalHeapBlock

Size = 0x68 + ((0x1000/blocksize) +3)&0x0FFFFFFFC



0x1000/blocksize(each byte represent the heap object type)							
0x07	0x06	0x05	0x04	0x03	0x02	0x01	0x00
0x68 smallheapblock data							

SmallNormalHeapBlock

Attribute Array
SmallNormalHeapBlock
0x04: StartAddress
0x20:pNextSmallHeapblock
0x24: pFreeHeapObject
0x2c: pValidPointersBuffer
0x34: blockSize
0x36: objectCapacity
0x44: pMarkBitMapTable
0x48: freeBitVector

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f

blockSize	blockSize	blockSize
……	……	……
blockSize	blockSize	blockSize

HeapInfo::ValidPointersMap::validPointersBuffer

0	1	2	……	fe	ff
100	101	102	……	1fe	1ff

L2MapChunk

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f

Markbitmap,freeBitvector

- SmallHeapBlock managers one page(4k)Memory.
  - 2^12/2^4=256
- markbitmap 32 bytes, 256 bit.
  - bit 1: mark
  - bit 0: unmark
- freeBitVector 32 bytes, 256 bit.
  - bit 1: free
  - bit 0: unfree

validPointersBuffer

- Each SmallHeapBlock mangers one page(4k) memory
  - 2^12/2^4=256
- ValidPointer: the beginaddres of the object is Valid pointer, the interior address is invalid pointer.
- Each validPointersBuffer element contains two part, each part is an array,array length is 256.
  - First part: Chakra GC
  - Second part: MemGc

HeapInfo::ValidPointersMap::validPointersBuffer

SmallNormalHeapBlock
blocksize 0x20
pageaddress 0x15100000

```
0:001> dw  chakra!HeapInfo::ValidPointersMap::validPointersBuffer +400 l100/4
6179fa08  0000 ffff 0001 ffff 0002 ffff 0003 ffff
6179fa18  0004 ffff 0005 ffff 0006 ffff 0007 ffff
6179fa28  0008 ffff 0009 ffff 000a ffff 000b ffff
6179fa38  000c ffff 000d ffff 000e ffff 000f ffff
6179fa48  0010 ffff 0011 ffff 0012 ffff 0013 ffff
6179fa58  0014 ffff 0015 ffff 0016 ffff 0017 ffff
6179fa68  0018 ffff 0019 ffff 001a ffff 001b ffff
6179fa78  001c ffff 001d ffff 001e ffff 001f ffff
0:001> dw  chakra!HeapInfo::ValidPointersMap::validPointersBuffer +400 + 200 l1
6179fc08  0000 0000 0001 0001 0002 0002 0003 0003
6179fc18  0004 0004 0005 0005 0006 0006 0007 0007
6179fc28  0008 0008 0009 0009 000a 000a 000b 000b
6179fc38  000c 000c 000d 000d 000e 000e 000f 000f
6179fc48  0010 0010 0011 0011 0012 0012 0013 0013
6179fc58  0014 0014 0015 0015 0016 0016 0017 0017
6179fc68  0018 0018 0019 0019 001a 001a 001b 001b
6179fc78  001c 001c 001d 001d 001e 001e 001f 001f
```

validPointersBuffer example

- 15100000,15100020,15100040……
- 15100010
  - Chakra GC:
    - index = validPointerBuffer_chakra[(address – pageaddress)/0x10] = 0xffff
  - MemGC
    - Index = validPointerBuffer_memgc[(address – pageaddress)/0x10] = 0x00
    - Realaddress = pageaddress + index*blocksize = 0x15100000

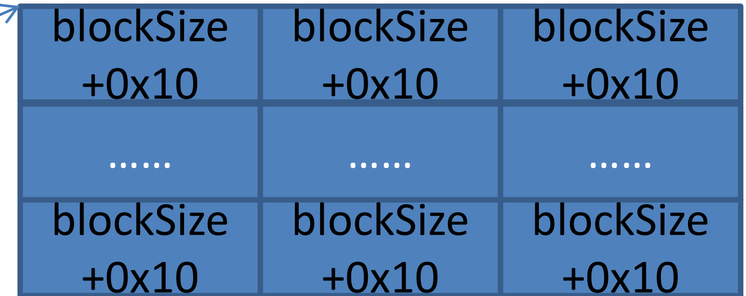
LargeHeapBlock

- pagenums = ((blocksize*4 + 10) + 0xfff)/2^12
- arrayLength = ((pagenums*2^12) – blocksize -0x10)/2^10 +1
- largeheapblockSize = 0x64 + 4*arrayLength

LargeHeapBlock

LargeHeapBlock
0x04 pageAddress
0x28 allocblockcount
0x2c blockCapacity
0x30 allocAddress
0x34 endAddress
0x38 pNextLargeHeapBlock
0x44 pPrevFreeList
0x48 pNextFreeList
0x4c pFreeHeapObject
0x64 allocBlockAddressArray[]

0	1	2	…	blockCapacity-1



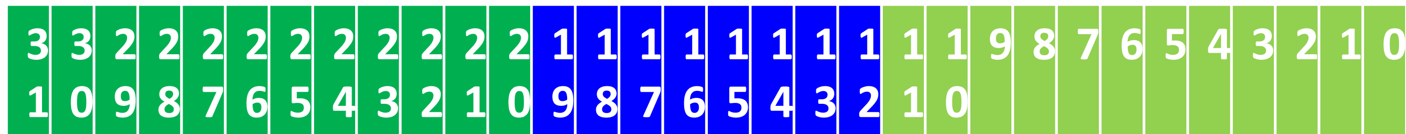
LargeObjectHeader

LargeObjectHeader(**inuse**)	
0x00	index
0x04	blocksize
0x08	initialzero
0x0c	encode

LargeObjectHeader(**free**)	
0x00	index
0x04	blocksize
0x08	pLargeHeapBlock
0x0c	pNextFreeHeapObject

pageaddress to HeapBlock

- pageaddres



  - High 12 bit: first_index
  - Middle 8 bit: second_index
  - Low 12 bit: not used

HeapBlock32Map

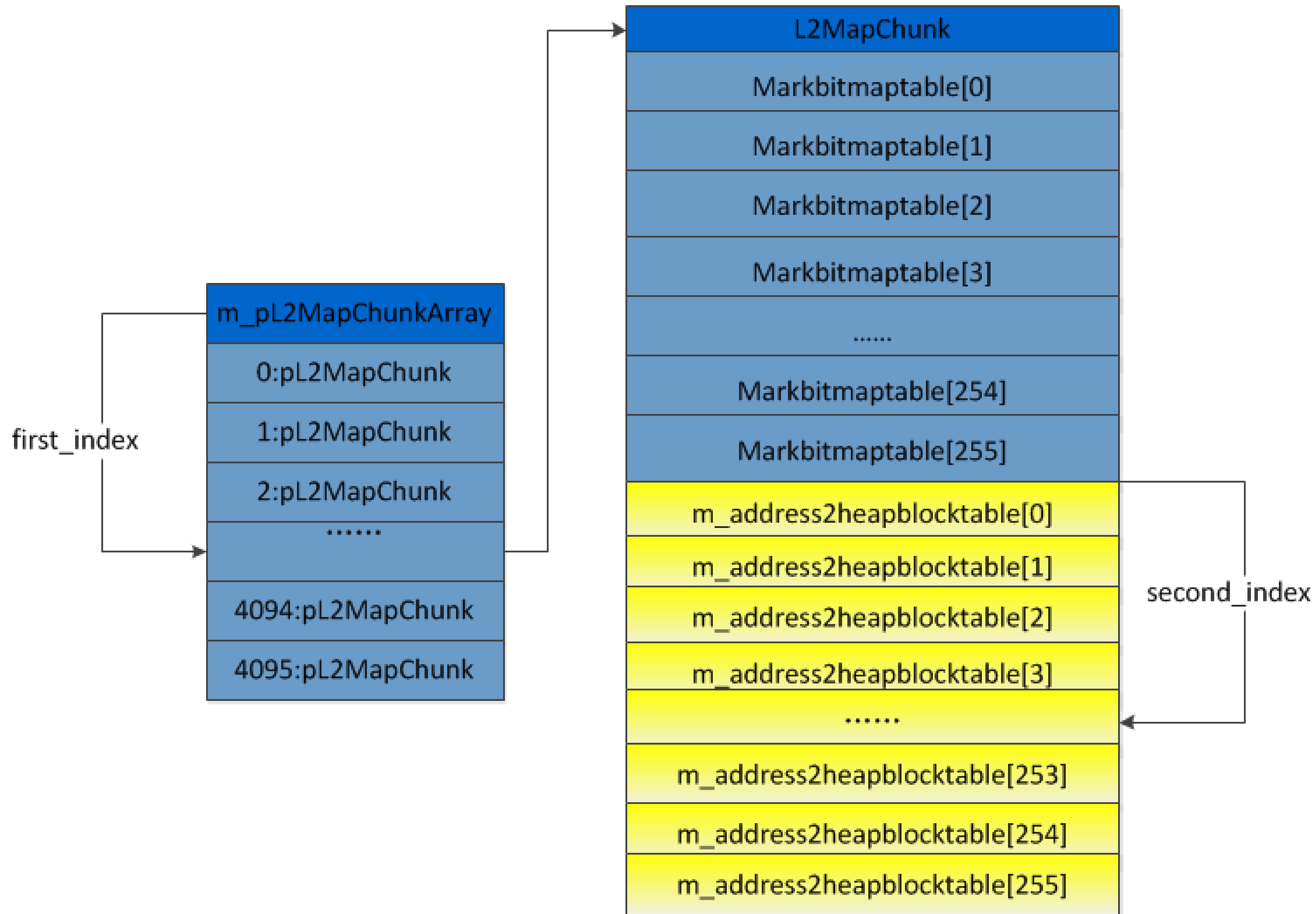
HeapBlock32Map	
0x00	count
0x04	m_pL2MapChunkArray[4096]

count: the number of L2MapChunk in m_pL2MapChunkArray
m_pL2MapChunkArray: an L2MapChunk array.

L2MapChunk	
0x0000	markbitmaptable[256]
0x2000	m_address2heapblocktable[256]

markbitmaptable: markbitmap array. each element 32 bytes
m_address2heapblocktable : an array, each element is an pointer to HeapBlock

pageaddress to HeapBlock



MemGC Free

- edgehtml! MemoryProtection::HeapFree
  - Edgehtml!MemoryProtection::CMemoryGC::ProtectedFree
    - chakra!MemProtectHeapUnrootAndZero

MemProtectHeapUnrootAndZero

1、 memset zero

2、 unroot

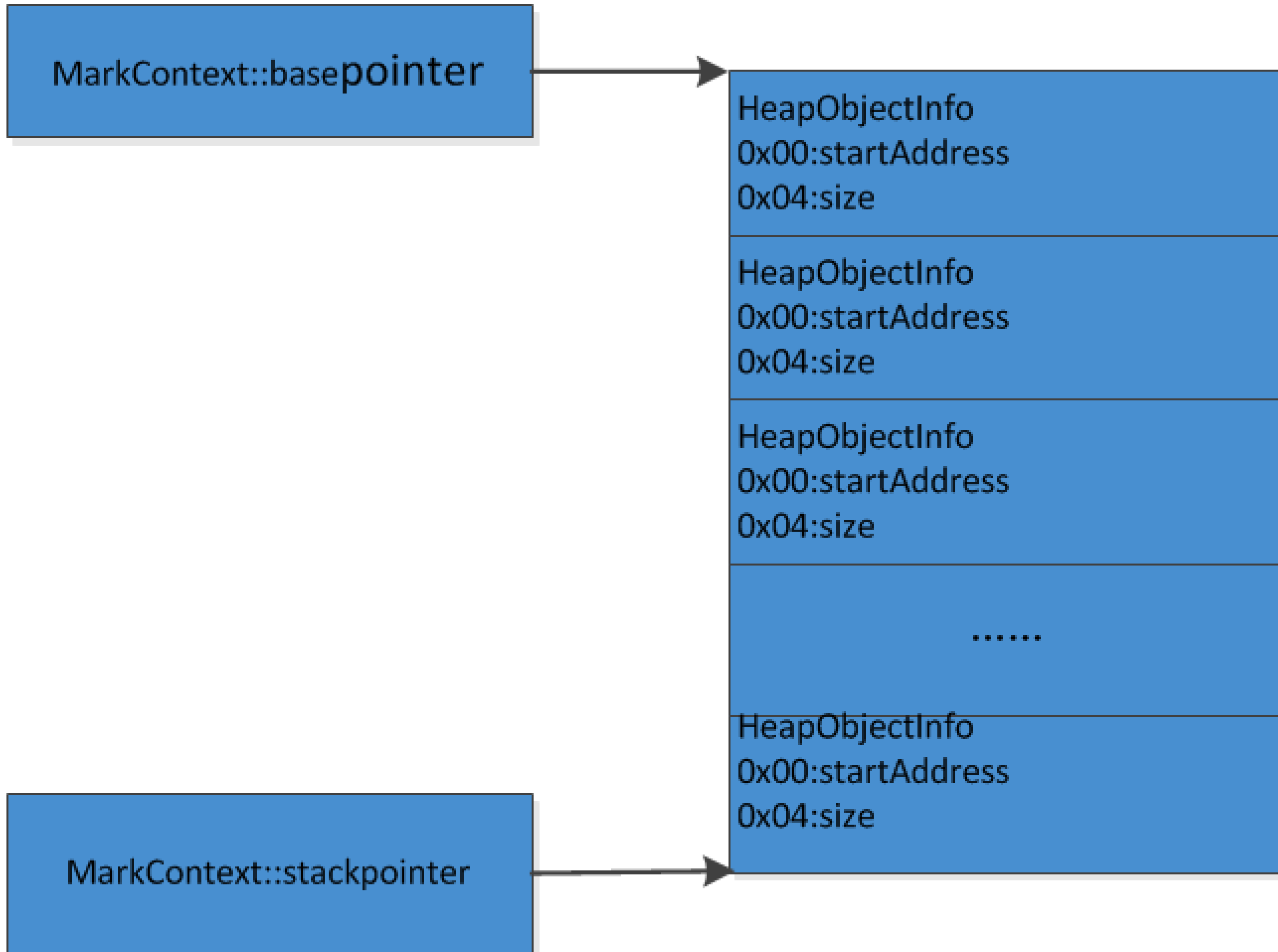
```
MemProtectHeapUnrootAndZero(MemProtectHeap* pMemProtectHeap, void* freeBlockAddress)
{
    MemProtectThreadContext* pMemProtectThreadContext = TlsGetValue( pMemProtectHeap->m_tlsIndex);
    RecyclerHeapObjectInfo tempRecyclerHeapObjectInf;
    if( pMemProtectThreadContext )
    {
        *(_BYTE*)(pMemProtectThreadContext+8) = 1;
        MemProtectHeap* pMemProtectHeapFromContext = pMemProtectThreadContext->pMemProtectHeap;
        Recycler* pRecycler = &(pMemProtectHeapFromContext->m_Recycler);
        if(pRecycler->FindHeapObject( freeblockAddress,2, &tempRecyclerHeapObjectInf ))
        {
            if( !tempRecyclerHeapObjectInf.IsLeaf( ))
            {
                int objectsize = tempRecyclerHeapObjectInf.pHeapBlock->GetObjectSize( );
                //set the freeblockaddress content zero
                memset( freeBlockAddress, 0, objectsize);
            }
            if(tempRecyclerHeapObjectInf.ClearImplictRootbit( ))
            {
                pMemProtectThreadContext->NotifyUnroot( &RecyclerHeapObjectInfo);
            }
        }
    }
}
```

Mark

chakra!markcontext	
0x08	stackpointer
0x0c	basepointer
0x10	endAddress
0x14	arrayStartAddress

- stackpointer: the stack current element address.
- basepointer: the stack begin address
- endAddress: the stack endaddress
- arrayStartAddress : the begin address which maintance the stack information.

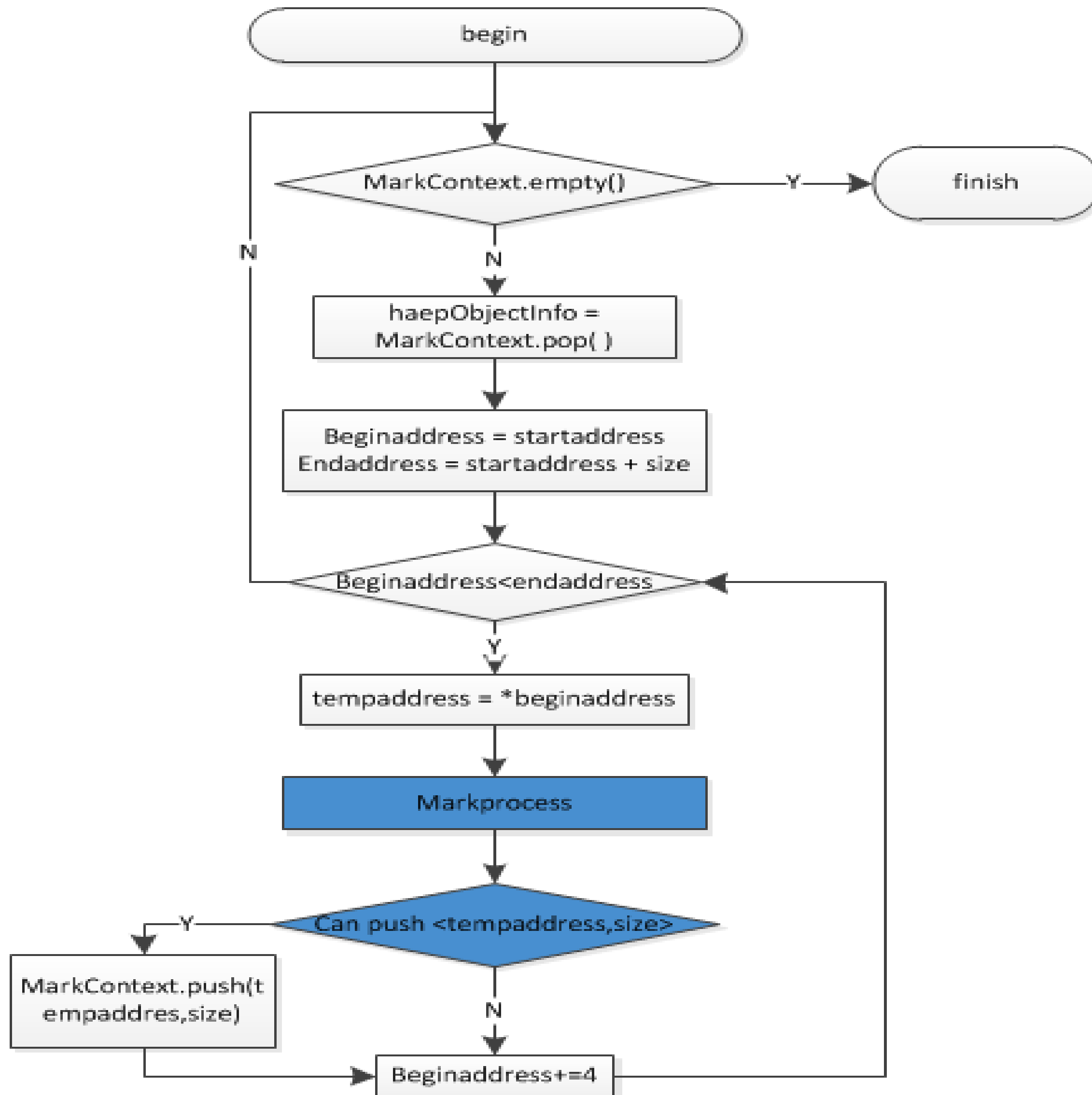
Heap Object Info stack



find roots

- MemProtectHeap::FindRoots
  - MemProtectThreadContext::ScanStack
  - Recycler::ScanImplicitRoots
- push the root object into makecontext.

processmarkcontext

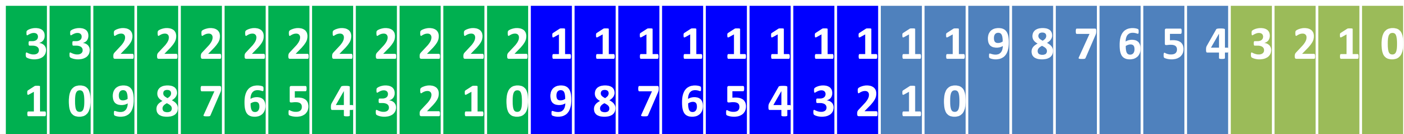


Address mark

- Address > 0x10000
- Address -> HeapBlock
- realaddress = GetRealAddressFromInterior
  - LargeHeapBlock:: GetRealAddressFromInterior
  - SmallHeapBlock::GetRealAddressFromInterior

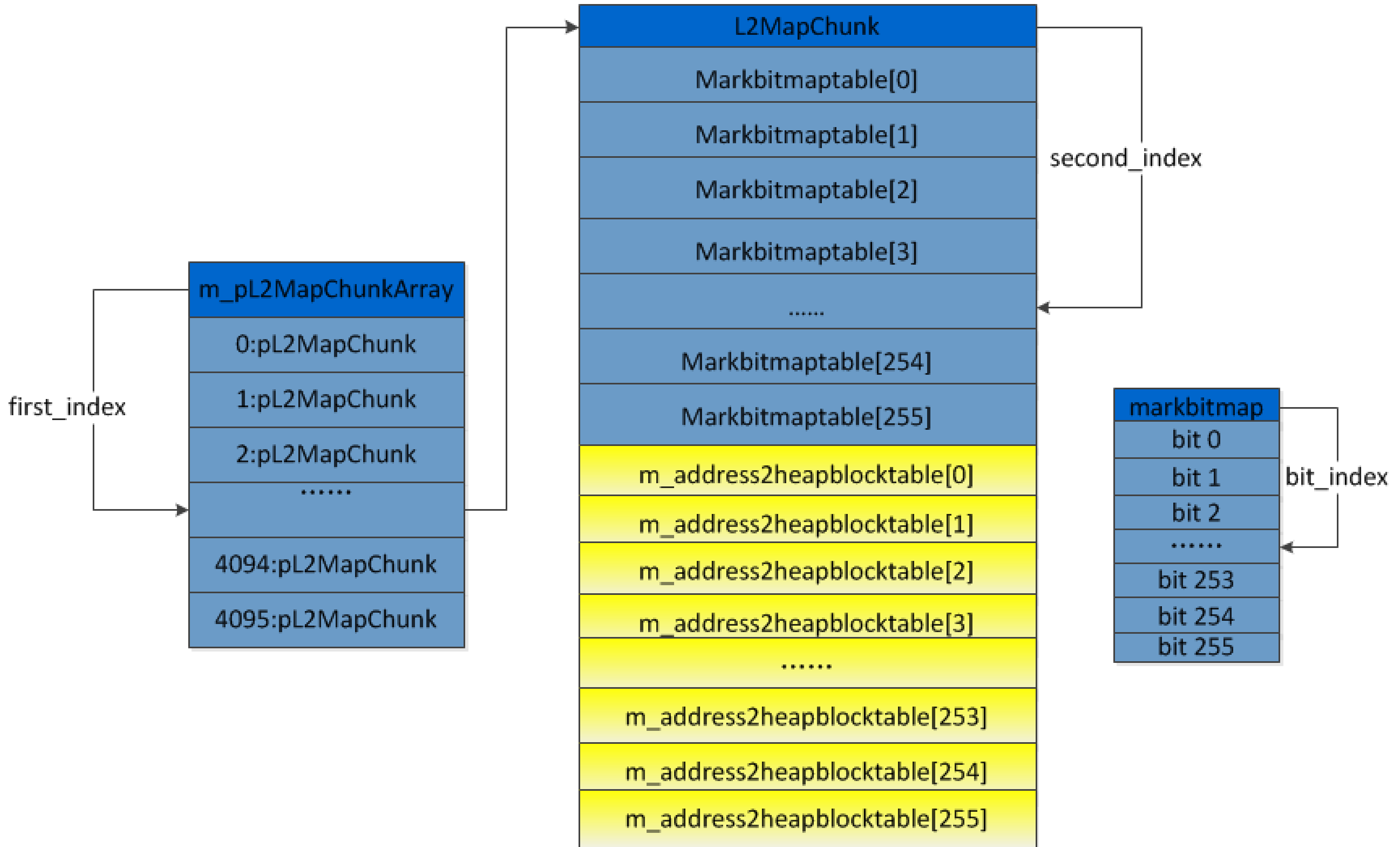
address mark

- Addres



  - High 12 bit: first_index
  - Middle 8 bit: second_index
  - Low 8 bit: bit_index
  - Last low 4 bit: 0x10 bytes alignment

address mark



push (address,size) to stack

- The address is the first time mark
- Address -> HeapBlock
- SmallNormalHeapBlock::ProcessMarkedObject
- LargeHeapBlock::Mark

SmallNormalHeapBlock::ProcessMarkedObject

```
chakra!SmallNormalHeapBlock::ProcessMarkedObject(SmallHeapBlock* pSmallHeapBlock, int
address, MarkContext* pMarkContext)
{
    int blockSize = this->blockSize;
    BYTE bit_index = (address>>4) &0xff int invalidBitsIndex = blockSize/0x10;
    //32 bytes
    invalidBits = HeapInfo::ValidPointersMap:: invalidBitsData[ invalidBitsIndex ];
    //1,invalid,0 valid

    if(!bittest( invalidBits, bit_index ))
    {
        if( pMarkContext->stackpointer != pMarkContext->arrayEndAddress)
        {
            //push one element into the stack.
            stackpointer = pMarkContext->stackpointer;
            *stackpointer = address;
            *(currentAddress+4) = blockSize;
            pMarkContext->stackpointer +=8;
        }
    }
}
```

invalidBitsData

- each smallheapblock manager one page memory(4k)
  - 2^12/2^4 = 256
- Each invalidBitsData element is 32 bytes, 256 bit , each bit indicates whether the address is a valid pointer
  - bit 1: invalid pointer
  - bit 0: valid pointer

HeapInfo::ValidPointersMap::invalidBitsData

- blocksize 0x20

```
0:034> dc 5eb50000 + 474e70 l30
5efc4e70  00000000 00000000 00000000 00000000  ................
5efc4e80  00000000 00000000 00000000 00000000  ................
5efc4e90  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa  ................
5efc4ea0  aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa  ................
5efc4eb0  b6db6db6 6db6db6d db6db6db b6db6db6  .m..m..m..m..m..
5efc4ec0  6db6db6d db6db6db b6db6db6 edb6db6d  m..m..m..m..m...
5efc4ed0  eeeeeeee eeeeeeee eeeeeeee eeeeeeee  ................
5efc4ee0  eeeeeeee eeeeeeee eeeeeeee eeeeeeee  ................
5efc4ef0  bdef7bde ef7bdef7 7bdef7bd def7bdef  .{....{....{....
5efc4f00  f7bdef7b bdef7bde ef7bdef7 fbdef7bd  {....{....{.....
5efc4f10  befbefbe efbefbef fbefbefb befbefbe  ................
5efc4f20  efbefbef fbefbefb befbefbe ffbefbef  ................
0:034> .formats aaaaaaaa
Evaluate expression:
  Hex:     aaaaaaaa
  Decimal: -1431655766
  Octal:   25252525252
  Binary:  10101010 10101010 10101010 10101010
  Chars:   ....
  Time:    ***** Invalid
  Float:   low -3.03165e-013 high 0
  Double:  1.41466e-314
```

LargeHeapBlock::Mark

```
chakra!LargeHeapBlock::Mark( int address, MarkContext* pMarkContext)
{
    //get the largeobjectheader
    LargeObjectHeader* pLargeObjectHeader = (LargeObjectHeader*)(address-0x10)
    //check the object is a valid object
    //one: address-0x10 > LargeHeapblock::pageaddress
    //two: LargeObjectHeader::index < LargeHeapblock::allocblockcount
    //three: address-0x10 == allocBlockAddressArray[pLargeObjectHeader->index]
    if(pLargeObjectHeader >= this->pageaddress && pLargeObjectHeader->index <this->allocblockcount  &&
        this->allocBlockAddressArray[pLargeObjectHeader->index] == pLargeObjectHeader)
    {
            //push one object info into stack.
            stackpointer = pMarkContext->stackpointer;
            *strackpointer = address;
            *(stackpointer+4) = pLargeObjectHeader->blockSize;
            pMarkContext->stackpointer +=8;
    }
}
```

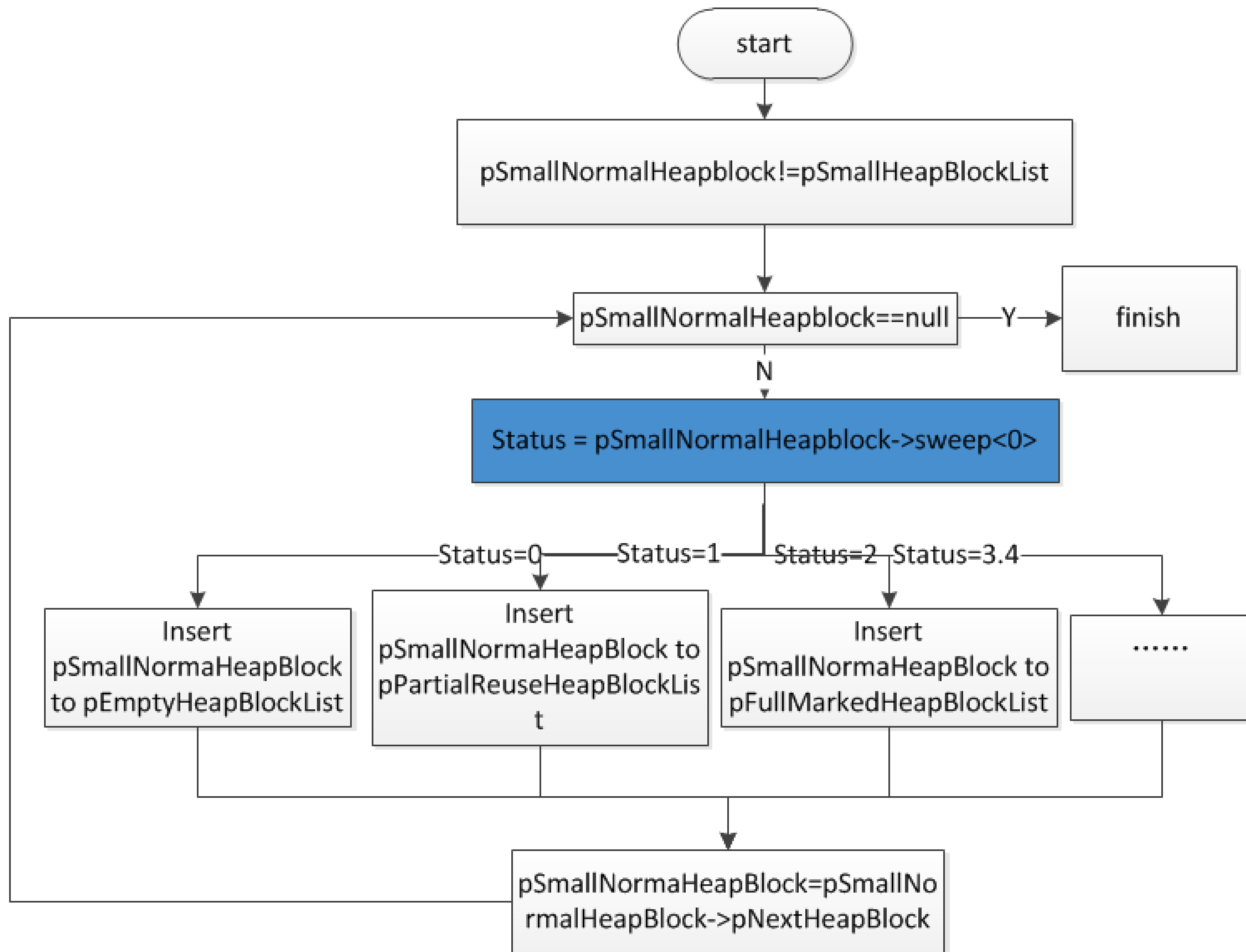
HeapInfo::Sweep<0>

```
HeapInfo::Sweep<0>( RecyclerSweep* pRecyclerSweep, bool flag)
{

    for( var i=0;i<0x40;i++)
    {
        HeapBucketGroup pHeapBucketGroup = &this->m_HeapBucketGroup[i];
        SmallFinalizableHeapBucketT<SmallFinalizableHeapBlock>* pHeapBucket =
&pHeapBucketGroup->m_HeapBucketT<SmallFinalizableHeapBlock>;
        pHeapBucket->Sweep0( pRecyclerSweep );
        SmallFinalizableHeapBucketT<SmallFinalizableWithBarrierHeapBlock>* pHeapBucket =
&pHeapBucketGroup->m_HeapBucketT<SmallFinalizableWithBarrierHeapBlock>;
        pHeapBucket->Sweep0( pRecyclerSweep );
    }

    this->SweepSmallNonFinalizable<0>(pRecyclerSweep);

    for( var i=0;i<0x20;i++)
    {
        LargeHeapBucket* pLargeHeapBucket = &this->m_LargeHeapBucket[i];
        pLargeHeapBucket->Sweep<0>( pRecyclerSweep);
    }
    this->m_LastLargeHeapBucket.Sweep<0>( pRecyclerSweep);
}
```

HeapBucketT<SmallNormalHeapBlock>

HeapBucketT<SmallNormalHeapBlock>	
0x20	pPartialReuseHeapBlockList
0x24	pEmptyHeapBlockList
0x28	pFullMarkedHeapBlockList
0x2c	pPendingNewHeapBlockList

SweepSmallHeapBlock



SmallHeapBlock::sweep<0>

- return status
  - 0: no object marked
  - 1 : partial object marked
  - 2 : all object marked
  - 3: never come here
  - 4: some pending heapblock

`SmallHeapBlock::Sweep<0>`

1、Calculation freecount

2、Calculation markcount

3、Sweepcount = objectCapacity – markcount-freecount

SmallHeapBlock::Sweep<0>: Part I

```
SmallHeapBlock::Sweep<0>(RecyclerSweep *pRecyclerSweep,pendingflag,flag,pSmallHeapBlock->unknowncount,flagDispose )
 {
     if(flag)
     {
         //if pFreeHeapObject ! = pLatestSweepFreeHeapObject, after the last sweep, some free object alloced
         //need
         if(this->pFreeHeapObject != this->pLatestSweepFreeHeapObject)
         {
             this->freeobjectCount = this->BuildFreeBitVector( this->pFinalizeBitMap);
             this->pLatestSweepFreeHeapObject = this->pFreeHeapObject;
         }

         pRecycler = *pRecyclerSweep;
         if(*(pRecyler->bPartialCollectMode)
         {
             int freeblock =this->freeobjectCount;
             int  lastSweepfreeblock = this->lastetSweepfreeblockCount;
             int blockSize = this->blockSize;
             int sub = lastSweepfreeblock - freeblock;
             int subtotal = sub*blockSize;
             this->lastetSweepfreeblockCount = this->freeobjectCount;
            * (pRecyclerSweep + 0x1424) +=aa;
             *(pRecycler + b5c0) += aa;
         }
         else
         {
            //update lastetSweepfreeblockCount
            this->lastetSweepfreeblockCount= this->freeobjectCount;
            this->freeblock3e = this->freeobjectCount;
         }
     }
```

SmallHeapBlock::Sweep<0>: Part II

```
    if( this->freeobjectCount)
    {
        pMarkBitMapTable = this->pMarkBitMapTable;
        pMarkBitMapTableEnd = pMarkBitMapTable+32;
        pTemp=pMarkBitMapTable;
        pFreeBitVector = &this->freeBitVector
        //Recalculated markbittable
        do(
            *pTemp =*pTemp & ( ~*(pFreeBitVector +pTemp-
pMarkBitMapTable ) );
            pTemp += 4;
        )while( pTemp !=pMarkBitMapTableEnd )
    }
```

SmallHeapBlock::Sweep<0>: Part III

```
     blockSize = this->blockSize;
     pinvalidBitsData =
chakra!HeapInfo::ValidPointersMap::invalidBitsData;
     pInvalidBitsDataBegin = pinvalidBitsData+32*( blockSize>>4);
     pMarkBitMapTable = this->pMarkBitMapTable;
     pMarkBitMapTableEnd = pMarkBitMapTable+32;
     //Recalculated markbittable
     for(; pMarkBitMapTable!=pMarkBitMapTableEnd;pMarkBitMapTable+=4)
     {
         int value = *pInvalidBitsDataBegin;
         pInvalidBitsDataBegin+=4;
         *pMarkBitMapTable= *pMarkBitMapTable&(~*value);
     }
```

SmallHeapBlock::Sweep<0>: Part IV

```
      int markcount = 0;
      pMarkBitMapTable = this->pMarkBitMapTable;
      //Calculation mark object number
      for(int i=0;i<8;i++)
      {
          markcount
+=BVUnitT<unsigned_int>::CountBit(pMarkBitMapTable + 4*i );
      }
```

SmallHeapBlock::Sweep<0>: Part V

```
      if( pendingDisposeCount || markcount!=0)
      {
         result = 1;
LABEL_25:
         //have dispose object, return 3.
         if( flagDispose)
             result = 3;
         if( unmark)
         {
             if(pendingflag)
             {
                 *(pRecyclerSweep+0x141a) = 1;
                 result = 4;
                 this->0x0e = 1;
                 return result;
             }
             this->SweepObjects<0>(* pRecyclerSweep);
             if( this->IsIsAnyFinalizableBlock()&& (this-
>0x6aunknownDispose1))
                 return 3;
         }
         else
             //all object marked, return 2
             if(!freeblock)
                 return 2;
         return result;
      }

      if( flagDispose)
         goto Label_25:
      return 0;
 }
```

Calculation markcount

markbitmap

&~

invalidBitsData

&~

freebitvector

=

New markbitmap

Why need invalidBitsData, freebitvector

- MemGC is a Conservative GC, does not distinguish between data and pointers.
- X is a data manager by MemGC
  - X.value == freeobjectA.address or X.value == InvalidPointer
  - In MemGC mark phase, it will mark the address(x.value).

Sweepcount!=0

- sweepcount= objectCapacity - freeCount-markcount

SmallHeapBlock::SweepObjects I

```
SmallHeapBlock: : SweepObjects<0>(Recycler* pRecycler)
{
    startaddress = this->startaddress;
    pMarkBitMapTable = pSmallHeapBlock->pMarkBitMapTable  ;
    blockCount = this->blockCount( 0x34)
    blocksize = this->blocksize ( 0x36)
    tyeparray=  (*BYTE)this-1;
    vartype = *typearray
    tempaddress = startaddress
    freeBitVector = pSmallHeapBlock->freeBitVector
    //while( ) process each object in the SmallHeapBlock
    where(blockcount--)
    {
        //tempaddress unmarked
        if(tempaddress unmark in pMarkBitMapTable and tempaddress unkmark in
freeBitVector)// check in Bitmapaddress)
        {
            //vartype not implict root
            if(vartype & 0x80 ==0)
            {
                //link the tempaddress into the freeheapobject list
                pFreeHeapObject = this->pFreeHeapObject;
                pFreeHeapObject = pFreeHeapObject | 1;
                *tempaddress = pFreeHeapObject;
                this->pFreeHeapObject = tempaddress;
                *typearray = 0;
            }
```

pFreeHeapObject

SmallNormalHeapBlock
0x24: pFreeHeapObject

0x00:pNextHeapObject

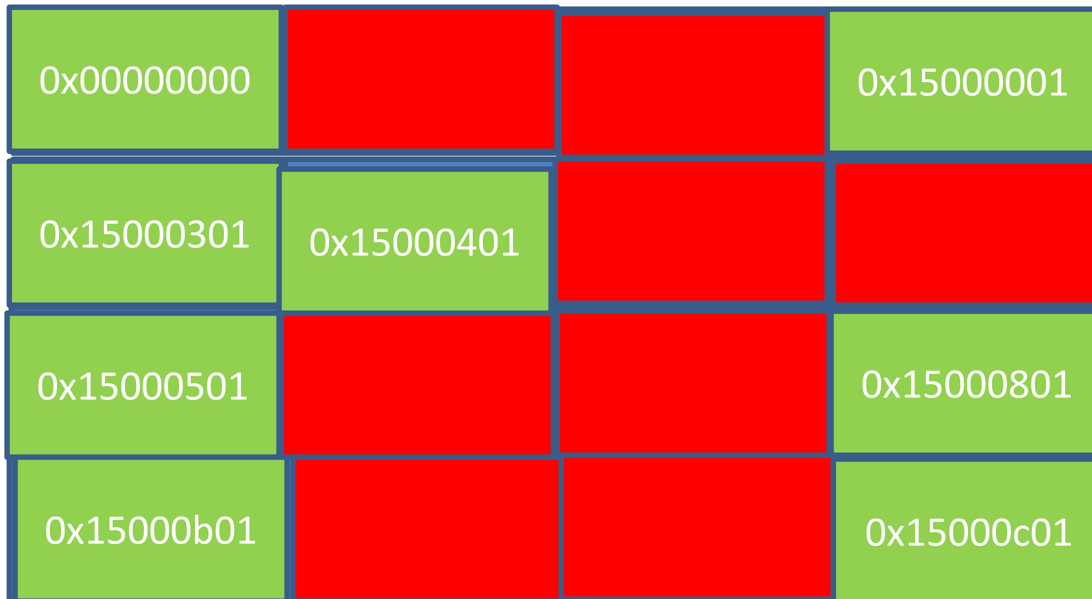
0x00:pNextHeapObject

0x00:pNextHeapObject

0x00:pNextHeapObject

SmallHeapBlock mark-sweep example

PageAddress: 0x15000000



0x15000f01

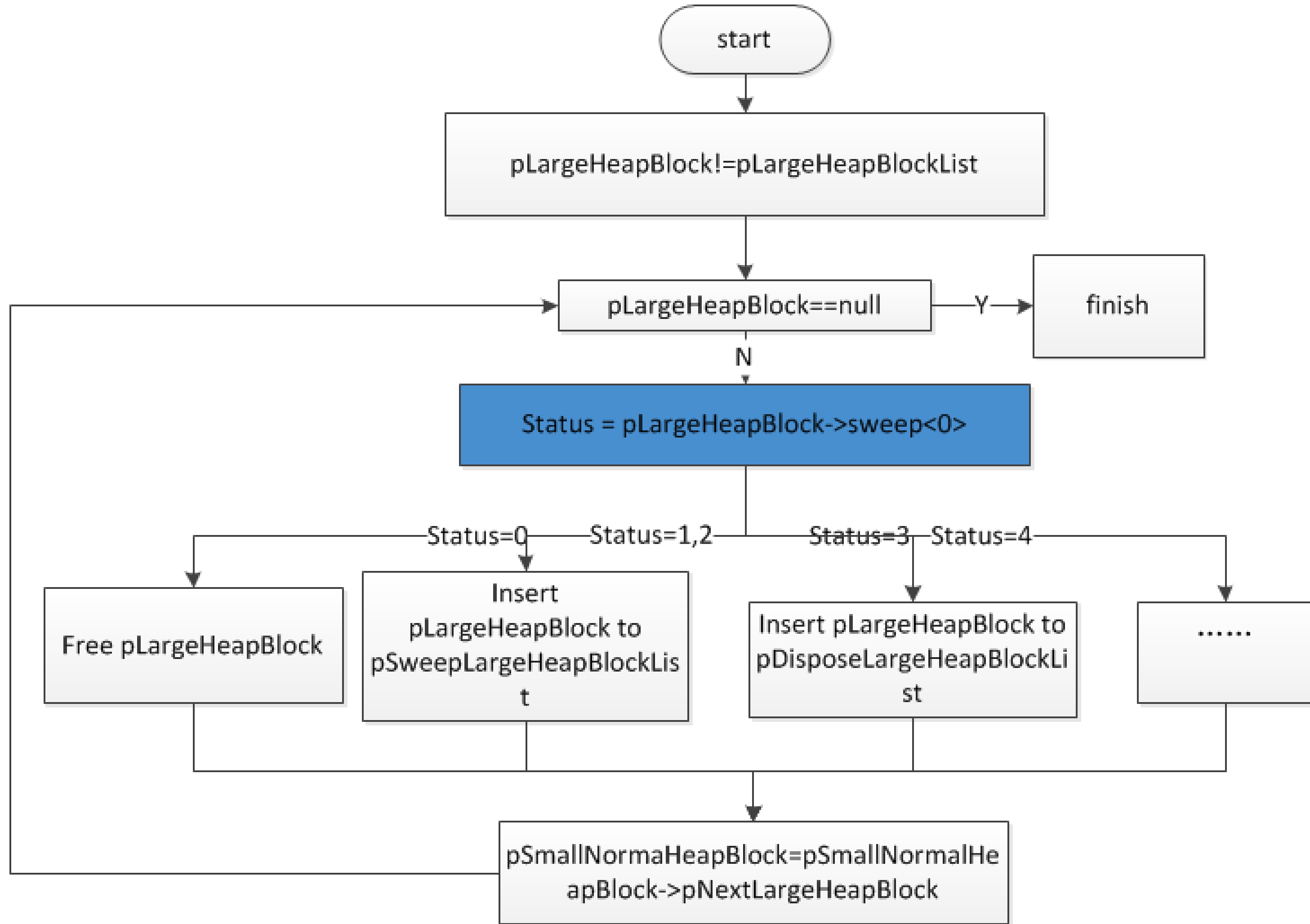
LargeHeapBucket::Sweep<0>

```
LargeHeapBucket::Sweep<0>(RecyclerSweep* pRecyclerSweep)
{
    LargeHeapBlock*
pNewLargeHeapBlockList,pSweepLargeHeapBlockList,pUnknown1,pDisposeLargeHeapBlockLi
st;
    pNewLargeHeapBlockList = this->pNewLargeHeapBlockList;
    pSweepLargeHeapBlockList  = this->pSweepLargeHeapBlockList;
    //pUnknown1 always null.
    pUnknown1 = this->pUnknown1;
    pDisposeLargeHeapBlockList = this->pDisposeLargeHeapBlockList;

    this->pNewLargeHeapBlockList =0;
    this->pSweepLargeHeapBlockList =0;
    this->pUnknown1 = 0;
    if( this->freelistflag)
    {
        this->pExplicitFreeListHead = 0;
        this->pFreeListHead = 0;
    }
    //sweep pNewLargeHeapBlockList, pSweepLargeHeapBlockList,
pUnknown1,pDisposeLargeHeapBlockList list.
    this->SweepLargeHeapBlockList<0>(pRecyclerSweep,pNewLargeHeapBlockList );
    this->SweepLargeHeapBlockList<0>(pRecyclerSweep,pSweepLargeHeapBlockList );
    this->SweepLargeHeapBlockList<0>(pRecyclerSweep,pUnknown1 );
    this->SweepLargeHeapBlockList<0>(pRecyclerSweep,pDisposeLargeHeapBlockList );
}
```

LargeHeapBucket

LargeHeapBucket	
0x0c	pSweepLargeHeapBlockList
0x10	pNewLargeHeapBlockList
0x14	pUnknown1
0x18	pDisposeLargeHeapBlockList
0x1c	pPendingLargeHeapBlockList

SweepLargeHeapBlock



LargeHeapBlock::Sweep<0>

- Return status
  - 0: no object marked
  - 1 : partial object marked
  - 2 : all object marked
  - 3: never come here
  - 4: some pending heapblock

LargeHeapBlock::Sweep<0>

- calculation markcount

LargeHeapBlock::Sweep<0>

```
int LargeHeapBlock::Sweep<0>( )
{
    int result;
    //Calculate marked object number in the LargeHeapBlock
    int markCount = this->GetMarkCount( );
    //markcount==0 && this->sweepFlag==0, return zero, It indicates that the memory
in the largeheapBlock can be released
    if( markCount ==0 && this->sweepFlag==0)
    {
        Recycler::EventWriteFreeMemoryBlock( );
        result = 0;
    }
    else
    {
        //some object alloc from LargeHeapblock need sweep.
        if( markCount != this->allocblockcount)
        {
            this->SweepObjects<0>()
        }
        if( this->pDisposeObjectList) //largeheapblock->0x3c
            result = 3;
        else
        {
            //have reuse heap object, return 1.
            if( (this->blockCapacity != this->allocblockcount  && this->endAddress-
this->allocAddress >=0x400) || this->pFreeHeapObject!=0)
            {
                result = 1;
            }
            else
            //all object marked, return 2.
                result = 2;
        }
    }
```

calculation markcount

```
for( i=0;i< allocblockcount;i++)
{
        objectheader = allocBlockAddressArray[i];
        if( ismarked( objectheader + 0x10 ))
                markcount++;
}
```

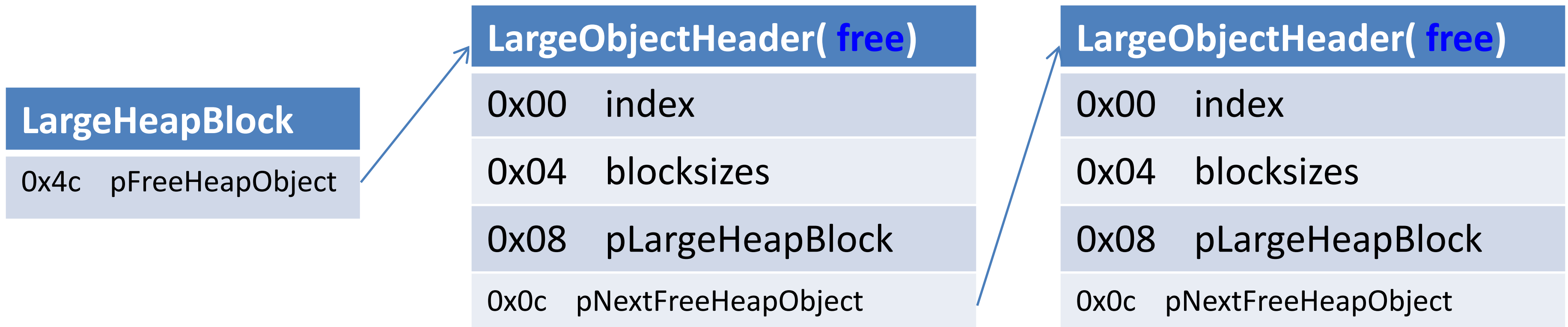
Need sweep

- `markCount!=allocblockcount`

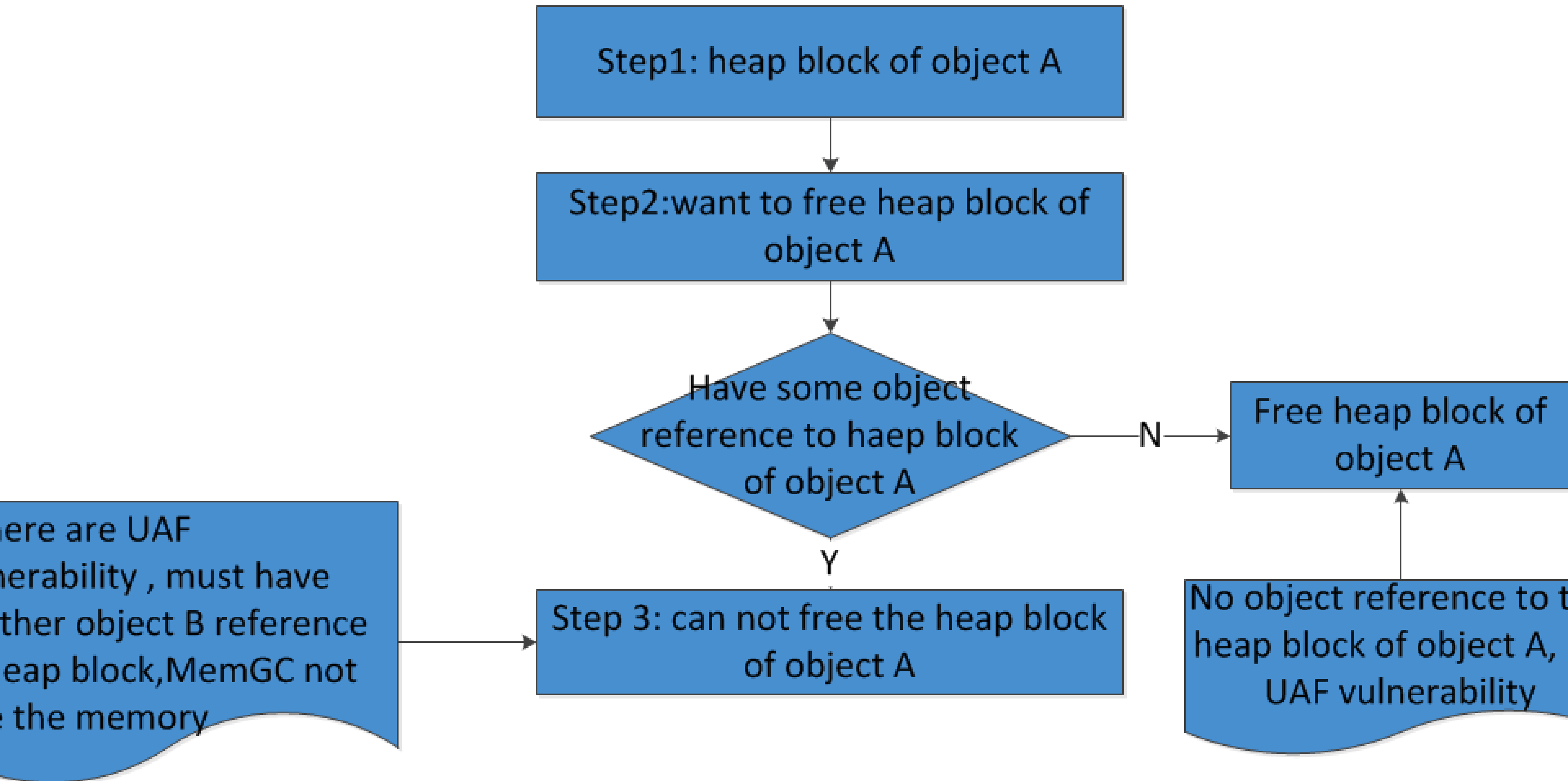
int LargeHeapBlock::SweepObjects<0>

```
int LargeHeapBlock::SweepObjects<0>(Recycler* pRecycler )
{
    flag = this->0x54;
    if (flag)
    {
        pAllocBlockAddressArray = this->allocBlockAddressArray;
        int index = 0;
        //do{}while{} get each object alloc from the LargeHeapBlock, check the object status,if unmark, sweep
the object.
        do
        {
            tempAllocAddress = *pAllocBlockAddressArray;
            //check the tempAllocAddress is valid
            if( (tempAllocAddress & 1) ==0 && tempAllocAddress )
            {
                //get the tempAllocAddress  index in m_pL2MapChunkArray
                int first_index = (tempAllocAddress + 0x10) >> 20;
                L2MapChunk* pL2MapChunk = pRecycler->m_pL2MapChunkArray[ first_index];
                //if tempAllocAddress not marked, sweep the object.
                if(!tempAllocAddress mark in pL2MapChunk->m_markbitmaptable)
                {
                    LargeObjectHeader* pLargeObjectHeader =(LargeObjectHeader*) tempAllocAddress;
                    blockSize = pLargeObjectHeader->blocksizes;
                    this->SweepObject<0>( pRecycler,tempAllocAddress );
                    //if LargeHeapBlock::pLargeHeapBucket not null, link the object into pFreeHeapObject.
                    if( this->pLargeHeapBucket)
                    {
                        pFreeHeapObject = this->pFreeHeapObject;
                        pLargeObjectHeader->index = index;
                        pLargeObjectHeader->pLargeHeapBlock = this;
                        pLargeObjectHeader->blocksize = blockSize;
                        pLargeObjectHeader->pNextFreeHeapObject = pFreeHeapObject;
                        this->pFreeHeapObject = tempAllocAddress;
                    }
                }
```

free largeobject list



Prevent the UAF'S exploit



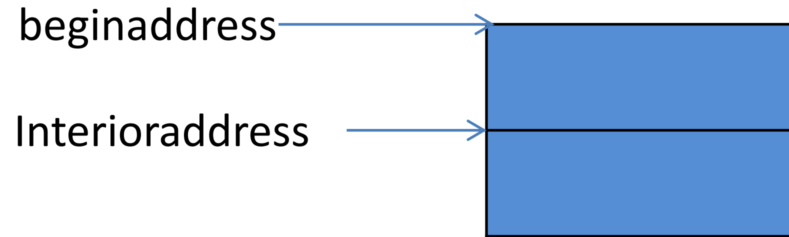
Weakness of MemGC

- Conservative GC
- Interior Pointer
- Cross-reference in different heap
- MemGC heap metadata

Conservative GC

- MemoryProtection weakness: bypass ASLR
- Yuange find the jscript9 GC infoleak vulnerability. Ga1ois first finish the poc on IE11.

Interior Pointer



Chakra GC Interior Pointer

- Interior Address >0x10000
- Interior Address & 0x0f ==0
- GetHeapBlock !=null
- Mark
- invalidBitsData
- can UAF

MemGC Interior Pointer

- Interior Address >0x10000
- GetHeapBlock !=null
- Mark
  - Realaddress =SmallHeapBlock::GetRealAddressFormInterio
    - validPointersBuffer
  - realaddress=LargeHeapBlock::GetRealAddressFormInterior
- May be memory leak

Cross-reference in different heap

- CVE-2015-2425 UAF
- FREE OBJECT IN CustomHeap::Heap
- reuse object in chakra GC Heap

MemGC heap metadata

- LargeHeapBlock::pFreeHeapObject
  - LargeobjectHeader
- SmallHeapBlock::pFreeHeapObject
- LargeHeapBlock::pPrevFreeList
- LargeHeapBlock::pNextFreeList

Thank you!

Any question?