



[Development Lifecycle](#) » [Debug Tips](#) » [General](#)

## Debug Tutorial Part 3: The Heap

By [Toby Opferman](#)

Introduction to the heap.

ASM, VC6, VC7, VC7.1Win2K, WinXP, Win2003, Visual Studio, Dev, QA

Posted: **21 Mar 2004**

Views: **168,576**

Bookmarked: **191 times**

82 votes for this article.

Popularity: **9.47** Rating: **4.95** out of 5



[QuickView: System Explorer - 90.6 Kb](#)

### Introduction

In the previous installment of this debug series, we learned about the stack. The stack is a temporary storage location for local variables, parameters, return addresses, and just about anything the compiler wants to use it for. In this installment of the debug series, we will learn about the heap in usermode.

### What's the heap?

The heap is just a block of memory within the process space that can be used to provide memory to an application when requested. The system APIs generally allocate a block of memory and serve it to the application as requested. The APIs add a header to each requested memory location that helps the system mark it as in-use and its size. This helps the system when it frees the memory. (Global variables are also on the heap).

The methods I am talking about above are completely in usermode. This block of memory is already allocated into your process space for your application to use. This chunk of memory is allocated when the specific library starts, for example in the `DllMain()` of `MSVCRT.DLL`. The code that handles `malloc()` generally works of this already allocated chunk of memory.

Memory heaps can be allocated into your process using functions such as `HeapCreate`. This function allocates a heap segment and returns you the segment number. Your application can then pass this segment into `HeapAlloc` to allocate memory from this heap. This works the same way as I described `malloc` as working, the `Heap*` functions perform all management of memory on the segment you allocated. This is actually what `malloc()` uses internally. A segment is allocated specifically for `malloc()` function calls.

There is another function termed `VirtualAlloc`. This function allocates memory on a larger scale for application use and commits the pages to memory. This function does not use a pre-allocated heap such as `HeapAlloc` does (the one you supply from `HeapCreate`) and you can even specify the location of the memory, but it's not required. This is a more advanced method of allocation and generally is not used nor needed in most application programs.

This is one of the main reasons you cannot free memory with a different function pair than you allocated (i.e., allocate with `LocalAlloc()` and free with C's `free()`). This is because they could have been allocated using different methods or different underlying heaps. The underlying code is generally stupid and usually attempts to free it anyway causing corruption in memory. This is also why any module that allocates memory should always free it. What if you allocate memory in one DLL and free in another? It may work some of the time, but maybe, one day you throw a debug DLL to replace one of them. I guarantee you will trap, as the debug heap allocations will contain different information than the release allocations! That's a bad habit! Allocate and free in the same module using the matching function pair!

### Allocated Memory is not zero?

I mentioned this in the last tutorial that you may see values on the stack that don't make sense such as return values to function calls that currently haven't happened and etc. The stack is generally initialized to zero, however, during its use it becomes dirty. You know that local variables aren't always initialized, so if you make a stack call, those values aren't reset to zero when the stack moves up. If you pop a value off the stack, the stack may decrement but the values stay unless they are physically cleaned up. Sometimes the stack optimizes things and doesn't clean up variables as well. So, seeing "ghost" values on the stack is very common. The same thing can occur with the heap.

Freeing a location on the heap does not clear its value to zero unless you have done so before you freed the memory. So, it's very common that after allocating memory, your pointer will point to "garbage". This is generally why you usually initialize memory to zero after you've allocated it.

It may seem funny to "zero memory" before freeing it, however, if your application contains sensitive information such as passwords, or the like, it's probably best to zero out even stack variables before returning. This helps erase the sensitive information after their use. You wouldn't want your program to trap, and right there on the stack or on the heap is the user's password!

## Heap Problems

In this article, I will go over the two most common problems with the heap. These are "Memory Leaks" and "Heap Corruption".

### Memory Leaks

If you've ever noticed that over time you eventually start to see your program's memory foot print grow, using Task Manager, you may realize you have a memory leak. A memory leak basically means that you are allocating memory and not freeing it. The "leak" really occurs when the program "forgets" or drops the allocated memory into space never to reference it or even remember it's there all together. This is when we start the actual "leak". The memory footprint of a process can grow without leaking, the program may just require a lot of memory. So, where do we begin?

The first thing I would do is check task manager. If it's a fast leak, the memory may go up very fast. If it's a slow leak, the memory may go up gradually over time. First, you need to know how to read task manager.

1. **Virtual Memory** - This field is how much memory is unique to this process. If the process was to be paged out to disk, this is how much it would take up in the paging file.
2. **Memory Usage** - This is the "working set" or how much memory is in physical memory. Some people get confused when they see this number bigger than the Virtual Memory size. This is because this number does not just include memory "unique" to this process but also includes memory that is shared by other processes. For example, a lot of processes use *kernel32.dll* and to duplicate the same code for every process would be wasteful.

So, once you have determined the leak, you need to find the problem. Heap problems are some of the hardest problems to track down. One thing that can help you is that, if it is a leak, most likely they came from the same source. Even if there are multiple sources, they all will leak a lot of data. This means that usually the case is that the stored data will all be similar.

If the stored data is similar, you just need to find any memory that has been allocated in large amounts and examine it. To find the data, here are some tips:

1. Generally, memory allocated from the same place are "usually" the same size. This is true for fixed structures, so you need to find allocations that are many and of similar size. This does not always have to be the case, for example, dynamically allocated strings based on a varying size.
2. The memory allocated from the same place generally will contain the same or similar information or type of information. This means once you find memory, examine it. Compare different memory locations and see if they are similar. If you know data structures, etc., you could verify the size and match the data in the memory to attempt to come up with an educated "guess" as to what the memory actually is. This can then help narrow down where to look for such memory allocations in your code.

I have written a small program that causes a memory leak. This program is very simple and we will

go through a simple scenario. This will help to get you acquainted with the heap.

Now, when we see a memory leak in Windows in your program, it doesn't always mean it's from your application. If you use a third party library or DLL, perhaps they're leaking the memory. Perhaps if you see a memory leak, but you're not directly "allocating" memory, then something you are doing could be indirectly allocating memory. For example, say you want to open a registry key and you use `RegOpenKey`. Do you know how that's implemented? No. So, if you leaked the "handle", could there be some memory associated with it? There could be. "Handle" leaks are another problem I will deal with in a later tutorial.

Now, I'm not saying that leaking a registry key would cause memory to grow and I'm not saying that leaking memory in another module would be shown by a handle leak. I'm just stating that manipulating and interacting with other modules could indirectly cause leaks that you are still responsible for. If the library you are using specifies you must free or call some destroy function, you should follow the instructions!

So, we have ran the program and we noticed the memory has gone up. Let's see if we can find out where it is. First, we find the PID of the process and then we can use "CDB -P <PID>". You could also use WinDbg's GUI and select the process from a list. The first thing we do once we are broken into the program is "!heap" to display the heaps of the process.

```
0:000> !heap
NtGlobalFlag enables following debugging aids for new heaps:      tail checking
                        disable coalescing of free blocks
Index  Address  Name           Debugging options enabled
  1:   00140000                tail checking free checking validate parameters
  2:   00240000                tail checking free checking validate parameters
  3:   00250000                tail checking free checking validate parameters
  4:   00320000                tail checking free checking validate parameters
0:000>
```

The next thing we will do is examine all the heaps and find out which ones are hording the most memory.

```
0:000> !heap 00140000
Index  Address  Name           Debugging options enabled
  1:   00140000
      Segment at 00140000 to 00240000 (00100000 bytes committed)
      Segment at 00510000 to 00610000 (00100000 bytes committed)
      Segment at 00610000 to 00810000 (00051000 bytes committed)
  2:   00240000
  3:   00250000
  4:   00320000
0:000> !heap 00240000
Index  Address  Name           Debugging options enabled
  1:   00140000
  2:   00240000
      Segment at 00240000 to 00250000 (00006000 bytes committed)
  3:   00250000
  4:   00320000
0:000> !heap 00250000
Index  Address  Name           Debugging options enabled
  1:   00140000
  2:   00240000
  3:   00250000
      Segment at 00250000 to 00260000 (00001000 bytes committed)
  4:   00320000
0:000> !heap 00320000
Index  Address  Name           Debugging options enabled
  1:   00140000
  2:   00240000
  3:   00250000
  4:   00320000
      Segment at 00320000 to 00330000 (00010000 bytes committed)
      Segment at 00410000 to 00510000 (000ee000 bytes committed)
0:000>
```

In bold, we have the segments that have the most memory committed. We will start with the first heap.

```

0:000> !heap 00140000 -a
Index  Address  Name           Debugging options enabled
1:    00140000
Segment at 00140000 to 00240000 (00100000 bytes committed)
Segment at 00510000 to 00610000 (00100000 bytes committed)
Segment at 00610000 to 00810000 (00051000 bytes committed)
Flags:                50000062
ForceFlags:           40000060
Granularity:          8 bytes
Segment Reserve:      00400000
Segment Commit:       00002000
DeCommit Block Thres: 00000200
DeCommit Total Thres: 00002000
Total Free Size:      00000226
Max. Allocation Size: 7fffdefff
Lock Variable at:     00140608
Next TagIndex:        0000
Maximum TagIndex:     0000
Tag Entries:          00000000
PsuedoTag Entries:    00000000
Virtual Alloc List:   00140050
UCR FreeList:         00140598
FreeList Usage:       00040000 00400000 00000000 00000000
FreeList[ 00 ] at 00140178: 00660118 . 00660118
Unable to read nt!_HEAP_FREE_ENTRY structure at 00660118
FreeList[ 12 ] at 00140208: 0023ff78 . 0023ff78
Unable to read nt!_HEAP_FREE_ENTRY structure at 0023ff78
FreeList[ 36 ] at 00140328: 0060fe58 . 0060fe58
Unable to read nt!_HEAP_FREE_ENTRY structure at 0060fe58
Segment00 at 00140640:
  Flags:                00000000
  Base:                 00140000
  First Entry:          00140680
  Last Entry:           00240000
  Total Pages:          00000100
  Total UnCommit:       00000000
  Largest UnCommit:     00000000
  UnCommitted Ranges:  (0)

```

```

Heap entries for Segment00 in Heap 00140000
00140000: 00000 . 00640 [01] - busy (640)
00140640: 00640 . 00040 [01] - busy (40)
00140680: 00040 . 01818 [07] - busy (1800),
  tail fill - unable to read heap entry extra at 00141e90
00141e98: 01818 . 00040 [07] - busy (22),
  tail fill - unable to read heap entry extra at 00141ed0
00141ed8: 00040 . 00020 [07] - busy (5),
  tail fill - unable to read heap entry extra at 00141ef0
00141ef8: 00020 . 002f0 [07] - busy (2d8),
  tail fill - unable to read heap entry extra at 001421e0
001421e8: 002f0 . 00330 [07] - busy (314),
  tail fill - unable to read heap entry extra at 00142510
00142518: 00330 . 00330 [07] - busy (314),
  tail fill - unable to read heap entry extra at 00142840
00142848: 00330 . 00040 [07] - busy (24),
  tail fill - unable to read heap entry extra at 00142880
00142888: 00040 . 00040 [07] - busy (24),
  tail fill - unable to read heap entry extra at 001428c0
001428c8: 00040 . 00028 [07] - busy (10),
  tail fill - unable to read heap entry extra at 001428e8
001428f0: 00028 . 00058 [07] - busy (40),
  tail fill - unable to read heap entry extra at 00142940
00142948: 00058 . 00058 [07] - busy (40),
  tail fill - unable to read heap entry extra at 00142998
001429a0: 00058 . 00060 [07] - busy (44),
  tail fill - unable to read heap entry extra at 001429f8
00142a00: 00060 . 00020 [07] - busy (1),
  tail fill - unable to read heap entry extra at 00142a18
00142a20: 00020 . 00028 [07] - busy (10),
  tail fill - unable to read heap entry extra at 00142a40
00142a48: 00028 . 00050 [07] - busy (36),
  tail fill - unable to read heap entry extra at 00142a90
00142a98: 00050 . 00210 [07] - busy (1f4),
  tail fill - unable to read heap entry extra at 00142ca0
00142ca8: 00210 . 00210 [07] - busy (1f4),
  tail fill - unable to read heap entry extra at 00142eb0

```

In order to shorten the list, I hit "Control Break" a few times to get out of the listing. I notice that after a while, the memory blocks are the same size and they're larger sizes. The listing goes like this:

```
<ADDRESS>: <Current Size> . <PREVIOUS Size>
```

If I dump one of the addresses, this is what I see:

```
0:000> dd 001457f8
001457f8 00420042 001c0700 00006968 00000000
00145808 00000000 00000000 00000000 00000000
00145818 00000000 00000000 00000000 00000000
00145828 00000000 00000000 00000000 00000000
00145838 00000000 00000000 00000000 00000000
00145848 00000000 00000000 00000000 00000000
00145858 00000000 00000000 00000000 00000000
00145868 00000000 00000000 00000000 00000000
```

The first **DWORD** is the size, but it's split into two sections. The low word is the current size and the high word is the previous size. In order to get the true size, you must shift the value left by 3. All memory is being allocated in a granularity of 8.  $42 \ll 3 = 210$  or 528. The second **DWORD** is the flags and the rest is the memory allocated to the program.

```
0:000> dc 001457f8
001457f8 00420042 001c0700 00006968 00000000 B.B.....hi.....
00145808 00000000 00000000 00000000 00000000 .....
00145818 00000000 00000000 00000000 00000000 .....
00145828 00000000 00000000 00000000 00000000 .....
00145838 00000000 00000000 00000000 00000000 .....
00145848 00000000 00000000 00000000 00000000 .....
00145858 00000000 00000000 00000000 00000000 .....
00145868 00000000 00000000 00000000 00000000 .....
0:000>
```

If I do "DC" to dump the characters, I notice the allocated memory contains the word "hi". If I continue dumping the memory, I continue to see "hi" in the memory allocated at 528 bytes. This is now starting to become a pattern. Let's move onto the next heap.

```

0:000> !heap
NtGlobalFlag enables following debugging aids for new heaps:    tail checking
                        disable coalescing of free blocks
Index  Address  Name           Debugging options enabled
  1:   00140000
  2:   00240000
  3:   00250000
  4:   00320000
0:000> !heap 00320000 -a
Index  Address  Name           Debugging options enabled
  1:   00140000
  2:   00240000
  3:   00250000
  4:   00320000
Segment at 00320000 to 00330000 (00010000 bytes committed)
Segment at 00410000 to 00510000 (000ee000 bytes committed)
Flags:                50001062
ForceFlags:           40000060
Granularity:          8 bytes
Segment Reserve:      00200000
Segment Commit:       00002000
DeCommit Block Thres: 00000200
DeCommit Total Thres: 00002000
Total Free Size:      000000b3
Max. Allocation Size: 7ffdefff
Lock Variable at:     00320608
Next TagIndex:        0000
Maximum TagIndex:     0000
Tag Entries:          00000000
PsuedoTag Entries:    00000000
Virtual Alloc List:   00320050
UCR FreeList:         00320598
FreeList Usage:       00000800 00000000 00000000 00000000
FreeList[ 00 ] at 00320178: 004fdac8 . 004fdac8
Unable to read nt!_HEAP_FREE_ENTRY structure at 004fdac8
FreeList[ 0b ] at 003201d0: 0032ffb0 . 0032ffb0
Unable to read nt!_HEAP_FREE_ENTRY structure at 0032ffb0
Segment00 at 00320640:
  Flags:                00000000
  Base:                  00320000
  First Entry:           00320680
  Last Entry:            00330000
  Total Pages:           00000010
  Total UnCommit:       00000000
  Largest UnCommit:00000000
  UnCommitted Ranges: (0)

Heap entries for Segment00 in Heap 00320000
00320000: 00000 . 00640 [01] - busy (640)
00320640: 00640 . 00040 [01] - busy (40)
00320680: 00040 . 01818 [07] - busy (1800),
      tail fill - unable to read heap entry extra at 00321e90
00321e98: 01818 . 000a0 [07] - busy (88),
      tail fill - unable to read heap entry extra at 00321f30
00321f38: 000a0 . 00498 [07] - busy (480),
      tail fill - unable to read heap entry extra at 003223c8
003223d0: 00498 . 00098 [07] - busy (80),
      tail fill - unable to read heap entry extra at 00322460
00322468: 00098 . 00028 [07] - busy (d),
      tail fill - unable to read heap entry extra at 00322488
00322490: 00028 . 000e0 [07] - busy (c8),
      tail fill - unable to read heap entry extra at 00322568
00322570: 000e0 . 000e0 [07] - busy (c8),
      tail fill - unable to read heap entry extra at 00322648
00322650: 000e0 . 000e0 [07] - busy (c8),
      tail fill - unable to read heap entry extra at 00322728
00322730: 000e0 . 000e0 [07] - busy (c8),
      tail fill - unable to read heap entry extra at 00322808
00322810: 000e0 . 000e0 [07] - busy (c8),
      tail fill - unable to read heap entry extra at 003228e8
003228f0: 000e0 . 000e0 [07] - busy (c8),
      tail fill - unable to read heap entry extra at 003229c8
003229d0: 000e0 . 000e8 [07] - busy (c8),
      tail fill - unable to read heap entry extra at 00322ab0
00322ab8: 000e8 . 00238 [07] - busy (220),
      tail fill - unable to read heap entry extra at 00322ce8

```

In this heap, I see a lot of <address>: e0 . e0 allocations. This could be my leak. If I take a look at them, I notice they are the same:

```
0:000> dc 00325188
00325188 001c001c 00180700 66647361 61667361 .....asdfasfa
00325198 73666473 73666461 73666461 61666164 sdfsadfsadfsdafa
003251a8 61736673 73616664 61736664 73616664 sfsadfasdfsadfas
003251b8 66736166 73666473 66736661 66647361 fasfsdfsafsfasdf
003251c8 00617364 baadf00d baadf00d baadf00d dsa.....
003251d8 baadf00d baadf00d baadf00d baadf00d .....
003251e8 baadf00d baadf00d baadf00d baadf00d .....
003251f8 baadf00d baadf00d baadf00d baadf00d .....
```

Another thing you could do besides !heap is to simply take the starting address of the heap allocations and continuously "DC" on the heap looking for strings or other patterns.

01c << 3 = e0 or 224. So, let's look at the source code.

```
char *p, *x;
while(1)
{
    p = malloc(200);
    strcpy(p, "asdfasfasdfsadfsadfsdafasfsadfasdfsadfasfasfsdfsafsfasdfdsa");

    x = LocalAlloc(LMEM_ZEROINIT, 500);
    strcpy(x, "hi");

    Sleep(1);
}
```

Very simple, and obviously would cause a fast leak. Notice that we allocated "224" bytes instead of "200". This is because the allocation size includes the 2 **DWORD** headers as well as must be on an 8 byte boundary. If you take the memory that you were given in your program and subtract 8, you have the header information. If you know, take the allocated size, shift left by 3, then add this to that address, you will get the next allocated block of memory. Take our last allocation as an example:

```
0:000> dc 0325188 + e0
00325268 001c001c 00180700 66647361 61667361 .....asdfasfa
00325278 73666473 73666461 73666461 61666164 sdfsadfsadfsdafa
00325288 61736673 73616664 61736664 73616664 sfsadfasdfsadfas
00325298 66736166 73666473 66736661 66647361 fasfsdfsafsfasdf
003252a8 00617364 baadf00d baadf00d baadf00d dsa.....
003252b8 baadf00d baadf00d baadf00d baadf00d .....
003252c8 baadf00d baadf00d baadf00d baadf00d .....
003252d8 baadf00d baadf00d baadf00d baadf00d .....
0:000>
```

I will not go over the heap flags as I really never bother with them myself unless really necessary. The most important flag would basically be to tell you if the memory is currently allocated or not. This is told to you when you do *!heap <heap> -a* as "busy" generally means not free and "free" means the memory is free. The "00180700" when allocated generally turns into "00180400" when free. You can experiment by writing a program to allocate and free memory while watching the flags. Just be careful that there's no other thread or the memory was allocated in the time after you freed it but before you checked its flags!

## Private Heaps Verse Global Heaps

I have introduced you to the !heap function, but there's something I should mention to you. It does not show the global heap. If you have created a global variable, it will not show up in any of the heaps listed in !heap and will not follow the same rules. The global heap cannot be destroyed while these private heaps can.

The global heap you will have problems with corruption though, not leaking; which will be the next topic for discussion. I will not go over the global heap in this tutorial though.

## Keeping Track Of Allocations

There is another method that you could use in your program to track down memory leaks. This would be the creation of your own memory allocation routines. These routines would simply behave as the following:

```
PVOID MyAllocationRoutine(DWORD dwSize)
{
    PVOID pMem = malloc(dwSize + sizeof(_DEBUG_STRUCTURE));

    if (pMem)
    {
        _DEBUG_STRUCTURE *pDebugStruc = (_DEBUG_STRUCTURE *)pMem;

        /* Fill In Your Debug Information Here */

        /* Make Sure You Give the Application the memory AFTER your debug structure */
        pMem = pDebugStruc + 1;
    }

    return pMem;
}
```

You simply append your own header to all allocations. This information could be as complex as returning the return address of who allocated the memory, thread identifier, etc. This is how programs like *boundschecker* work. They can simply replace your allocations and find out who's allocating the memory and keep track of it. Your program could do this itself. You could even create a global variable and add all the memory into a linked list and create a debug helper extension to walk through it dumping all the memory and information. Creating a debug helper extension would be in a later tutorial.

As you can see, you can simply create your own memory allocation thanks to add useful information to the allocated block to help you keep track of memory and find leaks or even memory corruption. You could even `#define` this function to only be your code when in a certain mode, such as debug. The function you used could be redefined to `LocalAlloc` or `malloc` on a normal build. You could even always use it and enable it with a special registry key or flag. There're a lot of possibilities.

Also, remember that you now need to create a "free" routine. This routine should simply subtract the size of your structure before calling the `free` API.

## Heap Corruption

Also known as "Memory Corruption", is basically when variables start overwriting their boundaries. The most common cause of simple heap corruption is caused when you use the wrong method to free the memory. For example, if you allocate with `malloc()` but free with `LocalFree()`. They use different heaps and could even use different underlying methods to allocate or keep track of memory all together. The problem is that these functions may still try to `free` the memory using their algorithms without doing validation on the heap being freed. This is when the corruption occurs. Look at the above section on creating your own `malloc()` thunk function. Other APIs could be performing their own thunks so it's always best to use the matching APIs to free the memory!

The other common causes are simply over running your boundary, such as writing to more memory than you've allocated and even simply just randomly writing to memory that the program does not own. Heap corruption can be harder to track down than memory leaks. The strategy of tracking allocations does not help, especially if you want to compile that in after the fact. This is because now you change the allocation sizes, so it's possible you will not corrupt the heap the same or perhaps there's enough padding it won't become corrupted.

I have written a program with a few heap problems. Now, heap problems do not surface immediately, they usually surface over time. They can surface when another part of the program goes to use the corrupted memory and trap, or when you free or allocate another variable.

I ran this program and got a few dialog boxes about invalid memory referencing. So, I run it in the debugger to see what happens.

```

C:\programs\DirectX\Games\src\Games\temp\bin>cdb temp

Microsoft (R) Windows Debugger Version 6.3.0005.1
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: temp
Symbol search path is: SRV*c:\symbols*http://msdl.microsoft.com/download/symbols

Executable search path is:
ModLoad: 00400000 00404000 temp.exe
ModLoad: 77f50000 77ff7000 ntdll.dll
ModLoad: 77e60000 77f46000 C:\WINDOWS.0\system32\kernel32.dll
ModLoad: 77c10000 77c63000 C:\WINDOWS.0\system32\MSVCRT.dll
(a20.710): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffdf000 ecx=00000004 edx=77f51310 esi=00241eb4 edi=00241f48
eip=77f75a58 esp=0012fb38 ebp=0012fc2c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
77f75a58 cc          int     3
0:000> g
(a20.710): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=61736664 ebx=00000004 ecx=73616664 edx=00142ab8 esi=00142ab8 edi=00140000
eip=77f8452d esp=0012f7e4 ebp=0012f9fc iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010246
ntdll!RtlAllocateHeapSlowly+0x6bd:
77f8452d 8901          mov     [ecx],eax          ds:0023:73616664=????????
0:000>

```

We receive a first chance exception. This generally means if I hit "g", the program will continue to run. If I get a "second chance" exception, the program is then trapped. However, this still does not look good. It's a good sign of memory corruption. Let's take a look. The first thing we want to do is remember what we were taught in the first tutorials. We need to find out why this memory is being referenced, who's referencing it and where it came from. The first way to do this? Our old friend, the stack trace!

```

0:000> kb
ChildEBP RetAddr  Args to Child
0012f9fc 77f9d959 00140000 50140169 00000006 ntdll!RtlAllocateHeapSlowly+0x6bd
0012fa80 77f83eb1 00140000 50140169 00000006 ntdll!RtlDebugAllocateHeap+0xaf
0012fcac 77f589f2 00140000 40140068 00000006 ntdll!RtlAllocateHeapSlowly+0x41
0012fee4 77e7a6d4 00140000 40140068 00000006 ntdll!RtlAllocateHeap+0xe44
0012fff30 00401024 00000040 00000006 00000000 kernel32!LocalAlloc+0x58
0012fff4c 0040113b 00000001 00322470 00322cf8 temp!main+0x24
0012fffc0 77e814c7 00000000 00000000 7ffdf000 temp!mainCRTStartup+0xe3
0012fff0 00000000 00401058 00000000 78746341 kernel32!BaseProcessStart+0x23
0:000>

```

So, we are allocating memory and basically NTDLL is trapping! This is a good sign of memory corruption. Let's look further, what is the memory attempted to be referenced.

```

0:000> u eip - 20
ntdll!RtlAllocateHeapSlowly+0x69d:
77f8450d 058845b356          add     eax,0x56b34588
77f84512 8b7de4             mov     edi,[ebp-0x1c]
77f84515 57                 push   edi
77f84516 e85eeaffff         call   ntdll!RtlpUpdateIndexRemoveBlock (77f82f79)
77f8451b 8b4608             mov     eax,[esi+0x8]
77f8451e 89855cffff         mov     [ebp-0xa4],eax
77f84524 8b4e0c             mov     ecx,[esi+0xc]
77f84527 898d58ffff         mov     [ebp-0xa8],ecx
0:000> u
ntdll!RtlAllocateHeapSlowly+0x6bd:
77f8452d 8901             mov     [ecx],eax

```

From the looks of this, it appears to be some type of linked list or similar data structure. I have put in bold the assembly of interest. We first see that **ECX** is being de-referenced as a pointer. **[ecx]**, as stated in another tutorial, is the same as **DWORD \*pECX; \*pECX = EAX;** in C. So, we need to

find where the `ECX` value came from. We see "`ECX, [ESI + 0Ch]`", which would basically be:

```
DWORD *pECX, *pESI; pECX = pESI[12/4];
```

Remember, in assembly there's no types so the arrays are always indexed as byte sizes rather than data type sizes. (I assume if you're reading this you know about pointers!) So, let's dump `[ESI + C]` and see what's there.

```
0:000> dc esi + c
00142ac4 73616664 66736166 73666473 66736661 dfasfasfsdfsafsf
00142ad4 66647361 00617364 feefefee feefefee asdfdsa.....
00142ae4 feefefee feefefee feefefee feefefee .....
00142af4 feefefee feefefee feefefee feefefee .....
00142b04 feefefee feefefee feefefee feefefee .....
00142b14 feefefee feefefee feefefee feefefee .....
00142b24 feefefee feefefee feefefee feefefee .....
00142b34 feefefee feefefee feefefee feefefee .....
```

The trap:

```
77f8452d 8901 mov [ecx],eax ds:0023:73616664=????????
```

This is very simple now! This is a string, so we just need to look through our program to find out where we allocated this string.

```
x = LocalAlloc(LMEM_ZEROINIT, 5);
strcpy(x, "asdfasfasdfsadfsadfsdafasfsadfasdfsadfasfasfsdfsafsfasdfsafds");
p = LocalAlloc(LMEM_ZEROINIT, 6);
strcpy(p, "hi");
LocalFree(x);
free(p);
```

As we can see, we have overwritten a lot of memory beyond our "5" bytes we wanted to allocate. This was a very simple example. Sometimes, it could just be one byte overwritten and you may need to search backwards through memory to find out if it belongs to a string or the "null" of a string. A very common mistake is to forget to leave 1 character for the `NULL`. A lot of times, this is OK as we see we allocate on granularities of 8, so it's unnoticed. Other times though, it's a nightmare to track down because once the memory has been overwritten, the culprit is **long gone** once its effect actually takes hold!

Another good thing to do is step through the program. If the place being overwritten does not change, it's simple. Step through and keep checking if it's been overwritten. You can walk functions to narrow down the location. This is also assuming a simple problem, not a complex scenario. You may also want to check other memory locations and etc. to track down.

Another good tool is the "breakpoint". "ba r1 xxxx" means "break if anyone attempts to read or write to this address". This is very useful as well if the address is constant. It will break as soon as someone attempts to write to it. There's other methods you could attempt as well, such as narrowing down the functionality that causes the problems, enabling "global flags", etc.

I know that I mentioned in the above first paragraph that the code used to check for memory leaks could not be used to check for heap corruption. This is true and false. I basically meant, in its current state, you could not expect to use it to help find memory corruption. With a little tweak, you **can** pad the end of the allocation and the beginning with values that you would not expect the program to use. Then, at the end, when you free the data, check the signature to validate it was unchanged. If it did change, it's a possible memory overwrite. This assumes your corruption is continuous or will hit the boundary and not skip it. This also assumes the problem lies in the allocated buffer and not somewhere else in your code!

## Other Tools

There are other tools that can help you spot and track down memory corruption and leaks.

## Performance Monitor

This tool comes with Windows, "perfmon". This allows you to monitor and record system performance using the options you choose. This can help you study a process' memory footprint over time. This is a better alternative than "Task Manger", for spotting slow leaks.

## Bounds Checker

This is a tool which I described above. It will help you to track down memory that is not freed at the close of your application. It will also help to find other types of leaks and even help find memory corruption. This is a very easy tool to use and will display the locations in your code which caused the allocation that was never freed. It will even tell you how much memory was not freed.

## Global Flags

There are options in the registry to enable heap validation and heap checking. There is a utility that comes with the debug tools called "gflags" that can help set these flags in the registry for you. I may go over these in a future tutorial, however the debug tools do come with documentation.

## QuickView: System Explorer

This is a utility that I have written and it is a general Windows 2000 and higher debug helper. It does not do real time data at this time and is just for exploring different aspects of the system to help you track down problems. If you want to use it, there is an .RTF file installed with it that helps explain all the features.

- [QuickView: System Explorer - 90.6 Kb](#)

## Conclusion

This tutorial should hopefully have introduced you to memory leaks and heap corruption from usermode. This tutorial should also have helped educate you in how to approach such problems with your own applications.

## License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found [here](#)

## About the Author

### Toby Opferman

Toby Opferman has worked in just about all aspects of Windows development including applications, services and drivers.

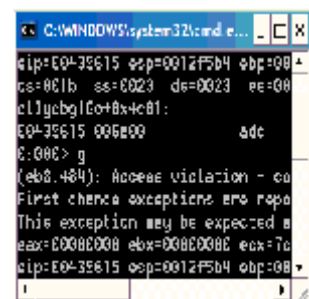
He has also played a variety of roles professionally on a wide range of projects. This has included pure researching roles, architect roles and developer roles. He also was also solely responsible for debugging traps and blue screens for a number of years.

Previously of Citrix Systems he is very experienced in the area of Terminal Services. He currently works on Operating Systems and low level architecture at Intel.

Occupation: Engineer


Company: Intel

Location:  United States



Member

## Discussions and Feedback

 **28 messages** have been posted for this article. Visit <http://www.codeproject.com/KB/debug/cdbntsd3.aspx> to post and view comments on this article, or click [here](#) to get a print view with messages.

[PermaLink](#) | [Privacy](#) | [Terms of Use](#)  
Last Updated: 21 Mar 2004  
Editor: [Smitha Vijayan](#)

Copyright 2004 by Toby Opferman  
Everything else Copyright © [CodeProject](#), 1999-2009  
Web21 | [Advertise on the Code Project](#)