

Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano
(From the NT Debugging Blog)

Hello - It's Ryan again with the second installment of my list corruption walkthrough. The previous blog post is here - Reversing in Reverse: Linked-List Pool Corruption, a Complete Walkthrough (Part 1)

In part one we walked through the analysis of a memory.dmp collected during a bugcheck caused by pool corruption. The post also discussed doubly linked lists and demonstrated an unconventional order of debugging steps in which we did not begin our examination with the backtrace of the bad pointer value.

Today's continuation will consist of another crash dump 'debugging walkthrough' explaining a lot of the typical commands used for debugging. This one involves pool corruption affecting a linked list which led us to a kernel-mode crash. I will also discuss the removal of an entry from the head of a linked list. As in the previous post, I shall provide demonstrations of the topics within the debugger in an attempt to relate the information to a real-world problem. Despite the title, we won't be working in reverse today.

As is typical, I began this windbg session with a quick `!analyze -v` to get a feel for what went wrong. `IRQL_NOT_LESS_OR_EQUAL (a)` An attempt was made to access a pageable (or completely invalid) address at an interrupt request level (IRQL) that is too high. This is usually caused by drivers using improper addresses. If a kernel debugger is available get the stack backtrace.

Arguments:

Arg1: 00000004, memory referenced <-- Looks like a BAD pointer

Arg2: d0000002, IRQL

Arg3: 00000001, bitfield :

bit 0 : value 0 = read operation, 1 = write operation

bit 3 : value 0 = not an execute operation, 1 = execute

operation (only on chips which support this level of status)

Arg4: 808436e8, address which referenced memory

TRAP_FRAME: f7906cdc -- (.trap 0xfffffffff7906cdc)

The output from this command lets us know that the code trapped accessing the invalid address 0x00000004. When we crash attempting to access low addresses like this, it is almost always due to dereferencing a null pointer. Such invalid address values are often obtained by adding some value to the pointer in an attempt to access fields in the structure that the pointer should have been referencing. Recall in part one of this blog, the invalid address was also close to zero. The invalid address in the previous blog's memory.dmp was obtained by subtracting negative values from a null pointer. For example, we may find in our analysis today that we attempted to access a structure field that was located at offset 0x4 within the structure. However, as I mentioned in part one, there are other ways that we could have ended up this value besides a NULL pointer. Some examples would include a hardware problem, following a corrupted pointer to a location that had some bad value, or some code improperly writing some value over a pointer. Let's start our analysis and identify what happened.

Dumping the stack without setting the trap frame shows the code executing within the PageFault Trap Handler

0: kd> kC

nt!_KiTrap0E ← We were in the handler for Trap 0xE

nt!MiRemoveUnusedSegments

nt!MiDereferenceSegmentThread

Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano
(From the NT Debugging Blog)

```
nt!PspSystemThreadStartup
nt!KiThreadStartup
```

```
0: kd> .formats 0xE
Evaluate expression:
Hex:      0000000e
Decimal: 14 ←Trap number 14 which per the x86 Intel Manuals is a
PageFault Trap
```

```
0: kd> rcr2 ←PageFault trap results in the address that triggered the
trap being stored in the CR2 register
Last set context:
cr2=00000004 ← Here is the invalid address
```

Next, we'll set the trap frame

```
0: kd> .trap 0xffffffff7906cdc
```

After setting the trap frame, I looked at the stack again just to get a quick overview of what calls were made in this thread:

```
0: kd> kC
nt!MiRemoveUnusedSegments <-- This is the function executing when the trap
occurred. Take notice of the function name.
nt!MiDereferenceSegmentThread
nt!PspSystemThreadStartup
nt!KiThreadStartup
```

Next, I went straight to the assembly to explore what happened. I'll begin by examining the faulting instruction.

```
0: kd> r
Last set context:
eax=00000000 ebx=80a5e540 ecx=808ab4a8 edx=00000000 esi=86f4c658 edi=80a5e4d0
eip=808436e8 esp=f7906d50 ebp=f7906d90 iopl=0          nv up ei ng nz ac po cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010293
nt!MiRemoveUnusedSegments+0x716:
808436e8 c74004a8b48a80 mov     dword ptr [eax+4],offset nt!MmUnusedSegmentList
(808ab4a8) ds:0023:00000004=????????
```

As you can see, the trap was triggered by attempting to add the value four to the null value in EAX. Also, I would like to point out that parameter number three of the bugcheck provides supporting information which is sometimes important in developing a clear understanding of what occurred:

```
0: kd> .bugcheck
Bugcheck code 0000000A
Arguments 00000004 d0000002 00000001 808436e8
```

This is illustrated by the output of !analyze -v

Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano

(From the NT Debugging Blog)

Arguments:

Arg1: 00000004, memory referenced

Arg2: d0000002, IRQL

Arg3: 00000001, bitfield :

bit 0 : value 0 = read operation, 1 = write operation

bit 3 : value 0 = not an execute operation, 1 = execute operation

(only on chips which support this level of status)

Arg4: 808436e8, address which referenced memory

0: kd> .formats 0x1

Evaluate expression:

Binary: 00000000 00000000 00000000 00000001 ←Write Operation, Not an execute operation.

If you do not need to see the entire output of !analyze -v, you can get a very abbreviated version of the !analyze-v output by using !analyze -a as displayed below.

0: kd> !analyze -a

Bugcheck Analysis

Use !analyze -v to get detailed debugging information.

BugCheck A, {4, d0000002, 1, 808436e8}

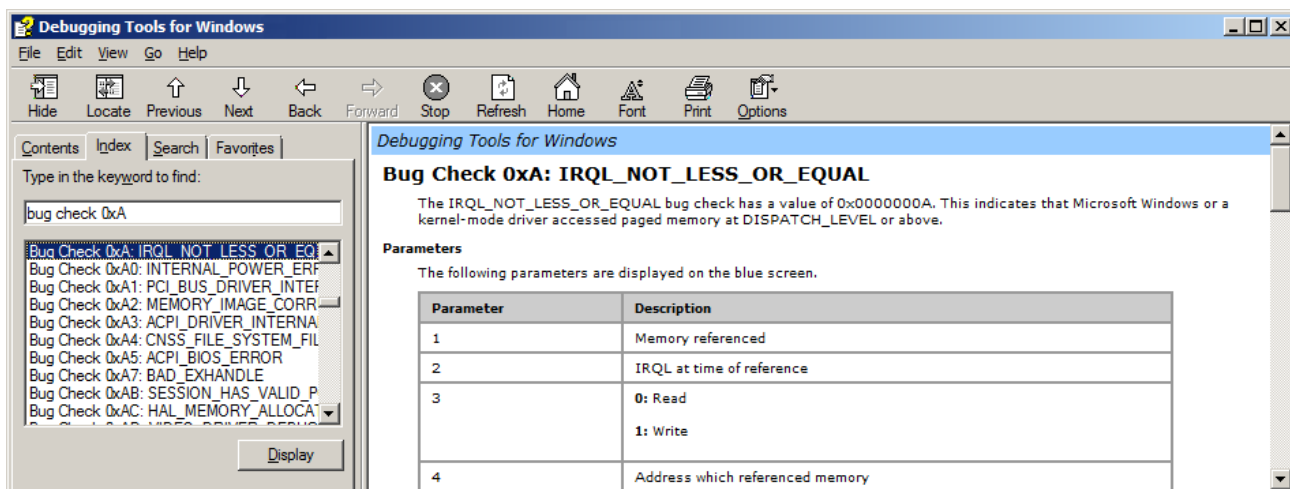
Probably caused by : memory_corruption (nt!MiRemoveUnusedSegments+716)

Followup: MachineOwner

You can get further information about this bugcheck by typing .hh bug check 0xA into the debugger

windbg> .hh bug check 0xA

This will bring up the following screen which displays helpful information.



Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano
(From the NT Debugging Blog)

Now that we understand that this crash was due to a NULL value present in EAX, the next goal of this postmortem memory.dmp debug session becomes identification of the source of this NULL value. This task can often prove extremely challenging due to the fact that it sometimes involves backtracing through several highly optimized functions full of jumps without symbols. Other times, we may be at the beginning of a very simple function for which we have private symbols. Let's see how we luck out today.

I'll proceed by disassembling the code around the point of the trap. Here is my favorite command for accomplishing this.

```
0: kd> ub @$ip L4;u . L3;r$ip
nt!MiRemoveUnusedSegments+0x70a
808436dc 8bf0          mov     esi,eax
808436de 8b06          mov     eax,dword ptr [esi]
```

The first two instructions (highlighted) together accomplish the following "Overwrite the value in EAX with the data that EAX is presently pointing to"

```
808436e0 a3a8b48a80    mov     dword ptr [nt!MmUnusedSegmentList (808ab4a8)],eax
808436e5 83c6fc        add     esi,0FFFFFFFCh
nt!MiRemoveUnusedSegments+0x716
808436e8 c74004a8b48a80  mov     dword ptr [eax+4],offset nt!MmUnusedSegmentList
(808ab4a8)      ←However EAX was unexpectedly NULL here.
808436ef ff0d8cb48a80  dec     dword ptr [nt!MmUnusedSegmentCount (808ab48c)]
808436f5 33c9          xor     ecx,ecx
Last set context:
$ip=808436e8
```

```
0: kd> reax
Last set context:
eax=00000000
```

In reviewing the code above, I observed that we had just moved this value from EAX to nt!MmUnusedSegmentList two instructions ago. To summarize, the code moved a value (that obviously wasn't expected to be NULL) to nt!MmUnusedSegmentList and then dereferenced this value (plus four) which of course caused us to crash because 0x4 is not a valid address.

Let's take a quick look at nt!MmUnusedSegmentList

```
0: kd> x nt!MmUnusedSegmentList
808ab4a8 nt!MmUnusedSegmentList = <no type information>

0: kd> dt 808ab4a8 nt!_LIST_ENTRY
[ 0x0 - 0x86f05234 ]
+0x000 Flink          : (null)
+0x004 Blink          : 0x86f05234 _LIST_ENTRY [ 0x808ab4a8 -
0x8724392c ]
```

MmUnusedSegmentList is the head of a doubly-linked list. You can see that the value that we just moved to this list head's Flink(forward link) entry is NULL. As covered in part one of this blog, a Flink is a pointer to the next _LIST_ENTRY in the list. A Blink (backwards link) is an entry to the previous

Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano

(From the NT Debugging Blog)

_LIST_ENTRY in the list . In this case we moved a NULL Flink from EAX into the list head (MmUnusedSegmentList->Flink) and crashed trying to dereference offset 4 from this same NULL value. The flink of the list head should contain a pointer to the first _LIST_ENTRY. The flink and blink should never contain NULL values. If the list is empty, both entries will be pointing to the list head ListHead->Flink and ListHead->Blink will both contain a pointer to the ListHead itself.

The symbols have provided us with some clues to what the assembly was doing. Based on the name of this list head MmUnusedSegmentList, the fact that the code seems to be decrementing MmUnusedSegmentCount, and the function name MiRemoveUnusedSegments, it appears obvious that we are trying to remove an entry from a list of unused segments. Also, I can tell that we are in a memory manager function by the Mm function prefix. You can find a list of Commonly Used Prefixes in chapter two of Mark Russinovich's book "Windows Internals 4th Edition".

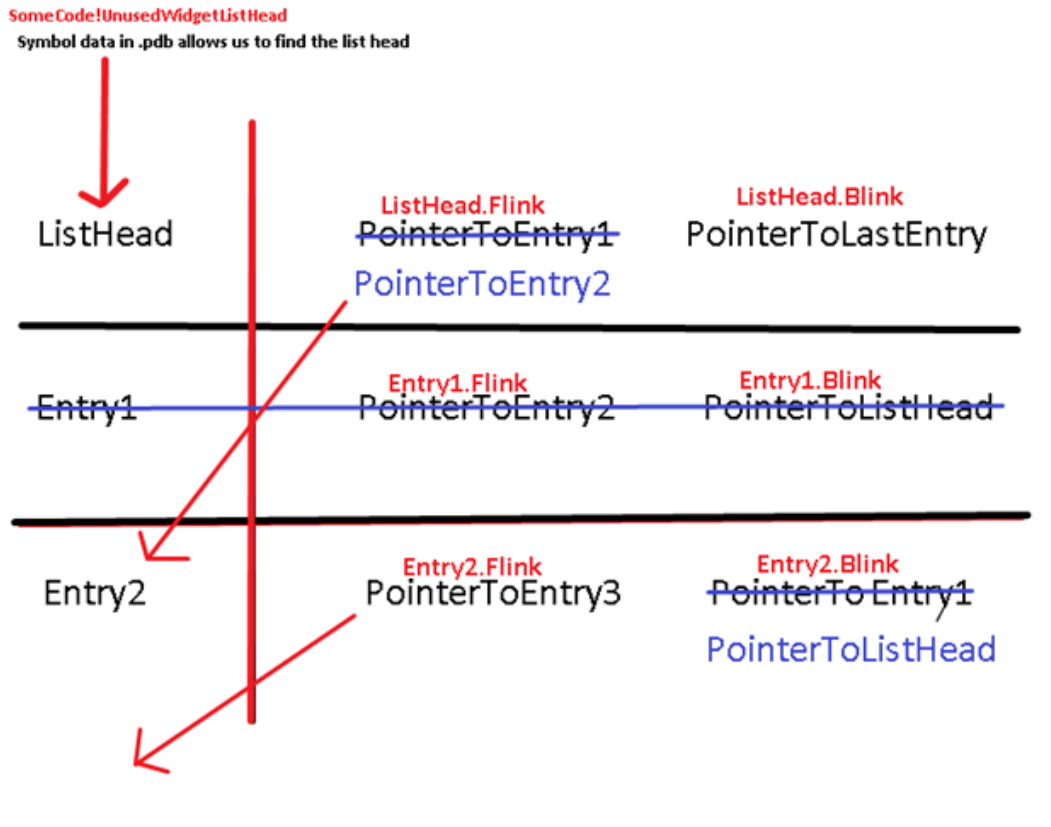
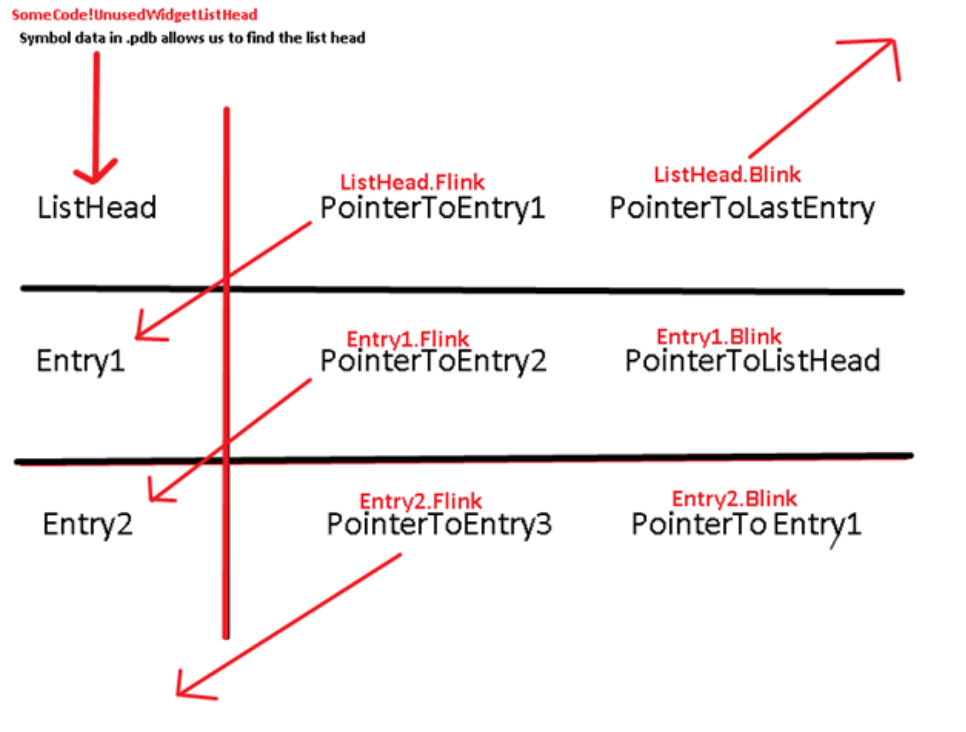
It is a common operation when working with a linked list to remove an entry from the beginning of a doubly linked list of LIST_ENTRY structures. I'll explain this by providing a simplified fictitious example. Let's pretend that we have a doubly linked list that is used in keeping track of unused Widgets. Let's also pretend that we have a function that code can call as follows:

```
pWidget = giveMeAWidget ()
```

This function first finds the head for the list of unused widgets which we are going to store in a global variable called UnusedWidgetListHead. It then finds an available widget to return by following UnusedWidgetListHead->Flink. Recall from part one of this blog, that unless the Linked List was @ offset 0x0 from the start of the WIDGET structure, the code would have to subtract the offset of the LIST_ENTRY to reach the base of the actual WIDGET structure. Before returning a pointer to the widget back to the calling code, it will be necessary to remove this widget from the doubly linked list of unused widgets since this widget will now be in use. Ignoring possible synchronization requirements, the process of removing this first entry from the list would typically involve code that sets ListHead->Flink to point to the second entry in the list instead of the first entry. It would also be necessary to update the second widget's blink member to point to the ListHead instead of the first entry that it pointed to before. This would now make entry number two the first widget therefore removing entry number one from the list.

Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano
(From the NT Debugging Blog)



Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano
(From the NT Debugging Blog)

While reviewing the code, I noticed that a pattern in the assembly that matched the type of operation that I described above where we are removing the first entry from the list. Let's review the assembly again, this time in more detail. We will break down the instructions to identify what transpired. You can use `.` or `@$ip` to represent the current instruction pointer. In this case, we'll use the pseudo-register `@$ip`.

```
0: kd> ub @$ip L4;u @$ip L1
nt!MiRemoveUnusedSegments+0x70a
808436dc 8bf0          mov     esi,eax
```

This is copying the `ListHead->Flink` to `esi`. The code uses the `ListHead` flink to find `ListMember1->Flink` (Not an actual name, I am simply referring to the `_LIST_ENTRY` as `ListMember#` just to demonstrate what is taking place). Why does it need `ListMember1->Flink`? Because it points to `ListMember2` which it needs to convert to the new `ListMember1` in the manner described above. This is very important for our debugging since it may be able to obtain the value of `Member1` from `esi`.

```
808436de 8b06          mov     eax,dword ptr [esi]
```

follow this flink that it just moved by dereferencing it to obtain `ListMember1->Flink` and place it in `eax`. This should be a pointer to `ListMember2`, however it was null.

```
808436e0 a3a8b48a80   mov     dword ptr [nt!MmUnusedSegmentList (808ab4a8)],eax
```

move this null value to `ListHead->Flink`. This operation should be setting the list head to point to `Member2`, therefore converting it to `Member1`. However since `Member1->Flink` was null, it now contains a NULL value.

```
808436e5 83c6fc       add     esi,0FFFFFFFCh
```

Earlier I mentioned that the code might be able to obtain the address of `Member1` from `ESI` since `esi` had the `ListHead->flink` pointing to it, however `ESI` was modified here. This means that it can't rely on the value of `esi`. The code added `0x0ffffffc` here; however, this was really just a compiler optimized (fancy) way of subtracting four. It doesn't care why that it's subtracting four from this value since the code crashed before ever using `address-4`, so I won't be investigating that today. However, I suspect we were subtracting the offset of the following field.

```
0: kd> dt nt!_CONTROL_AREA DereferenceList +0x004 DereferenceList : _LIST_ENTRY
```

Instead of digging into that, we simply want to identify the value of the list member. To obtain our value here, we will perform the reverse of the addition operation and simply subtract `0x0FFFFFFC` as a fancy way to add four :

```
0: kd> resi
Last set context:
esi=86f4c658
```

We effectively added 4 to get the original value of `esi`

Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano
(From the NT Debugging Blog)

```
0: kd> ? 86f4c658-0x0FFFFFFFFC
Evaluate expression: -2030778788 = 86f4c65c
```

So this address should contain a NULL flink

```
0: kd> dd 86f4c65c L2
86f4c65c  00000000 808ab4a8
```

And there it is (the underlined value above)

And as expected in the output above, the flink was null. Also note that the blink (highlighted) was in fact pointing to 808ab4a8, which is the list head. So this does appear to be the address of the original member1. If you can't recall the address of the list head, don't scroll up in the debugger text (or this blog text), we can prove this quickly as follows;

```
0: kd> dd nt!MmUnusedSegmentList L1
808ab4a8  00000000
```

Let's dump the address out:

```
0: kd> !address 86f4c65c
83041000 - 07fbf000
          Usage          KernelSpaceUsageNonPagedPool
0: kd> !pool 86f4c65c
Pool page 86f4c65c region is Nonpaged pool
86f4c000 size:  98 previous size:  0 (Allocated) File (Protected)
86f4c098 is not a valid large pool allocation, checking large session pool...
bf7f4000: Unable to get contents of pool block
```

So we are dealing with pool corruption. Now that we have followed the NULL value in the corrupt pool and loaded EAX with zero, lets proceed with the inspection of the next instruction which caused the trap leading to the bugcheck:

```
nt!MiRemoveUnusedSegments+0x716 808436e8
c74004a8b48a80  mov     dword ptr [eax+4],offset nt!MmUnusedSegmentList
(808ab4a8)
```

This instruction was attempting to place the address of the List Head into eax+4. Why? Well, EAX has the new Member1. If we add four to this value, this would bring us to Member1->Blink which should be pointing to the list head. Had the new Member1 actually been Member2 instead of NULL, then we would be overwriting the pointer to Member1 with a pointer to the list head. However, the pointer had been zeroed out and that brought us to address 4. Next, we trapped and the system bugchecked. As discussed earlier, we died on a write operation attempting to write a value to the address referenced by [EAX+4]

For completeness, let's review what the next two instructions would have executed had we not trapped

```
08436ef ff0d8cb48a80  dec     dword ptr [nt!MmUnusedSegmentCount (808ab48c)]
```

Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano
(From the NT Debugging Blog)

If the instruction above would not have trapped, then we would have decremented the count. However we did not make it this far. If you remove an item from a linked list, you should decrement the value of any variable tracking the number of items in such list. This instruction would have accomplished this.

```
808436f5 33c9          xor     ecx,ecx
```

This would have simply zeroed out ecx.

So this crash was in fact due to a null pointer. More specifically, it was caused by a null flink. Let's dump the linked list located at nt!MmUnusedSegmentCount using the dlb just to see what happens. We won't be able to go forward since the flink is null; however, we should be able to dump the linked list going backwards. If the list loops back on itself or if a null pointer is encountered, the dl command will stop traversing the list.

First, I would like to know how large this list is since the dl command accepts a value that limits its length.

The following global should tell us how many member are in this list.

```
0: kd> x nt!*UnusedSegmentCount*
808ab48c nt!MmUnusedSegmentCount = <no type information>
```

```
0: kd> dd 808ab48c L1
808ab48c 0000e7ce
```

```
0: kd> .formats 0xe7ce
Evaluate expression:
Hex:      0000e7ce
Decimal:  59342
```

This is a huge list, so let's see if we can traverse it. Based on the size above, I dumped out the list using ffff for the limit.

```
0: kd> dlb nt!MmUnusedSegmentList ffff 2
808ab4a8 00000000 86f05234
86f05234 808ab4a8 8724392c
8724392c 86f05234 8867fd94
8867fd94 8724392c 877d600c
877d600c 8867fd94 87849944
.....
.....
(omitting lengthy unneeded output, all links were valid)
.....
.....
86ecbaac 86f04354 86f95664
86f95664 86ecbaac 86f4c28c
86f4c28c 00000000 00000000 <-- After over a minute of output, we have a
null pointer, but it's at a different address.
```

Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano

(From the NT Debugging Blog)

First Address we found to be corrupt: 86f4c65c
Second Address we found to be corrupt: 86f4c28c

Now that we have located the address above which appear to be incorrectly zeroed out. Let's dump out these areas in an attempt to get more information. It would be great if we could verify if the pool that we are using is corrupt. It would be even better if we some pointers, text, or other clues that may lead us closer to the problem. Let's dump the two addresses out in various ways. I'll start by examining the pool for corruption.

The !pool extension as used below displays information about a specific pool allocation

```
0: kd> !pool 86f4c65c;!pool 86f4c28c
Pool page 86f4c65c region is Nonpaged pool
86f4c000 size: 98 previous size: 0 (Allocated) File (Protected)
86f4c098 is not a valid large pool allocation, checking large session pool...
bf7f4000: Unable to get contents of pool block
Pool page 86f4c28c region is Nonpaged pool
86f4c000 size: 98 previous size: 0 (Allocated) File (Protected)
86f4c098 is not a valid large pool allocation, checking large session pool...
bf7f4000: Unable to get contents of pool block
```

The !poolval extension analyzes the headers for a pool page and diagnoses any possible corruption.

```
0: kd> !poolval 86f4c65c;!poolval 86f4c28c
Pool page 86f4c65c region is Nonpaged pool
Validating Pool headers for pool page: 86f4c65c
Pool page [ 86f4c000 ] is __inVALID.
Analyzing linked list...
[ 86f4c000 --> 86f4c6c0 (size = 0x6c0 bytes)]: Corrupt region
Scanning for single bit errors...
None found
Pool page 86f4c28c region is Nonpaged pool
Validating Pool headers for pool page: 86f4c28c
Pool page [ 86f4c000 ] is __inVALID.
Analyzing linked list...
[ 86f4c000 --> 86f4c6c0 (size = 0x6c0 bytes)]: Corrupt region
Scanning for single bit errors...
None found
```

We can also dump out the memory around the addresses in question using various commands. As discussed previously, we are seeking clues in text format, pointers, etc. The output from the following command shows that the memory is mostly zeroed. One value appears to be present. Perhaps the entire region was overwritten and then this value was updated. Dd dumps the raw data as dwords. Dc will dump the data as type char.

```
0: kd> dd 86f4c65c-100 86f4c65c+100;dd 86f4c28c-100 86f4c28c+100
...<omitting zeros>
86f4c63c 00000000 00000000 00000000 00000000
86f4c64c 00000000 00000000 00000000 00000000
86f4c65c 00000000 808ab4a8 00000000 00000000
```

Reversing in Reverse Part 2 - More Linked-List Pool Corruption

Ryan Mangipano

(From the NT Debugging Blog)

```
86f4c66c 00000000 00000000 00000000 00000000
86f4c67c 00000000 00000000 00000000 00000000
86f4c68c 00000000 00000000 00000000
...<omitting zeros>
86f4c26c 00000000 00000000 00000000 00000000
86f4c27c 00000000 00000000 00000000 00000000
86f4c28c 00000000 00000000 00000000 00000000
86f4c29c 00000000 00000000 00000000 00000000
...<omitting zeros>
```

```
0: kd> dc 86f4c28c-1000 86f4c65c
<omitting>
86f4c62c 00000000 00000000 00000000 00000000 .....
86f4c63c 00000000 00000000 00000000 00000000 .....
86f4c64c 00000000 00000000 00000000 00000000 .....
86f4c65c 00000000 .....
0: kd> dc
86f4c660 808ab4a8
```

We can see from the command below that we are in fact dealing with a NonPagedPool address range.

```
0: kd> !address 86f4c65c;!address 86f4c28c
83041000 - 07fbf000
Usage KernelSpaceUsageNonPagedPool
83041000 - 07fbf000
Usage KernelSpaceUsageNonPagedPool
```

```
0: kd> dps 86f4c65c-8 86f4c65c+8
86f4c654 00000000
86f4c658 00000000
86f4c65c 00000000
86f4c660 808ab4a8 nt!MmUnusedSegmentList
86f4c664 00000000
```

Just as in the previous dump, in order to identify the source of the pool corruption we need to use Special Pool. Special Pool will use guard pages to catch a buffer overrun or underrun and should provide us with a dump that shows the code that causes the corruption. You can find more information on Special Pool in KB188831. Also, in some situations memory corruption can be caused by a driver overflowing on a DMA transfer causing corruption to physical pages rather than virtual pages as we see in typical pool. For more information on linked lists and list heads refer to the following MSDN article:

Singly and Doubly Linked Lists- <http://msdn.microsoft.com/en-us/library/aa489548.aspx>

For an example of a function that removes the first item of a linked list: <http://msdn.microsoft.com/en-us/library/ms804330.aspx>