

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

My name is Ryan Mangipano (ryanman) and I am a Sr. Support Escalation Engineer at Microsoft. Today's blog will consist of a complete walkthrough of my recent analysis of a stop 0x50 along with the steps that led me to identify that this crash was caused by pool corruption. In this particular case, I found the answer before completing the actual debug.

To begin the analysis, I entered the `!analyze -v` command into the debugger and examined the output relating to the bugcheck. This is always my first step since it usually provides an overview of what type of problem occurred. In the output listed below, you can see that the system has bugchecked due to an attempt by kernel mode code to access the invalid address `0xfffff50`.

**PAGE\_FAULT\_IN\_NONPAGED\_AREA (50)**

Invalid system memory was referenced. This cannot be protected by try-except, it must be protected by a Probe. Typically the address is just plain bad or it is pointing at freed memory.

Arguments:

Arg1: `fffff50`, memory referenced.

Arg2: 00000000, value 0 = read operation, 1 = write operation.

Arg3: 80846dd1, If non-zero, the instruction address which referenced the bad memory address.

Arg4: 00000000, (reserved)

Before proceeding with our debugging, I would like to point out that invalid addresses such as the one listed above are often obtained when code subtracts values from a null pointer. To illustrate that this particular address `0xfffff50` could be obtained in such a manner, I have provided output from the `.formats` debugger command below. Notice that the hex value `0xfffff50` can be used to represent negative 176 as a decimal. It is a relatively common situation to arrive at a negative value by subtracting values from a null pointer (which is zero). Since addresses are actually unsigned, this is really just an invalid positive address.

```
0: kd> .formats fffff50
Evaluate expression:
Hex:      fffff50
Decimal: -176
```

Let's proceed with our investigation. The output of `!analyze -v` also provided us with the following:

```
TRAP_FRAME: f4009940 -- (.trap 0xfffffffff4009940)
```

This provided a trap frame address in the form of a DML (Debugger Markup Language) link that can be clicked on to quickly enter the command into the debugger.

Passing `.trap` the address of a trap frame will show the register state when the trap was generated. This will also set the context to the state when the trap was generated. This information was saved into the trap frame by the trap handler that executed after the trap occurred.

The trap frame address listed above could have instead been obtained by dumping out the stack using the command `kv . !analyze -v` is very good at displaying the correct trapframe, however I feel

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

that it is a good practice to dump out the stack to verify that the trap frame you are entering is in fact the one that you are interested in. You can see in the output below that there are two trapframes listed. I am interested in the topmost one that matches the trap that led us to the bugcheck. I can see that analyze -v has listed the trapframe that we are interested in, so I entered it into the debugger.

```
0: kd> kv
ChildEBP RetAddr  Args to Child
f40098d8 808692ab 00000050 ffffffff 00000000 nt!KeBugCheckEx+0x1b (FPO: [5,0,0])
f4009928 80836c2a 00000000 ffffffff 00000000 nt!MmAccessFault+0x813 (FPO: [4,10,4])
f4009928 80846dd1 00000000 ffffffff 00000000 nt!KiTrap0E+0xdc (FPO: [0,0] TrapFrame @
f4009940)
f40099b0 809426b2 a9131e08 00000012 861ce318 nt!ObReferenceObjectSafe+0x3 (FPO: [0,0,0])
f40099c8 808544d9 861ce318 0007d960 00000000 nt!PsGetNextProcess+0x6c (FPO: [1,1,0])
f4009a58 8094292f 01210048 0007d960 00000000 nt!ExpGetProcessInformation+0x36d (FPO: [SEH])
f4009d4c 80833bef 00000005 01210048 0007d960 nt!NtQuerySystemInformation+0x11e0 (FPO:
[SEH])
f4009d4c 7c8285ec 00000005 01210048 0007d960 nt!KiFastCallEntry+0xfc (FPO: [0,0] TrapFrame
@ f4009d64)
```

```
0: kd> .trap 0xfffffffff4009940
```

The output from the .trap command will be similar to the output from the r command. The r command will dump out the registers and the current instruction that caused the error. Listed below are the output of the two commands:

```
0: kd> .trap 0xfffffffff4009940
ErrCode = 00000000
eax=861ce300 ebx=808b5be8 ecx=ffffff68 edx=ffffff50 esi=8747c8d0 edi=00000000
eip=80846dd1 esp=f40099b4 ebp=f40099c8 iopl=0         nv up ei pl nz ac pe cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010217
nt!ObReferenceObjectSafe+0x3:
80846dd1 8b0a                mov     ecx,dword ptr [edx]  ds:0023:ffffff50=?????????
```

```
0: kd> r
Last set context:
eax=861ce300 ebx=808b5be8 ecx=ffffff68 edx=ffffff50 esi=8747c8d0 edi=00000000
eip=80846dd1 esp=f40099b4 ebp=f40099c8 iopl=0         nv up ei pl nz ac pe cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010217
nt!ObReferenceObjectSafe+0x3:
80846dd1 8b0a                mov     ecx,dword ptr [edx]  ds:0023:ffffff50=?????????  □
This is the current instruction. We trapped on this instruction trying to
dereference edx which contained ffffffff50.
```

Next, we shall dump out the stack and try to get a feel for what happened based on the function names listed in the stack. Remember that the functions at the bottom of the stack called the functions above them. Here are the function names listed in the top four stack frames along with my annotation. In this case I read the stack starting at the bottom and working my way up instead of the traditional way of starting at the bad value and backtracing to the problem.

```
0: kd> kcL4
```

- nt!ObReferenceObjectSafe□ Finally, this function encountered the bad value and crashed. To observe this for yourself, please examine the output of the 'r' command listed above. You will

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

notice that the current instruction at the time of the trap (the address in EIP) was unassembled. Above this instruction the function name and offset are listed as nt!ObReferenceObjectSafe+0x3. This means that the instruction we were executing when the trap occurred was located at offset 0x3 from the location that the symbol ObReferenceObjectSafe references.

- nt!PsGetNextProcess □ Based on the function name here, it appears that we were trying to traverse the linked list of active processes on the system.
- nt!ExpGetProcessInformation □ NtQuerySystemInformation() called this function. Based on the function name GetProcessInformation and the call listed above, this was obviously an attempt to get information about the processes running on the system
- nt!NtQuerySystemInformation □ We were trying to query system information

Normally, my next step would typically be to unassemble the current function with `uf @$ip`, locate the current instruction in the output by pasting the address in eip into the find dialog box, and proceed to backtrace the source of the bad value until I had located the source. However, I have a bad habit of poking around for fun by dumping out various data while I am debugging. The function names on the stack seemed to indicate that we were working with the list of processes. The address we died on suggested that we may have encountered a null pointer. I couldn't resist pausing a moment to dump out the list of processes on the system to see if we had encountered a null pointer while traversing the list of processes on the system. After all, I've reviewed dumps before that were that simple. Even if I found a null pointer in the list, I would still have to perform the actual debug and walk the assembly to prove it and identify how we crashed. However, I felt it would be more exciting to work in the reverse of my normal methods. Besides, it only takes a moment once you know what you are doing.

Let me pause a moment to provide a little background information here which will later prove relevant in understanding my method of thinking and comprehending what went wrong. In the nt!\_EPROCESS object there is a \_LIST\_ENTRY structure located at offset 0x98 called ActiveProcessLinks. This forms a linked list of all the active NT!\_EPROCESS structures on the system. Let us first examine the \_LIST\_ENTRY structure by reviewing the output below:

```
1: kd> dt nt!_LIST_ENTRY
      +0x000 Flink           : Ptr32 _LIST_ENTRY
      +0x004 Blink           : Ptr32 _LIST_ENTRY
```

As you can see above, a \_LIST\_ENTRY contains two pointers that create what is called a linked list. Blink represents a backwards link and Flink represents a forward link. A double linked list may be traversed in either direction by following the pointer to the next or previous entry in the list. Linked lists are used heavily in operating system programming. This particular linked list contains a pointer to the previous \_EPROCESS object's ActiveProcessLinks field (blink) and the next \_EPROCESS object's ActiveProcessLinks field (flink).. The following command will display the activeprocesslinks field of the nt!\_EPROCESS structure.

```
0: kd> dt nt!_EPROCESS activeprocesslinks +0x098 ActiveProcessLinks : _LIST_ENTRY
```

the +0x098 lets us know that this field is located 0x98 from the start of each nt!\_EPROCESS structure. All of these EPROCESS structures on the system are linked together using the flink/blink

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

method. This allows us to enumerate all of the processes on the system by following the links in either direction.

The first entry of a linked list is commonly referred to as the list head. The list head is a `_LIST_ENTRY` structure that points to the first and last entry of the link list. Global variables are often used to store the location of the list head. Once you have the location of the head, you can then use it's `fblink`/`blink` values to traverse the list in either direction. `Nt!PsActiveProcessHead` is a global variable that points to the beginning of the list of Active Processes that we discussed in the previous paragraph. If you traverse this list starting at the head, you will end up in the `activeprocesslinks` field of each process on the system. This will allow code to find each `_EPROCESS` structure on the system by simply traversing this list. If you are an true geek like me and are learning this for the first time, you are probably going to fire up your debugger and dump out the list of active processes on your own system before you reach the end of this article.

Please note that by following the `fblink`/`blink` you will arrive at offset `0x098` from this structure, not the start of the structure. So the complete list consists of a `_LIST_ENTRY` field present in each `EPROCESS` structure + the `_LIST_ENTRY` located at the memory referenced by the global variable `nt!PsActiveProcessHead`. You can use the 'x' examine symbols command to display this variable. I included the wildcards below only to demonstrate the ability to use wildcards with these commands.

```
0: kd> x nt!*PsActiveProcessHead*
808b5be8 nt!PsActiveProcessHead = <no type information>
0: kd> dt nt!*PsActiveProcessHead*
808b5be8 ntkrnlmp!PsActiveProcessHead
```

Back to our investigation. At the point where we left off, we had not yet dug into the code to identify exactly what happened. We instead took an educated guess and decided to dump out the list of processes on the system. If it is found that we have a null value, this may very well allow us to work in reverse of our normal debugging. Armed with the above background information, we shall now dump out the list. If we get lucky, this will expedite the review of this memory dump by telling us what may have gone wrong before we even start digging into the code.

The beginning of the list sounds like a good place to start traversing the entries. The value highlighted in the global variable above marks the address where the linked list starts. The following command will dump out this location and provide you with the `fblink` and `blink`. The switch `/c1` was used in order to limit the output to 1 column for easy blog annotation. For more information on the `/c1` switch, type `.hh dd` into the debugger command interpreter and then press enter once the help file window appears.

```
0: kd> dd /c1 nt!PsActiveProcessHead L2
808b5be8 8a78c800 □ this is the forward link (fblink)
808b5bec 85ddd510 □ this is the backwards link (blink)
```

Now that we have the address that the list starts at, we need a way to dump it out. Windbg provides many ways to automatically walk a linked list. You can use `dt`, `!list`, or the `dl` commands. In this case, let's use the `dl` command due to its simplicity. More information about this command may be obtained from the help file that is included with windbg.

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

0: kd> dl 0x8a78c800 ff 2 □ Here is an example of how to use this command to traverse the linked list starting at address 0x8a78c800. The value ff represents the maximum number of links to display. The value 0x8a78c800 was obtained from the flink field `_LIST_ENTRY` present at the global variable listed above. The value 2 represents the number of values to dump from each address. Since this linked list contains a forward link (flink) and backwards link (blink), we should pass in a value of 2 to dump both addresses.

As displayed below, I decided to use the global `nt!PsActiveProcessHead` instead of specifying an address to the command `dl`. Here is the beginning of the output as displayed by the debugger.

```
0: kd> dl nt!PsActiveProcessHead ff 2
808b5be8 8a78c800 85ddd510
8a78c800 89af0370 808b5be8
888745a0 888492e0 89af0370
888492e0 89b17708 888745a0
89b17708 89b4e3d0 888492e0
... .. Ommitting the rest of the output
```

The plain output displayed above represents the type of output that you will see when entering the `dl` command into the debugger. In order to better illustrate the patterns and show the relationship between the flink and blink values, I have bolded, underlined, highlighted, and commented various portions of the output and listed it below. Note the annotations that demonstrate how each member of the linked list points to the next and previous item. You can follow the list all the way down until you see we do in fact end up at a zeroed out address.

```
0: kd> dl nt!PsActiveProcessHead ff 2
```

808b5be8 **8a78c800** 85ddd510 <-- Notice that the `dl` command first displays address 808b5be8 which is the address referenced by the symbol information for the global variable `nt!PsActiveProcessHead` that we provided. The next two values are the flink and the blink. I have highlighted the flink above. Notice how this flink is pointing to the next memory location listed below.

**8a78c800** 89af0370 808b5be8 <-- In this line of output, notice that the `dl` command has followed the flink to address 8a78c800 (which I have again highlighted) and is dumping out the new flink and blink pair. I have underlined the blink. Notice that the blink points back to the address of the previous flink/blink pair.

89af0370 888745a0 8a78c800 Notice the patterns displayed, follow the output all the way to the bottom, and see if you notice anything that may be a problem.

```
888745a0 888492e0 89af0370
888492e0 89b17708 888745a0
89b17708 89b4e3d0 888492e0
89b4e3d0 8883fe20 89b17708
8883fe20 89ab73e8 89b4e3d0
89ab73e8 887d7650 8883fe20
887d7650 888906a8 89ab73e8
888906a8 89483408 887d7650
89483408 88839d40 888906a8
```

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

... Ommitting output to reduce length. The links were all valid..

```
85ff6e20 8708be20 8663a278
8708be20 87176940 85ff6e20
87176940 861ce3b0 8708be20
861ce3b0 860010b8 87176940
```

860010b8 00000000 00000000 <-- Wait...This doesn't look correct. The blink should be 861ce3b0, not zero. The previous list entry's flink took us here to address 860010b8. It appears that our theory might just be correct. Observing this, I suspected that memory location 860010b8 has been overwritten with zeros. It is also possible, however, that the value in the flink above was incorrect causing us to end up at some random address that contained zeros. To know for sure I dumped the list backwards to see if the next flink/blink pair had a blink pointing to 860010b8.

0: kd> dlb nt!PsActiveProcessHead ff 2 This command dumps the list backwards. Below you can see the dlb command walking the linked list by starting at the list head and moving backwards (the list should make a circle). Don't be confused by the different patterns below. This is simply due to the fact that the list is being walked backward. The output displays the three value in the same order address flink blink.

```
0: kd> dlb nt!PsActiveProcessHead ff 2
808b5be8 8a78c800 85ddd510
85ddd510 808b5be8 85e58e20
85e58e20 85ddd510 8646ce20
8646ce20 85e58e20 85d92c08
85d92c08 8646ce20 8616ce20
... Ommitting output to reduce length. The links were all valid..
872ba380 86e456a0 86c8da10
86c8da10 872ba380 8692f708
8692f708 86c8da10 86d630b8
86d630b8 8692f708 8673eae0
8673eae0 86d630b8 87223a98
87223a98 8673eae0 860010b8
860010b8 00000000 00000000
```

We arrived at the same address of 860010b8. So when walking the list both forward and backward we encounter pointers that take us to this address. This means that there should have been a valid flink/blink pair (which would be a valid nt!\_EPROCESS activeprocesslinks \_LIST\_ENTRY ) at this address. This memory appears to have instead been zeroed out. It's flink pointer should be 87223a98

When dumping the list out forward using dl, the line of output representing 860010b8 should have contained these values below, however it did not.  
860010b8 87223a98 861ce3b0

So what happed? In dumping the area around the address, you can see it has been zeroed

```
0: kd> dd 860010b8-200 860010b8+200
86000eb8 00000000 00000000 00000000 00000000
```

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

```
86000ec8 00000000 00000000 00000000 00000000
86000ed8 00000000 00000000 00000000 00000000
86000ee8 00000000 00000000 00000000 00000000
... Ommitting output to reduce length. Entire range contained all zeros...
86001088 00000000 00000000 00000000 00000000
86001098 00000000 00000000 00000000 00000000
860010a8 00000000 00000000 00000000 00000000
860010b8 00000000 00000000 00000000 00000000
860010c8 00000000 00000000 00000000 00000000
860010d8 00000000 00000000 00000000 00000000
860010e8 00000000 00000000 00000000 00000000
... Ommitting output to reduce length. Entire range contained all zeros...
86001288 00000000 00000000 00000000 00000000
86001298 00000000 00000000 00000000 00000000
860012a8 00000000 00000000 00000000 00000000
860012b8 00000000
```

Out of curiosity, I ran the !address command to see what range this address falls into.

```
0: kd> !address 860010b8
82b7e000 - 07c82000
Usage KernelSpaceUsageNonPagedPool
```

Next, we dump the pool information out. Note the corruption indication listed below.

```
0: kd> !pool 860010b8
Pool page 860010b8 region is Nonpaged pool
86001000 is not a valid large pool allocation, checking large session
pool...
86001000 is freed (or corrupt) pool
Bad allocation size @86001000, zero is invalid
***
*** An error (or corruption) in the pool was detected;
*** Attempting to diagnose the problem.
***
*** Use !poolval 86001000 for more details.
***
```

```
0: kd> !poolval 86001000
Pool page 86001000 region is Nonpaged pool
Validating Pool headers for pool page: 86001000
Pool page [ 86001000 ] is __inVALID.
Analyzing linked list...
[ 86001000 --> 86001310 (size = 0x310 bytes)]: Corrupt region
Scanning for single bit errors...
None found
```

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

So now that we have opened the dump, set the trap frame, dumped the stack, then found the answer. Now, we need to do the actual debug of the assembly code to both prove our theory and determine how this caused us to die. Let's revisit the activeprocesslinks field that we discussed before.

```
0: kd> dt nt!_EPROCESS activeprocesslinks +0x098 ActiveProcessLinks : _LIST_ENTRY
```

the linked list we were following is located 0x98 from the start of the nt!\_EPROCESS. The previous \_LIST\_ENTRY provided us with the address of this field by in the next process object. To get the start of the actual process object, we need to subtract 0x98 from that value. For example, if you were passed in a null address instead of a valid pointer and subtracted 0x98 you would get:

```
0: kd> ? 0x0-0x098  
Evaluate expression: -152 = ffffffff68
```

If we start with a null value and backup 0x98 we get ffffffff68

As you can see below, that very value happens to be present in ECX. ECX should instead be the address of our process object. However, since our linked list pointer was null, we instead were following an invalid address. It's all downhill from here:

```
0: kd> recx
```

What's in ECX?

Last set context:

```
ecx=fffffff68
```

it's the offset from the null pointer

Let's keep digging to verify this by looking back in the assembly and walking through what happened in the final moments of the crash. To do this, I unassembled backwards using the ub command. Examination of the output of this command (listed below but limited to only one command for blog simplicity) will show that the code took the invalid address ffffffff68 that should have been a pointer to the process object and subtracted 0x18. This gave us the invalid address ffffffff50 that we died on. The . represents the current instruction pointer and the L1 tells us to only display 1 unassembled assembly language instruction.

```
0: kd> ub . L1  
nt!ObReferenceObjectSafe  
80846dce 8d51e8          lea     edx, [ecx-18h]
```

we took the process object pointer in ecx and subtracted 0x18 from it and put it into edx to dereference

```
0: kd> redx  
edx=fffffff50
```

```
0: kd> ? ffffffff68-0x18
```

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

Evaluate expression: -176 = **ffffff50**

Next, we tried to dereference this invalid pointer. Note that @\$ip is another way (just like .) to point to the instruction pointer.

```
0: kd> u @$ip L1
nt!ObReferenceObjectSafe+0x3:
80846dd1 8b0a          mov     ecx,dword ptr [edx]
```

edx contains the invalid value

You can also use the r command to dump out all of the registers and display this failed instruction as shown below.

```
0: kd> r
Last set context:
eax=861ce300 ebx=808b5be8 ecx=ffffff68 edx=ffffff50 esi=8747c8d0 edi=00000000
eip=80846dd1 esp=f40099b4 ebp=f40099c8 iopl=0          nv up ei pl nz ac pe cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010217
nt!ObReferenceObjectSafe+0x3:
80846dd1 8b0a          mov     ecx,dword ptr [edx]  ds:0023:ffffff50=????????
```

we trapped trying to dereference the bad address that edx was directing us to

So, why would we be backing up from this address? Each process object has an Object Header that precedes it. The following command ?? uses the current Expression Evaluator (C++ is the default) to run the sizeof() function against the **nt!\_OBJECT\_HEADER** symbol information and output the size of the an object header.

```
0: kd> ?? sizeof (nt!_OBJECT_HEADER)
unsigned int 0x20
```

You can dump out this header to see where the start of the process object is with the following command

```
0: kd> dt nt!_OBJECT_HEADER
+0x000 PointerCount      : Int4B
```

the code was accessing the offset 0 of the Object Header which is the PointerCount field

```
+0x004 HandleCount      : Int4B
+0x004 NextToFree       : Ptr32 Void
+0x008 Type             : Ptr32 _OBJECT_TYPE
+0x00c NameInfoOffset   : UChar
+0x00d HandleInfoOffset : UChar
+0x00e QuotaInfoOffset  : UChar
+0x00f Flags           : UChar
+0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
+0x010 QuotaBlockCharged : Ptr32 Void
+0x014 SecurityDescriptor : Ptr32 Void
```

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

```
+0x018 Body : _QUAD
```

The process object is located here at offset 0x18. This means that backing up 0x18 would put you at offset 0x0 of the object header which is the Pointer Count field.

So let's see if we can locate any stack data to validate this. Dumping the stack again, I observe the address of the process object.

```
0: kd> kvL2
ChildEBP RetAddr Args to Child
f40099b0 809426b2 a9131e08 00000012 861ce318 nt!ObReferenceObjectSafe+0x3
(FPO: [0,0,0])
f40099c8 808544d9 861ce318 0007d960 00000000 nt!PsGetNextProcess+0x6c (FPO:
[1,1,0])
```

Let's dump out this process field. Wow! Look what we have here. The address of the zeroed list entry from above.

```
0: kd> dt nt!_EPROCESS 0x861ce318 ActiveProcessLinks
+0x098 ActiveProcessLinks : _LIST_ENTRY [ 0x860010b8 - 0x87176940 ]
```

```
0: kd> dd 0x860010b8 L2
860010b8 00000000 00000000
```

So to summarize the problem, we crashed because a memory address that should have contained valid flink/blink pointers to a process object contained the invalid value 00000000. We expected this value to be pointing to a valid nt!\_EPROCESS ActiveProcessLinks field instead of containing a null pointer (zero). After obtaining this null pointer, we then subtracted 0x098 which normally would have brought us to the start of the process object. Instead we ended up with invalid pointer 0xfffff68. Finally, we subtracted another 0x18 which should have brought us to the PointerCount field of the object header. This gave us address 0xfffff50 instead of a valid object header address. We then dereferenced this invalid address which led us to the bugcheck.

This crash was due to pool corruption, as shown in the output of the !pool command above. This blog has been a walkthrough of debugging a memory dump and identifying how pool corruption led up to a crash. Identification of the reason for the corruption is a separate matter. These types of issues are often difficult to troubleshoot due to the fact that any driver in kernel mode could have corrupted this memory region. To further complicate matters, the actual corruption of this memory may have taken place millions of CPU cycles before the time this code stumbled across the corrupted region leading up to the crash that caused this memory dump to be initiated. A memory dump is simply a snapshot of information relating to the state of a system during one particular moment in time.

In this case, 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 corruption.

# Reversing in Reverse - Linked-List Pool Corruption a Complete Walkthrough Part 1

Ryan Mangipano  
(From the NT Debugging Blog)

Usually, a debug such as this one would have started with analysis of the function at the top of the stack and worked down the stack reading the assembly to definitively identify the source of the bad pointer. However in this case, we were lucky enough to find the answer right away and then validate it with conventional debugging methods by reversing in reverse.