

Bit Flips - Was That a Zero or a One

Ryan Mangipano

(Reprinted From WindowsItPro Magazine)

It's no fun when you're paged at 2:00 A.M. because your production server unexpectedly rebooted due to a bug check—a Windows system crash that can be caused by any of a number of conditions, such as malfunctioning hardware. But that's what happened to one server administrator when his hardware malfunctioned. The admin contacted Microsoft, and we analyzed the dump file. We found evidence of his hardware problem in the form of a bit flip. A bit flip occurs when you're copying data and one of the bits changes so that it's incorrect. A value of 1 incorrectly becomes a zero, or vice versa. Bit flips that lead to bug checks are a common way that Windows detects a hardware problem (e.g., bad memory, an overheating CPU).

In this article, I'll explain what a bit flip is and demonstrate an example of how we found one such bit flip, when a bit changed to an incorrect value as the CPU attempted to copy data caused the system to crash. That way, if Microsoft support reviews your memory dump, and the support engineer explains that we found evidence of a hardware problem in the form of a bit flip, you'll have a solid understanding of what the engineer is talking about. I'll also provide some background information about access violations, registers, and the mov assembly language instruction.

Access Violation

The customer's server generated a memory.dmp file, which the customer submitted to the Microsoft Global Escalation Services team for analysis. I loaded the crash dump into the Windows debugger and began my review. (For more information about how to load a dump file into the debugger on your system, see "Administrators' Intro to Debugging.")

Once the dump file was loaded into the debugger, I ran the command

```
!analyze -v
```

which provides basic information about the type of crash that occurred. The textual output from !analyze -v explained that invalid memory was referenced. Also, the debugger displayed the instruction that the CPU was attempting to execute when the crash occurred. This type of crash usually occurs when a pointer gets set to some value that it should not have been set to. Pointers should hold the address of where data is located in memory. If pointers are set to some bad value, the system can crash while attempting to follow that value. When this type of crash occurs as a result of the system accessing a garbage address, the crash is commonly referred to as an access violation. The !analyze -v output has also listed the assembly language instruction that caused the access violation. In the output that Figure 1 shows, 80546944 was not a valid address, as indicated by the question marks shown next to the address. When the code that was running on the CPU tried to access this address, a page fault trap occurred, followed by an access violation.

Figure 1
!analyze -v output showing access violation

```
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:
```

Bit Flips - Was That a Zero or a One

Ryan Mangipano

(Reprinted From WindowsItPro Magazine)

```
Arg1: 80546944, memory referenced.  
mov     eax, dword ptr [esi]      80546944=????????
```

Introducing the mov Command

Notice the mov command (which stands for move) in the output in Figure 1. Executing the mov command on the CPU copies the source to the destination. I'm not sure why this command wasn't called copy instead of mov, since the command doesn't delete the data from the source.

The mov command needs to know what data it must copy from and where to copy the data to. This information is provided in the form of operands. In the mov instruction in Figure 1, the EAX register is the first operand, and the first operand is the destination. (I'll explain what registers are shortly.) The second operand is dword ptr [esi]. This represents the address pointed to by the ESI register. How do we know that it is the address pointed to by ESI and not the ESI register itself? Because the debugger has surrounded the ESI register in brackets. The brackets tell us that the processor was not using the register itself but was instead using the contents of the register as a pointer to the virtual address where the data is located.

The debugger has also output dword ptr, which also tells us that the ESI register will be treated as a dword-sized pointer. Using a pointer as an address to get the actual data is called dereferencing the pointer. To summarize, the debugger has helped us identify that the command that was executing on the CPU when the crash occurred was trying to copy memory from the address contained in the ESI register to the EAX register. So there are registers with names, but what is a register anyway?

Viewing CPU Registers' Contents

Registers are small memory locations that are built into the CPU. They can be accessed very quickly as opposed to the amount of time it would take to access the memory located on the memory in slots on the motherboard slots (DIMMs, for example). Each of these registers has a name that the register is referred to, such as ESI or EAX.

You can use the r Windows debugger command to dump out the registers' contents. For example, if I run the r command from the debugger prompt and pass it the name of the ESI register, the following output will appear on screen, letting us know that ESI contains the value 0x86f4c658:

```
r esi  
esi=86f4c658
```

If you scroll up to the !analyze -v output listed in Figure 1, you can see that this value contained in the ESI register is the value of the bad address. So the system crashed because ESI had a bad value loaded into it.

Examining the Previous Assembly Language Instruction

To understand why this particular system crashed, we will need to look at the assembly instruction that executed just before the bad value in ESI was accessed causing the crash. We can use the ub (unassembled backwards) command to look at the assembly language instructions that executed right before this instruction. Here's the command followed by its output:

Bit Flips - Was That a Zero or a One

Ryan Mangipano

(Reprinted From WindowsItPro Magazine)

```
0: kd> ub . L1
      mov esi, dword ptr [edi]
```

The period (.) tells the command to review the instructions backward starting from the current instruction. The L1 tells the command that we want to see only one instruction.

From the output, we can see that we followed a pointer in EDI to get the value that we're loading into ESI. Remember that the brackets around EDI indicate that we are dereferencing, otherwise known as following, a pointer contained in the EDI register. So now we know how ESI got the bad value. We copied the value from the memory location that EDI is pointing to. Let's use the dd (display memory as dwords) command to examine the data that EDI is referencing. Here's the command and its output:

```
1: kd>dd @edi L1
      80566944
```

The @ tells the command that EDI is the name of a register. The L1 modifier tells the command that we want to see only one dword.

So if the hardware had correctly performed the assembly language commands that the software instructed it to do, the value 80566944 would have been copied to the ESI register. Instead, as we showed earlier, the value present in the ESI register was the bad value, 80546944.

Breaking Down the Bits to Find the Bit Flip

Notice that the address is very close to the invalid address—80566944—that the instruction pointer was referencing. Let's use the debugger's built-in .formats command to convert these two values to binary format:

```
1: kd> .formats 80546944
      Binary: 10000000 01010100 01101001 01000100
1: kd> .formats 80566944
      Binary: 10000000 01010110 01101001 01000100
```

You can see that these two addresses differ by only one bit. As previously discussed, this type of error is known as a bit flip. It's caused by a hardware problem that causes one of the bits to be set to an incorrect digit.

A Bit of Understanding

As you've seen, understanding bit flips can help you better understand the underlying cause of a bug check that results in a system crash. By identifying when a bit flip has occurred, you'll have more detailed information to provide to Microsoft support, and you'll be able to hone in on the nature of a system crash (e.g., a hardware problem) more quickly. And, as a side educational benefit, you'll also gain some insight into the workings of Windows registers!