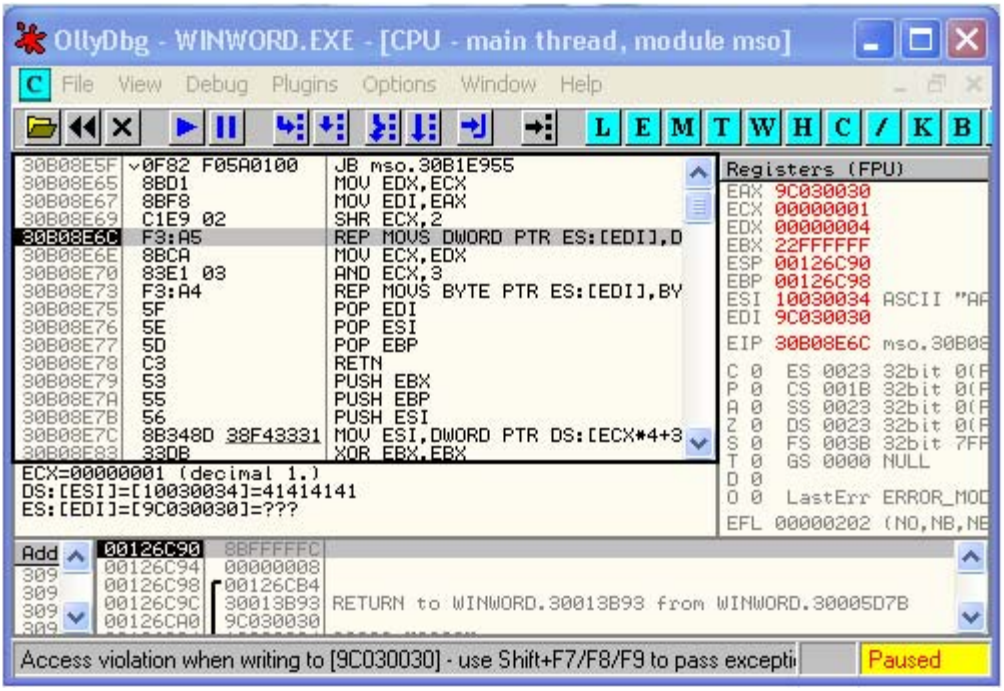
Analysis of Microsoft Word Heap Overflow Vulnerability, CVE-2006-5994

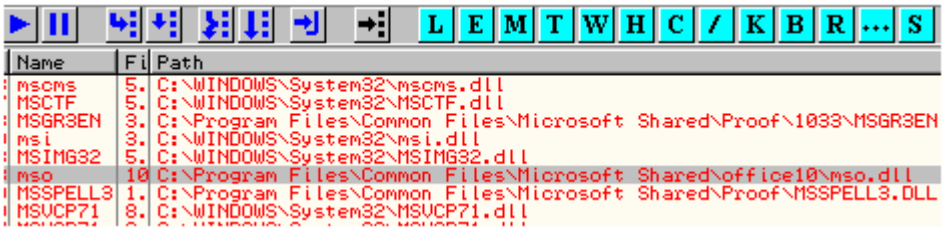
These are roughly outlined notes from an analysis of the public POC for this vulnerability.

First, make sure OllyDbg is set to JIT debugger. Then open the POC document.

Resources:
http://www.milw0rm.com/exploits/2922
http://www.microsoft.com/technet/security/advisory/929433.mspx



We can see an access violation occurs during a movsd ([esi]->[edi] * ecx) instruction. This happens because edi points to an address in kernel space (>0x80000000) which is not writable by the program. Also we know that the movsd instruction is within the mso module loaded by winword.exe (see title bar). So, we can search for mso module, or just click the cyan colored ‘E’ to access them and then sort alphabetically to find the full path:

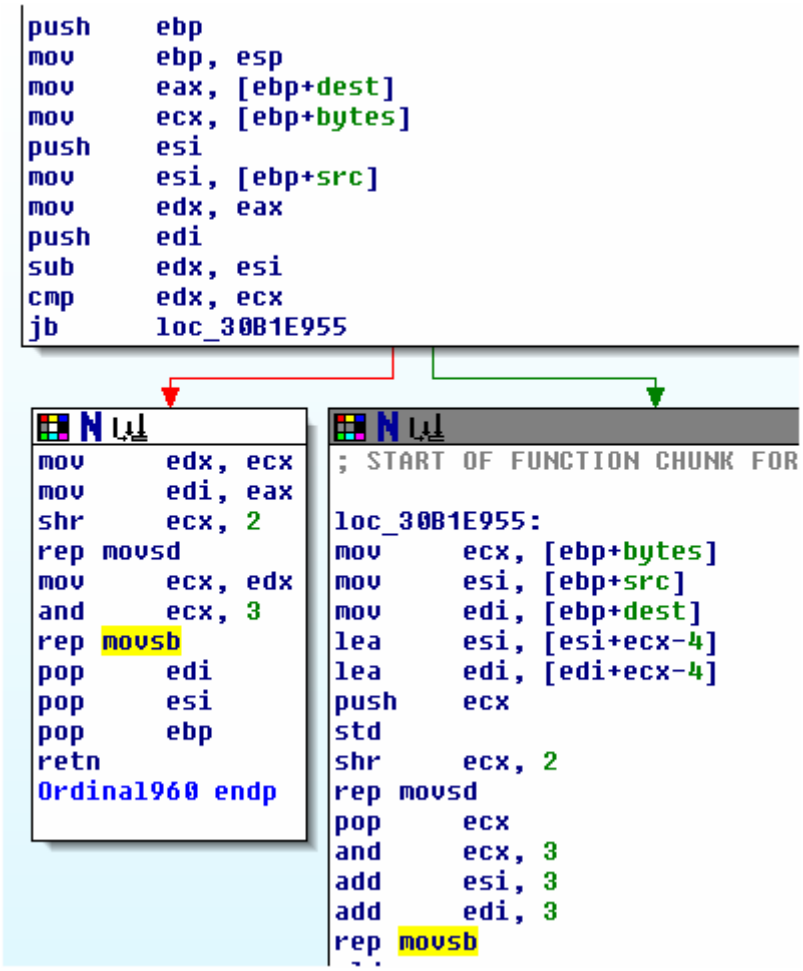


Open mso.dll in IDA and go to the address of the movsd instruction (0x30B08E6C). It is within exported function ordinal 960. This is a really small function that takes three parameters. We can tell that the first and second (arg_0 and arg_4) are both pointers because they are the values that get shoved into edi and esi, respectively. In other words, arg_0 is a pointer to the destination buffer and arg_4 is a pointer to the source buffer.

Also, the third parameter, or arg_8, is moved into ecx, which tells the rep movsd instruction how many times to repeat. This value is then divided by 4 using a shr,2 instruction. This is because 4 bytes will be copied for each iteration of the movsd loop, and we want to movsd once for every 4 byte chunk there is (but then copy the remainder too if bytes % 4 > 0).

The remainder stuff is shown below after the first movsd. There is an AND ecx,3 instruction followed by a rep movsb (move single byte instead of four byte). This may be misleading if trying to reverse too literally, because the original author did not do all this himself. It is basically an in-line call (e.g. call and return to function compiled out, and just replaced with the function's whole body of code) to memcpy(). All memcpy() actually does is use a movsd instruction followed by AND ecx,3 followed by movsb.

Here is the function in IDA:

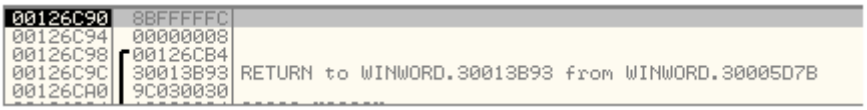


Here is the corresponding (roughly) reversed C source code:

```
void MSO_960(void *dest, void *src, unsigned int bytes)
{
	if ((dest-src) > bytes)
	{
		memcpy(dest, src, bytes);
	}
	else {
		/* adjust a little and then memcpy() */
	}
	return;
}
```

Now, based on this information, we know that the *dest, which ends up pointing to kernel space, is the bad guy (or girl, depending on your gender and discrimination beliefs). Input from the user, in the form of a hard-coded value in a Word document, must have some influence over this value. To find out for sure, we have to trace the call to this MSO_ 960 function. The cross-references pane in IDA shows about 100 different functions within mso.dll that call the function. No need to go analyze each one, because we have OllyDbg still running and it shows important information.

In particular, check the stack frame. This shows the return address of the function to which MSO_ 960 will return when it is done executing. Therefore, this return address is where we must go to find out more of why/how the *dest value ends up being invalid. As shown, the return address is 0x30013B93 within winword.exe itself. And so we load in IDA…

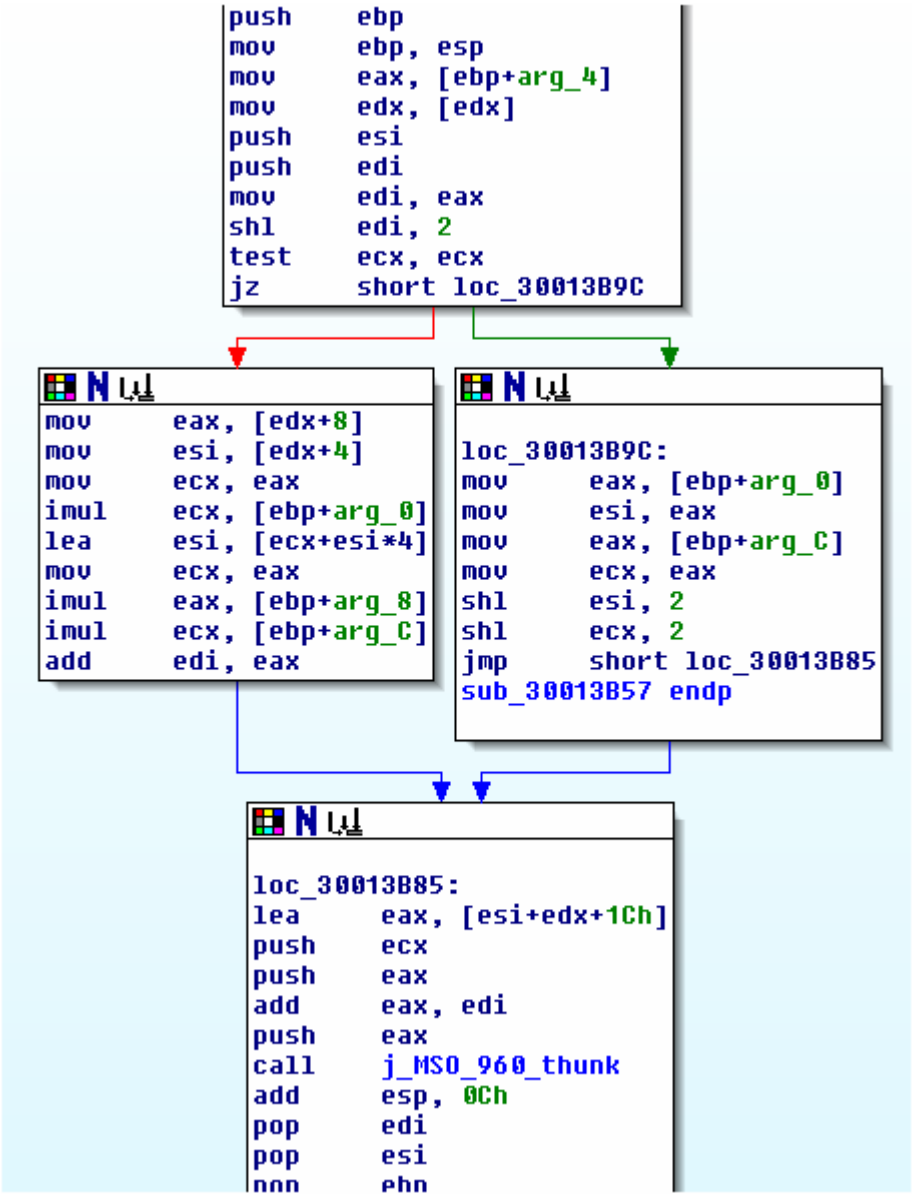


This function is nearly as small as MSO_960 and we can see the parameters it pushes on the stack before calling MSO_960 (referred to as j_MSO_960_thunk in the screen shot). Since we are most interested in the first parameter to MSO_960 because that becomes *dest, we look at the last parameter pushed on the stack before calling the MSO_960.

Basically, this value is computed by getting a pointer to the *src buffer. Then, it takes the number of 4-byte chunks to copy and multiplies that by 4 to generate the total number of bytes. This total number is added to the base of *src buffer in order to generate the address of the *dest buffer. In other words, assume *src points to 0x05000000 and there are two 4-byte chunks to copy. This would mean that *dest would end up as 0x05000008. However, the vulnerable code does not check the number of 4-byte chunks as specified in the specially-crafted Word document.

This means that an attacker can produce a specially-crafted Word document that specifies a really huge number of 4-byte chunks to copy (even if there is really only one 4-byte chunk to copy). The *dest buffer in this case will end up pointing to a memory address very far away from the *src buffer, even into kernel space…which will cause an access

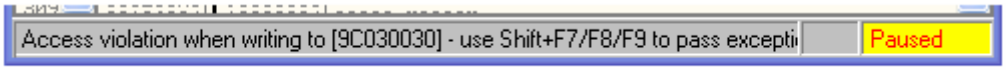
violation. However, the attacker does not need to aim as far as the POC shows. The attacker can carefully choose to write an arbitrary number of bytes to a memory region between the *src buffer and kernel memory space --- and code execution will result instead of DoS like in the POC. Here is the function:



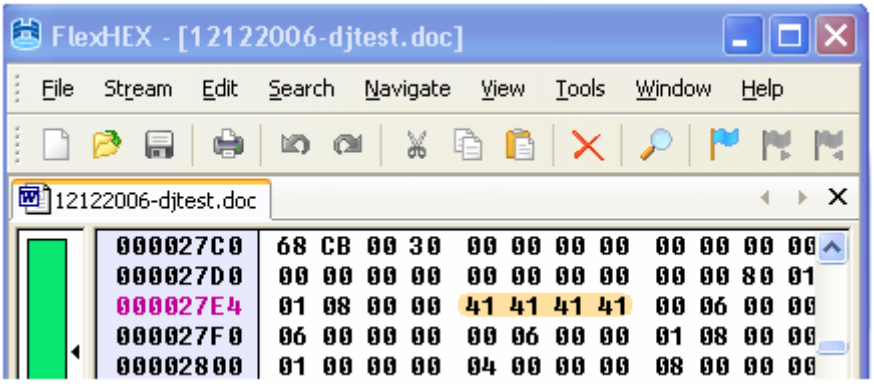
So trace [ebp+arg_4] from the start. It is first used when it is moved into eax, which in turn is then moved into edi and multiplied by 4. Then after computing *src (see instruction right under loc_30013B85), edi is added to *src to produce the first argument to MSO_960 which is *dest.

The *src buffer in our test machine is 0x10030034 (ymmv on that). As stated above, when the number of items (0x22FFFFFF during POC) is multiplied by 4 to produce the total amount of bytes to copy, it becomes 0x8BFFFFFC. Subsequently, when this total number of bytes is added to the base of *src buffer, it equals 0x9C030030 (0x10030034 + 0x8BFFFFFC = 0x9C030030). This pointer is familiar, because it is the same address at which our original on-write access violation occurred (just snipped from the first screen shot):

Note: originally I thought 0x22FFFFFF was hard-coded in the POC, because there actually is a sequence of bytes like 0x22FFFFFF, but this is just a coincidence. The 0x22FFFFFF value is formed when 1 is subtracted from 0x23000000 about 6 functions earlier in the execution path.



So, as explained in the proof of concept, data for the *src buffer comes from offset 0x27e4 of the specially-crafted Word document (see below).



The attacker who wishes to turn this POC DoS into code execution still has a bit of work to do, including calculating the reliable distance from *src to a desirable heap region to overwrite, and properly staging that data starting at 0x27e4 in the Word document.