



# Code Analysis

# Code Analysis

- Code Analysis involves analyzing the code
- Allows you to determine functionality that cannot be determined using dynamic analysis
- Code Analysis Tools: Disassemblers and Debuggers

# Disassembler and Debuggers

**Disassembler:** Allow you to examine the code statically

**Debugger:** Allows you to examine the aspects of the program while it is running

- Allows you to pause execution by setting breakpoints
- Single step through the code
- Step over
- Continue

**IDA Pro, x64dbg, Ollydbg, Immunity Debugger, Windbg** are popular disassemblers and debuggers



# Code Analysis Tools

# IDA Pro

- Disassembles the compiled executable into assembly instructions
- Powerful disassembler and debugger
- Commercial Tool
- Free Versions and demo version available

[https://www.hex-rays.com/products/ida/support/download\\_freeware.shtml](https://www.hex-rays.com/products/ida/support/download_freeware.shtml)

<http://out7.hex-rays.com/demo/request>

# x64dbg

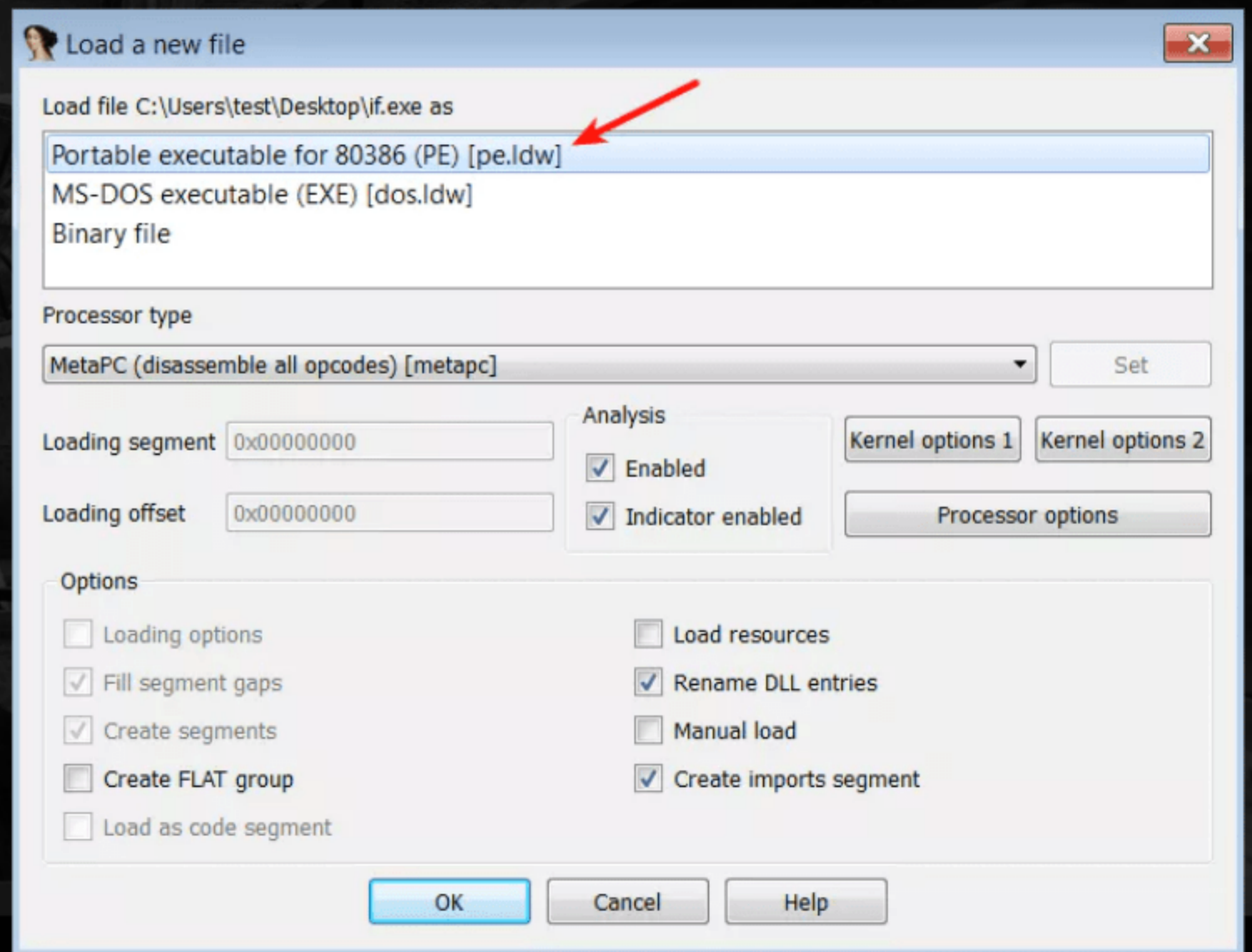
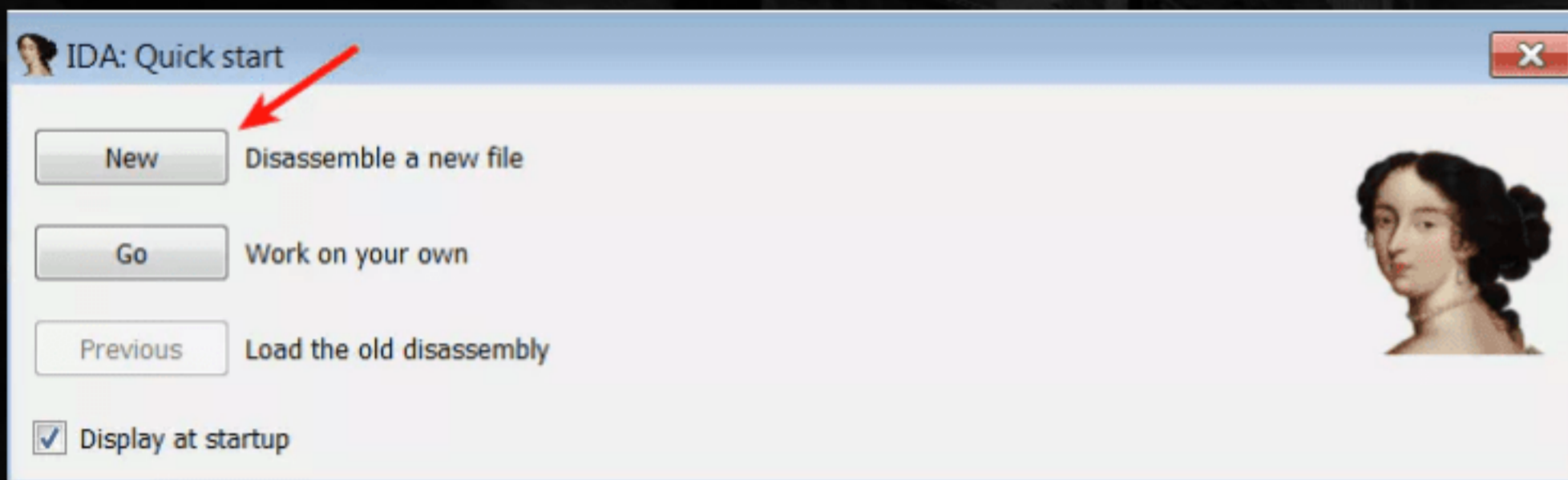
- Open source debugger
- Can debug 32-bit and 64-bit program
- Offers various debugging features

<https://x64dbg.com>

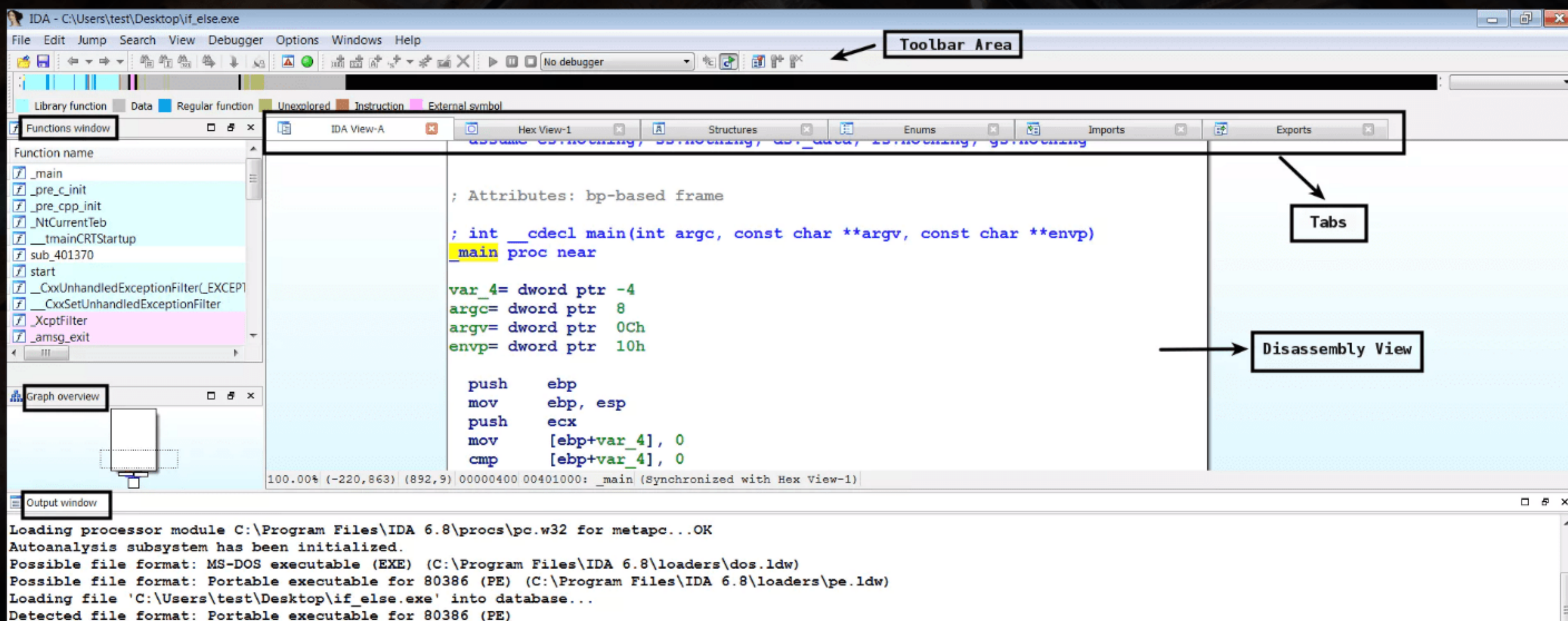


# Disassembly using IDA

To load an executable, launch **IDA pro (right-click / Run as administrator)**. Choose **New** and select the file you want to analyze. The file will be loaded into the memory and IDA determines the best possible loader to use during the disassembly process



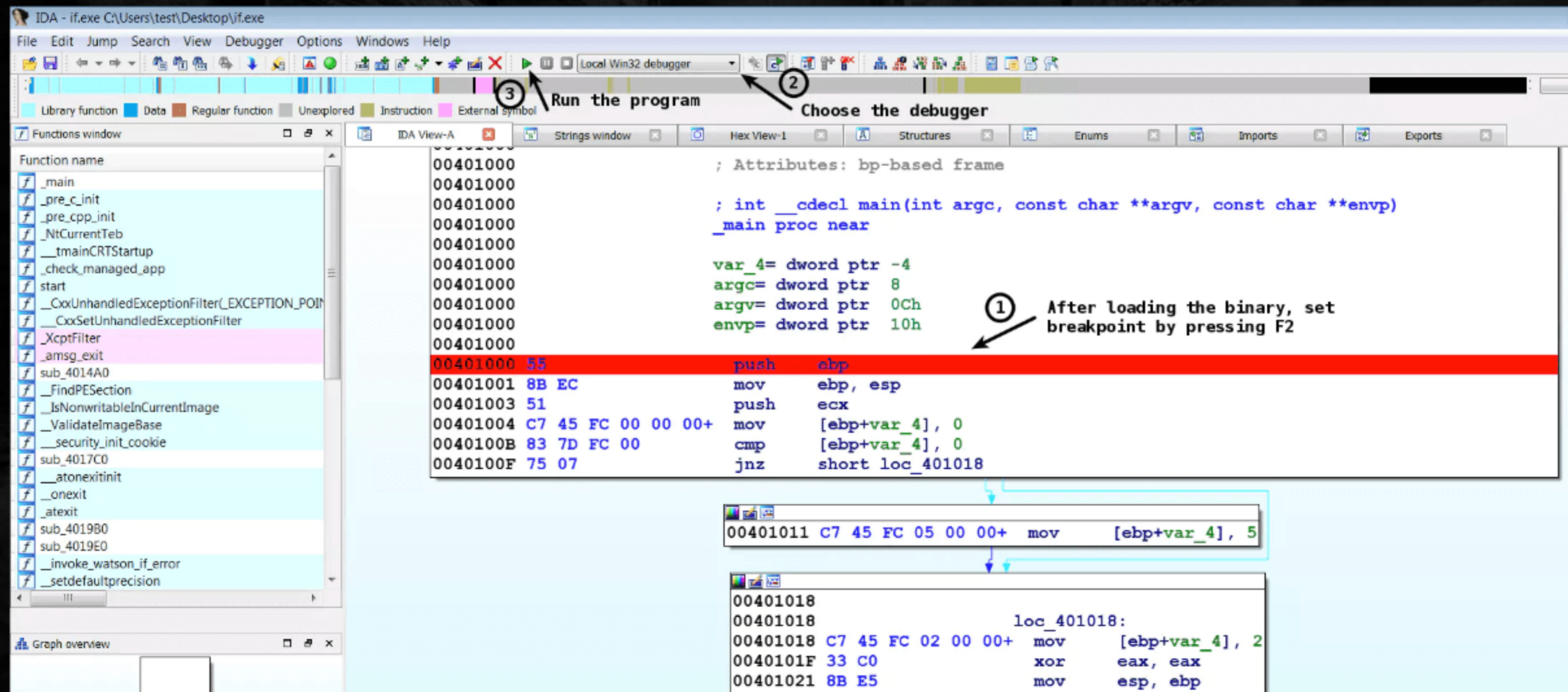
The following screenshot shows the IDA desktop after loading an executable file. The IDA desktop contains multiple tabs, clicking on each tab brings up different windows. Each window contains different information extracted from the binary





# Debugging using IDA

To debug a binary, first load the binary into IDA, set a breakpoint at the first instruction by pressing **F2** and then choose the appropriate debugger and Run the program (**F9**)



After you launch the program in IDA debugger, the process will be paused and the following debugger display will be presented to you.

The screenshot displays the IDA debugger interface for a program named 'if.exe'. The main window shows the disassembly of the program, with the following instructions highlighted:

```
00011000 push ebp
00011001 mov ebp, esp
00011003 push ecx
00011004 mov [ebp+var_4], 0
0001100B cmp [ebp+var_4], 0
0001100F jnz short loc_11018
```

The Hex View window shows the corresponding hex data for these instructions:

```
00011000 55 8B EC 51 C7 45 FC 00 00 00 00 83 7D FC 00 75 Ui8Q|En...â)n.u
00011010 07 C7 45 FC 05 00 00 00 C7 45 FC 02 00 00 00 33 .!En...!En...3
00011020 C0 8B E5 5D C3 CC CC CC CC CC CC CC CC CC CC CC CC +!s|+!!!!!!
00011030 55 8B EC E8 38 03 00 00 A3 20 30 01 00 6A 01 FF Ui8F8...ú'0..j..
00011040 15 84 20 01 00 83 C4 04 6A FF FF 15 18 20 01 00 .ä...â-.j.....
```

The Registers window shows the state of the general registers:

Register	Value
ECX	00356608
EDX	00000001
ESI	00000000
EDI	00000000
EBP	0028F8C4
ESP	0028F878
EIP	00011000
EFL	00000206

The Modules window shows the loaded modules:

Path	Base Address
C:\binary\if.exe	000
C:\Windows\system32\msvcr120.dll	61C
C:\Windows\system32\KernelBase.dll	75F

The Threads window shows the current thread:

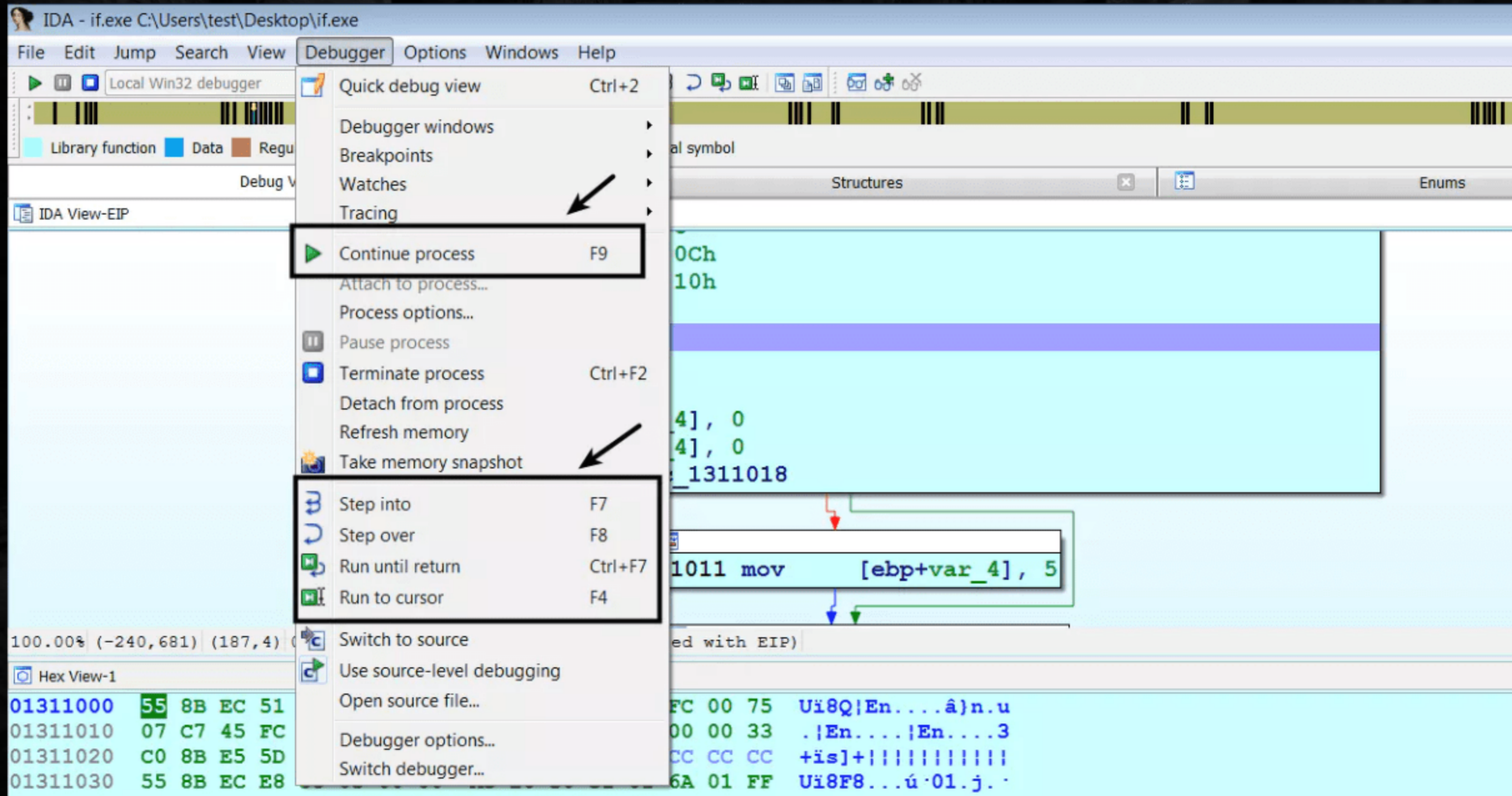
Decimal	Hex	State
1876	754	Ready

The Stack View window shows the current stack frame:

Address	Value
0028F878	000112C9
0028F87C	00000001
0028F880	00356608
0028F884	00355B38
0028F888	5A49881A

Other windows visible include the Toolbar, Debug View, Structures, and Enums.

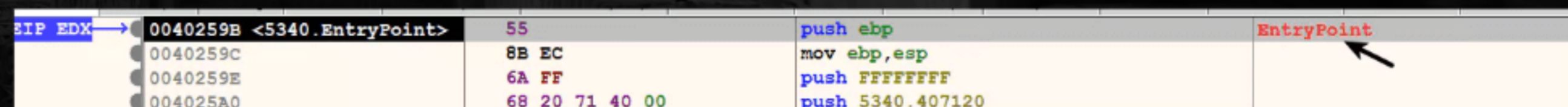
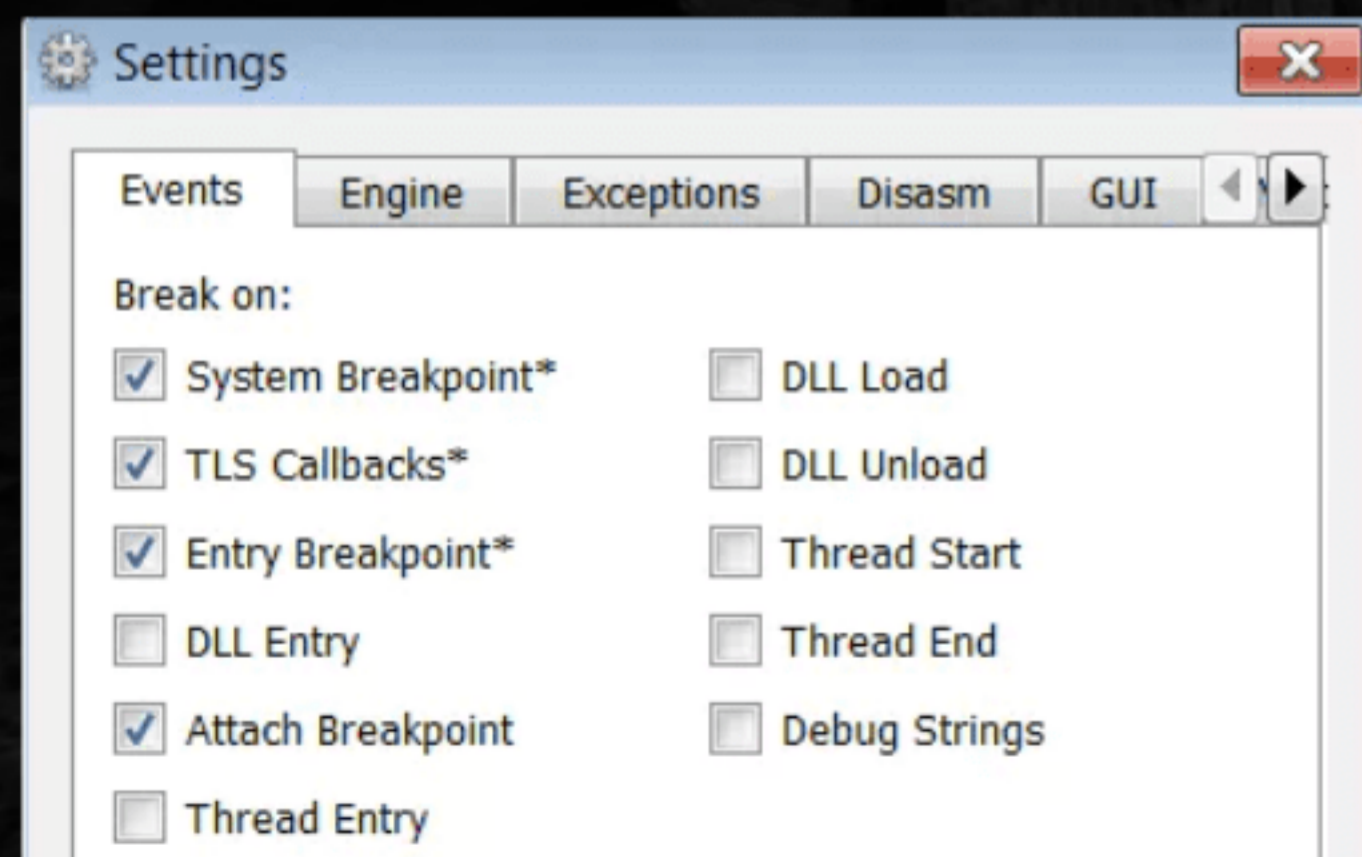
You can access various debugging options from the Debugger menu as shown below





# Debugging using x64dbg

To load an executable select **File / Open** and then browse the file you wish to debug, this will start the process and the debugger will pause at the **System Breakpoint, TLS callback** or the **program entry point** function depending on the configuration settings. You can access the settings dialog by choosing **Options / Preferences / Events** as shown below. If you want the execution to pause at the program's entry point directly then uncheck the **System Breakpoint** and **TLS Callbacks\*** option.



# x64dbg - Below screen shot shows the x64dbg interface

The screenshot displays the x64dbg debugger interface for a file named 'if.exe'. The main window shows the disassembly of the current instruction at address 00EA1000, which is 'push ebp'. The instruction list includes several other instructions such as 'mov ebp, esp', 'push ecx', and conditional jumps. The registers window on the right shows the current state of the CPU registers, with EAX containing the string '"ALLUSERSPROFILE=C:\\ProgramData"'. The stack window shows the current stack frame, with the return address 00EA12C9. The dump window at the bottom shows a memory dump of the string 'S.Y.S.T.E.M. ....'.

**Disassembly Window**

EIP	Hex	Disassembly
00EA1000	55	push ebp
00EA1001	8B EC	mov ebp, esp
00EA1003	51	push ecx
00EA1004	C7 45 FC 00 00	mov dword ptr ss:[ebp-4], 0
00EA100B	83 7D FC 00	cmp dword ptr ss:[ebp-4], 0
00EA100F	75 07	jne if.EA1018
00EA1011	C7 45 FC 05 00	mov dword ptr ss:[ebp-4], 5
00EA1018	C7 45 FC 02 00	mov dword ptr ss:[ebp-4], 2
00EA101F	33 C0	xor eax, eax

**Registers Window**

Register	Value	Comment
EAX	00315B90	&"ALLUSERSPROFILE=C:\\ProgramData"
EBX	7FFDE000	
ECX	00316E90	
EDX	00000001	
EBP	0019F998	
ESP	0019F94C	
ESI	00000000	
EDI	00000000	

**Stack Window**

Address	Hex	Comment
0019F94C	00EA12C9	return to if.00EA12C9 from if.00EA1001
0019F950	00000001	
0019F954	00316E90	
0019F958	00315B90	&"ALLUSERSPROFILE=C:\\ProgramData"
0019F95C	DA5BAF95	
0019F960	00000000	
0019F964	00000000	

**Dump Window**

Address	Hex	ASCII
77CF1000	53 00 59 00 53 00 54 00 45 00 4D 00 00 00 90 90	S.Y.S.T.E.M. ....
77CF1010	72 00 63 00 00 00 8B 46 0C 3B C7 0F 85 46 BC 09	r.c....F.;Ç..F..
77CF1020	00 64 A1 18 00 00 00 8B 40 30 56 57 FF 70 18 E8	.dj.....@0vwÿp.è
77CF1030	1E 18 05 00 33 C0 E9 AE 9B 06 00 33 C0 E9 8D 9B	....3Àé@...3Àé..



# Understanding API Calls

# Focus on Function Calls, mainly API calls

- **Call** is the assembly instruction that calls a function
- Functions perform some actions, parameters to functions are pushed on to the stack.
- Function returns value in **EAX/RAX** register
- Examining the functions allow you to determine the functionality
- API Calls are external functions that allow malware to interact with its environment. Search MSDN for documentation on API calls

<https://msdn.microsoft.com/en-us/default.aspx>

# Below example shows the MSDN documentation on **CreateFile** API call

https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx

## CreateFile function

Creates or opens a file or I/O device. The most commonly used I/O devices are as follows: file, file stream, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, and pipe. The function returns a handle that can be used to access the file or device for various types of I/O depending on the file or device and the flags and attributes specified.

To perform this operation as a transacted operation, which results in a handle that can be used for transacted I/O, use the [CreateFileTransacted](#) function.

### Syntax

```
C++  
HANDLE WINAPI CreateFile(  
    _In_      LPCTSTR          lpFileName,  
    _In_      DWORD            dwDesiredAccess,  
    _In_      DWORD            dwShareMode,  
    _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    _In_      DWORD            dwCreationDisposition,  
    _In_      DWORD            dwFlagsAndAttributes,  
    _In_opt_  HANDLE           hTemplateFile  
);
```

### Parameters

*lpFileName* [in]  
The name of the file or device to be created or opened. You may use either forward slashes (/) or backslashes (\) in this name.

In the ANSI version of this function, the name is limited to **MAX\_PATH** characters. To extend this limit to 32,767 wide characters, call the Unicode

# Below example shows the MSDN documentation for *InternetConnect* API function

## InternetConnect function

Opens an File Transfer Protocol (FTP) or HTTP session for a given site.

### Syntax

C++

```
HINTERNET InternetConnect(  
    _In_ HINTERNET    hInternet,  
    _In_ LPCTSTR     lpszServerName,  
    _In_ INTERNET_PORT nServerPort,  
    _In_ LPCTSTR     lpszUsername,  
    _In_ LPCTSTR     lpszPassword,  
    _In_ DWORD       dwService,  
    _In_ DWORD       dwFlags,  
    _In_ DWORD_PTR   dwContext  
);
```



### Parameters

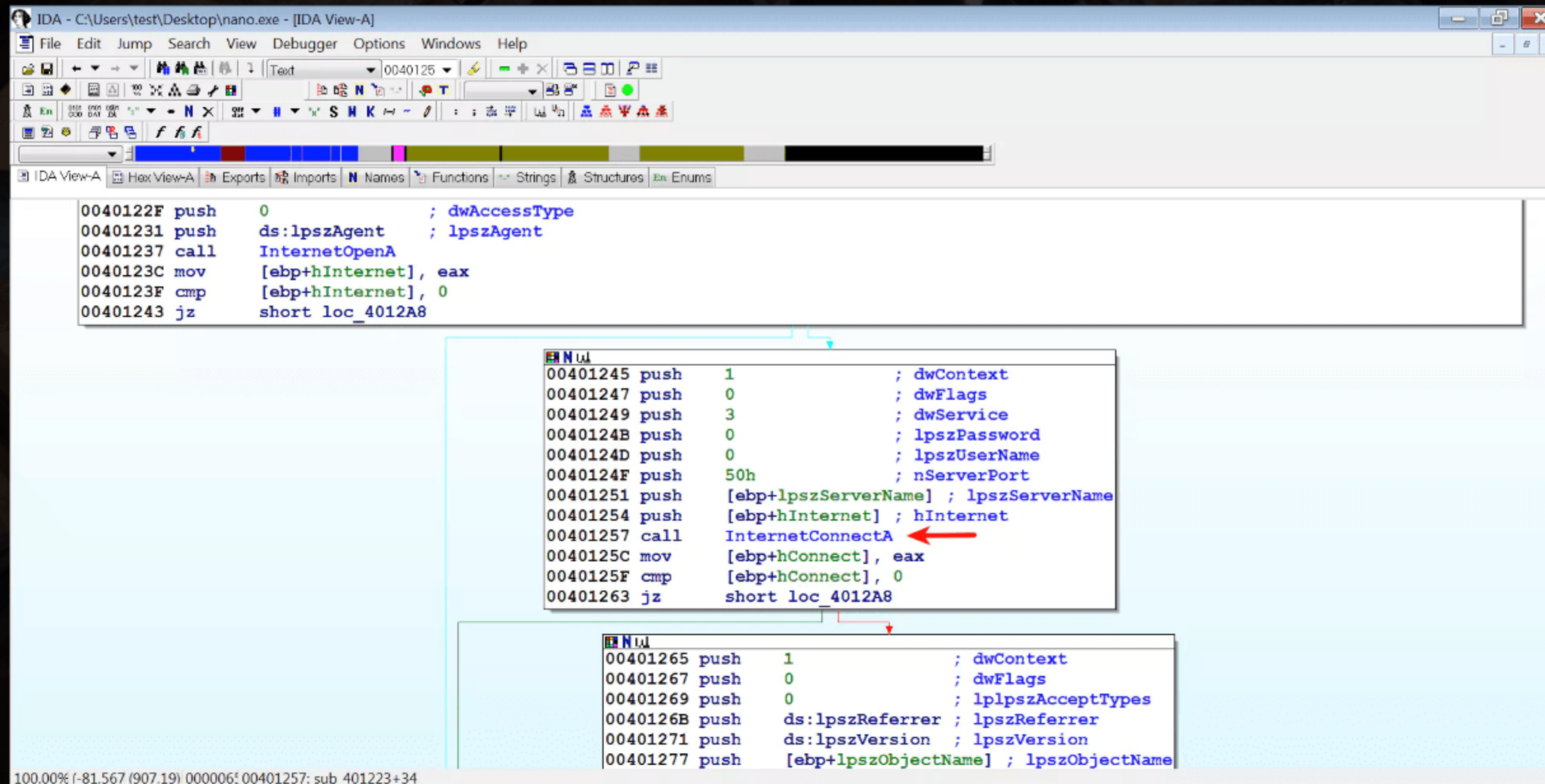
*hInternet* [in]

Handle returned by a previous call to [InternetOpen](#).

*lpszServerName* [in]

Pointer to a **null**-terminated string that specifies the host name of an Internet server. Alternately, the string can contain the IP number of the site, in ASCII dotted-decimal format (for example, 11.0.1.45).

Below screenshot shows the disassembled code of a malware sample using IDA, the malware uses **InternetConnectA** API call to open an HTTP session with C2 server which is the 2nd parameter to the API call but the server name is not clear just by looking at the disassembled code



```
IDA - C:\Users\test\Desktop\nano.exe - [IDA View-A]
File Edit Jump Search View Debugger Options Windows Help
0040122F push 0 ; dwAccessType
00401231 push ds:lpszAgent ; lpszAgent
00401237 call InternetOpenA
0040123C mov [ebp+hInternet], eax
0040123F cmp [ebp+hInternet], 0
00401243 jz short loc_4012A8

loc_401245:
00401245 push 1 ; dwContext
00401247 push 0 ; dwFlags
00401249 push 3 ; dwService
0040124B push 0 ; lpszPassword
0040124D push 0 ; lpszUserName
0040124F push 50h ; nServerPort
00401251 push [ebp+lpszServerName] ; lpszServerName
00401254 push [ebp+hInternet] ; hInternet
00401257 call InternetConnectA
0040125C mov [ebp+hConnect], eax
0040125F cmp [ebp+hConnect], 0
00401263 jz short loc_4012A8

loc_401265:
00401265 push 1 ; dwContext
00401267 push 0 ; dwFlags
00401269 push 0 ; lpVtblAcceptTypes
0040126B push ds:lpszReferrer ; lpszReferrer
00401271 push ds:lpszVersion ; lpszVersion
00401277 push [ebp+lpszObjectName] ; lpszObjectName
```

100.00% (-81.567 (907.19) 000006! 00401257: sub 401223+34

# Debugging the malware sample and inspecting the *2nd* parameter on the stack displays the name of the C2 server that malware will connect to

The screenshot shows the IDA Pro interface for debugging nano.exe. The assembly view displays the following instructions:

```
00401247 push 0 ; dwFlags
00401249 push 3 ; dwService
0040124B push 0 ; lpszPassword
0040124D push 0 ; lpszUserName
0040124F push 50h ; nServerPort
00401251 push [ebp+lpszServerName] ; lpszServerName
00401254 push [ebp+hInternet] ; hInternet
00401257 call InternetConnectA
0040125C mov [ebp+hConnect], eax
0040125F cmp [ebp+hConnect], 0
00401263 jz short loc_4012A8
```

The instruction at 00401257 is highlighted with a red arrow. Below it, a call window shows the arguments for InternetConnectA:

```
00401265 push 1 ; dwContext
00401267 push 0 ; dwFlags
00401269 push 0 ; lplpszAcceptTypes
```

The stack view at the bottom right shows the following data:

```
0012FF04 00CC0004
0012FF08 0040341B .data:aEj3tkl_hop_ru
0012FF0C 00000050
0012FF10 00000000
0012FF14 00000000
0012FF18 00000003
```

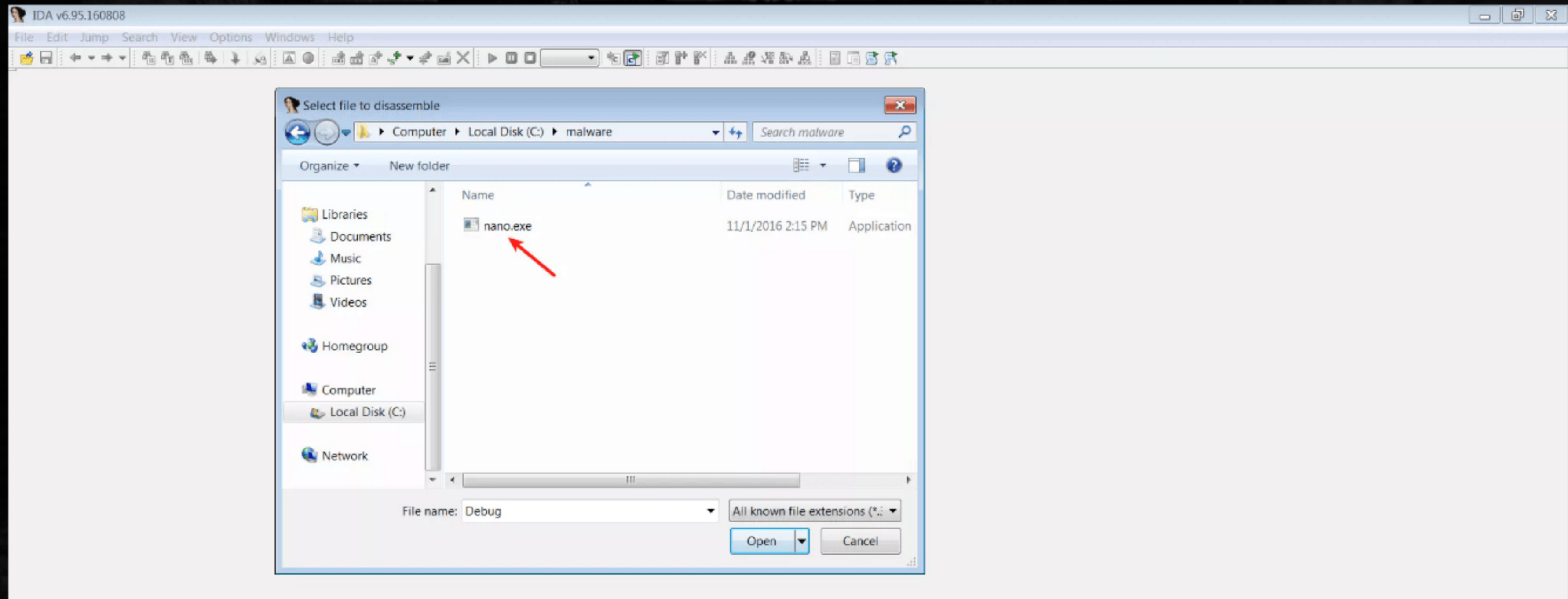
The address 0040341B is highlighted with a red box, and a red arrow points to it from the assembly view. The hex view at the bottom left shows the corresponding hex data:

```
0040341B 65 6A 33 74 6B 6C 2E 68 6F 70 2E 72 75 00 2F 76 ej3tkl.hop.ru /v
0040342B 74 2E 70 68 70 00 77 77 77 2E 67 6F 6F 67 6C 65 t.php.www.google
0040343B 2E 63 6F 6D 00 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 .com.Mozilla/5.0
0040344B 20 28 57 69 6E 64 6F 77 73 20 4E 54 20 36 2E 31 .(Windows·NT·6.1
0040345B 3B 20 57 4F 57 36 34 29 20 41 70 70 6C 65 57 65 ;·WOW64)·AppleWe
0040346B 62 4B 69 74 2F 35 33 37 2E 31 37 20 28 4B 48 54 bKit/537.17·(KHT
```

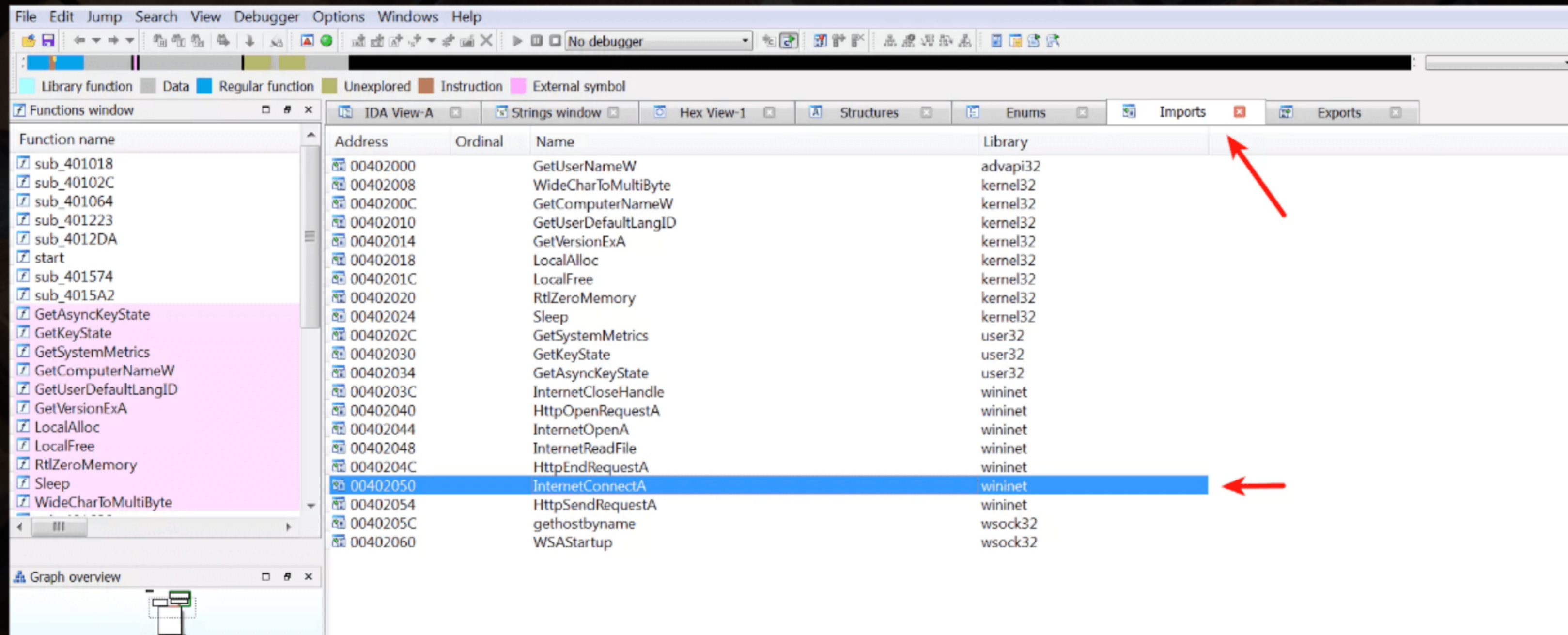


# Inspecting API Call references in IDA

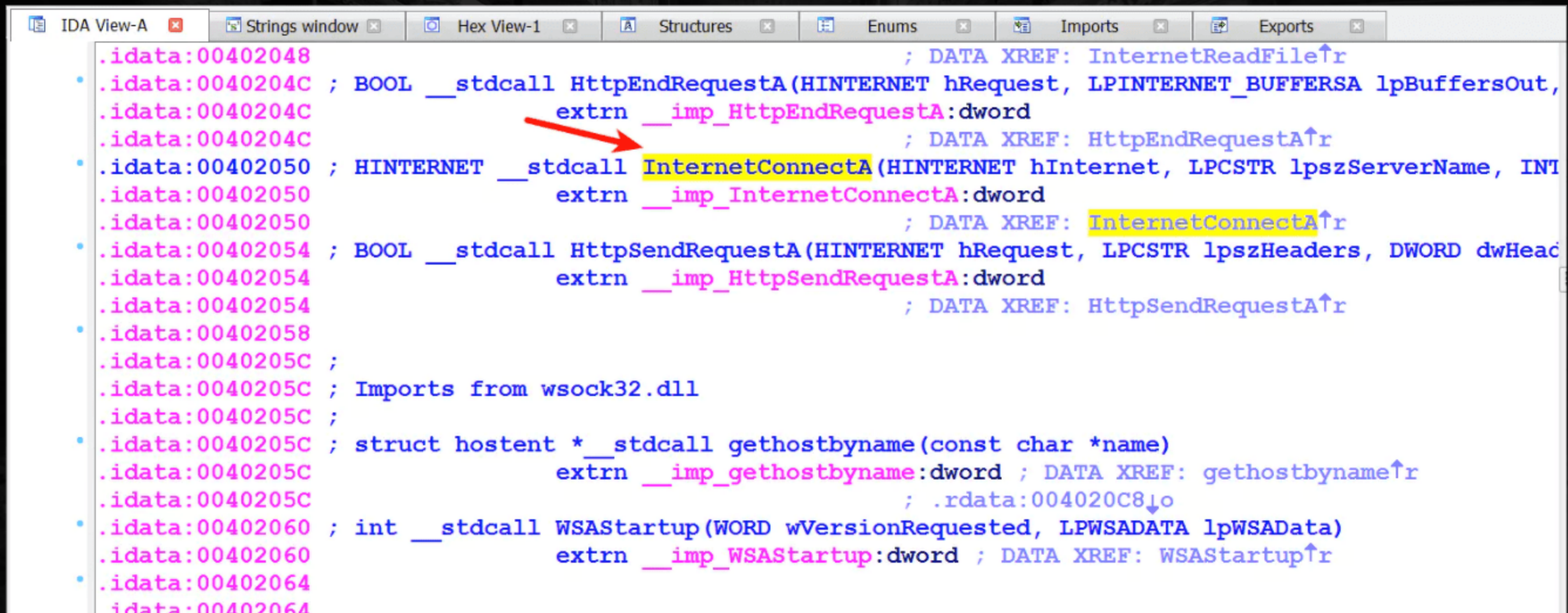
# Load the executable in IDA and click on '**Open**'



Click on the *imports* tab, this will display all the imported functions as shown below. In this example, Let's try to find where *InternetConnectA()* api call is used in the code

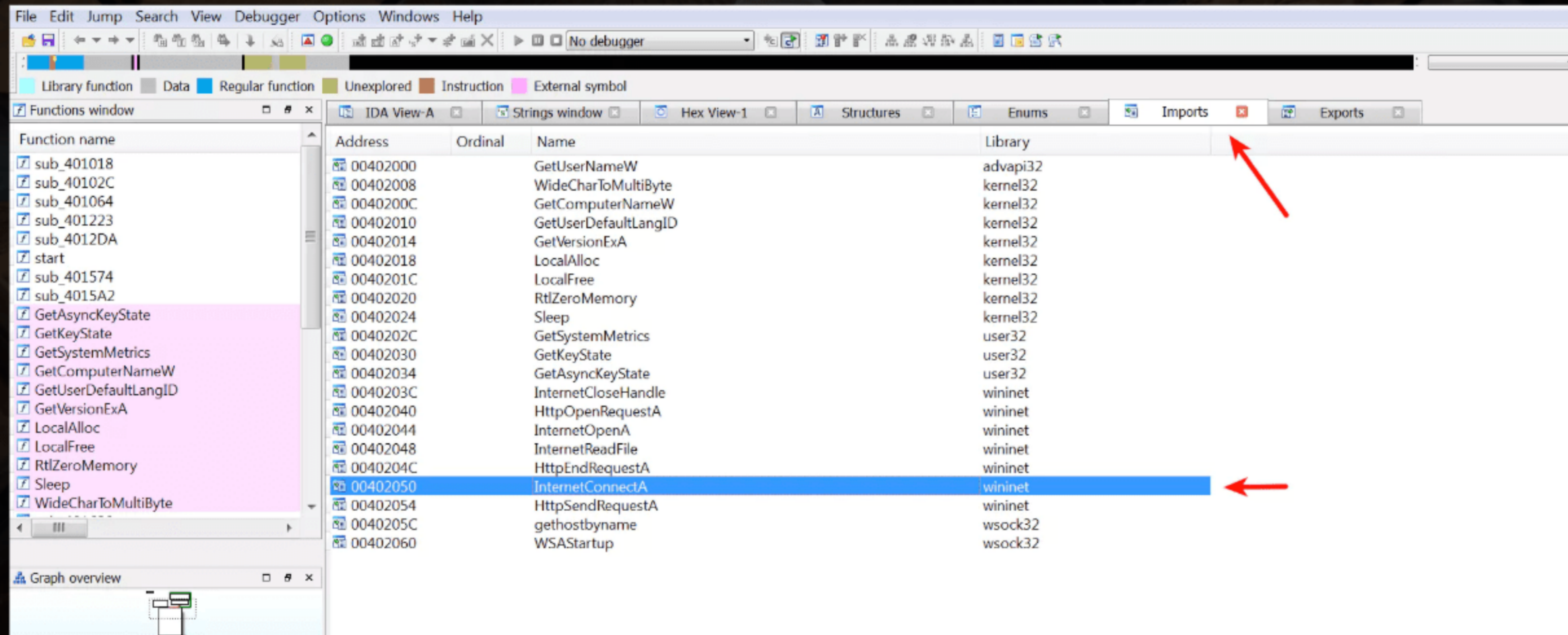


Double clicking on the desired API call will bring up the window shown below

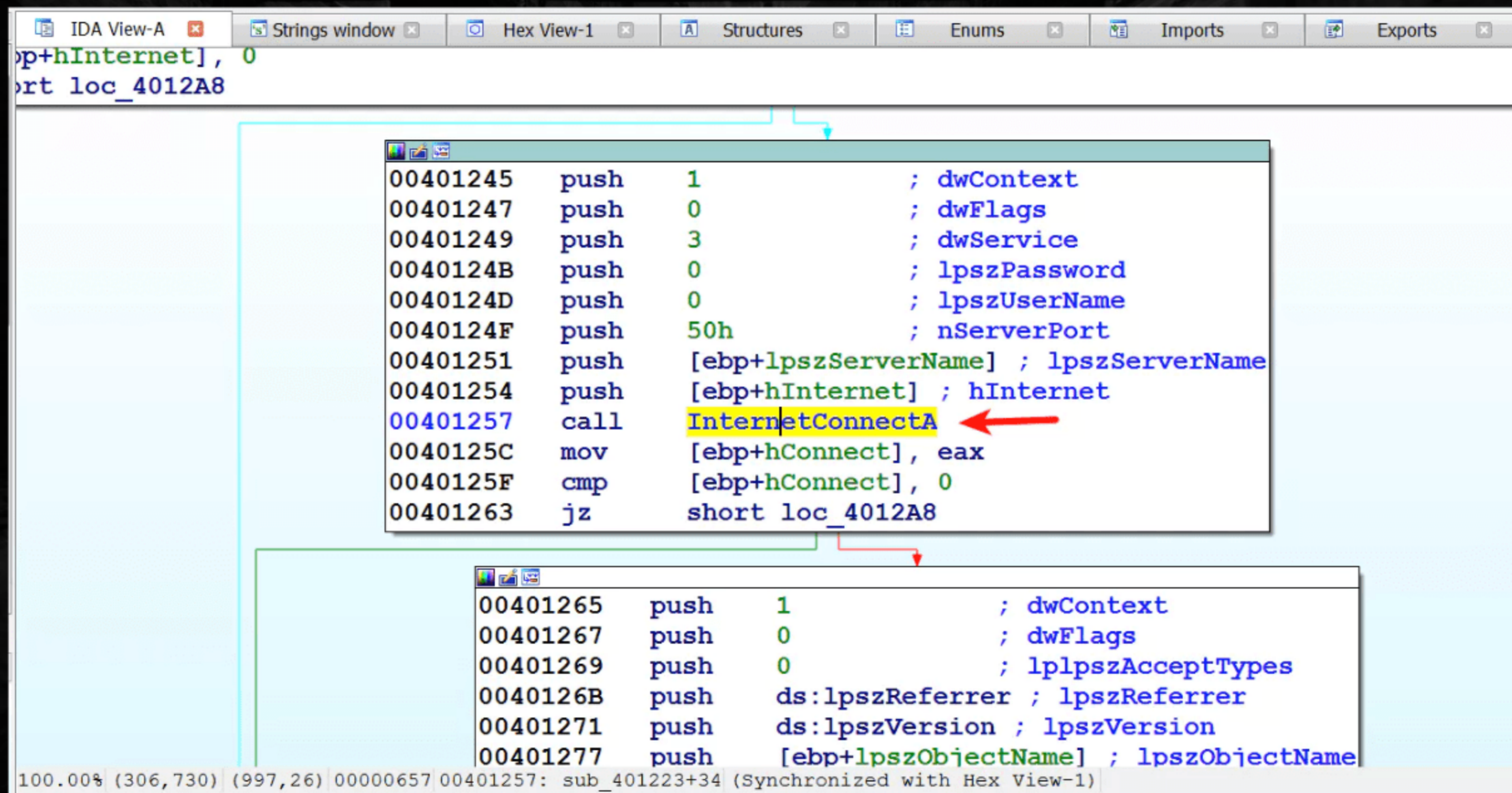


```
IDA View-A | Strings window | Hex View-1 | Structures | Enums | Imports | Exports
; DATA XREF: InternetReadFile↑r
• .idata:00402048 ; BOOL __stdcall HttpEndRequestA(HINTERNET hRequest, LPINTERNET_BUFFERSA lpBuffersOut,
; DATA XREF: HttpEndRequestA↑r
; DATA XREF: HttpEndRequestA↑r
; DATA XREF: HttpEndRequestA↑r
• .idata:00402050 ; HINTERNET __stdcall InternetConnectA(HINTERNET hInternet, LPCSTR lpszServerName, INT
; DATA XREF: InternetConnectA↑r
; DATA XREF: InternetConnectA↑r
• .idata:00402054 ; BOOL __stdcall HttpSendRequestA(HINTERNET hRequest, LPCSTR lpszHeaders, DWORD dwHead
; DATA XREF: HttpSendRequestA↑r
; DATA XREF: HttpSendRequestA↑r
; DATA XREF: HttpSendRequestA↑r
• .idata:00402058
;
; Imports from wsock32.dll
;
• .idata:0040205C ; struct hostent *__stdcall gethostbyname(const char *name)
; DATA XREF: gethostbyname↑r
; .rdata:004020C8↓o
• .idata:00402060 ; int __stdcall WSASStartup(WORD wVersionRequested, LPWSADATA lpWSADATA)
; DATA XREF: WSASStartup↑r
• .idata:00402064
; .idata:00402064
```

Highlight the API call and press 'x' key that will display all the cross-references to the API. In this case, highlighting ***InternetConnectA()*** api call and pressing x key shows reference to the code where the api call is used as shown below



Highlight the entry that you want to examine and click *ok*. The below screenshot shows the code where *InternetConnectA()* api is used now you can examine the code.



The screenshot displays the IDA Pro interface with several windows open: IDA View-A, Strings window, Hex View-1, Structures, Enums, Imports, and Exports. The main window shows assembly code for a function. A call instruction at address 00401257 is highlighted in yellow, and a red arrow points to it from the right. The code includes several push instructions for arguments and a call instruction. Below the call, there are mov, cmp, and jz instructions. A second assembly window is open below, showing further code starting with push instructions for arguments to another function call.

```
00401245  push  1          ; dwContext
00401247  push  0          ; dwFlags
00401249  push  3          ; dwService
0040124B  push  0          ; lpszPassword
0040124D  push  0          ; lpszUserName
0040124F  push  50h        ; nServerPort
00401251  push  [ebp+lpszServerName] ; lpszServerName
00401254  push  [ebp+hInternet] ; hInternet
00401257  call  InternetConnectA
0040125C  mov   [ebp+hConnect], eax
0040125F  cmp   [ebp+hConnect], 0
00401263  jz    short loc_4012A8

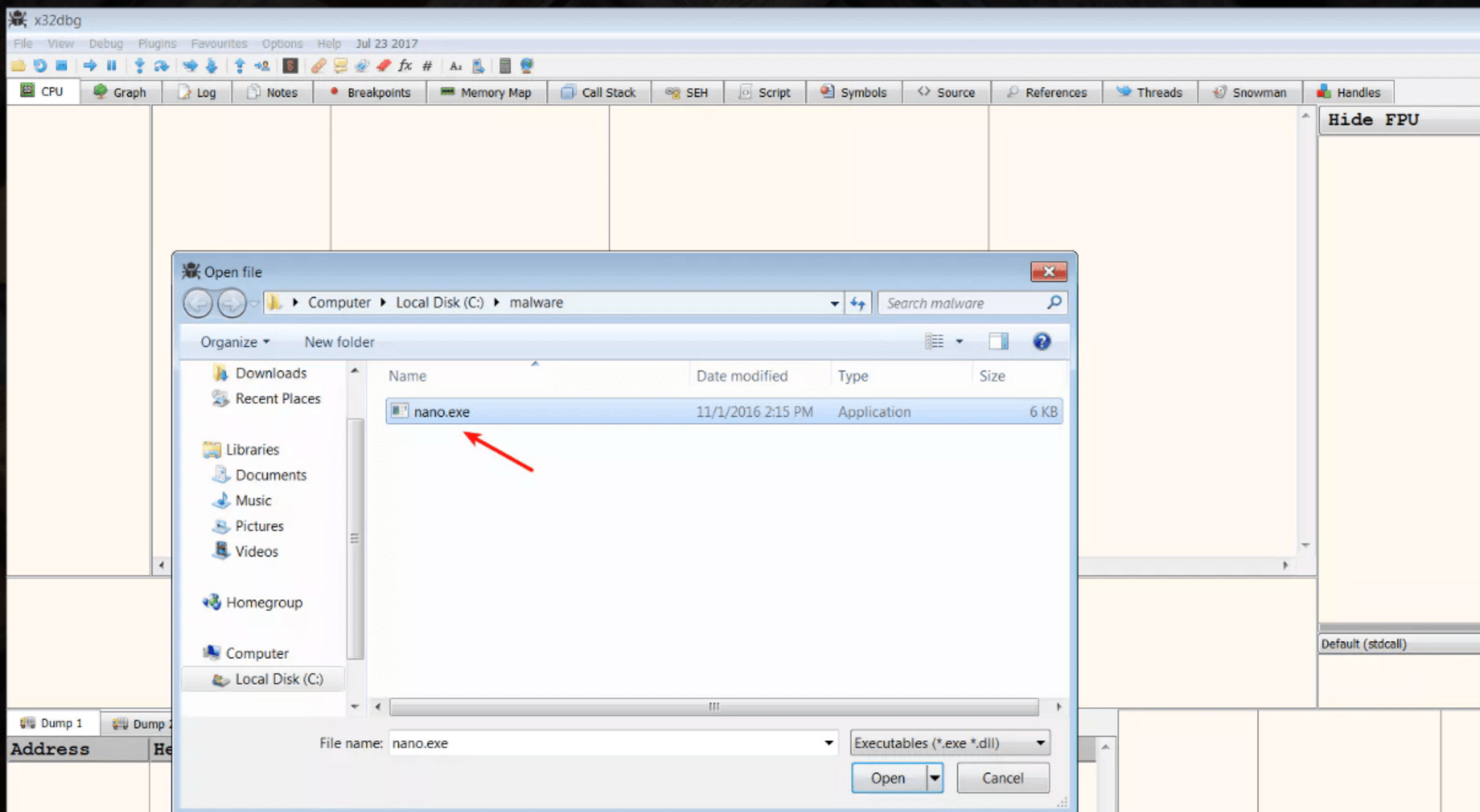
00401265  push  1          ; dwContext
00401267  push  0          ; dwFlags
00401269  push  0          ; lpIpszAcceptTypes
0040126B  push  ds:lpszReferrer ; lpszReferrer
00401271  push  ds:lpszVersion ; lpszVersion
00401277  push  [ebp+lpszObjectName] ; lpszObjectName
```

100.00% (306,730) (997,26) 00000657 00401257: sub\_401223+34 (Synchronized with Hex View-1)



# Inspecting API Call references in x64dbg

# Load the executable in x64dbg by selecting *File / Open*



Once the binary is loaded and paused at the entrypoint, right click on Disassembly window and choose **Search For / Current Module / Intermodular calls** as shown below

The screenshot displays the x32dbg interface with the following components:

- Disassembly Window:** Shows assembly code for nano.exe. The current instruction at EIP 00401458 is `mov eax, 2`. Other instructions include `jmp nano.401477`, `push D`, `call <nano.GetKeyState>`, `cmp eax, 1`, `jne nano.40145F`, `push 1`, `call <nano.GetAsyncKeyState>`, `cmp eax, 1`, `jne nano.40145F`, `push 410`, `push 40`, `call <nano.LocalAlloc>`, and `mov dword ptr ds:[403554], eax`.
- Context Menu:** A right-click menu is open over the disassembly window. The path **Search for** -> **Current Module** -> **Intermodular calls** is highlighted with red boxes and arrows.
- Register Window:** Shows the current state of registers: EAX (7791EF0A), EBX (7FFD4000), ECX (00000000), EDX (00401458), EBP (0012FF94), ESP (0012FF8C), ESI (00000000), EDI (00000000), and EIP (00401458).
- Memory Dump Window:** Shows a dump of memory at address 0012FF8C, containing the string "S.Y.S.T.E.M.....".
- Call Stack Window:** Shows the current call stack with frames for `return` at addresses 0012FF90, 0012FF94, 0012FF98, 0012FF9C, 0012FFA0, 0012FFA4, 0012FFA8, 0012FFAC, 0012FFB0, 0012FFB4, 0012FFB8, 0012FFBC, 0012FFC0, 0012FFC4, 0012FFC8, and 0012FFCC.

The following screenshot shows references to the API calls in the references Windows. You can double click on any entry to navigate to the code where the API is called

x32dbg - File: nano.exe - PID: BC8 - Module: nano.exe - Thread: Main Thread 5C8

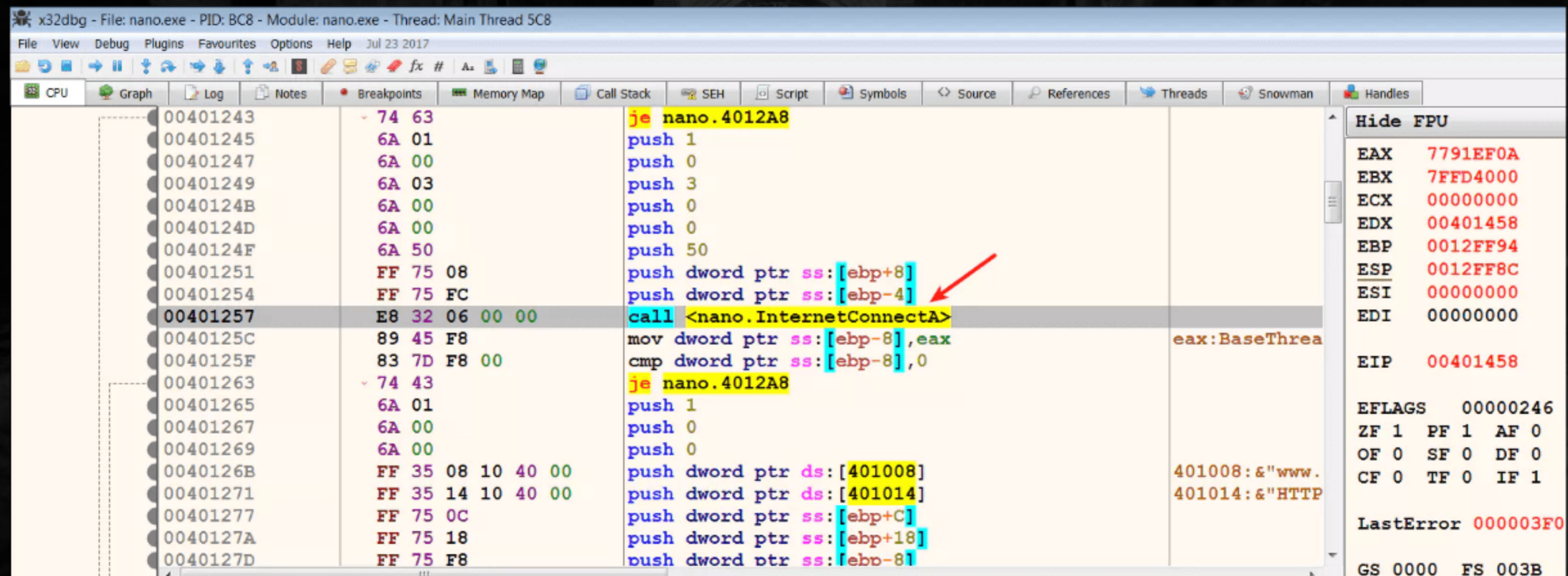
File View Debug Plugins Favourites Options Help Jul 23 2017

CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source **References** Threads Snowman Handles

Calls (nano.exe)

Address	Disassembly	Destination
004010E4	call <nano.WideCharToMultiByte>	<kernel32.WideCharToMultiByte>
00401113	call <nano.GetUserNameW>	<advapi32.GetUserNameW>
00401132	call <nano.RtlZeroMemory>	<ntdll.RtlZeroMemory>
00401156	call <nano.WideCharToMultiByte>	<kernel32.WideCharToMultiByte>
00401177	call <nano.GetUserDefaultLangID>	<kernel32.GetUserDefaultLangID>
0040119B	call <nano.GetVersionExA>	<kernel32.GetVersionExA>
00401200	call <nano.GetSystemMetrics>	<user32.GetSystemMetrics>
00401210	call <nano.GetSystemMetrics>	<user32.GetSystemMetrics>
00401237	call <nano.InternetOpenA>	<wininet.InternetOpenA>
00401257	call <nano.InternetConnectA>	<wininet.InternetConnectA>
00401280	call <nano.HttpOpenRequestA>	<wininet.HttpOpenRequestA>
0040129F	call <nano.HttpSendRequestA>	<wininet.HttpSendRequestA>
004012AB	call <nano.InternetCloseHandle>	<wininet.InternetCloseHandle>
004012B3	call <nano.InternetCloseHandle>	<wininet.InternetCloseHandle>
004012C1	call <nano.HttpEndRequestA>	<wininet.HttpEndRequestA>
00401311	call <nano.RtlZeroMemory>	<ntdll.RtlZeroMemory>
00401328	call <nano.InternetReadFile>	<wininet.InternetReadFile>
00401335	call <nano.InternetCloseHandle>	<wininet.InternetCloseHandle>
0040133D	call <nano.InternetCloseHandle>	<wininet.InternetCloseHandle>
0040134B	call <nano.HttpEndRequestA>	<wininet.HttpEndRequestA>
004013A0	call <nano.LocalAlloc>	<kernel32.LocalAlloc>
004013C8	call <nano.InternetReadFile>	<wininet.InternetReadFile>
004013EB	call <nano.LocalAlloc>	<kernel32.LocalAlloc>
00401412	call <nano.LocalFree>	<kernel32.LocalFree>
00401422	call <nano.InternetCloseHandle>	<wininet.InternetCloseHandle>
0040142A	call <nano.InternetCloseHandle>	<wininet.InternetCloseHandle>
00401438	call <nano.HttpEndRequestA>	<wininet.HttpEndRequestA>
0040144A	call <nano.LocalFree>	<kernel32.LocalFree>
00401461	call <nano.GetKeyState>	<user32.GetKeyState>
0040146D	call <nano.GetAsyncKeyState>	<user32.GetAsyncKeyState>
0040147E	call <nano.LocalAlloc>	<kernel32.LocalAlloc>
004014B3	call <nano.WSASStartup>	<ws2_32.WSASStartup>

Double-clicking the entry in the *references* window will bring up the code where the API is referenced as shown below. At this point, you can set a breakpoint and run the program which pauses the execution before the call to API function and then you inspect the parameters passed to the API



x32dbg - File: nano.exe - PID: BC8 - Module: nano.exe - Thread: Main Thread 5C8

File View Debug Plugins Favourites Options Help Jul 23 2017

CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Snowman Handles

Address	Disassembly	Comment
00401243	je nano.4012A8	
00401245	push 1	
00401247	push 0	
00401249	push 3	
0040124B	push 0	
0040124D	push 0	
0040124F	push 50	
00401251	push dword ptr ss:[ebp+8]	
00401254	push dword ptr ss:[ebp-4]	
00401257	call <nano.InternetConnectA>	
0040125C	mov dword ptr ss:[ebp-8],eax	eax:BaseThrea
0040125F	cmp dword ptr ss:[ebp-8],0	
00401263	je nano.4012A8	
00401265	push 1	
00401267	push 0	
00401269	push 0	
0040126B	push dword ptr ds:[401008]	401008:&"www.
00401271	push dword ptr ds:[401014]	401014:&"HTTP
00401277	push dword ptr ss:[ebp+C]	
0040127A	push dword ptr ss:[ebp+18]	
0040127D	push dword ptr ss:[ebp-8]	

Hide FPU

EAX	7791EF0A
EBX	7FFD4000
ECX	00000000
EDX	00401458
EBP	0012FF94
ESP	0012FF8C
ESI	00000000
EDI	00000000
EIP	00401458
EFLAGS	00000246
ZF	1 PF 1 AF 0
OF	0 SF 0 DF 0
CF	0 TF 0 IF 1
LastError	000003F0
GS	0000 FS 003B