Dynamic Analysis: Debugging

Introduction

In the Basic Dynamic Analysis room, we learned to identify malware traces in the infected system during execution. However, malware authors understand that the malware will be analyzed and want to thwart it. This can be done through various evasion techniques. To defeat these evasion techniques, a malware analyst desires more control over malware execution. In this room, we will learn how a malware analyst can control malware execution better to achieve the desired results.

Learning Objectives

In this room, we will learn:

- The evasion techniques used to evade basic dynamic analysis.
- Introduction to debuggers and how they can help us control the execution flow of malware.
- Manipulating execution flow at runtime by changing registers or other parameters.
- Patching malware to force it to move past the evasion techniques and onto the actual malicious content.

Pre-requisites

To get the best learning outcome from this room, it is recommended that you complete the following rooms:

- Basic Static Analysis
- Basic Dynamic Analysis
- Advanced Static Analysis

The Need for Advanced Dynamic Analysis

Analyzing malware is like a cat-and-mouse game. Malware analysts keep devising new techniques to analyse malware, while malware authors devise new techniques to evade detection. This task will review some techniques that hamper our efforts to analyse malware using static or basic dynamic analysis.

Evasion of Static Analysis:

In the static analysis rooms, we learned techniques to perform static analysis on malware. Malware often hides or tries to look like legitimate software to evade the prying eyes of a malware analyst. Because we are not executing the malware during static analysis, the main focus of evading static analysis is to obfuscate the true functionality of the program until the program is executed. The following techniques can be commonly used to achieve this.

- Changing the hash: We have learned previously that every file has a unique hash. Malware authors exploit this functionality by slightly changing their malware. This way, the malware's hash changes, bypassing the hash-based detection mechanism. Hashes can change even if one bit of the malware is changed (unless we are talking Context-Triggered Piecewise Hashes or fuzzy hashes), so just adding a NOP instruction or other such change can defeat the hash-based detection techniques.
- Defeating AV signatures: Anti-virus signatures and other signature-based detection often depend on static patterns found inside malware. Malware authors change those patterns to try to evade signatures. This technique is often accompanied by general obfuscation of malware code.
- Obfuscation of strings: Some malware authors obfuscate the strings in malware by decoding them at runtime. When we search the malware for strings, we might find nothing useful. However, when the malware runs, it decodes those strings during execution. Malware authors might obfuscate important strings, such as URLs, C2 domains, etc., to avoid burning the infrastructure based on a single-string search.
- Runtime loading of DLLs: Since we can identify malware imports while analyzing PE headers, malware authors often use the Windows libraries' LoadLibrary or LoadLibraryEx to load a DLL at runtime. When analyzing this malware statically, we might not see all the functions it is linked to while analyzing its headers.
- Packing and Obfuscation: Packing is very popular amongst malware authors. Packing malware is like packing a present. When we look at a packed present, we can't say what might be inside it unless we unpack the wrapper and take out the present. Similarly, packers pack the malware in a wrapper by writing code that decodes the malware at runtime. So when performing static analysis, we might be unable to see what is inside the packer. However, when we execute the malware, it unpacks the code, loads the actual malicious code into the memory, and then executes it.

As we might have observed, most of these techniques are suitable for hiding in plain sight, but they can be defeated when the malware is executed while performing dynamic analysis.

Evasion of Basic Dynamic Analysis:
Malware authors do not just accept their fate and let dynamic analysis detect their samples. For evasion of dynamic analysis, a host of techniques are employed. These

techniques generally depend on identifying whether the malware runs in a controlled analysis environment. The following techniques are commonly used for this purpose:

- Identification of VMs: Though some of these techniques might backfire nowadays since a lot of enterprise infrastructure is hosted on VMs, one of the favourites of malware authors has been to identify if the malware is running inside a VM. For this, malware often checks for registry keys or device drivers associated with popular virtualization software such as VMWare and Virtualbox. Similarly, minimal resources, such as a single CPU and limited RAM, might indicate that the malware is running inside a VM. In this scenario, malware will take a different execution path that is not malicious to fool the analyst.
- Timing attacks: Malware will often try to time out automated analysis systems. For example, when malware is executed, it will try to sleep for a day using the Windows Sleep library. After a few minutes, the automated analysis system will shut down, finding no traces of malicious activity. Newer malware analysis systems can identify these attacks and try to mitigate them by shortening the time the malware sleeps. However, malware can identify those mitigations by performing targeted timing checks to see if the time is being manipulated. This can be done by noting the time of execution and comparing it with the current time after the execution of the sleep call.
- Traces of user activity: Malware tries to identify if there are traces of user activity in the machine. If no or very few traces are found, malware will decide that it is being executed inside a controlled system and take a different, benign execution path. Traces of user activity can include no mouse or keyboard movement, lack of browser history, no recently opened files, little system uptime, etc.
- Identification of analysis tools: Malware can ask the Windows OS for a running process list using Process32First, Process32Next, or similar functions. If popular monitoring tools are identified among the list of running processes, malware can take a benign execution path. For example, if ProcMon or ProcExp is running, malware can identify that and switch to benign activities. Another way to identify analysis tools is by looking at the names of different windows open in a system. If the malware finds Ollydbg or ProcMon in the open Windows, it can switch to a different execution path.

By employing these techniques, malware authors make it difficult for malware analysts to perform analysis. However, malware analysts can use some tools and techniques to take greater control over malware execution, helping them defeat these evasion techniques. In the upcoming tasks, we will learn about some of these techniques.

****

Answer the questions below:

Malware sometimes checks the time before and after the execution of certain instructions to find out if it is being analysed. What type of analysis technique is bypassed by this attack?
Answer: Basic Dynamic Analysis

What is a popular technique used by malware authors to obfuscate malware code from static analysis and unwrap it at runtime?
Answer: Packing

Introduction to Debugging

The term Debugging is widely used by software programmers to identify and fix bugs in a program. Similarly, a malware sample trying to evade detection or reverse engineering can also be considered a program having a bug. A malware reverse engineer often has to debug a program to remove any roadblocks that prevent it from performing its malicious activity. Therefore, interactive debugging becomes an essential part of advanced malware analysis. Debuggers provide a malware analyst with the control desired to monitor a running program more closely, looking at the changes in different registers, variables, and memory regions as each instruction is executed. A debugger also allows a malware analyst to change the variables' values to control the program's flow at runtime, providing greater control over the execution path the malware follows.

Types of Debuggers:
We can loosely categorize debuggers into one of the following three categories.

Source-Level Debuggers:
Source Level Debuggers work on the source code level. Most software programmers use source-level debuggers while writing code to check their code for bugs. A source-level debugger is a high-level debugger compared to the other two options. When using a source-level debugger, we often see the local variables and their values.

Assembly-Level Debuggers:
When a program has been compiled, its source code is lost and can't be recovered. This is the case with malware analysis. We don't have the malware's source code we are investigating; instead, we have a compiled binary. An assembly-level debugger can help us debug compiled programs at the assembly level. While debugging with an assembly-level debugger, we often see the CPU registers' values and the debuggee's memory. This is the most common type of debugger used for malware reverse

engineering. The debugger attaches to the program that has to be debugged and executes it as per the analyst's desire.

Kernel-Level Debuggers:
Kernel-level debuggers are a step even lower than assembly-level debuggers. As the name suggests, these debuggers debug a program at the Kernel Level. For this level of debugging, generally, two systems are required. One system is used for debugging the code running on the other system. This is because if the kernel is stopped using a breakpoint, the whole system will stop.

Now let's move to the next task to learn how to debug malware.

****
Answer the questions below:

Can we recover the pre-compilation code of a compiled binary for debugging purposes? Write Y for Yes or N for No
Answer: N

Which type of debugger is used for debugging compiled binaries?
Answer: Assembly-level Debugger

Which debugger works at the lowest level among the discussed debuggers?
Answer: Kernel-level Debugger

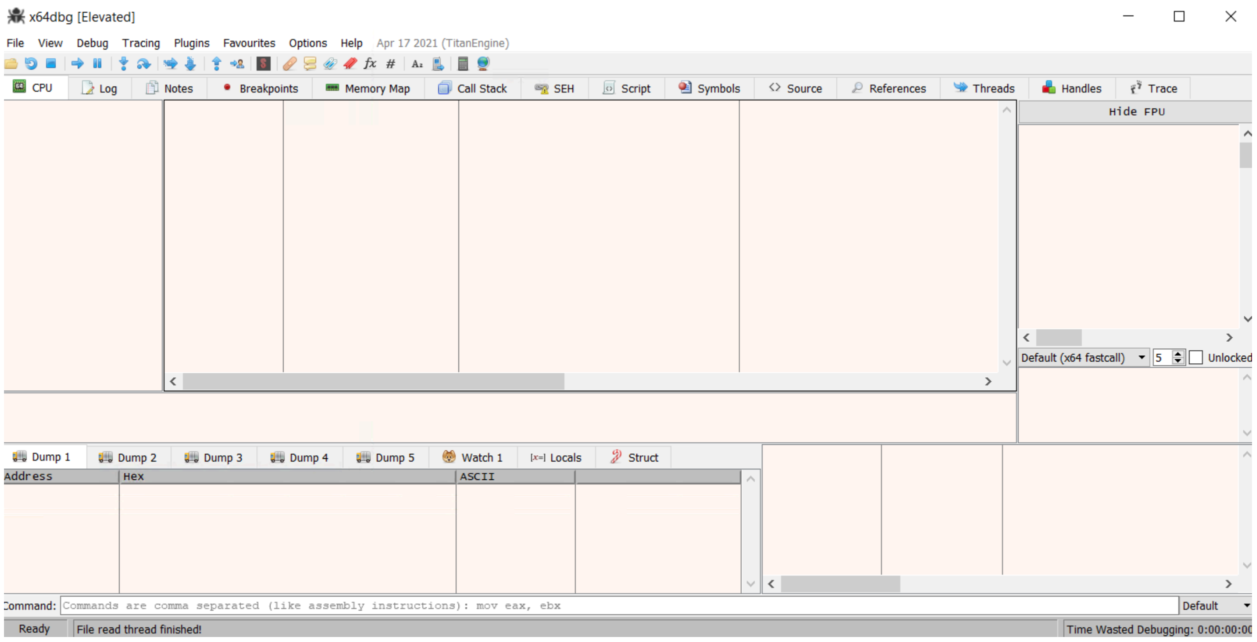
Familiarization with a Debugger

For malware analysis, there are many options to choose a debugger from. These options include Windbg, Ollydbg, IDA, and Ghidra. For this room, we will be using x32dbg and x64dbg. Before proceeding, please start the machine by clicking the option in the top-right corner. The machine will open in the split view. Alternatively, you can connect to the machine using the following credentials:

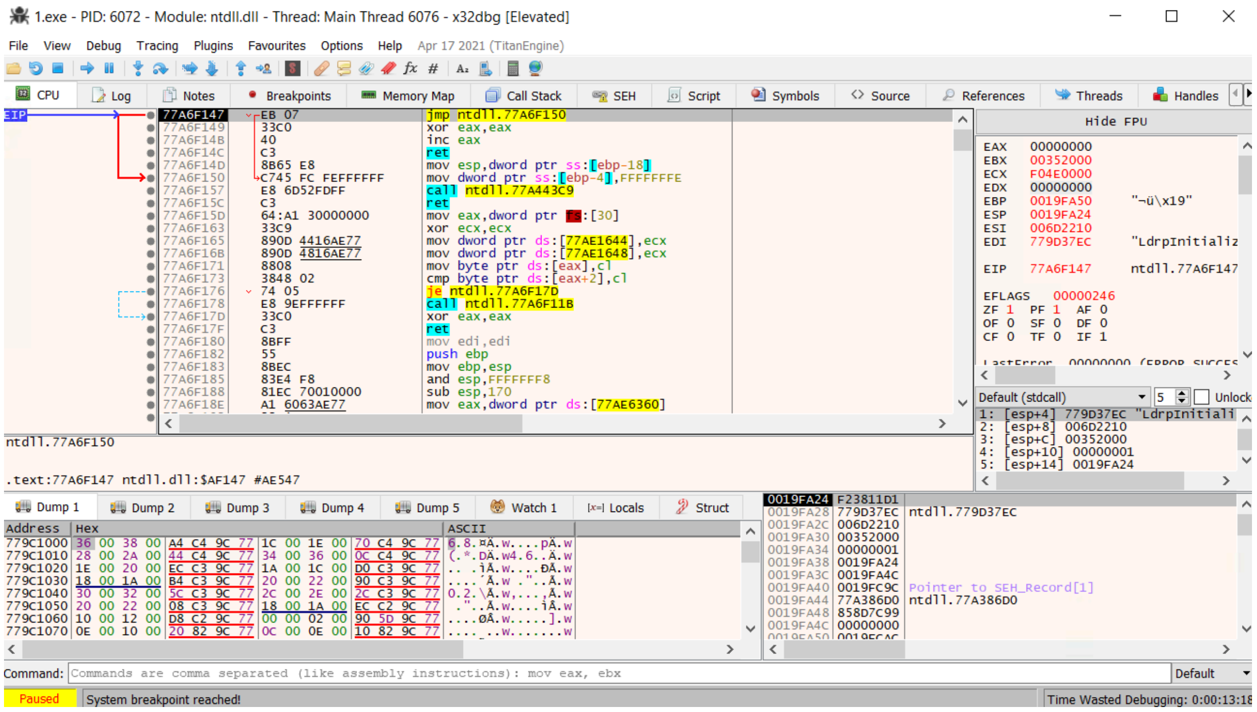
- Username: administrator
- Password: Passw0rd!

The attached VM is essentially a FLARE VM, distributed by Mandiant (previously FireEye, now part of Google), widely used for reverse engineering malware. It includes all the tools required for our analysis in this room. To start, we can navigate to Desktop

> Tools > debuggers > x32dbg.exe to run the debugger. We will be greeted with the following interface.


x64dbg [Elevated]
File View Debug Tracing Plugins Favourites Options Help Apr 17 2021 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace
Hide FPU
Default (x64 fastcall) 5 Unlocked
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 Locals Struct
Address Hex ASCII
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Default
Ready File read thread finished!
Time Wasted Debugging: 0:00:00:00

To open a file in the debugger, we can navigate to File > Open and open our desired file. The below screenshot shows the interface with a sample opened in the debugger. We are currently seeing the CPU tab in the interface. Please note that we must use x32dbg for 32-bit samples and x64dbg for 64-bit samples.


1.exe - PID: 6072 - Module: ntdll.dll - Thread: Main Thread 6076 - x32dbg [Elevated]
File View Debug Tracing Plugins Favourites Options Help Apr 17 2021 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles
Hide FPU
Default (stdcall) 5 Unlock
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 Locals Struct
Default

```
EIP  77A6F147  EB 07              jmp ntdll.77A6F150
     77A6F149  33C0               xor eax,eax
     77A6F14B  40                 inc eax
     77A6F14C  C3                 ret
     77A6F14D  8B65 E8            mov esp,dword ptr ss:[ebp-18]
     77A6F150  C745 FC FEFFFFFF   mov dword ptr ss:[ebp-4],FFFFFFFE
     77A6F157  E8 6D52FDFF        call ntdll.77A443C9
     77A6F15C  C3                 ret
     77A6F15D  64:A1 30000000     mov eax,dword ptr fs:[30]
     77A6F163  33C9               xor ecx,ecx
     77A6F165  890D 4416AE77      mov dword ptr ds:[77AE1644],ecx
     77A6F16B  890D 4816AE77      mov dword ptr ds:[77AE1648],ecx
     77A6F171  8808               mov byte ptr ds:[eax],cl
     77A6F173  3848 02            cmp byte ptr ds:[eax+2],cl
     77A6F176  74 05              je ntdll.77A6F17D
     77A6F178  E8 9EFFFFFF        call ntdll.77A6F11B
     77A6F17D  33C0               xor eax,eax
     77A6F17F  C3                 ret
     77A6F180  8BFF               mov edi,edi
     77A6F182  55                 push ebp
     77A6F183  8BEC               mov ebp,esp
     77A6F185  83E4 F8            and esp,FFFFFFF8
     77A6F188  81EC 70010000      sub esp,170
     77A6F18E  A1 6063AE77        mov eax,dword ptr ds:[77AE6360]

ntdll.77A6F150

.text:77A6F147 ntdll.dll:$AF147 #AE547
```

```
EAX     00000000
EBX     00352000
ECX     F04E0000
EDX     00000000
EBP     0019FA50     "¬ü\x19"
ESP     0019FA24
ESI     006D2210
EDI     779D37EC     "LdrpInitializ

EIP     77A6F147     ntdll.77A6F147

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCES

1: [esp+4] 779D37EC "LdrpInitiali
2: [esp+8] 006D2210
3: [esp+C] 00352000
4: [esp+10] 00000001
5: [esp+14] 0019FA24
```

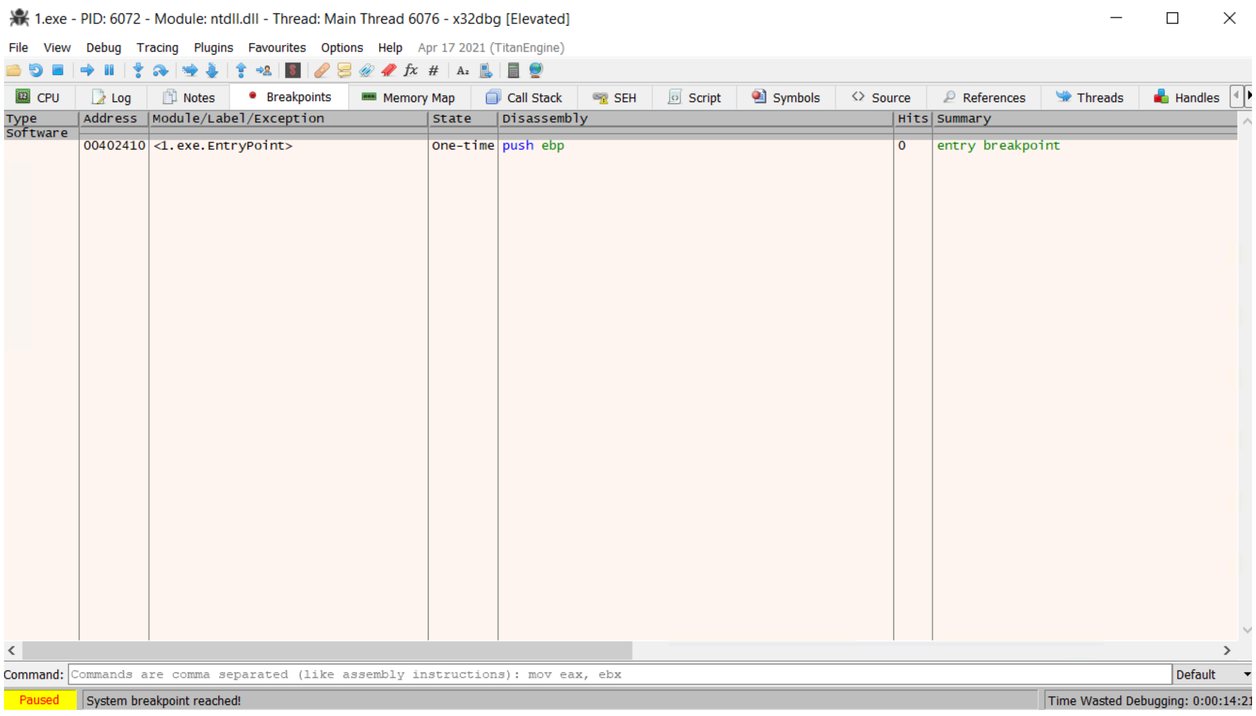
```
Address  Hex                                              ASCII
779C1000 36 00 38 00 A4 C4 9C 77 1C 00 1E 00 70 C4 9C 77  6.8.¤Ä.w....pÄ.w
779C1010 28 00 2A 00 44 C4 9C 77 34 00 36 00 0C C4 9C 77  (.*.DÄ.w4.6..Ä.w
779C1020 1E 00 20 00 EC C3 9C 77 1A 00 1C 00 D0 C3 9C 77  .. .ìÃ.w....ÐÃ.w
779C1030 18 00 1A 00 B4 C3 9C 77 20 00 22 00 90 C3 9C 77  ....´Ã.w .".. Ã.w
779C1040 30 00 32 00 5C C3 9C 77 2C 00 2E 00 2C C3 9C 77  0.2.\Ã.w,...,Ã.w
779C1050 20 00 22 00 08 C3 9C 77 18 00 1A 00 EC C2 9C 77   ."..Ã.w....ìÂ.w
779C1060 10 00 12 00 D8 C2 9C 77 00 00 02 00 90 5D 9C 77  ....ØÂ.w.....].w
779C1070 0E 00 10 00 20 82 9C 77 0C 00 0E 00 10 82 9C 77  .... ..w.......w
```

```
0019FA24  F23811D1
0019FA28  779D37EC  ntdll.779D37EC
0019FA2C  006D2210
0019FA30  00352000
0019FA34  00000001
0019FA38  0019FA24
0019FA3C  0019FA4C
0019FA40  0019FC9C  Pointer to SEH_Record[1]
0019FA44  77A386D0  ntdll.77A386D0
0019FA48  858D7C99
0019FA4C  00000000
0019FA50  0019FCAC
```

```
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused  System breakpoint reached!
Time Wasted Debugging: 0:00:13:18
```

As we can see in the bottom left corner, the execution of the program is paused because a System breakpoint has been reached. We can control whether to execute one instruction at a time or the whole program. But before that, let's take a look at the screenshot above. Here, we can see Disassembly in the middle pane, with the Instruction pointer pointing to the next instruction executed if we start the program. In the right pane, we can see the registers and their values. We can note that the value in EIP is the address EIP is pointing to in the disassembly pane. Similarly, on the bottom pane, we can see the stack view (right), the dump view (left), and the timer showing the time we spent debugging the sample (right corner).

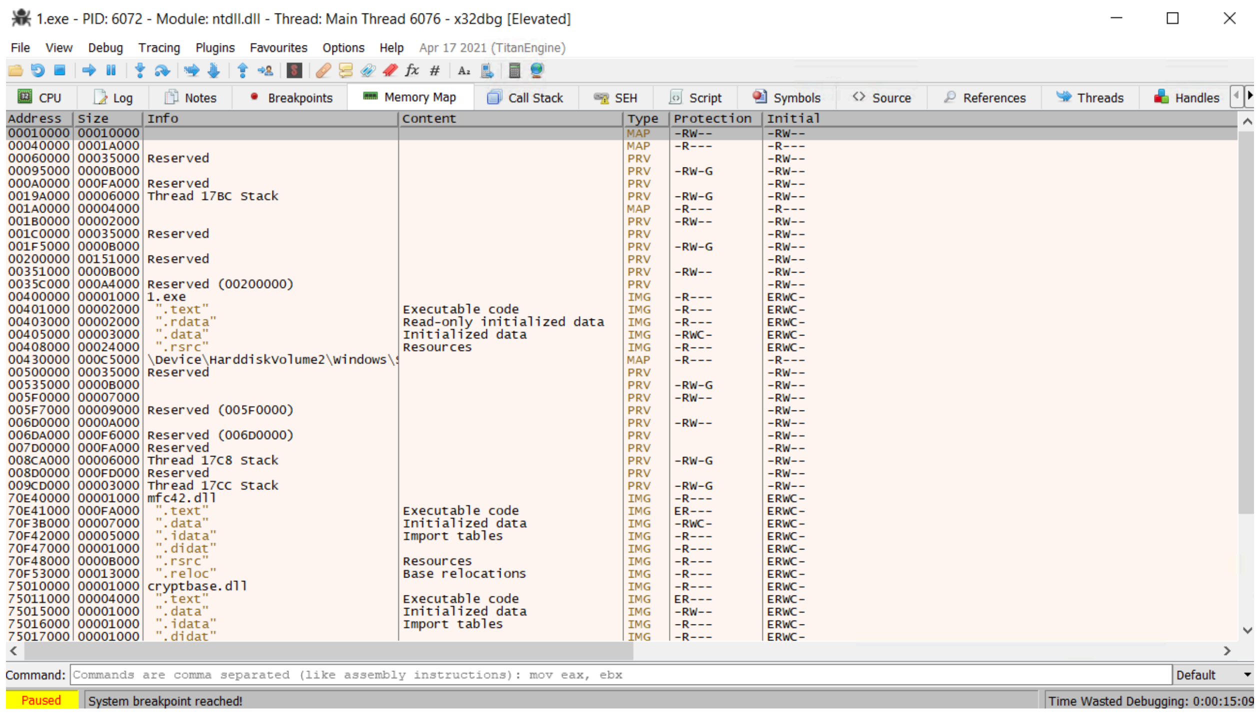
Let's look at some of the other tabs. The breakpoints tab shows the current status of breakpoints. Breakpoints are points where the execution of the program is paused for the analyst to analyze the registers and memory. A breakpoint on a specific instruction can be enabled by clicking the dot in front of that instruction in the CPU tab.


1.exe - PID: 6072 - Module: ntdll.dll - Thread: Main Thread 6076 - x32dbg [Elevated]
File View Debug Tracing Plugins Favourites Options Help Apr 17 2021 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles
Command:
Default
Paused

Type	Address	Module/Label/Exception	State	Disassembly	Hits	Summary
Software						
	00402410	<1.exe.EntryPoint>	One-time	push ebp	0	entry breakpoint

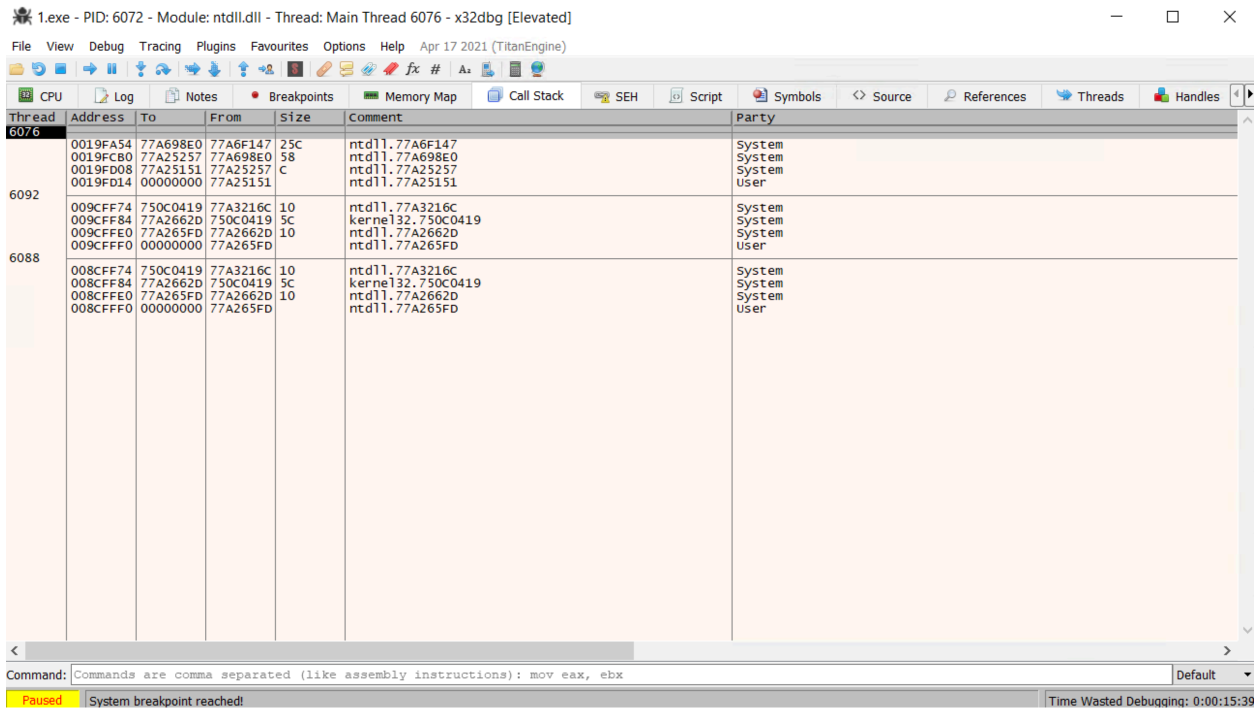
```
Commands are comma separated (like assembly instructions): mov eax, ebx
System breakpoint reached!
Time Wasted Debugging: 0:00:14:21
```

The Memory Map shows the memory of the program.



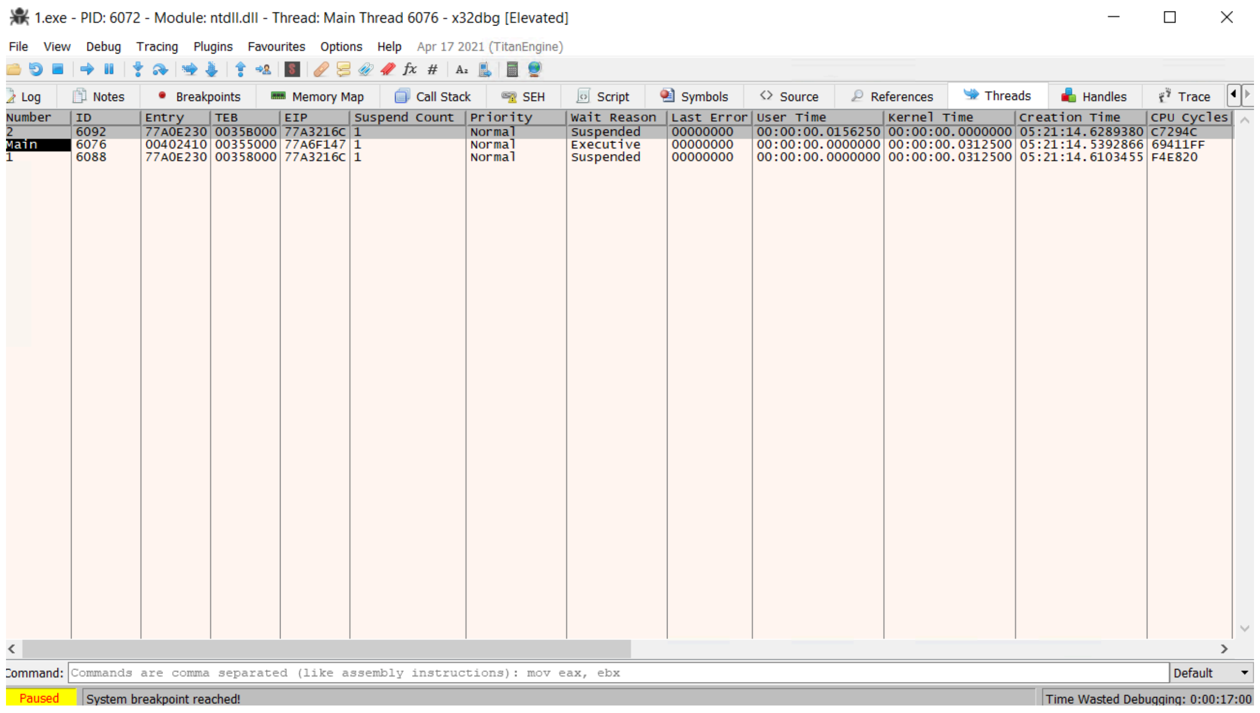
```
Address  Size     Info                                  Content                         Type Protection Initial
00010000 00010000                                                                       MAP  -RW--      -RW--
00040000 0001A000                                                                       MAP  -R---      -R---
00060000 00035000 Reserved                                                              PRV             -RW--
00095000 0000B000                                                                       PRV  -RW-G      -RW--
000A0000 000FA000 Reserved                                                              PRV             -RW--
0019A000 00006000 Thread 17BC Stack                                                     PRV  -RW-G      -RW--
001A0000 00004000                                                                       MAP  -R---      -R---
001B0000 00002000                                                                       PRV  -RW--      -RW--
001C0000 00035000 Reserved                                                              PRV             -RW--
001F5000 0000B000                                                                       PRV  -RW-G      -RW--
00200000 00151000 Reserved                                                              PRV             -RW--
00351000 0000B000                                                                       PRV  -RW--      -RW--
0035C000 000A4000 Reserved (00200000)                                                   PRV             -RW--
00400000 00001000 1.exe                                                                 IMG  -R---      ERWC-
00401000 00002000  ".text"                              Executable code                 IMG  -R---      ERWC-
00403000 00002000  ".rdata"                             Read-only initialized data      IMG  -R---      ERWC-
00405000 00003000  ".data"                              Initialized data                IMG  -RWC-      ERWC-
00408000 00024000  ".rsrc"                              Resources                       IMG  -R---      ERWC-
00430000 000C5000 \Device\HarddiskVolume2\Windows\S                                     MAP  -R---      -R---
00500000 00035000 Reserved                                                              PRV             -RW--
00535000 0000B000                                                                       PRV  -RW-G      -RW--
005F0000 00007000                                                                       PRV  -RW--      -RW--
005F7000 00009000 Reserved (005F0000)                                                   PRV             -RW--
006D0000 0000A000                                                                       PRV  -RW--      -RW--
006DA000 000F6000 Reserved (006D0000)                                                   PRV             -RW--
007D0000 000FA000 Reserved                                                              PRV             -RW--
008CA000 00006000 Thread 17C8 Stack                                                     PRV  -RW-G      -RW--
008D0000 000FD000 Reserved                                                              PRV             -RW--
009CD000 00003000 Thread 17CC Stack                                                     PRV  -RW-G      -RW--
70E40000 00001000 mfc42.dll                                                             IMG  -R---      ERWC-
70E41000 000FA000  ".text"                              Executable code                 IMG  ER---      ERWC-
70F3B000 00007000  ".data"                              Initialized data                IMG  -RWC-      ERWC-
70F42000 00005000  ".idata"                             Import tables                   IMG  -R---      ERWC-
70F47000 00001000  ".didat"                                                             IMG  -R---      ERWC-
70F48000 0000B000  ".rsrc"                              Resources                       IMG  -R---      ERWC-
70F53000 00013000  ".reloc"                             Base relocations                IMG  -R---      ERWC-
75010000 00001000 cryptbase.dll                                                         IMG  -R---      ERWC-
75011000 00004000  ".text"                              Executable code                 IMG  ER---      ERWC-
75015000 00001000  ".data"                              Initialized data                IMG  -RW--      ERWC-
75016000 00001000  ".idata"                             Import tables                   IMG  -R---      ERWC-
75017000 00001000  ".didat"                                                             IMG  -R---      ERWC-

Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused  System breakpoint reached!                                        Time Wasted Debugging: 0:00:15:09
```

We can also see the Call stack of the program.



```
Thread Address  To       From     Size Comment               Party
6076
       0019FA54 77A698E0 77A6F147 25C  ntdll.77A6F147        System
       0019FCB0 77A25257 77A698E0 58   ntdll.77A698E0        System
       0019FD08 77A25151 77A25257 C    ntdll.77A25257        System
       0019FD14 00000000 77A25151      ntdll.77A25151        User
6092
       009CFF74 750C0419 77A3216C 10   ntdll.77A3216C        System
       009CFF84 77A2662D 750C0419 5C   kernel32.750C0419     System
       009CFFE0 77A265FD 77A2662D 10   ntdll.77A2662D        System
       009CFFF0 00000000 77A265FD      ntdll.77A265FD        User
6088
       008CFF74 750C0419 77A3216C 10   ntdll.77A3216C        System
       008CFF84 77A2662D 750C0419 5C   kernel32.750C0419     System
       008CFFE0 77A265FD 77A2662D 10   ntdll.77A2662D        System
       008CFFF0 00000000 77A265FD      ntdll.77A265FD        User

Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused  System breakpoint reached!                                        Time Wasted Debugging: 0:00:15:39
```

The threads running in the program are shown in the threads tab.

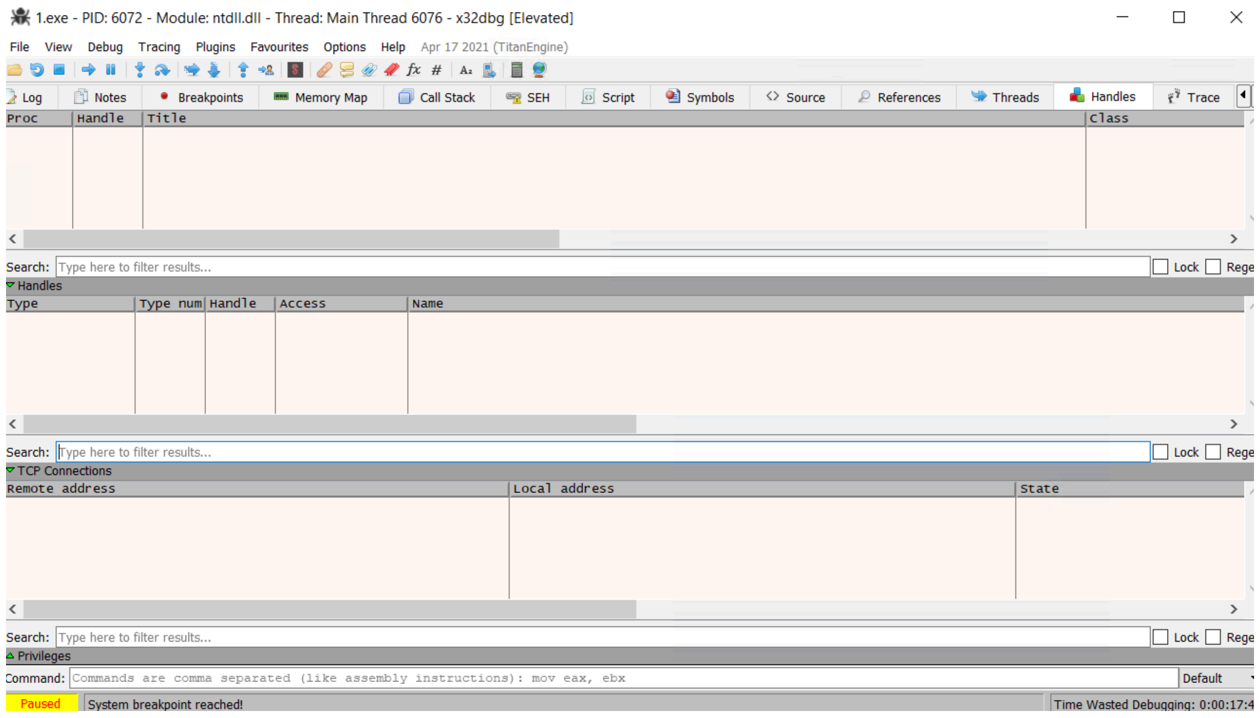


1.exe - PID: 6072 - Module: ntdll.dll - Thread: Main Thread 6076 - x32dbg [Elevated]

Number	ID	Entry	TEB	EIP	Suspend Count	Priority	Wait Reason	Last Error	User Time	Kernel Time	Creation Time	CPU Cycles
2	6092	77A0E230	0035B000	77A3216C	1	Normal	Suspended	00000000	00:00:00.0156250	00:00:00.0000000	05:21:14.6289380	C7294C
Main	6076	00402410	00355000	77A6F147	1	Normal	Executive	00000000	00:00:00.0000000	00:00:00.0312500	05:21:14.5392866	69411FF
1	6088	77A0E230	00358000	77A3216C	1	Normal	Suspended	00000000	00:00:00.0000000	00:00:00.0312500	05:21:14.6103455	F4E820

```
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused   System breakpoint reached!   Time Wasted Debugging: 0:00:17:00
```

Any handles to files, processes, or other resources the process accesses are shown in the handles tab.



1.exe - PID: 6072 - Module: ntdll.dll - Thread: Main Thread 6076 - x32dbg [Elevated]

Proc	Handle	Title	Class

Search: Type here to filter results...

Handles

Type	Type num	Handle	Access	Name

Search: Type here to filter results...

TCP Connections

Remote address	Local address	State

Search: Type here to filter results...

Privileges

```
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused   System breakpoint reached!   Time Wasted Debugging: 0:00:17:4
```

In the next task, we will learn how to utilize this information for malware analysis.

****

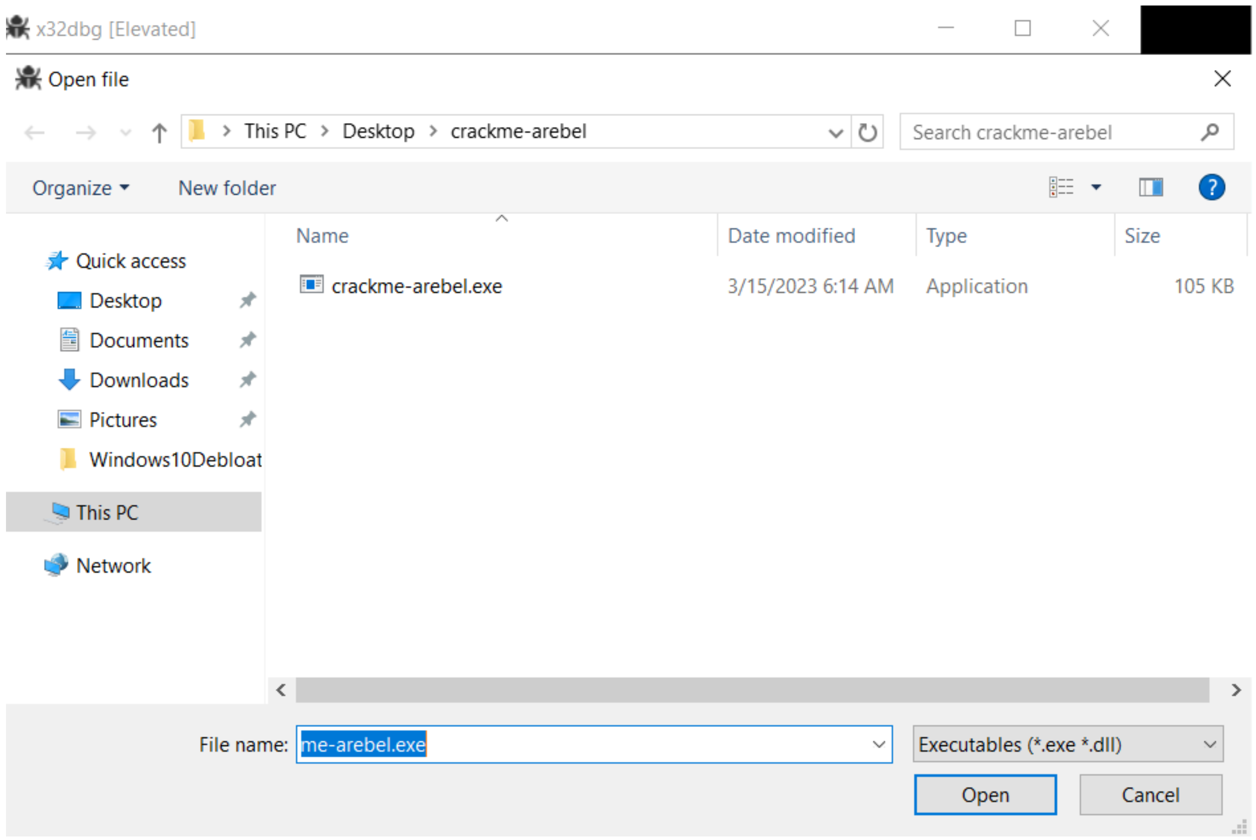
Answer the questions below:

In which tab is the disassembly view shown in x32dbg?
Answer: CPU Tab

If a process opens a file or a process, where can we see information regarding that opened file or process?
Answer: Handles Tab

Debugging in Practice

Now that we are familiar with the UI of x32dbg, let's learn about debugging a program by executing it step-by-step. Let's start by opening one of the crackmes from the attached VM. We can do this by clicking File > Open. We will see the below window asking us to select the file we want to select. Select crackme-arebel.exe as the file we want to debug.

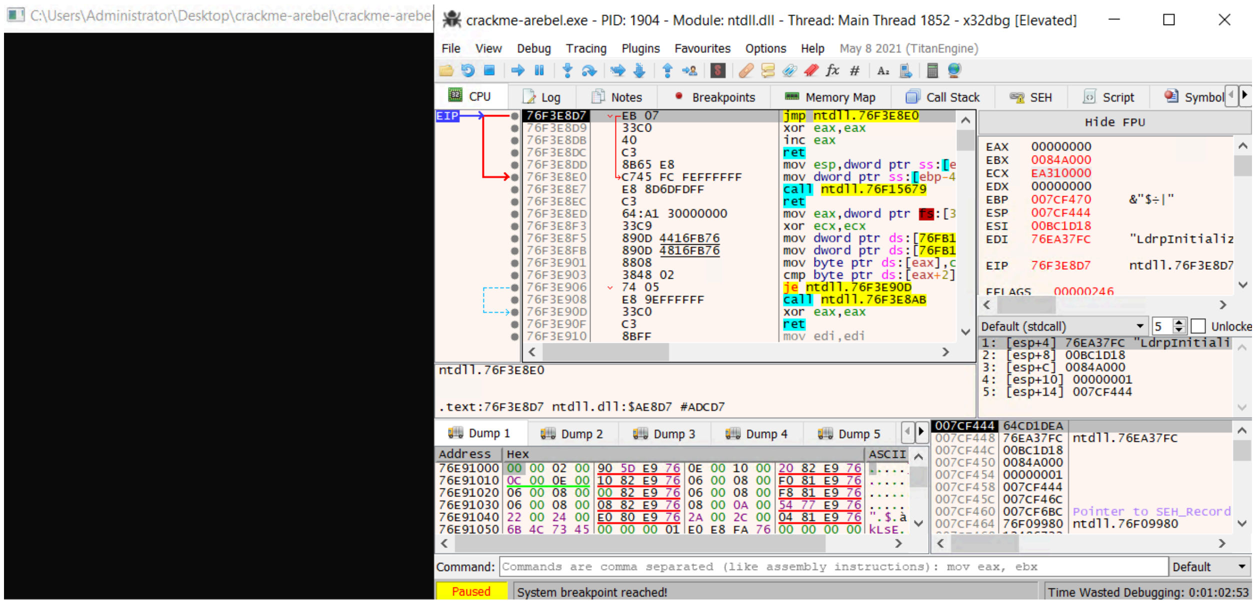


Name	Date modified	Type	Size
crackme-arebel.exe	3/15/2023 6:14 AM	Application	105 KB

File name: me-arebel.exe
Executables (*.exe *.dll)

We select the file we need to open. The debugger attaches itself to the process and pauses it before it starts. We can see that a blank Command window opens in the background. This shows that the process has started but was stopped by the debugger. Please note that this window might not open with all processes. It will depend on the

User Interface of the process. In the debugger window, we can see the familiar disassembly, the registers, the stack and other useful information. Please note that the address might not always be as shown in the images.



C:\Users\Administrator\Desktop\crackme-arebel\crackme-arebe

crackme-arebel.exe - PID: 1904 - Module: ntdll.dll - Thread: Main Thread 1852 - x32dbg [Elevated]

File View Debug Tracing Plugins Favourites Options Help May 8 2021 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbol

```
EIP 76F3E8D7   EB 07              jmp ntdll.76F3E8E0
    76F3E8D9   33C0               xor eax,eax
    76F3E8DB   40                 inc eax
    76F3E8DC   C3                 ret
    76F3E8DD   8B65 E8            mov esp,dword ptr ss:[e
    76F3E8E0   C745 FC FEFFFFFF   mov dword ptr ss:[ebp-4
    76F3E8E7   E8 8D6DFDFF        call ntdll.76F15679
    76F3E8EC   C3                 ret
    76F3E8ED   64:A1 30000000     mov eax,dword ptr fs:[3
    76F3E8F3   33C9               xor ecx,ecx
    76F3E8F5   890D 4416FB76      mov dword ptr ds:[76FB1
    76F3E8FB   890D 4816FB76      mov dword ptr ds:[76FB1
    76F3E901   8808               mov byte ptr ds:[eax],c
    76F3E903   3848 02            cmp byte ptr ds:[eax+2]
    76F3E906   74 05              je ntdll.76F3E90D
    76F3E908   E8 9EFFFFFF        call ntdll.76F3E8AB
    76F3E90D   33C0               xor eax,eax
    76F3E90F   C3                 ret
    76F3E910   8BFF               mov edi,edi

ntdll.76F3E8E0

.text:76F3E8D7 ntdll.dll:$AE8D7 #ADCD7
```

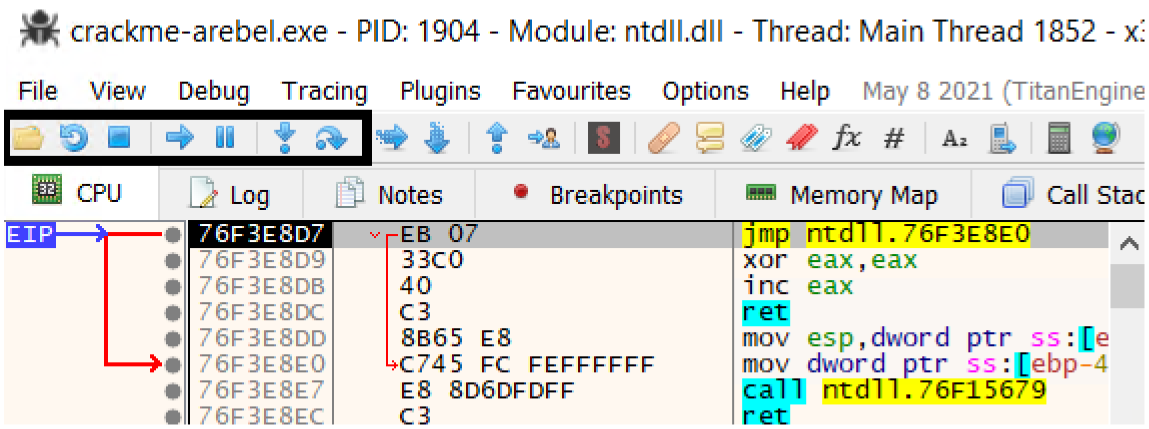
```
Hide FPU
EAX   00000000
EBX   0084A000
ECX   EA310000
EDX   00000000
EBP   007CF470      &"$÷|"
ESP   007CF444
ESI   00BC1D18
EDI   76EA37FC      "LdrpInitializ

EIP   76F3E8D7      ntdll.76F3E8D7

EFLAGS   00000246

Default (stdcall)   5   Unlocke
1: [esp+4] 76EA37FC "LdrpInitiali
2: [esp+8] 00BC1D18
3: [esp+C] 0084A000
4: [esp+10] 00000001
5: [esp+14] 007CF444
```

```
Address  Hex                                              ASCII
76E91000 00 00 02 00 90 5D E9 76 0E 00 10 00 20 82 E9 76 .....
76E91010 0C 00 0E 00 10 82 E9 76 06 00 08 00 F0 81 E9 76 .....
76E91020 06 00 08 00 00 82 E9 76 06 00 08 00 F8 81 E9 76 .....
76E91030 06 00 08 00 08 82 E9 76 08 00 0A 00 54 77 E9 76 .....
76E91040 22 00 24 00 E0 80 E9 76 2A 00 2C 00 04 81 E9 76 ".$.à
76E91050 6B 4C 73 45 00 00 00 01 E0 E8 FA 76 00 00 00 00 kLsE.
```

```
007CF444 64CD1DEA
007CF448 76EA37FC ntdll.76EA37FC
007CF44C 00BC1D18
007CF450 0084A000
007CF454 00000001
007CF458 007CF444
007CF45C 007CF46C
007CF460 007CF6BC Pointer to SEH_Record
007CF464 76F09980 ntdll.76F09980
```

```
Command: Commands are comma separated (like assembly instructions): mov eax, ebx   Default
Paused   System breakpoint reached!   Time Wasted Debugging: 0:01:02:53
```

In the debugger window, we have some features that help us control the execution. In the below screenshot, we can see some of them highlighted.



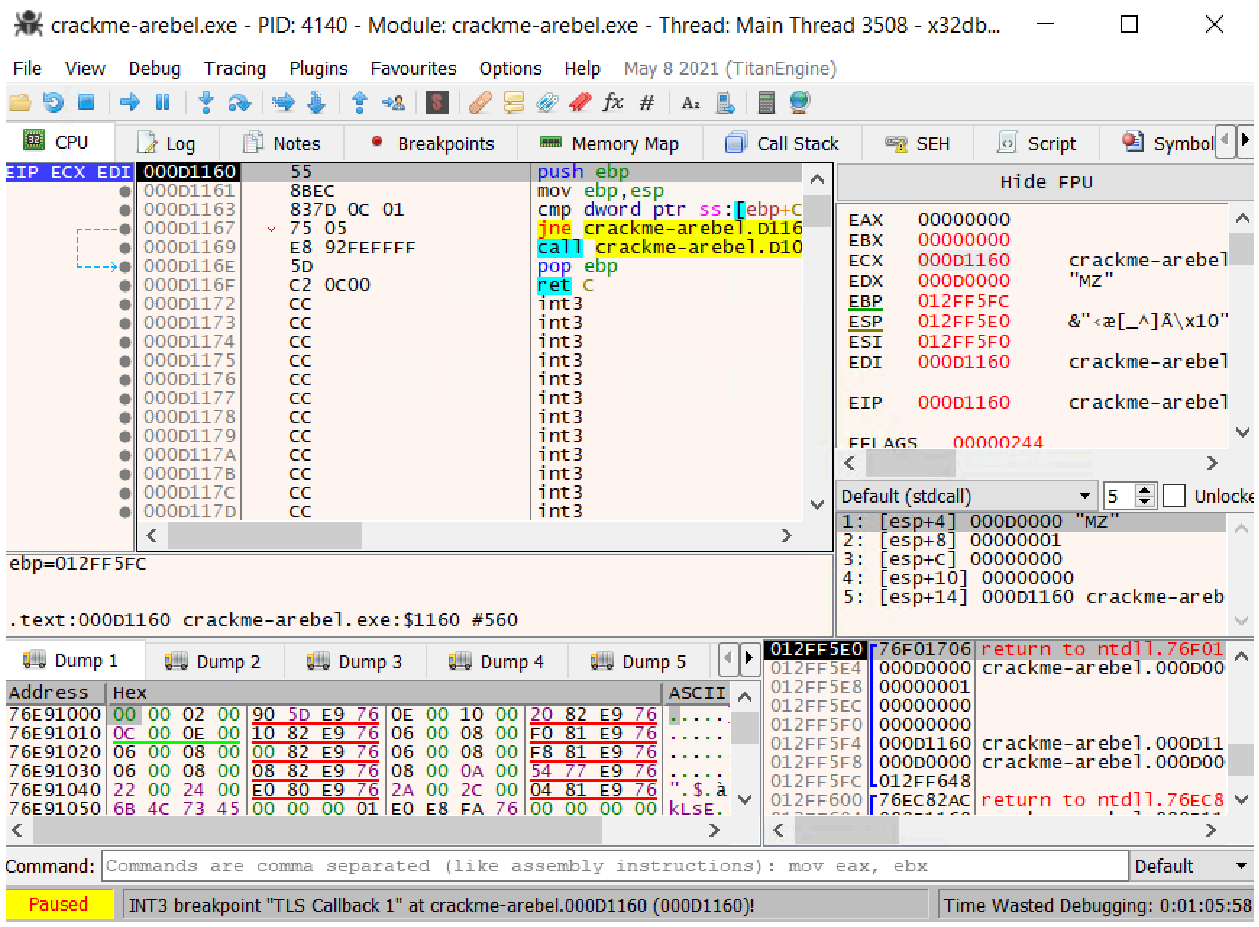
crackme-arebel.exe - PID: 1904 - Module: ntdll.dll - Thread: Main Thread 1852 - x

File View Debug Tracing Plugins Favourites Options Help May 8 2021 (TitanEngine
CPU Log Notes Breakpoints Memory Map Call Stac

```
EIP 76F3E8D7   EB 07              jmp ntdll.76F3E8E0
    76F3E8D9   33C0               xor eax,eax
    76F3E8DB   40                 inc eax
    76F3E8DC   C3                 ret
    76F3E8DD   8B65 E8            mov esp,dword ptr ss:[e
    76F3E8E0   C745 FC FEFFFFFF   mov dword ptr ss:[ebp-4
    76F3E8E7   E8 8D6DFDFF        call ntdll.76F15679
    76F3E8EC   C3                 ret
```

Among these, from left to right, we have the feature for opening a new file, restarting the execution of the opened file from the start, and stopping the execution. The arrow key will execute the program until it is stopped or paused by some other control (Please do not press this button right now as it might freeze the program. If you did, kindly restart the program by terminating it through the Task Manager). Then we have the pause option. The last two options are for stepping into and stepping over. The rest of the options are also useful, but for the scope of this room, we will only be using the ones mentioned above.

So now that we know how to control the execution, let's execute the program using the arrow key (Please only press it once). When we do that, we can see in the bottom right corner that the program's status changes to Running, then Paused. Along with the status, we have the reason for pausing the execution. It says "INT3 breakpoint "TLS Callback 1"....". This means that we have hit a TLS callback, and the debugger was programmed to break execution on TLS callbacks.


crackme-arebel.exe - PID: 4140 - Module: crackme-arebel.exe - Thread: Main Thread 3508 - x32db...
File View Debug Tracing Plugins Favourites Options Help May 8 2021 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbol
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5
Hide FPU
Default (stdcall) 5 Unlocke
Command:
Default

```
EIP ECX EDI  000D1160   55             push ebp
             000D1161   8BEC           mov ebp,esp
             000D1163   837D 0C 01     cmp dword ptr ss:[ebp+C
             000D1167   75 05          jne crackme-arebel.D116
             000D1169   E8 92FEFFFF    call crackme-arebel.D10
             000D116E   5D             pop ebp
             000D116F   C2 0C00        ret C
             000D1172   CC             int3
             000D1173   CC             int3
             000D1174   CC             int3
             000D1175   CC             int3
             000D1176   CC             int3
             000D1177   CC             int3
             000D1178   CC             int3
             000D1179   CC             int3
             000D117A   CC             int3
             000D117B   CC             int3
             000D117C   CC             int3
             000D117D   CC             int3

ebp=012FF5FC

.text:000D1160 crackme-arebel.exe:$1160 #560
```

```
EAX     00000000
EBX     00000000
ECX     000D1160     crackme-arebel
EDX     000D0000     "MZ"
EBP     012FF5FC
ESP     012FF5E0     &"‹æ[_^]Å\x10"
ESI     012FF5F0
EDI     000D1160     crackme-arebel

EIP     000D1160     crackme-arebel

EFLAGS     00000244
```

```
1: [esp+4] 000D0000 "MZ"
2: [esp+8] 00000001
3: [esp+C] 00000000
4: [esp+10] 00000000
5: [esp+14] 000D1160 crackme-areb
```

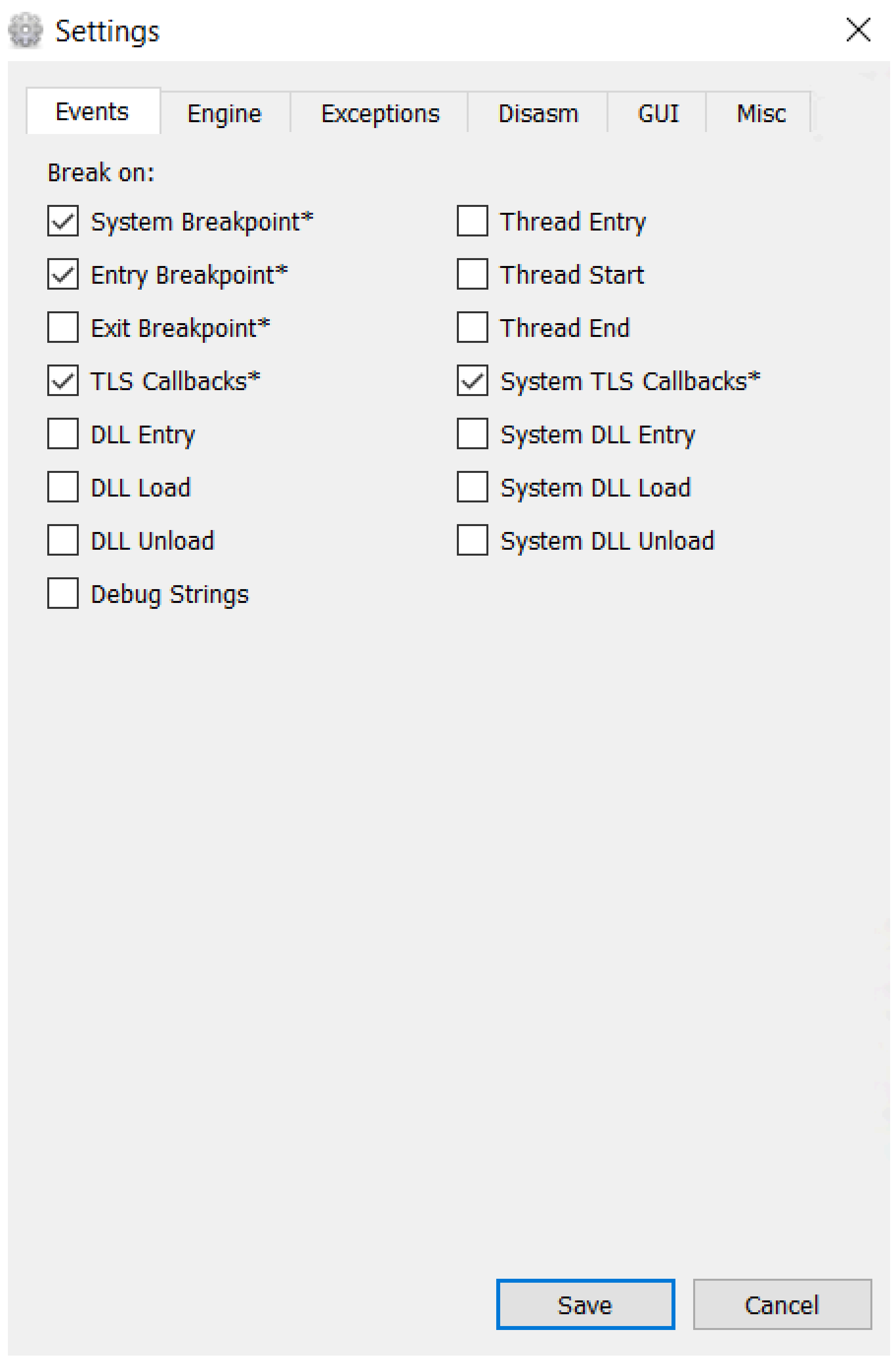
```
Address   Hex                                               ASCII
76E91000  00 00 02 00 90 5D E9 76 0E 00 10 00 20 82 E9 76  .....
76E91010  0C 00 0E 00 10 82 E9 76 06 00 08 00 F0 81 E9 76  .....
76E91020  06 00 08 00 00 82 E9 76 06 00 08 00 F8 81 E9 76  .....
76E91030  06 00 08 00 08 82 E9 76 08 00 0A 00 54 77 E9 76  .....
76E91040  22 00 24 00 E0 80 E9 76 2A 00 2C 00 04 81 E9 76  ".$.à
76E91050  6B 4C 73 45 00 00 00 01 E0 E8 FA 76 00 00 00 00  kLsE.
```

```
012FF5E0  76F01706  return to ntdll.76F01
012FF5E4  000D0000  crackme-arebel.000D00
012FF5E8  00000001
012FF5EC  00000000
012FF5F0  00000000
012FF5F4  000D1160  crackme-arebel.000D11
012FF5F8  000D0000  crackme-arebel.000D00
012FF5FC  012FF648
012FF600  76EC82AC  return to ntdll.76EC8
```

```
Commands are comma separated (like assembly instructions): mov eax, ebx
```

```
Paused   INT3 breakpoint "TLS Callback 1" at crackme-arebel.000D1160 (000D1160)!   Time Wasted Debugging: 0:01:05:58
```

In the debugger, we can set where to put automatic breakpoints on a program by going to the Options > Preferences menu. The following screenshot shows the different points where breakpoints are automatically placed. We can see that TLS Callbacks and System TLS Callbacks are checked, which means execution will be broken on these events.



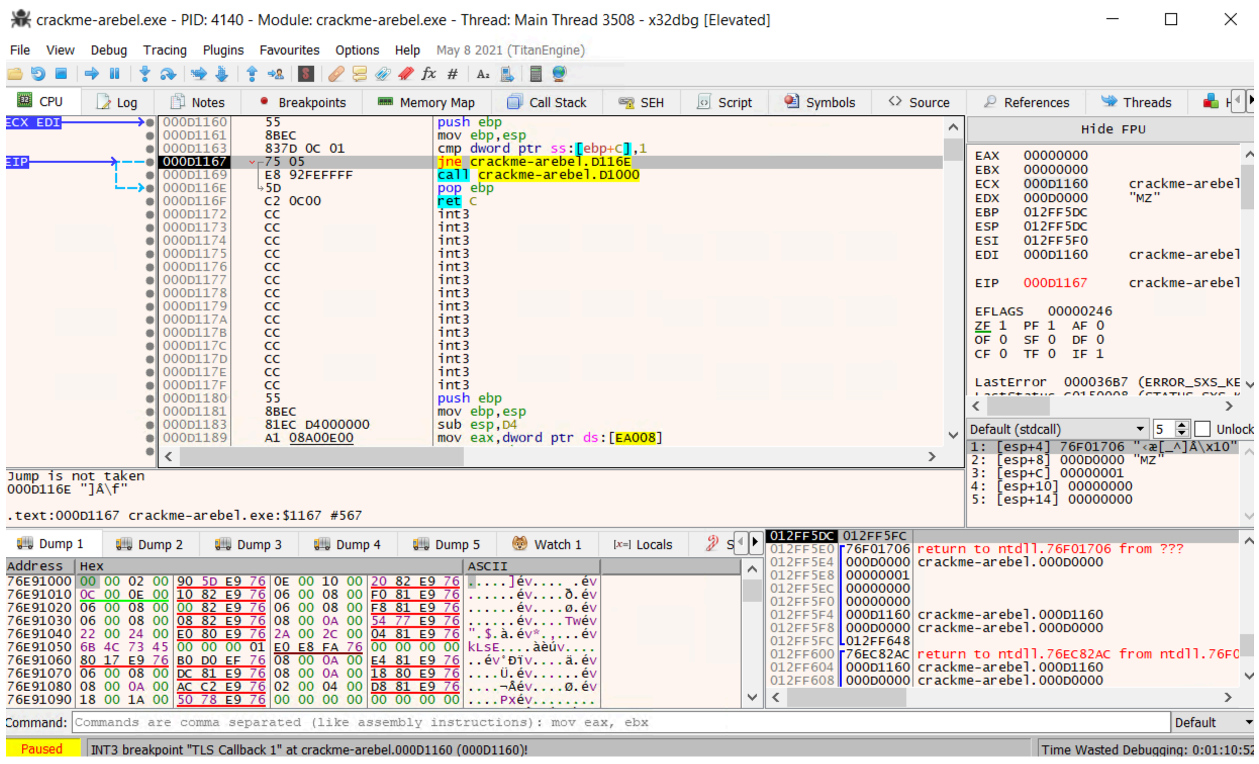
Settings

Events | Engine | Exceptions | Disasm | GUI | Misc

Break on:

- [✓] System Breakpoint*
- [✓] Entry Breakpoint*
- [ ] Exit Breakpoint*
- [✓] TLS Callbacks*
- [ ] DLL Entry
- [ ] DLL Load
- [ ] DLL Unload
- [ ] Debug Strings
- [ ] Thread Entry
- [ ] Thread Start
- [ ] Thread End
- [✓] System TLS Callbacks*
- [ ] System DLL Entry
- [ ] System DLL Load
- [ ] System DLL Unload

Since TLS callbacks are often used as an anti-reverse engineering technique, we should be careful when navigating this TLS callback. Therefore, it will be prudent to single-step each instruction while we are in the callback. We can do that by clicking the step-into option we discussed above. After stepping into every instruction, we see the EIP moving to the next instruction, and the values in the registers and stack change accordingly. After stepping in a few instructions, we reach a conditional jump instruction. In the pane below the disassembly, we can see that the debugger tells us that the jump is not taken. The registers pane shows that the ZF is set, so the jump is not taken.


crackme-arebel.exe - PID: 4140 - Module: crackme-arebel.exe - Thread: Main Thread 3508 - x32dbg [Elevated]

```
000D1160   55              push ebp
000D1161   8BEC            mov ebp,esp
000D1163   837D 0C 01      cmp dword ptr ss:[ebp+C],1
000D1167   75 05           jne crackme-arebel.D116E
000D1169   E8 92FEFFFF     call crackme-arebel.D1000
000D116E   5D              pop ebp
000D116F   C2 0C00         ret C
000D1172   CC              int3
000D1173   CC              int3
000D1174   CC              int3
000D1175   CC              int3
000D1176   CC              int3
000D1177   CC              int3
000D1178   CC              int3
000D1179   CC              int3
000D117A   CC              int3
000D117B   CC              int3
000D117C   CC              int3
000D117D   CC              int3
000D117E   CC              int3
000D117F   CC              int3
000D1180   55              push ebp
000D1181   8BEC            mov ebp,esp
000D1183   81EC D4000000   sub esp,D4
000D1189   A1 08A00E00     mov eax,dword ptr ds:[EA008]

Jump is not taken
000D116E "]Å\f"

.text:000D1167 crackme-arebel.exe:$1167 #567
```

```
EAX  00000000
EBX  00000000
ECX  000D1160     crackme-arebel
EDX  000D0000     "MZ"
EBP  012FF5DC
ESP  012FF5DC
ESI  012FF5F0
EDI  000D1160     crackme-arebel

EIP  000D1167     crackme-arebel

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  000036B7 (ERROR_SXS_KE
```

Default (stdcall) 5 Unlock

```
1: [esp+4] 76F01706 "‹æ[_^]Å\x10"
2: [esp+8] 000D0000 "MZ"
3: [esp+C] 00000001
4: [esp+10] 00000000
5: [esp+14] 00000000
```

```
Address   Hex                                              ASCII
76E91000  00 00 02 00 90 5D E9 76 0E 00 10 00 20 82 E9 76  .....]év.... .év
76E91010  0C 00 0E 00 10 82 E9 76 06 00 08 00 F0 81 E9 76  ......év....ð.év
76E91020  06 00 08 00 00 82 E9 76 06 00 08 00 F8 81 E9 76  ......év....ø.év
76E91030  06 00 08 00 08 82 E9 76 08 00 0A 00 54 77 E9 76  ......év....Twév
76E91040  22 00 24 00 E0 80 E9 76 2A 00 2C 00 04 81 E9 76  ".$.à.év*.,...év
76E91050  6B 4C 73 45 00 00 00 01 E0 E8 FA 76 00 00 00 00  kLsE....àèúv....
76E91060  80 17 E9 76 B0 D0 EF 76 08 00 0A 00 E4 81 E9 76  ..év°Ðïv....ä.év
76E91070  06 00 08 00 DC 81 E9 76 08 00 0A 00 18 80 E9 76  ....Ü.év......év
76E91080  08 00 0A 00 AC C2 E9 76 02 00 04 00 D8 81 E9 76  ....¬Âév....Ø.év
76E91090  18 00 1A 00 50 78 E9 76 00 00 00 00 00 00 00 00  ....Pxév........
```

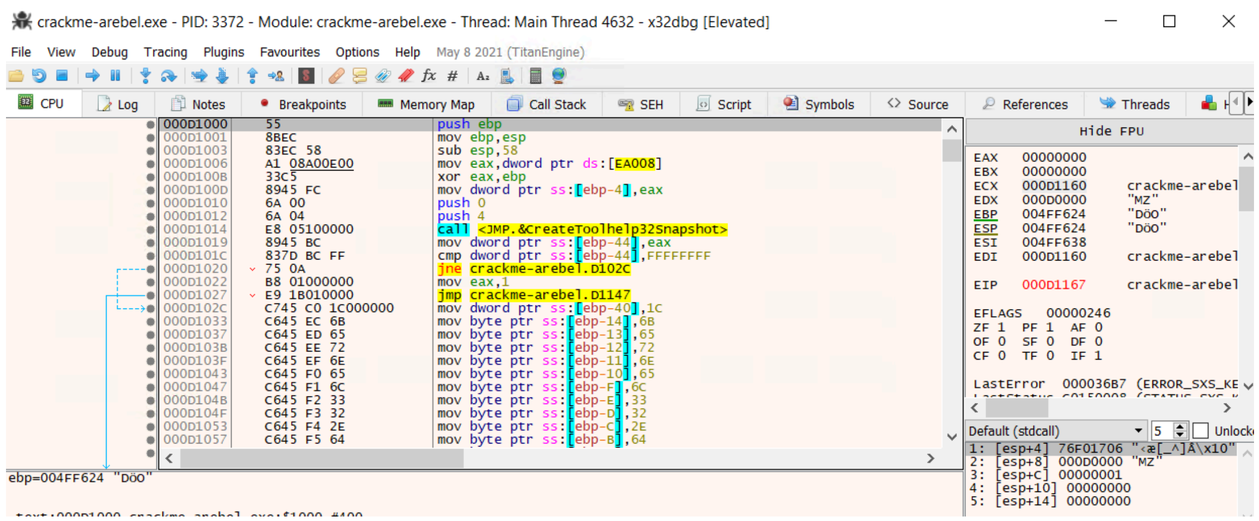
```
012FF5DC  012FF5FC
012FF5E0  76F01706  return to ntdll.76F01706 from ???
012FF5E4  000D0000  crackme-arebel.000D0000
012FF5E8  00000001
012FF5EC  00000000
012FF5F0  00000000
012FF5F4  000D1160  crackme-arebel.000D1160
012FF5F8  000D0000  crackme-arebel.000D0000
012FF5FC  012FF648
012FF600  76EC82AC  return to ntdll.76EC82AC from ntdll.76F0
012FF604  000D1160  crackme-arebel.000D1160
012FF608  000D0000  crackme-arebel.000D0000
```

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Paused INT3 breakpoint "TLS Callback 1" at crackme-arebel.000D1160 (000D1160)!

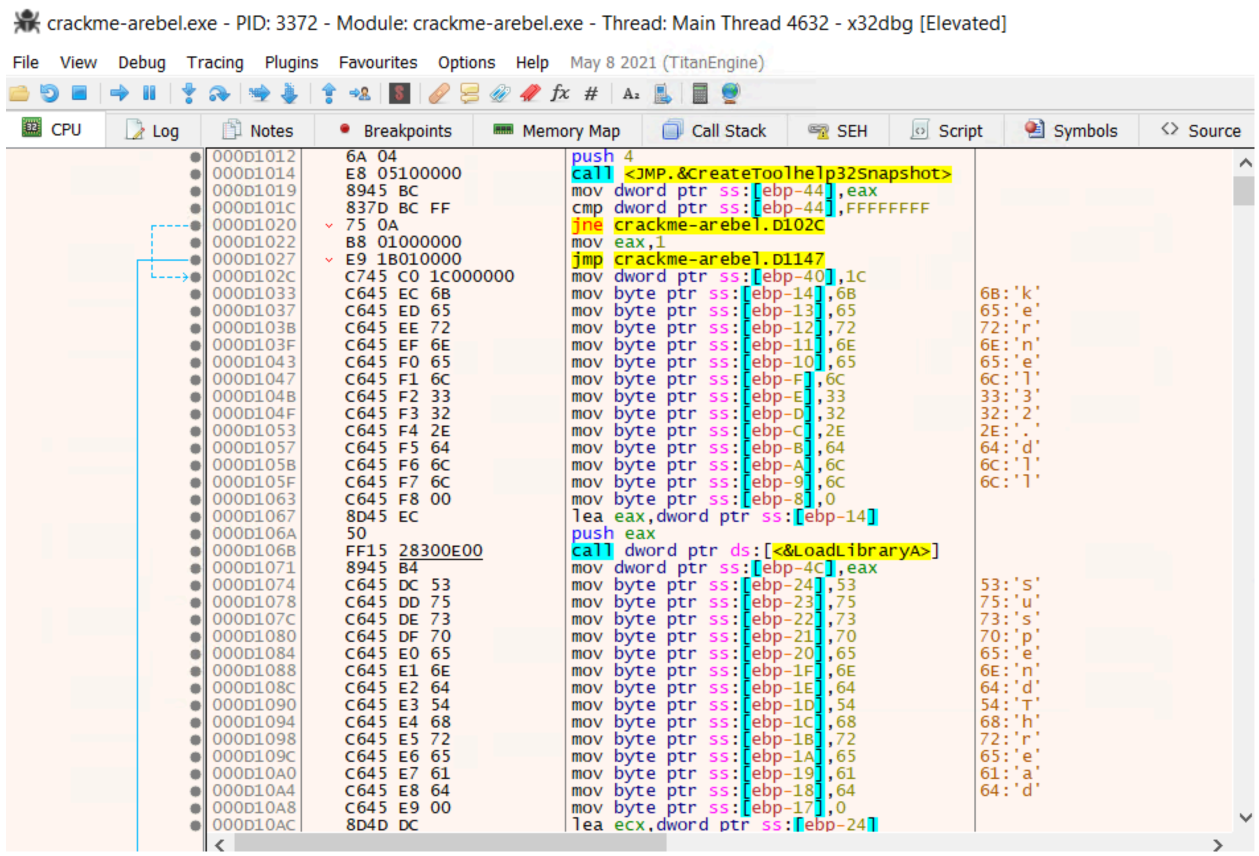
Time Wasted Debugging: 0:01:10:52

If we analyze both execution paths, we see if the jump is taken, it goes to address D116E, which pops EBP and returns. On the contrary, the current execution path, where the jump is not taken, takes us to address D1000. We can see what is on this address by double-clicking on the address. So let's go there. Alternatively, we can hover over the address to get a glimpse of the instructions on that address. The below screenshot shows the code when we follow the address. Here we see a few API calls like CreateToolhelp32Snapshot, LoadLibrary and GetProcAddress if we follow it down further. If we were sure that this function call was intended and we wanted to return after the function was executed, we could step over, which will bring us back after the function has been executed. However, the function seems too important to bypass straight away.



```
crackme-arebel.exe - PID: 3372 - Module: crackme-arebel.exe - Thread: Main Thread 4632 - x32dbg [Elevated]

000D1000  55                  push ebp
000D1001  8BEC                mov ebp,esp
000D1003  83EC 58             sub esp,58
000D1006  A1 08A00E00         mov eax,dword ptr ds:[EA008]
000D100B  33C5                xor eax,ebp
000D100D  8945 FC             mov dword ptr ss:[ebp-4],eax
000D1010  6A 00               push 0
000D1012  6A 04               push 4
000D1014  E8 05100000         call <JMP.&CreateToolhelp32Snapshot>
000D1019  8945 BC             mov dword ptr ss:[ebp-44],eax
000D101C  837D BC FF          cmp dword ptr ss:[ebp-44],FFFFFFFF
000D1020  75 0A               jne crackme-arebel.D102C
000D1022  B8 01000000         mov eax,1
000D1027  E9 1B010000         jmp crackme-arebel.D1147
000D102C  C745 C0 1C000000    mov dword ptr ss:[ebp-40],1C
000D1033  C645 EC 6B          mov byte ptr ss:[ebp-14],6B
000D1037  C645 ED 65          mov byte ptr ss:[ebp-13],65
000D103B  C645 EE 72          mov byte ptr ss:[ebp-12],72
000D103F  C645 EF 6E          mov byte ptr ss:[ebp-11],6E
000D1043  C645 F0 65          mov byte ptr ss:[ebp-10],65
000D1047  C645 F1 6C          mov byte ptr ss:[ebp-F],6C
000D104B  C645 F2 33          mov byte ptr ss:[ebp-E],33
000D104F  C645 F3 32          mov byte ptr ss:[ebp-D],32
000D1053  C645 F4 2E          mov byte ptr ss:[ebp-C],2E
000D1057  C645 F5 64          mov byte ptr ss:[ebp-B],64

ebp=004FF624 "DöO"

Hide FPU
EAX     00000000
EBX     00000000
ECX     000D1160     crackme-arebel
EDX     000D0000     "MZ"
EBP     004FF624     "DöO"
ESP     004FF624     "DöO"
ESI     004FF638
EDI     000D1160     crackme-arebel

EIP     000D1167     crackme-arebel

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  000036B7 (ERROR_SXS_KE

Default (stdcall)   5   Unlock
1: [esp+4] 76F01706 "‹æ[_^]Â\x10"
2: [esp+8] 000D0000 "MZ"
3: [esp+C] 00000001
4: [esp+10] 00000000
5: [esp+14] 00000000
```

These libraries can be used to evade detection or for legitimate purposes, but we don't know that for now. So we can move along this path, and if we see a red flag, we just restart it again using the restart execution option. Moving forward, we see that the Library being loaded is Suspend Thread.



```
crackme-arebel.exe - PID: 3372 - Module: crackme-arebel.exe - Thread: Main Thread 4632 - x32dbg [Elevated]

000D1012  6A 04               push 4
000D1014  E8 05100000         call <JMP.&CreateToolhelp32Snapshot>
000D1019  8945 BC             mov dword ptr ss:[ebp-44],eax
000D101C  837D BC FF          cmp dword ptr ss:[ebp-44],FFFFFFFF
000D1020  75 0A               jne crackme-arebel.D102C
000D1022  B8 01000000         mov eax,1
000D1027  E9 1B010000         jmp crackme-arebel.D1147
000D102C  C745 C0 1C000000    mov dword ptr ss:[ebp-40],1C
000D1033  C645 EC 6B          mov byte ptr ss:[ebp-14],6B      6B:'k'
000D1037  C645 ED 65          mov byte ptr ss:[ebp-13],65      65:'e'
000D103B  C645 EE 72          mov byte ptr ss:[ebp-12],72      72:'r'
000D103F  C645 EF 6E          mov byte ptr ss:[ebp-11],6E      6E:'n'
000D1043  C645 F0 65          mov byte ptr ss:[ebp-10],65      65:'e'
000D1047  C645 F1 6C          mov byte ptr ss:[ebp-F],6C       6C:'l'
000D104B  C645 F2 33          mov byte ptr ss:[ebp-E],33       33:'3'
000D104F  C645 F3 32          mov byte ptr ss:[ebp-D],32       32:'2'
000D1053  C645 F4 2E          mov byte ptr ss:[ebp-C],2E       2E:'.'
000D1057  C645 F5 64          mov byte ptr ss:[ebp-B],64       64:'d'
000D105B  C645 F6 6C          mov byte ptr ss:[ebp-A],6C       6C:'l'
000D105F  C645 F7 6C          mov byte ptr ss:[ebp-9],6C       6C:'l'
000D1063  C645 F8 00          mov byte ptr ss:[ebp-8],0
000D1067  8D45 EC             lea eax,dword ptr ss:[ebp-14]
000D106A  50                  push eax
000D106B  FF15 28300E00       call dword ptr ds:[<&LoadLibraryA>]
000D1071  8945 B4             mov dword ptr ss:[ebp-4C],eax
000D1074  C645 DC 53          mov byte ptr ss:[ebp-24],53      53:'S'
000D1078  C645 DD 75          mov byte ptr ss:[ebp-23],75      75:'u'
000D107C  C645 DE 73          mov byte ptr ss:[ebp-22],73      73:'s'
000D1080  C645 DF 70          mov byte ptr ss:[ebp-21],70      70:'p'
000D1084  C645 E0 65          mov byte ptr ss:[ebp-20],65      65:'e'
000D1088  C645 E1 6E          mov byte ptr ss:[ebp-1F],6E      6E:'n'
000D108C  C645 E2 64          mov byte ptr ss:[ebp-1E],64      64:'d'
000D1090  C645 E3 54          mov byte ptr ss:[ebp-1D],54      54:'T'
000D1094  C645 E4 68          mov byte ptr ss:[ebp-1C],68      68:'h'
000D1098  C645 E5 72          mov byte ptr ss:[ebp-1B],72      72:'r'
000D109C  C645 E6 65          mov byte ptr ss:[ebp-1A],65      65:'e'
000D10A0  C645 E7 61          mov byte ptr ss:[ebp-19],61      61:'a'
000D10A4  C645 E8 64          mov byte ptr ss:[ebp-18],64      64:'d'
000D10A8  C645 E9 00          mov byte ptr ss:[ebp-17],0
000D10AC  8D4D DC             lea ecx,dword ptr ss:[ebp-24]
```

The SuspendThread API is being loaded. This TLS callback will suspend the thread based on detecting a running process, such as a debugger (the CreateToolhelp32Snapshot API helps identify running processes). This is why the program will freeze if we proceed with the execution. We can verify this hypothesis by moving forward along this code path. On verification, we find that this is an evasion code path. Therefore, we would like to bypass it. For that, we would like to take the jump in the start of the TLS callback. Let's restart the execution and get to the TLS callback in the next task.

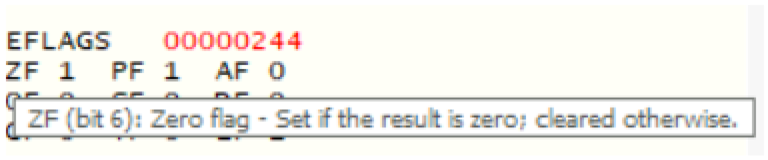
****

Answer the questions below:

The attached VM has a crackme in the directory Desktop > crackme-arebel. In that crackme, there is a TLS callback with a conditional jump. Is this conditional jump taken? Write Y for Yes or N for No

Answer: N

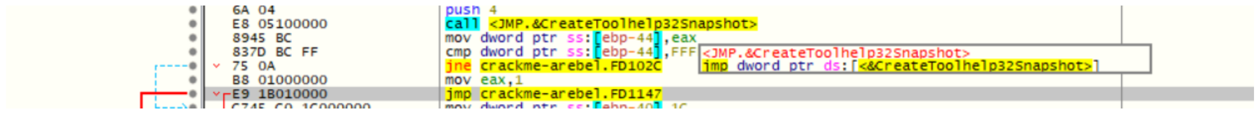
What is the value of the Zero Flag in the above-mentioned conditional jump?



```
EFLAGS    00000244
ZF 1   PF 1   AF 0
ZF (bit 6): Zero flag - Set if the result is zero; cleared otherwise.
```

Answer: 1

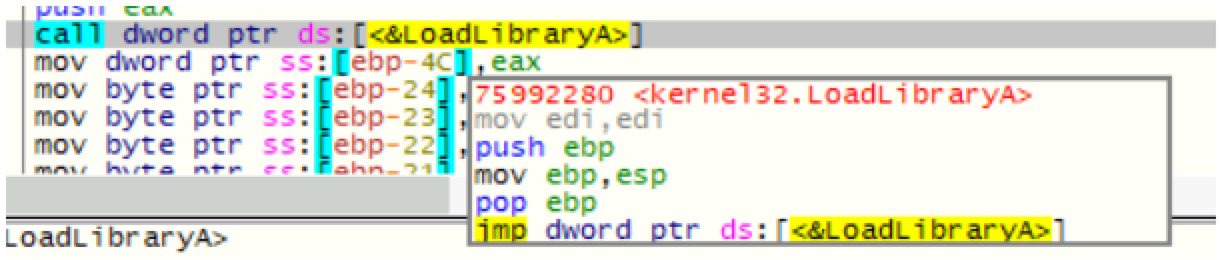
Which API call in the mentioned sample is used for enumerating running processes?



```
6A 04              push 4
E8 05100000        call <JMP.&CreateToolhelp32Snapshot>
8945 BC            mov dword ptr ss:[ebp-44],eax
837D BC FF         cmp dword ptr ss:[ebp-44],FFF    <JMP.&CreateToolhelp32Snapshot>
75 0A              jne crackme-arebel.FD102C         jmp dword ptr ds:[<&CreateToolhelp32Snapshot>]
B8 01000000        mov eax,1
E9 1B010000        jmp crackme-arebel.FD1147
```

Answer: CreateToolhelp32Snapshot

From which Windows DLL is the API SuspendThread being called?

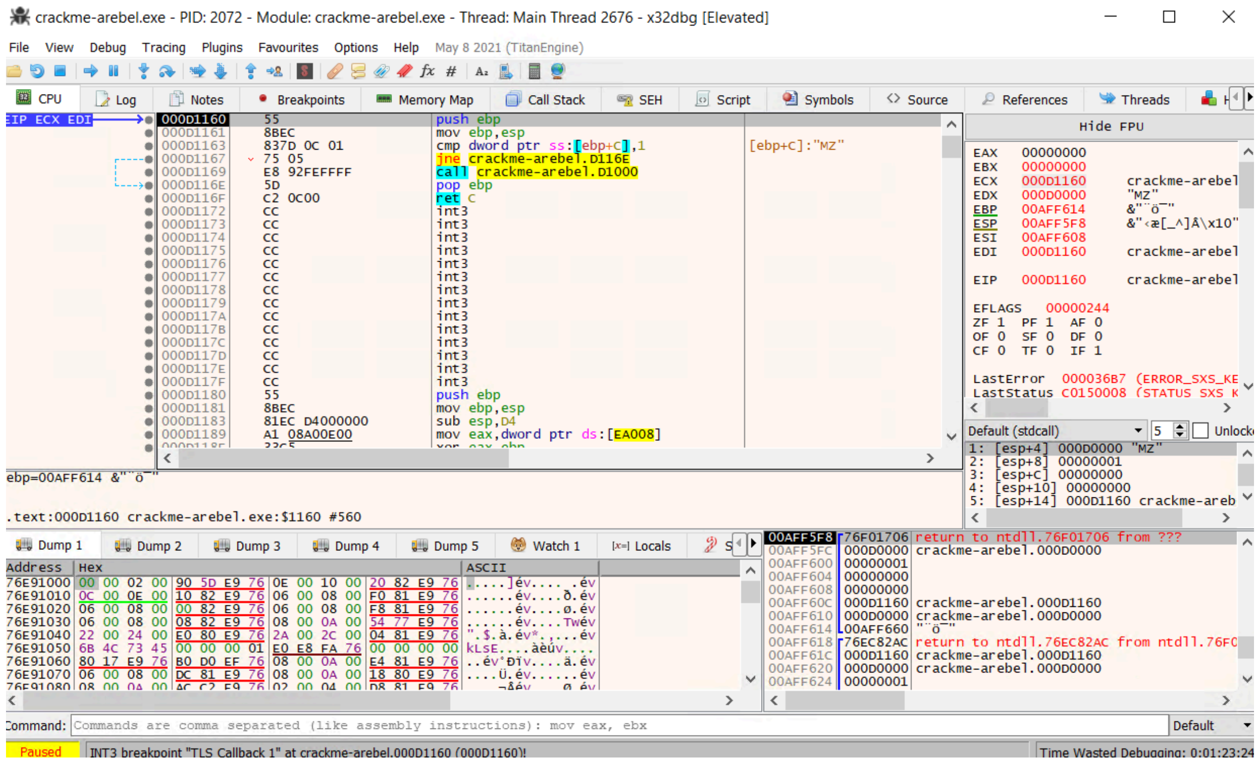


```
call dword ptr ds:[<&LoadLibraryA>]
mov dword ptr ss:[ebp-4C],eax
mov byte ptr ss:[ebp-24],   75992280 <kernel32.LoadLibraryA>
mov byte ptr ss:[ebp-23],   mov edi,edi
mov byte ptr ss:[ebp-22],   push ebp
                            mov ebp,esp
                            pop ebp
                            jmp dword ptr ds:[<&LoadLibraryA>]
LoadLibraryA>
```

Answer: kernel32.dll

Bypassing Unwanted Execution Path

In the below screenshot, we can see that the execution has been restarted and brought back to the branch of code where we identified a detection evasion, that is, at the TLS callback breakpoint.



crackme-arebel.exe - PID: 2072 - Module: crackme-arebel.exe - Thread: Main Thread 2676 - x32dbg [Elevated]

```
000D1160  55              push ebp
000D1161  8BEC            mov ebp,esp
000D1163  837D 0C 01      cmp dword ptr ss:[ebp+C],1        [ebp+C]:"MZ"
000D1167  75 05           jne crackme-arebel.D116E
000D1169  E8 92FEFFFF     call crackme-arebel.D1000
000D116E  5D              pop ebp
000D116F  C2 0C00         ret C
000D1172  CC              int3
000D1173  CC              int3
000D1174  CC              int3
000D1175  CC              int3
000D1176  CC              int3
000D1177  CC              int3
000D1178  CC              int3
000D1179  CC              int3
000D117A  CC              int3
000D117B  CC              int3
000D117C  CC              int3
000D117D  CC              int3
000D117E  CC              int3
000D117F  CC              int3
000D1180  55              push ebp
000D1181  8BEC            mov ebp,esp
000D1183  81EC D4000000   sub esp,D4
000D1189  A1 08A00E00     mov eax,dword ptr ds:[EA008]

ebp=00AFF614 &"¨ö¯"

.text:000D1160 crackme-arebel.exe:$1160 #560
```

```
EAX   00000000
EBX   00000000
ECX   000D1160     crackme-arebel
EDX   000D0000     "MZ"
EBP   00AFF614     &"¨ö¯"
ESP   00AFF5F8     &"‹æ[_^]Å\x10"
ESI   00AFF608
EDI   000D1160     crackme-arebel

EIP   000D1160     crackme-arebel

EFLAGS   00000244
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  000036B7 (ERROR_SXS_KE
LastStatus C0150008 (STATUS SXS K

Default (stdcall)  5
1: [esp+4] 000D0000 "MZ"
2: [esp+8] 00000001
3: [esp+C] 00000000
4: [esp+10] 00000000
5: [esp+14] 000D1160 crackme-areb
```

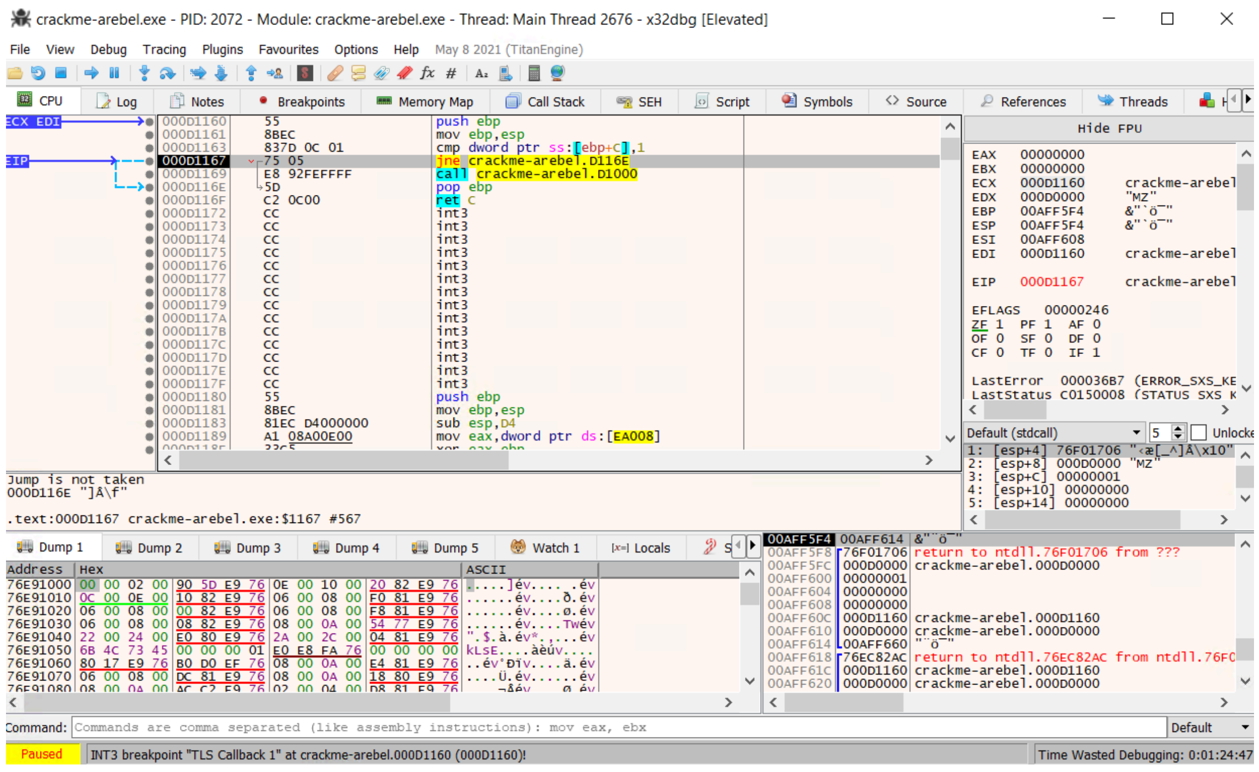
```
Address  Hex                                              ASCII
76E91000 00 00 02 00 90 5D E9 76 0E 00 10 00 20 82 E9 76 .....]év.... .év
76E91010 0C 00 0E 00 10 82 E9 76 06 00 08 00 F0 81 E9 76 ......év....ð.év
76E91020 06 00 08 00 00 82 E9 76 06 00 08 00 F8 81 E9 76 ......év....ø.év
76E91030 06 00 08 00 08 82 E9 76 08 00 0A 00 54 77 E9 76 ......év....Twév
76E91040 22 00 24 00 E0 80 E9 76 2A 00 2C 00 04 81 E9 76 ".$.à.év*.,...év
76E91050 6B 4C 73 45 00 00 00 01 E0 E8 FA 76 00 00 00 00 kLsE....àèúv....
76E91060 80 17 E9 76 B0 D0 EF 76 08 00 0A 00 E4 81 E9 76 ..év°Ðïv....ä.év
76E91070 06 00 08 00 DC 81 E9 76 08 00 0A 00 18 80 E9 76 ....Ü.év......év
```

```
00AFF5F8  76F01706  return to ntdll.76F01706 from ???
00AFF5FC  000D0000  crackme-arebel.000D0000
00AFF600  00000001
00AFF604  00000000
00AFF608  00000000
00AFF60C  000D1160  crackme-arebel.000D1160
00AFF610  000D0000  crackme-arebel.000D0000
00AFF614  00AFF660  "¨ö¯"
00AFF618  76EC82AC  return to ntdll.76EC82AC from ntdll.76F0
00AFF61C  000D1160  crackme-arebel.000D1160
00AFF620  000D0000  crackme-arebel.000D0000
00AFF624  00000001
```

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Paused INT3 breakpoint "TLS Callback 1" at crackme-arebel.000D1160 (000D1160)! Time Wasted Debugging: 0:01:23:24

Stepping a few instructions to reach the conditional jump, we get to the following point.



crackme-arebel.exe - PID: 2072 - Module: crackme-arebel.exe - Thread: Main Thread 2676 - x32dbg [Elevated]

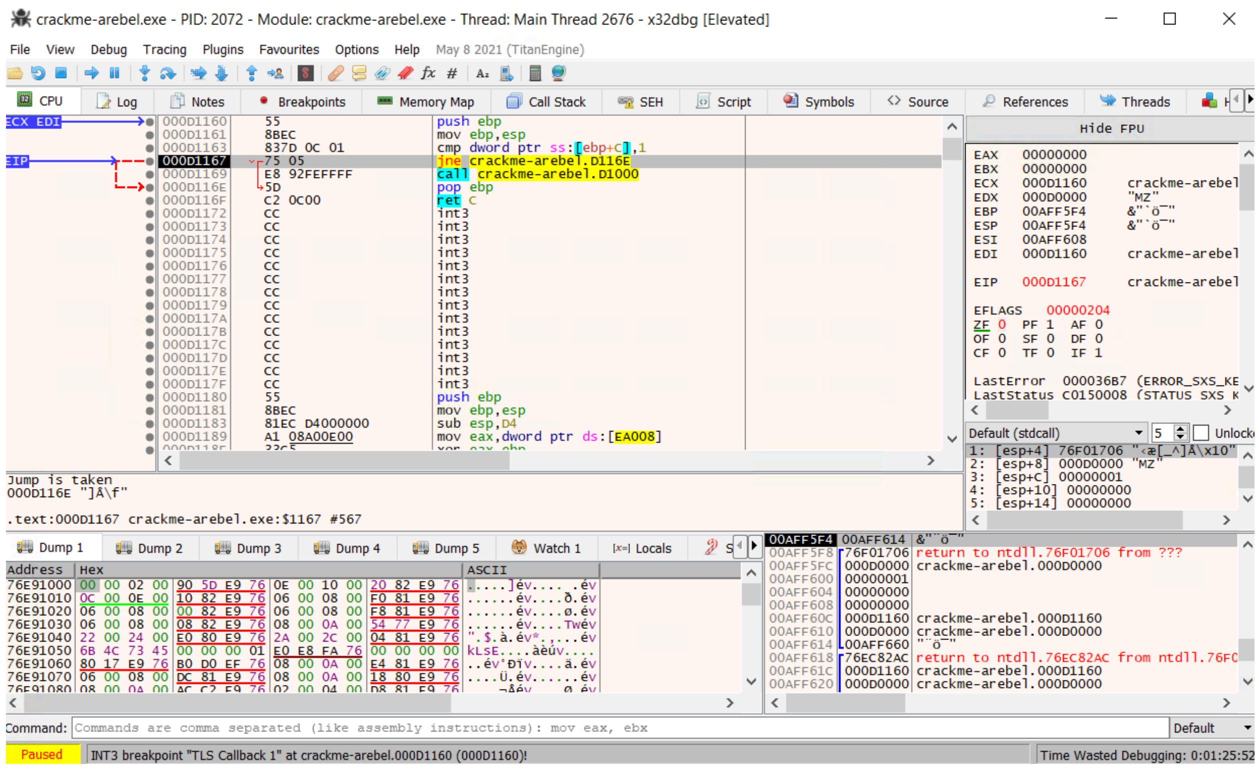
```
000D1160  55              push ebp
000D1161  8BEC            mov ebp,esp
000D1163  837D 0C 01      cmp dword ptr ss:[ebp+C],1
000D1167  75 05           jne crackme-arebel.D116E
000D1169  E8 92FEFFFF     call crackme-arebel.D1000
000D116E  5D              pop ebp
000D116F  C2 0C00         ret C
000D1172  CC              int3
...
000D117F  CC              int3
000D1180  55              push ebp
000D1181  8BEC            mov ebp,esp
000D1183  81EC D4000000   sub esp,D4
000D1189  A1 08A00E00     mov eax,dword ptr ds:[EA008]

Jump is not taken
000D116E "]Â\f"

.text:000D1167 crackme-arebel.exe:$1167 #567
```

```
EAX  00000000
EBX  00000000
ECX  000D1160   crackme-arebel
EDX  000D0000   "MZ"
EBP  00AFF5F4
ESP  00AFF5F4
ESI  00AFF608
EDI  000D1160   crackme-arebel

EIP  000D1167   crackme-arebel

EFLAGS  00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  000036B7 (ERROR_SXS_KE
LastStatus C0150008 (STATUS SXS K
```

```
00AFF5F8  76F01706  return to ntdll.76F01706 from ???
00AFF5FC  000D0000  crackme-arebel.000D0000
00AFF600  00000001
00AFF604  00000000
00AFF608  00000000
00AFF60C  000D1160  crackme-arebel.000D1160
00AFF610  000D0000  crackme-arebel.000D0000
00AFF614  00AFF660
00AFF618  76EC82AC  return to ntdll.76EC82AC from ntdll.76F0
00AFF61C  000D1160  crackme-arebel.000D1160
00AFF620  000D0000  crackme-arebel.000D0000
```

Paused   INT3 breakpoint "TLS Callback 1" at crackme-arebel.000D1160 (000D1160)!   Time Wasted Debugging: 0:01:24:47

We see that the jump is not being taken at this point. In the previous task, we identified that the ZF was the reason for the jump not being taken. By double-clicking ZF in the right pane, we can change the ZF to 0.



crackme-arebel.exe - PID: 2072 - Module: crackme-arebel.exe - Thread: Main Thread 2676 - x32dbg [Elevated]

```
000D1160  55              push ebp
000D1161  8BEC            mov ebp,esp
000D1163  837D 0C 01      cmp dword ptr ss:[ebp+C],1
000D1167  75 05           jne crackme-arebel.D116E
000D1169  E8 92FEFFFF     call crackme-arebel.D1000
000D116E  5D              pop ebp
000D116F  C2 0C00         ret C

Jump is taken
000D116E "]Â\f"

.text:000D1167 crackme-arebel.exe:$1167 #567
```

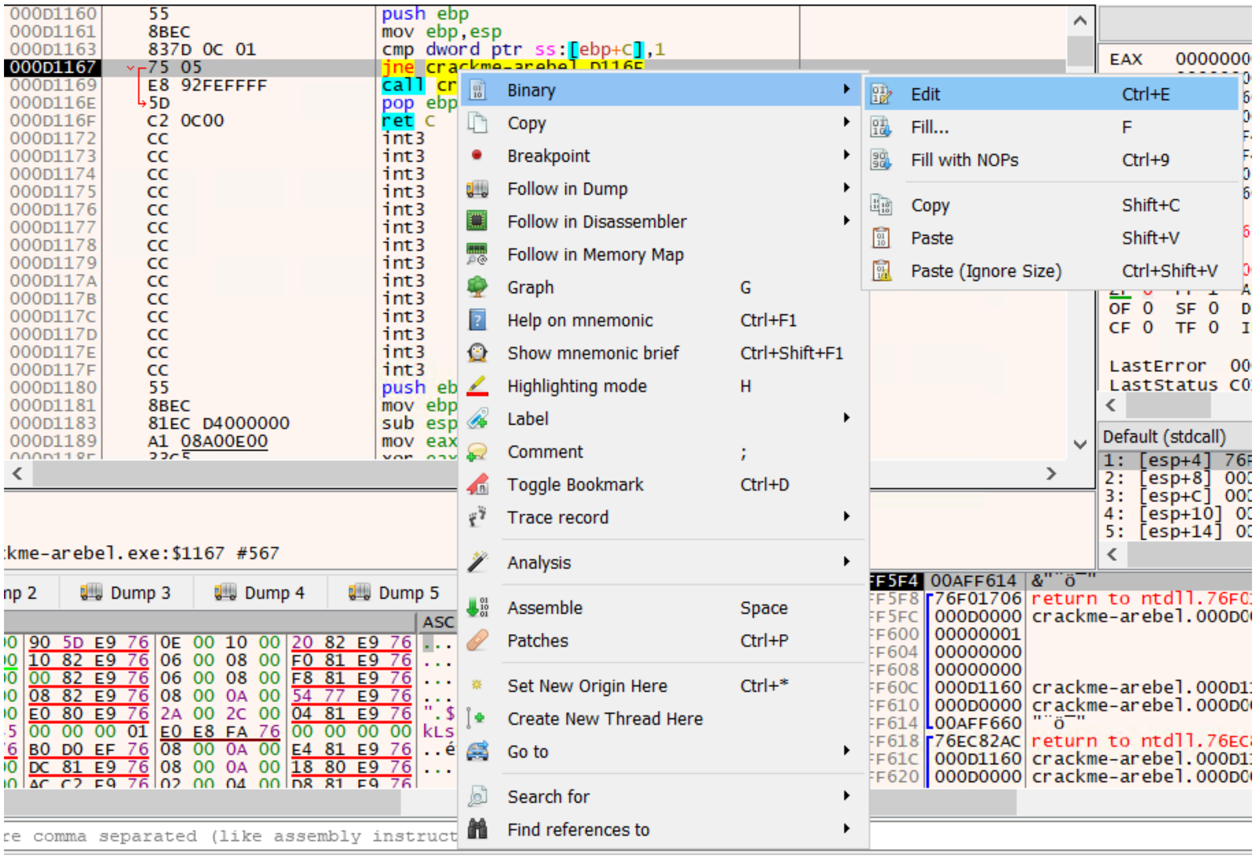
```
EIP  000D1167   crackme-arebel

EFLAGS  00000204
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1
```

Paused   INT3 breakpoint "TLS Callback 1" at crackme-arebel.000D1160 (000D1160)!   Time Wasted Debugging: 0:01:25:52

We can see here that the ZF is now 0, and the jump is taken. This way, we can bypass the unwanted execution path. If we step further or execute the program, it will take the jump and move to address D116E.

What we did here changed the execution path on runtime by manipulating the registers. However, if we execute the sample again, it will again go on to the evasion path unless we change the registers. If we wish to defang the binary, we will have to change it so that the evasion path is not an option if it is just run through double-clicking. This is called patching. This way, after being defanged, we can perform dynamic analysis to reach our desired conclusion. Let's see how we can do that.

We can right-click on the instruction we want to edit to find the options related to that instruction. Among these is the option to edit, as seen in the following screenshot.



```
000D1160    55              push ebp
000D1161    8BEC            mov ebp,esp
000D1163    837D 0C 01      cmp dword ptr ss:[ebp+C],1
000D1167    75 05           jne crackme-arebel.D116E
000D1169    E8 92FEFFFF     call cr
000D116E    5D              pop ebp
000D116F    C2 0C00         ret C
000D1172    CC              int3
000D1173    CC              int3
000D1174    CC              int3
000D1175    CC              int3
000D1176    CC              int3
000D1177    CC              int3
000D1178    CC              int3
000D1179    CC              int3
000D117A    CC              int3
000D117B    CC              int3
000D117C    CC              int3
000D117D    CC              int3
000D117E    CC              int3
000D117F    CC              int3
000D1180    55              push eb
000D1181    8BEC            mov ebp
000D1183    81EC D4000000   sub esp
000D1189    A1 08A00E00     mov eax
```

```
kme-arebel.exe:$1167 #567
```

```
EAX   0000000
OF 0   SF 0
CF 0   TF 0
LastError  00
LastStatus C0
```

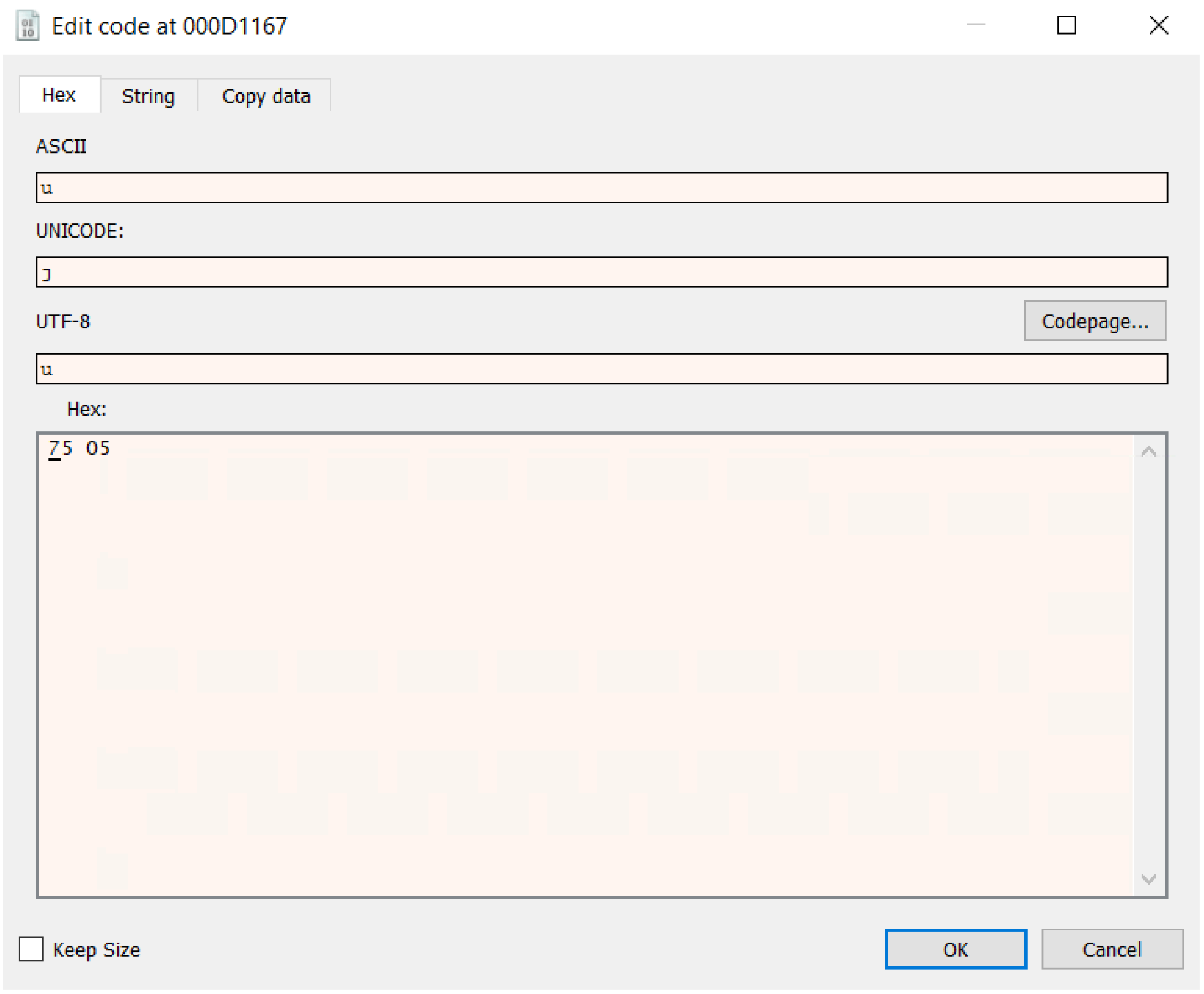
```
Default (stdcall)
1: [esp+4]  76F
2: [esp+8]  000
3: [esp+C]  000
4: [esp+10] 00
5: [esp+14] 00
```

```
90 5D E9 76  0E 00 10 00  20 82 E9 76  ...
10 82 E9 76  06 00 08 00  F0 81 E9 76  ...
00 82 E9 76  06 00 08 00  F8 81 E9 76  ...
08 82 E9 76  08 00 0A 00  54 77 E9 76  ...
E0 80 E9 76  2A 00 2C 00  04 81 E9 76  ".$
00 00 00 01  E0 E8 FA 76  00 00 00 00  kLs
B0 D0 EF 76  08 00 0A 00  E4 81 E9 76  ..é
DC 81 E9 76  08 00 0A 00  18 80 E9 76  ...
```

```
F5F4  00AFF614  &"¨ö¯"
F5F8  76F01706  return to ntdll.76F0
F5FC  000D0000  crackme-arebel.000D0
F600  00000001
F604  00000000
F608  00000000
F60C  000D1160  crackme-arebel.000D1
F610  000D0000  crackme-arebel.000D0
F614  00AFF660  "¨ö¯"
F618  76EC82AC  return to ntdll.76EC
F61C  000D1160  crackme-arebel.000D1
F620  000D0000  crackme-arebel.000D0
```

```
re comma separated (like assembly instruct
```

When we press edit, we see the following window.



ASCII: u

UNICODE: ȝ

UTF-8: u

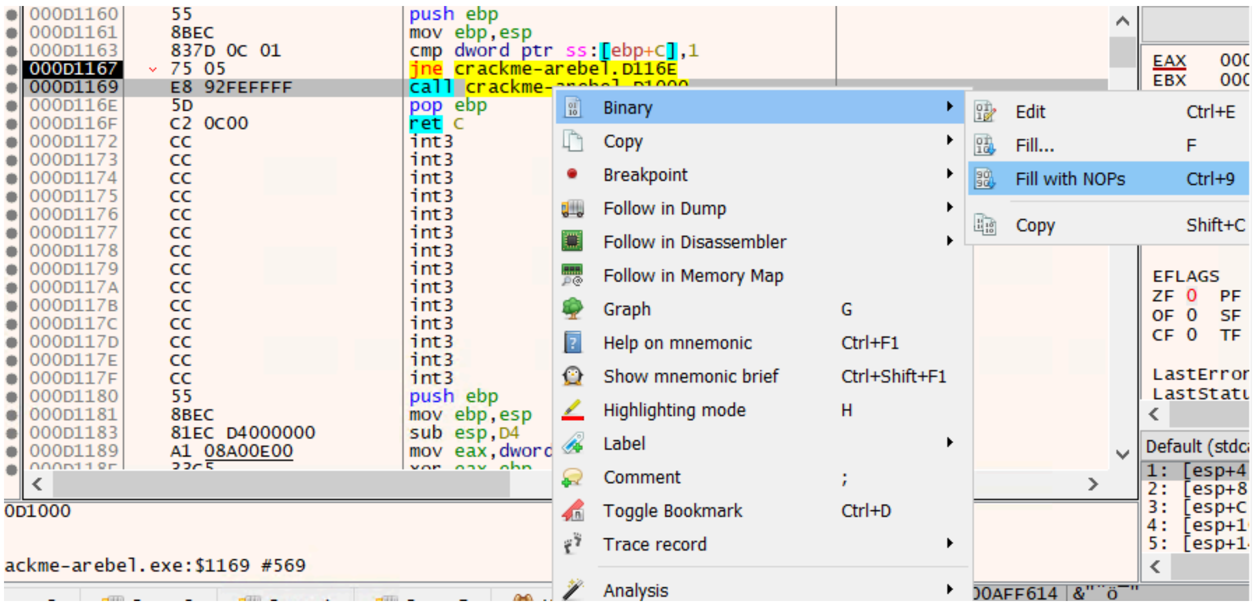
Hex:

```
75 05
```

In the lowermost pane, we see 75 05 in hex, the opcode for the conditional jump instruction. We can change this in the following ways.

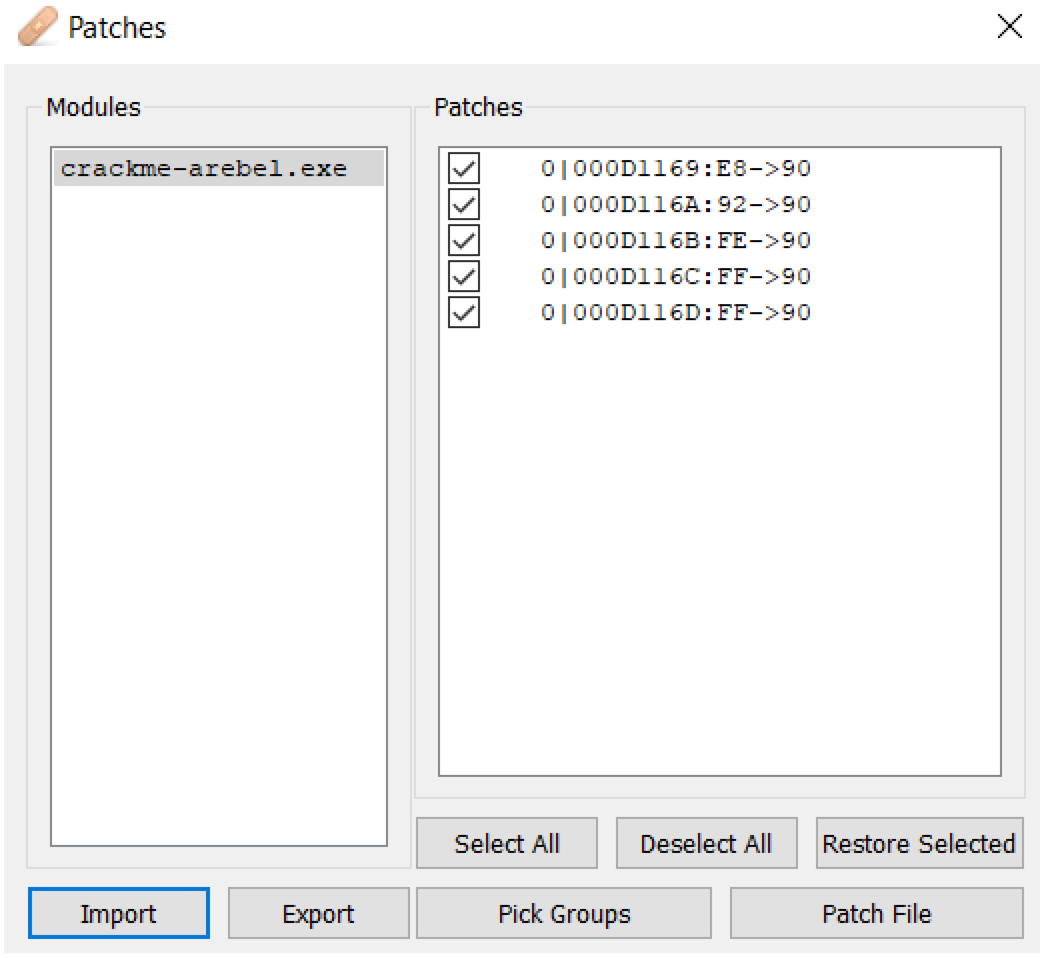
1. Change the jne to je. This way, the jump will be taken without us changing the ZF.
2. Change the conditional jump to an unconditional jump. This way, the jump will always be taken regardless of the condition above.

Among these two options, the second one is better as it gives us surety for the jump. However, there is one more way we can edit the binary for it to take our preferred execution path. That is, by filling the call instruction at address D1169 with NOPs. As discussed in a previous room, NOPs are instructions that don't do anything. The screenshot below shows that we can fill the instruction with NOP by right-clicking and going to Binary > Fill with NOPs.



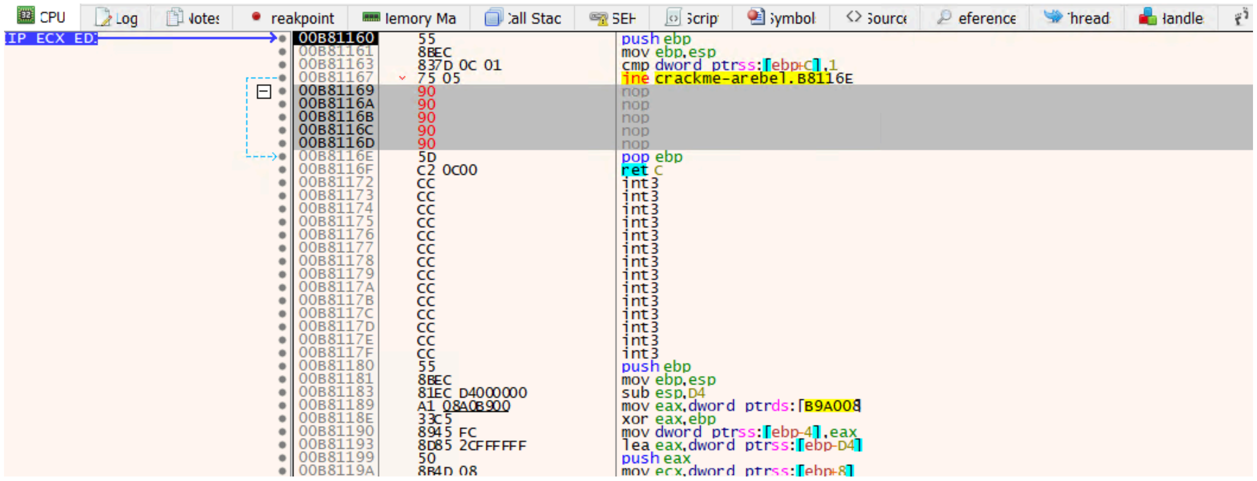
```
000D1160   55              push ebp
000D1161   8BEC            mov ebp,esp
000D1163   837D 0C 01      cmp dword ptr ss:[ebp+C],1
000D1167   75 05           jne crackme-arebel.D116E
000D1169   E8 92FEFFFF     call crackme-arebel.D1000
000D116E   5D              pop ebp
000D116F   C2 0C00         ret C
000D1172   CC              int3
000D1173   CC              int3
000D1174   CC              int3
000D1175   CC              int3
000D1176   CC              int3
000D1177   CC              int3
000D1178   CC              int3
000D1179   CC              int3
000D117A   CC              int3
000D117B   CC              int3
000D117C   CC              int3
000D117D   CC              int3
000D117E   CC              int3
000D117F   CC              int3
000D1180   55              push ebp
000D1181   8BEC            mov ebp,esp
000D1183   81EC D4000000   sub esp,D4
000D1189   A1 08A00E00     mov eax,dword

0D1000

ackme-arebel.exe:$1169 #569
```

Once done, we can go to File > Patch File to write this file to disk. We might see something like the below window once we navigate to that.



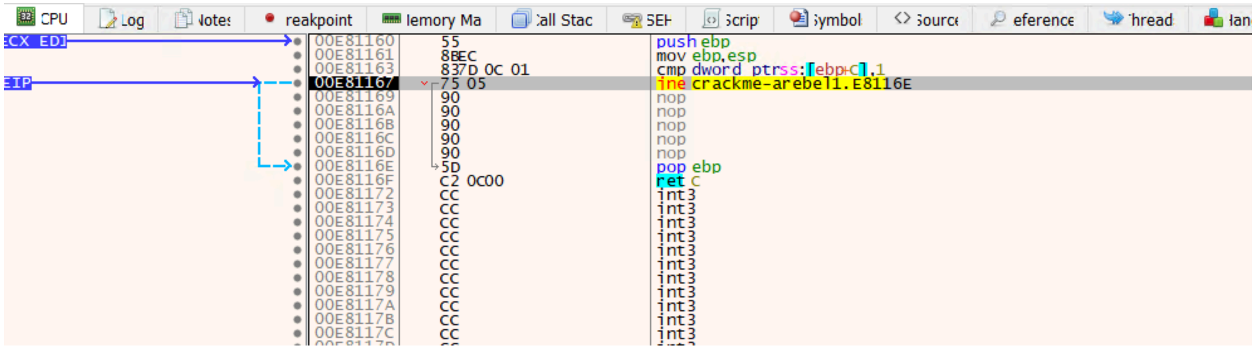
```
0|000D1169:E8->90
0|000D116A:92->90
0|000D116B:FE->90
0|000D116C:FF->90
0|000D116D:FF->90
```

By clicking Patch File on this window, the debugger will ask us where to save the patched file. This is how it will look once we have filled it with NOPs. Notice that a single instruction has been replaced with five NOP instructions. This is because the size of the single instruction was equal to five NOP instructions. This is more visible in the above screenshot, where each NOP instruction is mapped against equal sized Opcode.



```
00B81160  55          push ebp
00B81161  8BEC        mov ebp,esp
00B81163  837D 0C 01  cmp dword ptrss:[ebp+C],1
00B81167  75 05       jne crackme-arebel.B8116E
00B81169  90          nop
00B8116A  90          nop
00B8116B  90          nop
00B8116C  90          nop
00B8116D  90          nop
00B8116E  5D          pop ebp
00B8116F  C2 0C00     ret C
00B81172  CC          int3
00B81173  CC          int3
00B81174  CC          int3
00B81175  CC          int3
00B81176  CC          int3
00B81177  CC          int3
00B81178  CC          int3
00B81179  CC          int3
00B8117A  CC          int3
00B8117B  CC          int3
00B8117C  CC          int3
00B8117D  CC          int3
00B8117E  CC          int3
00B8117F  CC          int3
00B81180  55          push ebp
00B81181  8BEC        mov ebp,esp
00B81183  81EC D4000000  sub esp,D4
00B81189  A1 08A0B900 mov eax,dword ptrds:[B9A008]
00B8118E  33C5        xor eax,ebp
00B81190  8945 FC     mov dword ptrss:[ebp-4],eax
00B81193  8D85 2CFFFFFF  lea eax,dword ptrss:[ebp-D4]
00B81199  50          push eax
00B8119A  8B4D 08     mov ecx,dword ptrss:[ebp+8]
```

In the attached VM, we can see that the file crackme-arebel1.exe is patched. This is how the same file looks when patched and saved.



```
00E81160  55          push ebp
00E81161  8BEC        mov ebp,esp
00E81163  837D 0C 01  cmp dword ptrss:[ebp+C],1
00E81167  75 05       jne crackme-arebel1.E8116E
00E81169  90          nop
00E8116A  90          nop
00E8116B  90          nop
00E8116C  90          nop
00E8116D  90          nop
00E8116E  5D          pop ebp
00E8116F  C2 0C00     ret C
00E81172  CC          int3
00E81173  CC          int3
00E81174  CC          int3
00E81175  CC          int3
00E81176  CC          int3
00E81177  CC          int3
00E81178  CC          int3
00E81179  CC          int3
00E8117A  CC          int3
00E8117B  CC          int3
00E8117C  CC          int3
```

Hurray, we have used the debugger to patch a file evading debuggers!

****

Answer the questions below:

What is it called when a binary's assembly code is permanently altered to get the desired execution path?

Answer: Patching

Conclusion

This was it for this room. In this room, we learned the following:

- Common techniques to evade static and basic dynamic malware analysis.

- The use of debuggers for deeper analysis of malware.
- Using debuggers for changing the environment at runtime.
- Using debuggers to patch malware samples.

However, this is not it. Malware authors also use many techniques to evade dynamic analysis and debugging. Head over to the Anti-Reverse Engineering room to learn about more evasion techniques and how to counter them.