Module 2 - Introduction to Anti-Debugging Techniques

Module Overview

In this module, we'll be investigating a malware sample that uses a few new anti-sandbox techniques, as well as anti-debugging techniques to detect and evade debuggers. We'll explore how to identify these techniques in malware and how to circumvent these techniques.

A Note on Early Process Termination

Malware often terminates itself early when it detects that it's running in a virtual machine or a sandbox. This will often be in the form of **TerminateProcess** or **NtTerminateProcess** function calls. When reversing malware using a disassembler like IDA, it's helpful to closely inspect these function calls as they can reveal when the malware sample may be terminating itself.

When debugging malware, it's often helpful to put a breakpoint on **TerminateProcess** (*Hint: Set a breakpoint in x64dbg using the command **bp TerminateProcess***). This way, you'll catch the malware terminating itself early before the malware process can successfully terminate.

During this module, you should always have a breakpoint set on TerminateProcess, otherwise you'll have a difficult time debugging :)

Checking System Uptime / Time-bombing

Malware may attempt to evade sandboxes and other automated analysis tools by checking the current time, date, or system uptime. One common technique is checking system uptime via the **GetTickCount** API. **GetTickCount** returns the number of milliseconds that have elapsed since the system was started. Malware can use this to detect if it's running in a fresh environment like a sandbox, which usually has a low uptime.

Another (less common) technique is to include logic where the malware only executes if the date or time matches specific conditions - this is often referred to as **time-bombing**.

Let’s explore how this looks in practice:

1. Reboot your analysis VM (if you haven't already done so prior to the work on Module 1)
2. Run the malware sample in your VM.
3. Observe what happens. Does it execute as expected, or terminate early?

If the malware appears to be terminating prematurely, let's dig into why:

1. Open the malware in IDA Pro.
2. Look for time-related API calls. A good place to start is the **Imports** tab. Search for **GetTickCount**, or check the entry point and early code sections.
3. You may see a comparison involving the tick count. If it's above a certain threshold (e.g., 5 minutes), the malware proceeds. If not, it terminates:

```
mov     [rsp+78h+var_10], rax
lea     rcx, aCheckingTheSys ; "Checking the system's uptime...\n"
call    sub_140001010
call    cs:GetTickCount
cmp     eax, 300000
jnb     short loc_14000149E
```

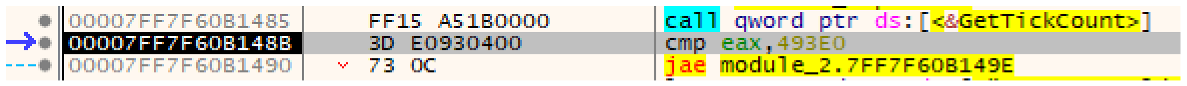
Shortly after this, you may also spot a call to **GetLocalTime**. **GetLocalTime** simply retrieves the current system time and date. Later, a date is compared (June 1, 2025) against the current system time. If the current system time is earlier than June 1, 2025, the sample will terminate.

```
loc_14000149E:          ; lpSystemTime
lea     rcx, [rsp+78h+SystemTime]
call    cs:GetLocalTime
mov     ecx, 7E9h
cmp     [rsp+78h+SystemTime.wYear], cx
jb      short loc_1400014C8
```

*Note: If the current date to your VM is June 1, 2025 or later and if the return value of **GetTickCount** is greater than 5 minutes, you'll bypass these checks. In this case, you won't need to circumvent these in a debugger, which is what we'll be doing now.*

If our malware sample terminated early because of these checks, we'll need to bypass these in a debugger. Let's do this now:

1. Launch the malware in x64dbg.
2. Click to **Run to user code** button or select **Debug -> Run to user code**. **Run to user code** is a very useful debugger option that skip's Windows process startup routines and "jumps" right to the malware's main logic.
3. Set a breakpoint on **GetTickCount** (**bp GetTickCount**) and press **F9** to run the malware.
4. When the breakpoint hits, select **Debug -> Run to use code**. This puts us back to the malware's code.
5. Examine the **cmp** instruction and the conditional jump afterward.



```
00007FF7F60B1485    FF15 A51B0000    call qword ptr ds:[<&GetTickCount>]
00007FF7F60B148B    3D E0930400      cmp eax,493E0
00007FF7F60B1490    73 0C            jae module_2.7FF7F60B149E
```

6. To bypass the **GetTickCount** check, modify the **cmp eax, 493E0** instruction to be **cmp eax, 0**. (Hint: To modify code, **Right-click** the instruction and select **Assemble**. Press **OK** once

done, and then **Cancel** to get exit the menu.)

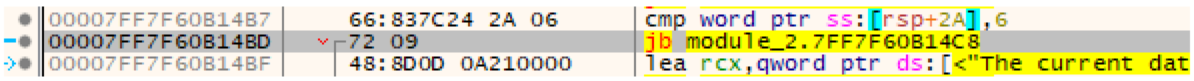
This will effectively bypass the **GetTickCount** check (as the malware is now comparing the current tick count to 0).

Now, let's circumvent the **GetLocalTime** check.

1. Set a breakpoint on **GetLocalTime** and press **F9** to run the malware.
2. When the breakpoint hits, select **Debug -> Run to use code**. This puts us back to the malware's code.
3. To bypass this check, we'll modify the final jump instruction after the date comparison. First, let's go back to IDA to inspect the code after the call to **GetLocalTime**. You should see something like the following:

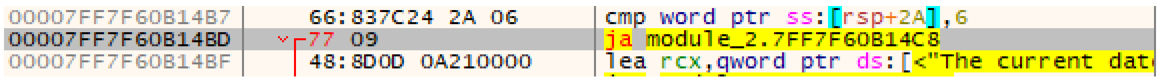
```
cmp     [rsp+78h+SystemTime.wMonth], 6
jb      short loc_1400014C8
```

4. If the **wMonth** value is below 6 (or, June) the control flow of the code jumps to another segment (in my case, **loc_1400014C8**). (Hint: The **jb** instruction means jump-if-below). To prevent the sample from terminating, we just need to modify this **jb** instruction in the debugger.



```
00007FF7F60B14B7 | 66:837C24 2A 06   | cmp word ptr ss:[rsp+2A],6
00007FF7F60B14BD | 72 09             | jb module_2.7FF7F60B14C8
00007FF7F60B14BF | 48:8D0D 0A210000  | lea rcx,qword ptr ds:[<"The current dat
```

5. In x64dbg, change the **jb** instruction to **ja** (jump-if-above). It should now look something like this:



```
00007FF7F60B14B7 | 66:837C24 2A 06   | cmp word ptr ss:[rsp+2A],6
00007FF7F60B14BD | 77 09             | ja module_2.7FF7F60B14C8
00007FF7F60B14BF | 48:8D0D 0A210000  | lea rcx,qword ptr ds:[<"The current dat
```

6. Before we go any further let's save the patched executable file. This will allow us to make permanent changes to the executable so that we don't have to re-patch during each run of the sample. To do this, navigate to **File -> Patch File** and then click **Patch File** and save it as something like "malware-patched.exe".
7. Close x64dbg and run the newly patched executable. You should have bypassed the **GetTickCount** and date check. What is happening now?

General Lab Tip: As its common for malware to check the uptime of the system as an anti-sandbox technique, it can be helpful to "fake" uptime. There are a few ways to do this, but perhaps the most straightforward is to simply start your VM, let it sit for 10 - 20 minutes, and take a snapshot of the VM's state. Next time you're about to analyze malware, your VM will already have been "up" for a good amount of time.

Basic Debugger Detection

Important Note: If you are using the ScyllaHide plugin for x64Dbg (or if you're using a very recent version of x64Dbg), you may bypass these next two techniques automatically. In this case, you won't need to manually bypass them. Still, it's very important to be aware of these next two techniques as they are very common in malware.

Two commonly used Windows API calls for anti-debugging are **IsDebuggerPresent** and **CheckRemoteDebuggerPresent**.

- **IsDebuggerPresent** checks the current process's **Process Environment Block (PEB)** to see if a debugger is attached. It returns `TRUE` if a debugger is present, `FALSE` otherwise.
- **CheckRemoteDebuggerPresent** performs a similar check but allows you to specify another process handle. It also examines the PEB of the target process to determine if a debugger is attached.

These functions access fields in the PEB such as **BeingDebugged** and **NtGlobalFlag**.

It's worth emphasizing: these functions are not inherently malicious. Many legitimate applications use them for various reason. Context matters! Analysts often mistakenly flag them as malicious without evaluating how they're being used.

Let's analyze how our sample uses them:

1. Open the malware in **IDA** or run it in **x64dbg**.
2. Look for calls to **IsDebuggerPresent** or **CheckRemoteDebuggerPresent**.
3. Try to identify the behavior that follows these calls - are they used to terminate execution or change control flow, for example?

Let's see if we can bypass this behavior to trick the malware into executing:

1. Launch the malware in x64dbg.
2. **Run to user code**.
3. Set breakpoints on **IsDebuggerPresent** and **CheckRemoteDebuggerPresent**.
4. Press F9 to run the sample.
5. When the breakpoint is hit, **Run to user code** again.
6. Modify the jump-if-equal instruction (**je**) after the call to **IsDebuggerPresent**. Change the first part of the instruction from **je** to **jmp**. This will make this an unconditional jump so that the malware always "jumps" past the termination code.
7. Do the same for the **je** instruction after **CheckRemoteDebuggerPresent**. This will bypass the **CheckRemoteDebuggerPresent** check.
8. As in the previous section, save the patched executable (**File -> Patch File**).

9. Close the debugger.

*Note: It's worth noting that there are lots of ways to bypass these sorts of techniques. Modifying the code is a more "permanent" method (especially if you are patching and saving the newly patched executable as we have been doing in this workshop. Alternatively, for example, you can also modify the return value in the CPU registers (such as RAX) after a function call like **IsDebuggerPresent**.)

Tip: As mentioned earlier, there is a plugin for x64Dbg called ScyllaHide that automates the circumvention of these debugger-detection techniques, as well as many other techniques. It's a very useful tool, but can occasionally cause unintended consequences such as crashing of the program, so use with care. (https://github.com/x64dbg/ScyllaHide)

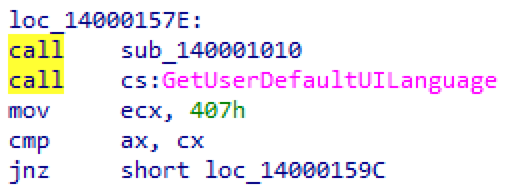
Checking System Language Settings

Malware may also check the system's language or locale settings for several reasons:

1. **Targeted Attacks:** If a threat actor wants to limit attacks to users in specific regions (e.g., targeting French speakers), the malware might only execute if the system language matches.
2. **Avoiding Certain Countries:** Many malware samples refuse to run on systems in countries like Russia or others in the Commonwealth of Independent States (CIS). This could be a tactic to avoid legal trouble or local enforcement.
3. **Sandbox Evasion:** Many sandbox environments use default English/US language settings. Malware that checks for non-default languages might evade sandbox detection simply because it doesn't meet the trigger conditions.

Let's look at language detection in our sample:

1. Open the malware in IDA or run it in x64dbg.
2. Look for calls to language-related APIs, such as **GetUserDefaultUILanguage**.
3. You should locate a call to **GetUserDefaultUILanguage**:

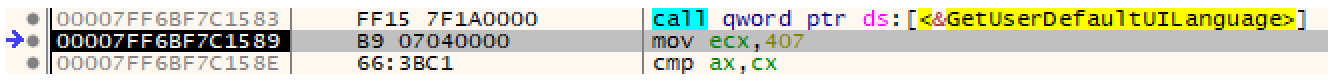


```
loc_14000157E:
call    sub_140001010
call    cs:GetUserDefaultUILanguage
mov     ecx, 407h
cmp     ax, cx
jnz     short loc_14000159C
```

Let's see if we can bypass these checks:

4. Launch the malware in x64dbg.
5. **Run to user code**.

6. Set a breakpoint on **GetUserDefaultUILanguage**.
7. Once the breakpoint hits, **Run to user code** again.
8. What value do you see in the RAX register? For example, yours may be 0x809, which is the language code for "English (United Kingdom)".
9. What language/country is this RAX register value being compared to? (Hint: 0x407):



```
00007FF6BF7C1583    FF15 7F1A0000    call qword ptr ds:[<&GetUserDefaultUILanguage>]
00007FF6BF7C1589    B9 07040000      mov ecx,407
00007FF6BF7C158E    66:3BC1          cmp ax,cx
```

10. To bypass this language check, let's change the instruction **mov ecx, 407** to **mov ecx, <your_language_code>**. In my case, this will be: **mov ecx, 809**.
11. Patch the file (**File -> Patch File**) and save it, close the debugger, and then execute the malware again. Does the malware execute any further?

Time-Based Detection

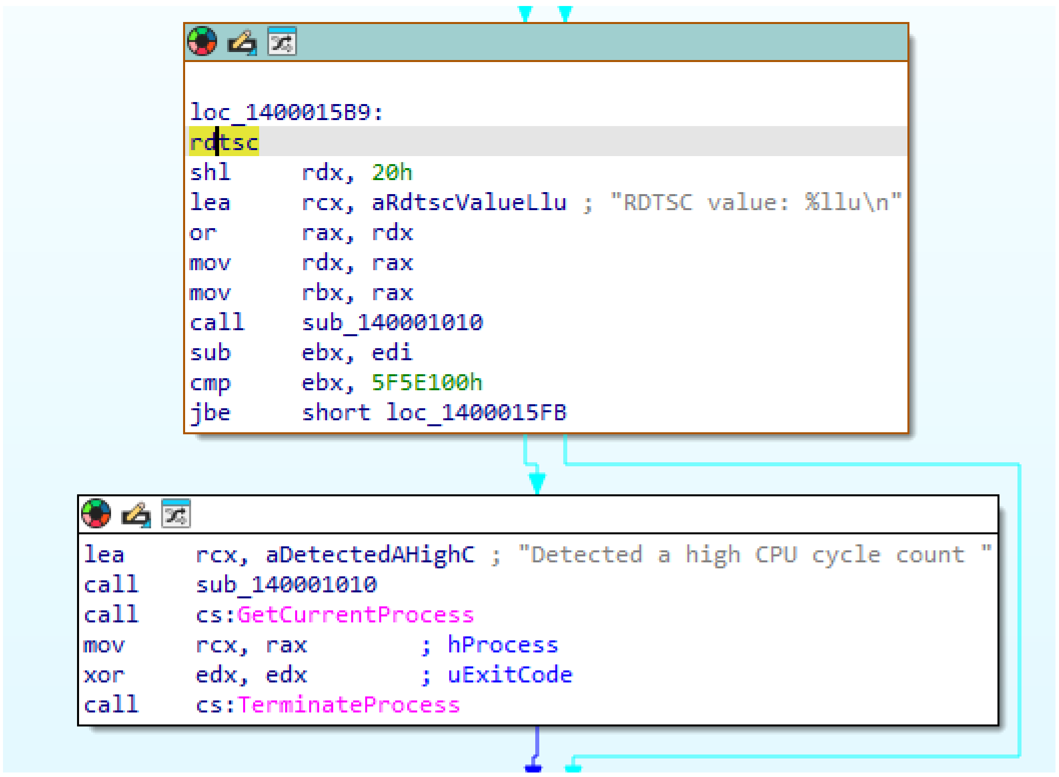
Malware often uses time-based techniques to detect analysis tools, sandboxes, and debuggers. "Time-based" in this context means relying on timing anomalies, usually involving CPU counters, to identify non-native execution environments. One common example you may have already seen is the use of **GetTickCount**. Another popular mechanism is the **Read Time-Stamp Counter** instruction, or **RDTSC**.

RDTSC is a CPU instruction that can be executed directly in assembly. It reads the current value of the processor's time-stamp counter which is basically a high-resolution tick counter that increments with each CPU cycle. Malware can call **RDTSC**, do some operations or call some APIs, then call **RDTSC** again to measure how many CPU cycles elapsed. If the delta is unusually high, the malware may conclude that it's running inside a VM (which often has slower, less consistent timing), or that it's being debugged (since breakpoints and stepping through code introduce delays). Let's dig into this behavior.

1. You may have already noticed a console message such as "I detected a high CPU cycle count!". This is because the language detection mechanism and debugger detection functions we discussed previously are "sandwiched" between two RDTSC instructions and a tick count comparison operation. Stepping through and debugging the code around the **GetUserDefaultUILanguage** and **IsDebuggerPresent** results in an unusually high tick count, which the malware doesn't like.
2. Use IDA or x64dbg to analyze the code prior to **IsDebuggerPresent** and after the **GetUserDefaultUILanguage** functionality. You should spot some code that looks like this:

```
loc_1400014EF:
rdtsc
shl     rdx, 20h
lea     rcx, aRdtscValueLlu ; "RDTSC value: %llu\n"
or      rax, rdx
mov     rdx, rax
mov     rdi, rax
call    sub_140001010
mov     [rsp+78h+pbDebuggerPresent], 0
call    cs:IsDebuggerPresent
test    eax, eax
```

And this:



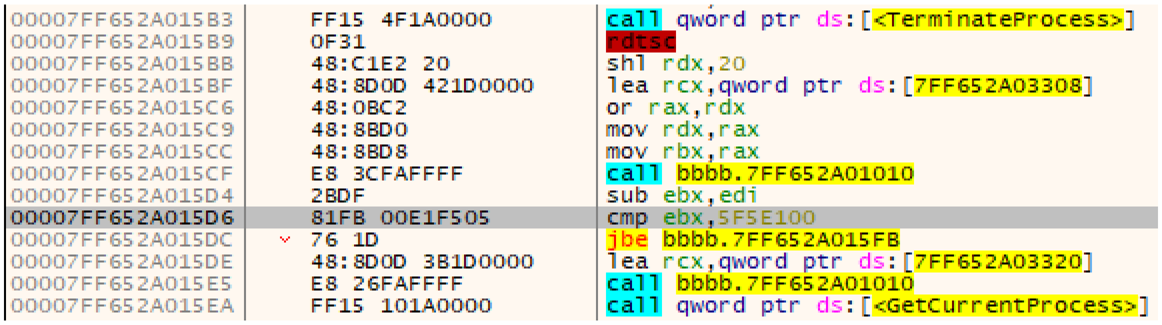
```
loc_1400015B9:
rdtsc
shl     rdx, 20h
lea     rcx, aRdtscValueLlu ; "RDTSC value: %llu\n"
or      rax, rdx
mov     rdx, rax
mov     rbx, rax
call    sub_140001010
sub     ebx, edi
cmp     ebx, 5F5E100h
jbe     short loc_1400015FB
```

```
lea     rcx, aDetectedAHighC ; "Detected a high CPU cycle count "
call    sub_140001010
call    cs:GetCurrentProcess
mov     rcx, rax        ; hProcess
xor     edx, edx        ; uExitCode
call    cs:TerminateProcess
```

This malware is using **RDTSC** in a clever way; It's measuring CPU timing to test if some pesky malware analyst is stepping through its code. Let’s talk about how to bypass this check. There are several ways to go about it, but for now, we’ll go with a relatively simple and practical method: patching the timing logic.

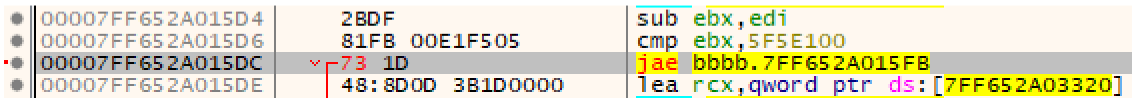
Once you’ve identified the two RDTSC instructions and the logic that compares their results, you can patch the conditional jump that triggers the "debugger detected" behavior.

1. Load the malware in x64dbg.
2. Set breakpoints on both **RDTSC** instructions so you can watch the flow. You can set the breakpoints by **right-clicking** on the target instruction and selecting **Breakpoint -> Toggle**.
3. Step through until you hit the comparison that checks the delta of the two RDTSC instructions. It should look something like this code (may not be exactly the same in your sample!)



```
00007FF652A015B3    FF15 4F1A0000        call qword ptr ds:[<TerminateProcess>]
00007FF652A015B9    0F31                 rdtsc
00007FF652A015BB    48:C1E2 20           shl rdx,20
00007FF652A015BF    48:8D0D 421D0000     lea rcx,qword ptr ds:[7FF652A03308]
00007FF652A015C6    48:0BC2              or rax,rdx
00007FF652A015C9    48:8BD0              mov rdx,rax
00007FF652A015CC    48:8BD8              mov rbx,rax
00007FF652A015CF    E8 3CFAFFFF          call bbbb.7FF652A01010
00007FF652A015D4    2BDF                 sub ebx,edi
00007FF652A015D6    81FB 00E1F505        cmp ebx,5F5E100
00007FF652A015DC    76 1D                jbe bbbb.7FF652A015FB
00007FF652A015DE    48:8D0D 3B1D0000     lea rcx,qword ptr ds:[7FF652A03320]
00007FF652A015E5    E8 26FAFFFF          call bbbb.7FF652A01010
00007FF652A015EA    FF15 101A0000        call qword ptr ds:[<GetCurrentProcess>]
```

4. To circumvent this check we can either modify the **cmp** operation or simply patch the **jbe** instruction so that the code jumps over the **TerminateProcess** function. Let's patch the jump instruction, and change it to **jae** (jump-if-above-or-equal):



```
00007FF652A015D4    2BDF                 sub ebx,edi
00007FF652A015D6    81FB 00E1F505        cmp ebx,5F5E100
00007FF652A015DC    73 1D                jae bbbb.7FF652A015FB
00007FF652A015DE    48:8D0D 3B1D0000     lea rcx,qword ptr ds:[7FF652A03320]
```

5. Save the patched executable file (similar to how we did this in the previous sections).
6. Execute the malware again. What happens now?

Delaying Execution

It's quite common for malware to delay its execution in order to evade sandboxes and debuggers. There are several ways to introduce these delays, but two of the most commonly used methods are the **Sleep** and **SleepEx** functions, which essentially "pause" the malware for a specified period. Another function that achieves the same goal is **NtDelayExecution**, which is actually called by **Sleep** and **SleepEx**. There are other techniques that can cause delays, such as inserting "stalling" code to waste CPU cycles, but we won't cover those here in this workshop.

As we've mentioned, delaying execution serves various purposes, but here are the key ones to keep in mind:

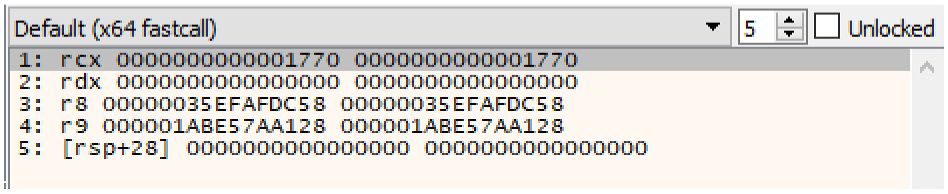
- **Evading sandboxes**: Sandboxes typically run malware for a limited amount of time (anywhere from 1 to 3 minutes). By introducing a delay of 5 minutes, the malware may ensure that the sandbox doesn't "observe" any malicious behavior.
- **Anti-debugging**: When malware calls functions like **Sleep** or **NtDelayExecution**, it continues executing in the debugger. This can cause the analyst to lose control of the sample, making it harder to debug and analyze.

Now, let's walk through an example of how malware might delay execution:

1. Execute the malware sample in **x64dbg**. At this point in the workshop, you should see a message in the console like: "Delaying execution to evade your sandbox".

It looks like this malware is trying to delay execution in order to avoid sandboxes and annoy malware analysts.

2. Set a breakpoint on the **SleepEx** function by using the command **bp SleepEx**, then run the malware again.
3. When the malware hits the breakpoint, take a look at the parameters passed to **SleepEx**. You can find this in the small window under the CPU register window:



```
1: rcx 0000000000001770 0000000000001770
2: rdx 0000000000000000 0000000000000000
3: r8 00000035EFAFDC58 00000035EFAFDC58
4: r9 000001ABE57AA128 000001ABE57AA128
5: [rsp+28] 0000000000000000 0000000000000000
```

How long is the malware planning to sleep for? (Hint: You should see 1770 in hex. What is this value in decimal?). If you wish, you can investigate this **SleepEx** call in IDA to see how the malware passes the time parameter to the function.

4. Allow the malware to continue execution (use the **Run** command). What happens next?
5. What occurs when you keep running the malware? Does the malware detect our breakpoint on **SleepEx**?

It seems like we've encountered another anti-debugging technique: breakpoint detection! This malware appears to be checking for breakpoints on the **SleepEx** function to disrupt our debugging efforts.

Hook- and Breakpoint Detection

Malware can use hook and breakpoint detection techniques to, as you might expect, identify the presence of hooks and breakpoints. By now, you're probably familiar with what a breakpoint is. A hook, however, is slightly different. It's similar to a breakpoint but is primarily used by sandboxes, analysis tools, and security solutions like EDR (Endpoint Detection and Response) or anti-malware software to monitor the function calls the malware makes. Just as reverse engineers place breakpoints on the function calls in malware to inspect behavior, sandboxes and EDR systems often inject hooks at the start of functions they wish to monitor.

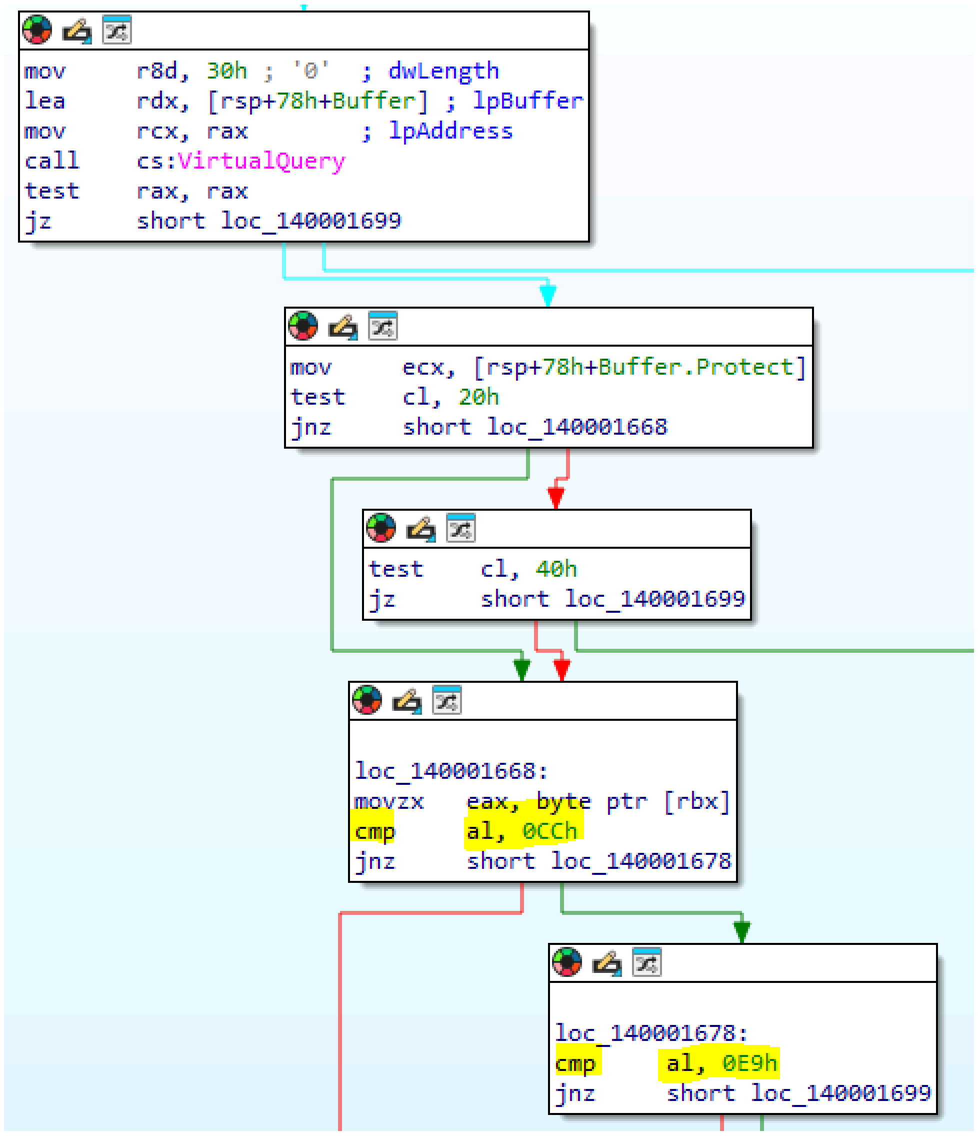
For example, sandboxes or EDR systems might want to monitor what files the malware is accessing or creating on disk. To do this, they might hook the **NtCreateFile** function, inserting code at the beginning of the function to allow the sandbox or EDR to log or monitor that specific function call.

When a breakpoint is set on a function in a debugger, a special assembly instruction, typically **INT3** (or **0xCC** in hexadecimal), is placed at the start of the target function. This instructs the debugger to "pause" execution at that point. Similarly, when creating a hook, sandboxes and

EDR systems often use a **JMP** instruction (**0xE9** in hexadecimal) at the start of the function, which redirects execution to the monitoring code. Because both breakpoints and hooks modify the target function's code, malware can detect these changes by scanning its loaded functions for **JMP** or **INT3** instructions.

Let's take a look at how this works in practice and how we can circumvent this technique as well the **SleepEx** execution delay technique we previously discussed:

1. Execute the malware sample in **x64dbg**. Make sure you still have a breakpoint on **SleepEx**.
2. At this point in the workshop, you should see a message in the console like: "Delaying execution to evade your sandbox" followed by a message telling you the malware detected a breakpoint.
3. The sample should terminate. Can you guess why?
4. Inspect the code in your debugger or in IDA Pro to identify why the sample is terminating. *(Hint: In IDA Pro, look for the **VirtualQuery** function. This function is used to query a memory page. In this case, the malware is calling this function to check for breakpoints on the **SleepEx** function memory)*.
5. You should see a comparison operation looking for 0xCC and for 0xE9. This is the malware looking for breakpoint and hook-related instructions!



```
mov     r8d, 30h ; '0'   ; dwLength
lea     rdx, [rsp+78h+Buffer] ; lpBuffer
mov     rcx, rax         ; lpAddress
call    cs:VirtualQuery
test    rax, rax
jz      short loc_140001699
```

```
mov     ecx, [rsp+78h+Buffer.Protect]
test    cl, 20h
jnz     short loc_140001668
```

```
test    cl, 40h
jz      short loc_140001699
```

```
loc_140001668:
movzx   eax, byte ptr [rbx]
cmp     al, 0CCh
jnz     short loc_140001678
```

```
loc_140001678:
cmp     al, 0E9h
jnz     short loc_140001699
```

Now we need to get around two things: the looping **SleepEx** function (used to stall analysis) *and* the malware's breakpoint detection. Let's tackle the breakpoint detection first.

There are a few ways to bypass this, one effective method is to use **hardware breakpoints**. These are similar to software breakpoints, but instead of modifying the code (like inserting an `INT3` instruction), hardware breakpoints use CPU debug registers to monitor execution - so there's no visible change to the code that malware can detect (bear in mind, though, that some malware *can* detect hardware breakpoints by inspecting CPU debug registers, but this is more uncommon.)

I won't cover this in depth in this lab guide, but to set a hardware breakpoint, simply **right-click** on an instruction to break on, and select **Breakpoint -> Set Hardware on execution**.

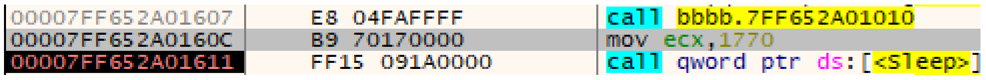
To bypass this sleep routine, let's instead set a breakpoint before the call to **SleepEx**.

1. In IDA, find the call to **SleepEx**.

2. In x64dbg, set a breakpoint on either a prior Windows function call, such as **GetUserDefaultUILanguage**, or a specific address (such as the **mov ecx, 1770** instruction prior to the **SleepEx** call). You can use IDA to find a suitable function or address.
3. Remove any software breakpoints you've previously set on the **SleepEx** function. (Hint: On the **SleepEx** breakpoint you have set, press **F2** to remove it.)

This will allow us to bypass the breakpoint detection since we won't be breaking on the actual function call to **SleepEx**. So what about the execution delay? To get around that:

1. After setting your breakpoint from the previous step, re-run the malware. When the malware hits your breakpoint you just set up, locate the sleep duration parameter (it will be a **mov ecx, 1770** instruction).



```
00007FF652A01607 | E8 04FAFFFF   | call bbbb.7FF652A01010
00007FF652A0160C | B9 70170000   | mov ecx,1770
00007FF652A01611 | FF15 091A0000 | call qword ptr ds:[<Sleep>]
```

2. Modify the instruction to be: **mov ecx, 0x0**
3. Let the malware **Run** and see what happens. You may need to remove your breakpoints in order for the malware to continue execution. We don't need this breakpoint anymore since we already modified the parameter to the **SleepEx** call.

With the sleep duration set to 0, the malware zips through all of its Sleep calls almost instantly - no more long delays. Eventually, the malware finishes executing completely and you should see a flag.

Nice work! You got the malware to fully execute. You now know the fundamentals to debug and dynamically patch problematic malware to get it to execute in a VM or sandbox!

You can now officially call yourself a **reverse engineer** 😎