

# Anti-Reverse Engineering

## Introduction

Malware authors are constantly looking for ways to evade detection and analysis to maintain the effectiveness of their malware. At the same time, security professionals are working to develop new methods and tools for detecting and mitigating the threat posed by malware. This ongoing "arms race" can lead to the development of increasingly sophisticated and effective malware defences as both sides seek to stay ahead of their counterparts.

Reverse engineering is the process of studying a technology product, software, or hardware to learn how it works and extract its functionality or design information. In cybersecurity, reverse engineering is used to understand how malware works, extract indicators of compromise (IOCs), and develop adequate detections, protections, and countermeasures.

As a response, malware authors are motivated to protect their malware from analysts. They use anti-reverse engineering techniques to make malware more challenging to analyze, so it can continue propagating and infecting more systems before security measures are implemented.

This constant back and forth between malware analysts and authors can be aptly described as an "arms race". Malware authors develop new and sophisticated techniques to avoid detection and analysis, and security professionals respond by developing new methods and tools to detect and mitigate the threat posed by malware. As each side grows new techniques, the other responds with even more advanced ones, creating a cycle of escalation.

In this room, we will explore some of the various anti-reverse engineering techniques malware uses. These include:

- VM Detection
- Obfuscation using Packers
- Anti-Debugging

Many more techniques exist, but we will focus on these three for this room.

## Learning Objectives

- Learn why malware authors use anti-reverse engineering techniques
- Learn about different anti-reverse engineering techniques

- Learn the techniques on how to circumvent anti-reverse engineering using various tools
- Learn how anti-reverse engineering techniques are implemented by reading the source code

### Room prerequisites

- Familiarity with [Basic](#) and [Advanced](#) Dynamic Analysis (How to use a debugger, read assembly code, and patching)
- Basic knowledge of Assembly (What registers are, etc.)
- Basic understanding of C programming concepts (Conditions, program flow, etc.)

### Connecting to the machine

Start the virtual machine in split-screen view by clicking on the green "Start Machine" button on the upper right section of this task. If the VM is not visible, use the blue "Show Split View" button at the top-right of the page. Alternatively, you can connect to the VM using the credentials below via "Remote Desktop".

- Username: Administrator
- Password: Passw0rd!
- IP: MACHINE\_IP'

## Anti-Debugging: Overview

Debugging is the process of examining software to understand its inner workings and identify potential vulnerabilities or issues. Debugging involves software tools called debuggers that allow analysts to step through the code and monitor its execution.

Here are the commonly used debuggers for malware analysis nowadays:

- x64dbg
- Ollydbg
- Ida Pro
- Ghidra

### Anti-Debugging techniques

Malware authors use anti-debugging measures to make it difficult for analysts to use debugging tools to analyze the malware's behaviour.

Some of the most common anti-debugging measures include, but are not limited to:

Checking for the presence of a debugger	Malware code looks for processes or files associated with debugging tools or hardware-based techniques, such as detecting hardware breakpoints. For example, a common
---	---

	anti-debugging technique uses the Windows API function <code>IsDebuggerPresent</code> to check if a debugger is running. You can see this in action in the <a href="#">Advanced Dynamic Analysis</a> room.
Tampering with debug registers	Malware may try to modify or corrupt debug logs, which debuggers use to control the execution of code. This prevents the debugger from functioning correctly.
Using self modifying code	This is a sophisticated technique where malware modifies itself while running, making it difficult for a debugger to follow the code flow.

In the next task, we will showcase an anti-debugging technique that uses the Windows system API function called `SuspendThread`.

\*\*\*\*\*

**Answer the questions below:**

**What is the name of the Windows API function used in a common anti-debugging technique that detects if a debugger is running?**

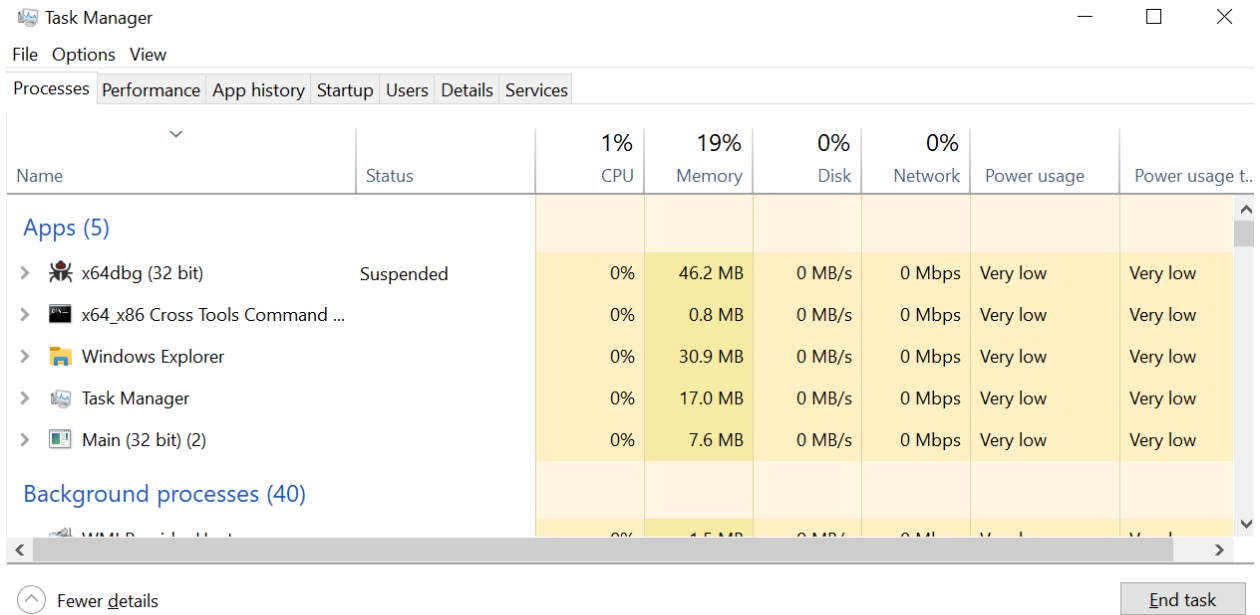
Answer: `IsDebuggerPresent`

## Anti-Debugging Using Suspend Thread

[Suspend Thread](#) is a Windows API function that is used to pause the execution of a thread in a running process. This function has legitimate uses but can also be called from within a malware process to suspend any threads attempting to debug or analyze it. What better way to thwart debugging than not making the debugger work at all?

We have created a simple program that does the above technique. You can try and run it to see how it behaves.

1. Run the x64dbg tool from the Desktop. Open the `suspend-thread.exe` file from the `C:\Malware\` directory.
2. Press F9 twice to continue the execution of the malware.
3. Notice that x64dbg will now become unresponsive. You can confirm the suspended process by opening Task Manager.



(To completely terminate x64dbg, right click on it and select "End Task").

The code snippet below shows how the malware works under the hood:

```

#include <windows.h>
#include <string.h>
#include <wchar.h>
#include <tlhelp32.h>
#include <stdio.h>

DWORD g_dwDebuggerProcessId = -1;

BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM dwProcessId)
{
    DWORD dwWindowProcessId;
    GetWindowThreadProcessId(hwnd, &dwWindowProcessId);

    if (dwProcessId == dwWindowProcessId)
    {
        int windowTitleSize = GetWindowTextLengthW(hwnd);
        if ( windowTitleSize <= 0 )
        {
            return TRUE;
        }
        wchar_t* windowTitle = (wchar_t*)malloc((windowTitleSize + 1) * sizeof(wchar_t));

        GetWindowTextW(hwnd, windowTitle, windowTitleSize + 1);

        if (wcsstr(windowTitle, L"dbg") != 0 ||
            wcsstr(windowTitle, L"debugger") != 0 )
        {
            g_dwDebuggerProcessId = dwProcessId;
            return FALSE;
        }

        return FALSE;
    }

    return TRUE;
}

DWORD IsDebuggerProcess(DWORD dwProcessId)
{
    EnumWindows(EnumWindowsProc, (LPARAM)dwProcessId);
    return g_dwDebuggerProcessId == dwProcessId;
}

```

```

DWORD IsDebuggerProcess(DWORD dwProcessId)
{
    EnumWindows(EnumWindowsProc, (LPARAM)dwProcessId);
    return g_dwDebuggerProcessId == dwProcessId;
}

DWORD SuspendDebuggerThread()
{
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if (hSnapshot == INVALID_HANDLE_VALUE)
    {
        printf("Failed to create snapshot\n");
        return 1;
    }

    THREADENTRY32 te32;
    te32.dwSize = sizeof(THREADENTRY32);

    if (!Thread32First(hSnapshot, &te32))
    {
        printf("Failed to get first thread\n");
        CloseHandle(hSnapshot);
        return 1;
    }

    do
    {
        HANDLE hThread = OpenThread(THREAD_QUERY_INFORMATION | THREAD_SUSPEND_RESUME, FALSE, te32.th32ThreadID);
        if (hThread != NULL)
        {
            DWORD dwProcessId = GetProcessIdOfThread(hThread);
            if ( IsDebuggerProcess(dwProcessId) )
            {
                printf("Debugger found with pid %i! Suspending!\n", dwProcessId);
                DWORD result = SuspendThread(hThread);
                if ( result == -1 )
                {
                    printf("Last error: %i\n", GetLastError());
                }
            }
            CloseHandle(hThread);
        }
    } while (Thread32Next(hSnapshot, &te32));

    CloseHandle(hSnapshot);

```

```

        return 0;
    }

    int main(void)
    {
        SuspendDebuggerThread();

        printf("Continuing malicious operation...");
        getchar();
    }

```

If you don't want to be bothered with reading the source code, here is a short explanation of the steps used by this technique:

1. The malware goes through all threads in the Windows system.
2. For each thread, it calls EnumWindow to go through each window displayed on the screen.
3. If the name of the window has the strings debugger, dbg, or debug, then the malware knows that a debugger is running.
4. If a debugger is present, the malware calls, which suspends the threads of the debugger, making it crash.
5. The malware proceeds with its malicious purpose.

As you can see, this method can effectively stop an analyst from continuing with the investigation. Fortunately, we have a way of dealing with this.

## Patching

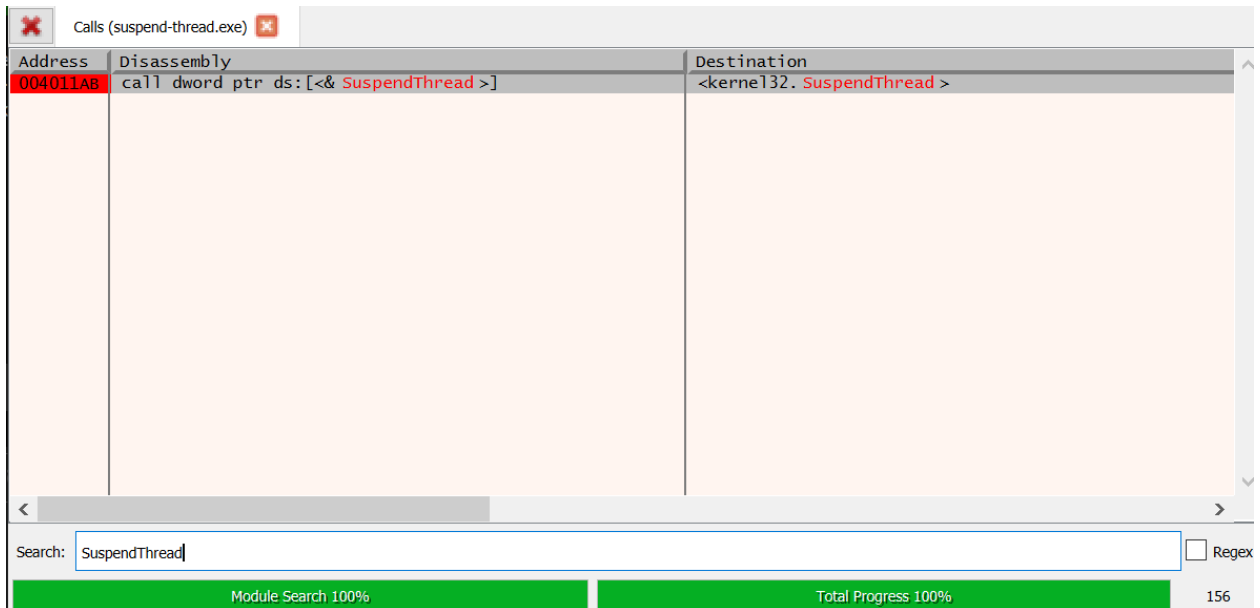
Patching is one of the most critical skills required of an analyst. The Advanced Dynamic Analysis room has already covered this skill, but we'll have a quick review here.

In this exercise, we want to patch the function SuspendThread so that our debugger won't get suspended.

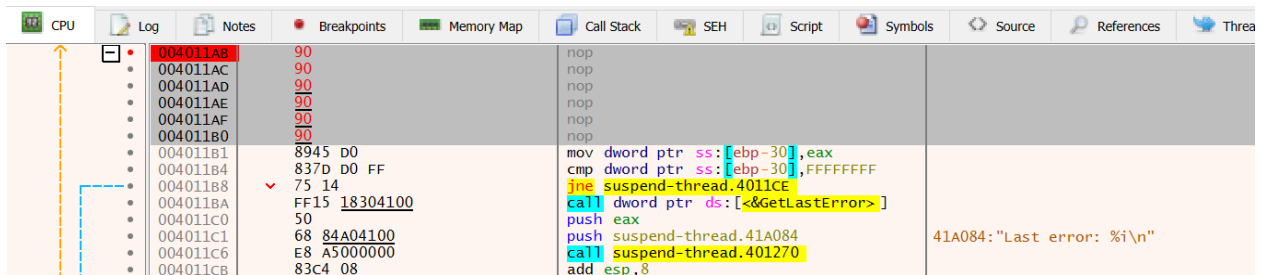
- Open x32dbg by double-clicking on its icon from the Desktop.
- Press the F3 key to show the "Open" dialogue window. Look for our sample file called suspend-thread.exe in the C:\Malware\ directory.
- Press the F9 key once to jump execution to the EntryPoint, as shown below:

EIP	ECX	EDX	E	00401502	E8 0D040000	call suspend-thread.401914	EntryPoint
				00401507	E9 74FEFFFF	jmp suspend-thread.401380	
				0040150C	55	push ebp	
				0040150D	8BEC	mov ebp,esp	
				0040150F	6A 00	push 0	
				00401511	FF15 38304100	call dword ptr ds:[<&SetUnhandledExcepti	
				00401517	FF75 08	push dword ptr ss:[ebp+8]	
				0040151A	FF15 34304100	call dword ptr ds:[<&UnhandledExceptionF	
				00401520	68 090400C0	push C0000409	
				00401525	FF15 3C304100	call dword ptr ds:[<&GetCurrentProcess>	
				0040152B	50	push eax	
				0040152C	FF15 40304100	call dword ptr ds:[<&TerminateProcess>]	
				00401532	5D	pop ebp	
				00401533	C3	ret	
				00401534	55	push ebp	

- Right-click anywhere on the main screen, highlight "Search for > Current Module", then select "Intermodular calls".
- On the search field at the bottom, type "SuspendThread". This will filter the list in the main window showing us one entry. Double-click on this entry.

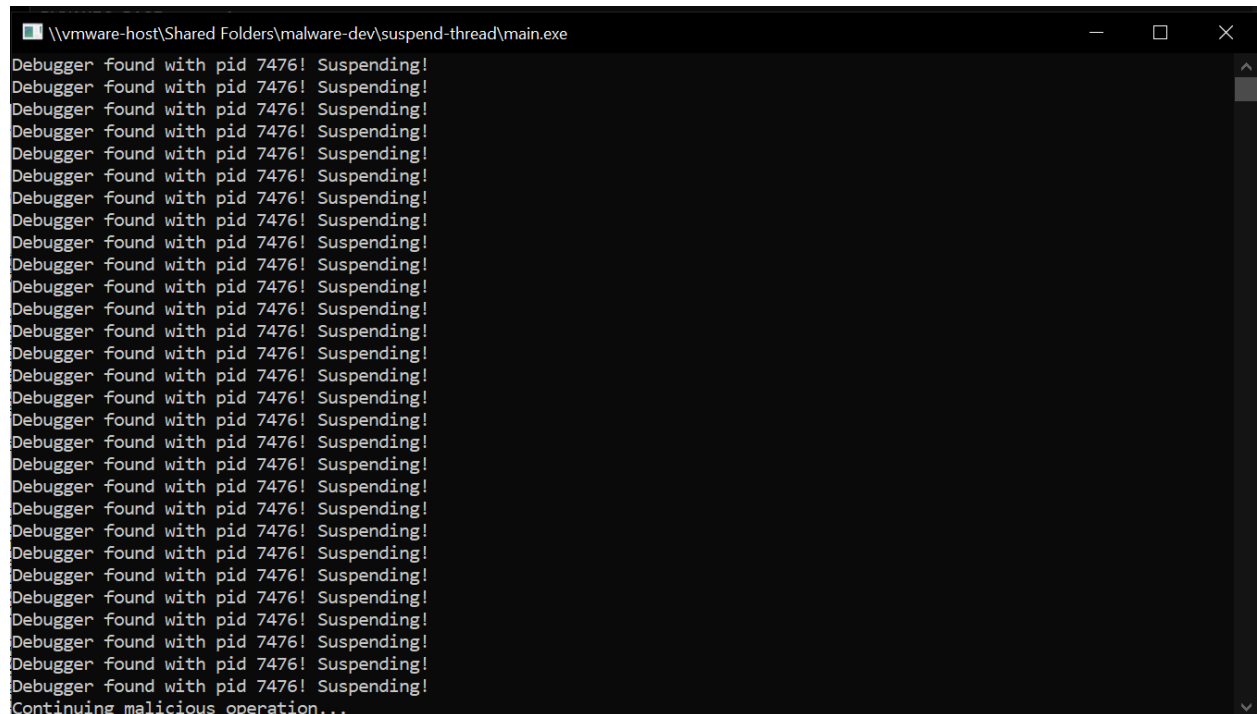


- You are now on the part of the code that calls the SuspendThread API. To skip this function, right-click on the Suspendthread line and select Binary > Fill with NOPs. Don't change anything here, and press "OK".



- You will see that memory locations 004011AB up to 004011B0 are now all set to the 90 hex value or nop (No Operation). If execution reaches this instruction, it will do nothing and proceed with the following line.
- Press F9 to continue execution, and we'll see that debugging continues at the very end without crashing our debugger. Note that while the console prints out lines that say "Debugger found with PID XXXX! Suspending!", it's not doing anything because we've skipped the function that does the suspending.





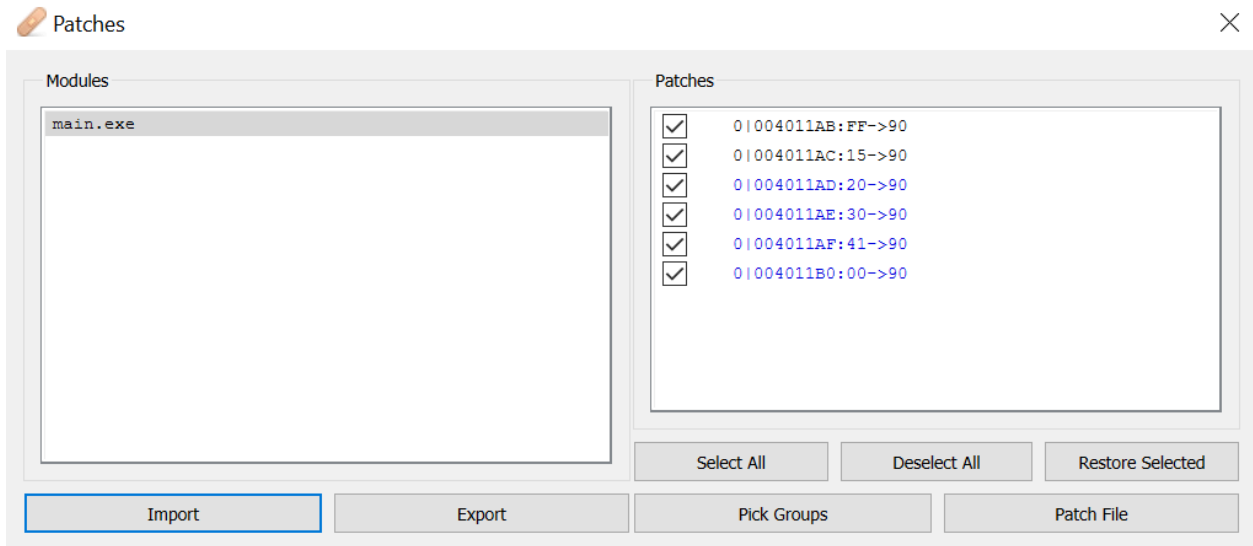
```
\\vmware-host\Shared Folders\malware-dev\suspend-thread\main.exe
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Debugger found with pid 7476! Suspending!
Continuing malicious operation...
```

Now, this is great. However, if you try debugging again, you'll discover it will start crashing again. This is because our patches are reset and are now gone. To avoid re-applying patches in the future, we can export and import our patches for future use.

**Exporting and Importing Patches**

Stop debugging by pressing Alt+F2 and go through the patching steps in the previous section again, but do not do the last step.

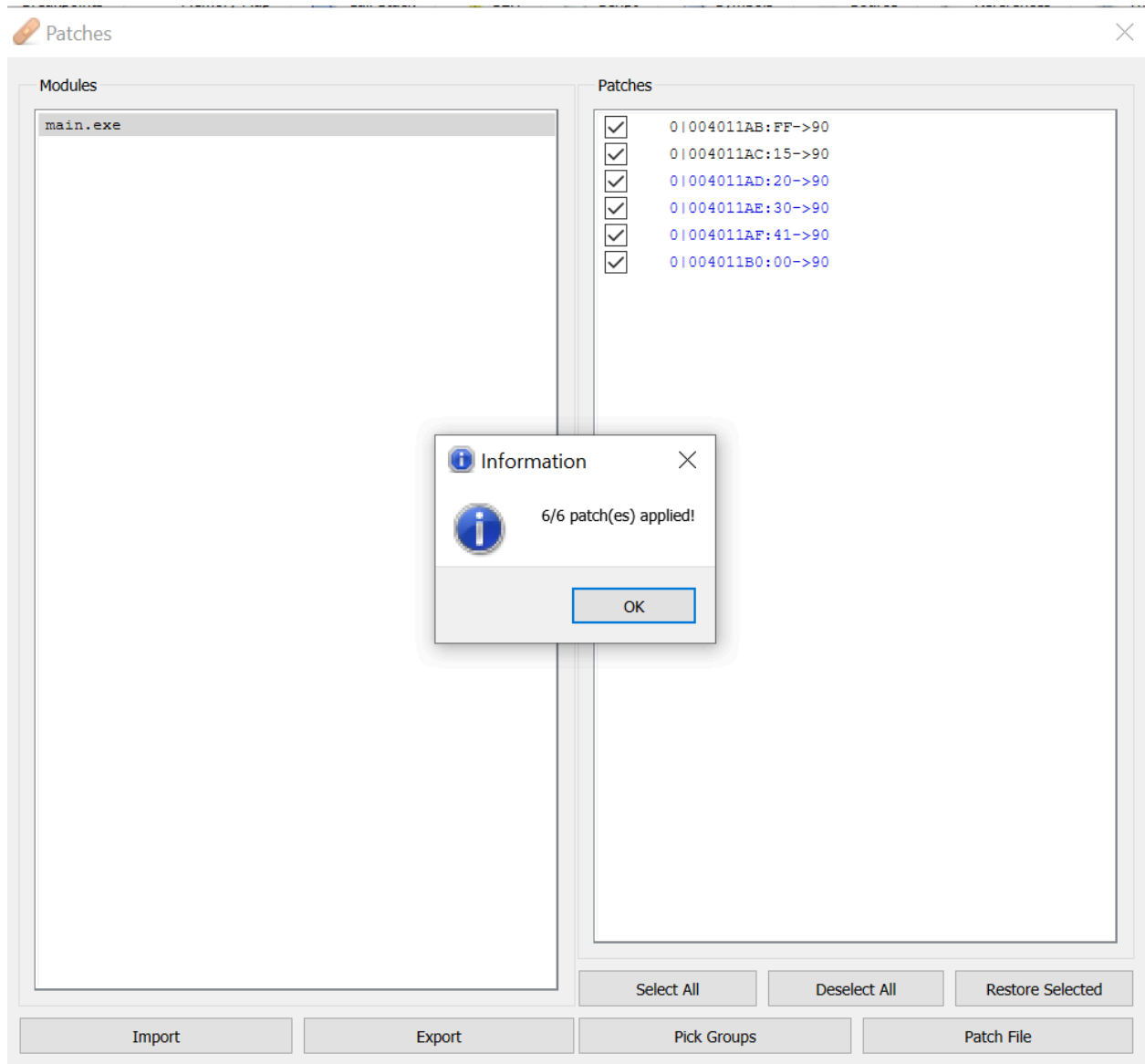
- Once the patches you want are in place, on the top bar, go to "View > Patch file..." to open the "Patch" dialogue window.
- A list of our patches will appear on the right panel. You can see there are six entries. This refers to the six memory locations that we have set to NOPs.



- Click on the "Export" button. On the next dialogue window, click "Yes" to continue. Save this file to a location of your choice.
- Stop the debugger by pressing Alt+F2.

Debug the file again. Because debugging has restarted, our previous patches have now been erased. We can restore it by importing the file we made a while ago.

- Go to the patches dialogue window again to "View > Patch file...".
- But this time, press the "Import" button and select the file we saved.



- Once imported, close the window and press F9 again to continue execution. You'll notice that the previous patches are now re-applied.

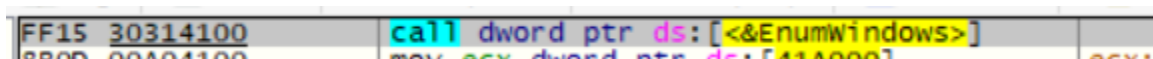
Exporting and importing patches are very helpful in reverse engineering programs as it allows you to save your work to save the hassle of doing them all over again.

In the next task, we'll examine another anti-reverse engineering technique and other ways to circumvent them.

\*\*\*\*\*

**Answer the questions below:**

**What is the Windows API function that enumerates windows on the screen so the malware can check the window name?**

A screenshot of assembly code. The instruction 'call dword ptr ds:[&EnumWindows]' is highlighted in yellow. The address 'FF15 30314100' is visible on the left, and 'mov ecx, dword ptr ds:[410000]' is visible on the right.

Answer: EnumWindows

**What is the hex value of a nop instruction?**

Answer: 90

**What is the instruction found at memory location 004011CB?**

A screenshot of assembly code. The instruction 'add esp, 8' is highlighted in yellow. The address '004011CB' is visible on the left, and the hex value '83C4 08' is visible in the middle.

Answer: add esp,8

## VM Detection: Overview

Virtual Machines (VMs) are software platforms that simulate a computer environment inside another computer system. These are useful in reverse engineering because they provide a cost-effective, controlled, and isolated environment for monitoring and analyzing suspicious software or malware. VMs also allow for the creation of snapshots and checkpoints that can be used to restore the system to a previous state, which helps test different scenarios and maintain a history of the analysis process.

When malware identifies that it is running on a VM, it may decide to respond differently; for example, it may change its behaviour by:

- Executing only a minimal subset of its functionality
- Self-destructing by deleting itself or overwriting parts of its code
- Cause damage to the system by deleting or encrypting files; or
- Not run at all

All of the above behaviours aim to reveal less information, making it harder for the analyst to progress.

## Detection Techniques

Malware can employ various techniques to detect and evade analysis in virtual machine environments. Some of them are listed below:

Checking Running Processes	VMs have easily identifiable processes; for example, VMWare runs a process called vmtools, while VirtualBox has vboxservice. Malware can use the
----------------------------	--

	<a href="#">EnumProcess</a> Windows API to list all the processes running on the machine and look for the presence of these tools.
Checking Installed Software	Malware can look in the Windows Registry for a list of installed software under the SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall Registry key. From here, it can check for installed programs like debuggers, decompilers, forensics tools, etc.
Network Fingerprinting	Malware can look for specific MAC and network addresses unique to VMs. For example, VMs autogenerate MAC addresses that start with any of the following numbers: 00-05-69, 00-0c-29, 00-1c-14 or 00-50-56. These numbers are unique and are specifically assigned to a VM vendor called the OUI (Organizationally Unique Identifier).
Checking Machine Resources	Malware can look at a machine's resources like RAM and CPU Utilization percentages. For example, a machine with RAM amounting to less than 8GB can indicate a virtual machine, as they are typically not assigned a significant amount.
Detecting Peripherals	Some malware checks for connected printers because this is rarely configured properly on VMs, sometimes not even configured at all.
Checking for Domain Membership	Corporate networks are a usual target for malware. An easy way to determine this is by checking if the current machine is part of an Active Directory domain. This can quickly be done without the use of API calls by checking the LogonServer and ComputerName environment variables.
Timing-based Attacks	Malware can measure the time it takes to execute specific instructions or access particular machine resources. For example, some instructions can be faster on a physical machine compared to a virtual machine.

Many techniques mentioned above are not foolproof, so it is not uncommon to see malware use a combination of one or more to ensure a higher chance of correct identification.

## Anti-VM Detection

To prevent malware from using some of the techniques above, we can apply several changes to the system that will remove VM-related artefacts making the VM look less like a VM. For example, we can remove or modify the Registry entries that malware checks for installed programs to hide our debuggers, change the MAC addresses, or configure the VM to appear connected to a printer.

But you can see how tedious this becomes. With the myriad of options available to malware, it would be challenging to cover all of them.

Some researchers have made scripts (See [VMwareCloak](#) and [VBoxCloak](#)) to help automate this process. Despite this, even if all the known techniques are addressed, minor architectural checks can still be made by malware, as you will see in the next task.

\*\*\*\*\*

**Answer the questions below:**

**What is the name of the identifiable process used by malware to check if the machine is running inside VirtualBox?**

Answer: **vboxservice**

**What is the OUI automatically assigned specifically to VMware?**

According to the VMware OUI allocation scheme, a MAC address has the format of **00:50:56:XX:YY:ZZ**, where **00:50:56** represents the VMware OUI, XX is calculated as (80 + vCenter Server ID), and YY and ZZ are random two-digit hexadecimal numbers.

Answer: **00:50:56**

**Using Task Manager, what process indicates that the machine for this room is an Amazon EC2 Virtual Machine?**

Name	Status	1% CPU	49% Memory
<b>Apps (3)</b>			
> suspend-thread.exe (32 bit)		0%	0.5 MB
> Task Manager		0%	11.6 MB
> x64dbg (32 bit)		0%	40.5 MB
<b>Background processes (21)</b>			
> amazon-ssm-agent.exe		0%	4.9 MB

Answer: **amazon-ssm-agent.exe**

## VM Detection by Checking the Temperature

Here is an example of a VM detection technique that checks the machine's temperature provided by the hardware.

Win32\_TemperatureProbe is a Windows Management Instrumentation (WMI) class that contains real-time temperature readings from the hardware through the SMBIOS (System Management BIOS) data structure. In a virtualized environment, the value returned is Not Supported, which is what malware looks for.

Note: Win32\_TemperatureProbe may also return Not Supported even on physical machines if the hardware doesn't support this SMBIOS feature. This makes it unreliable but valuable when used with other techniques mentioned in the previous task.

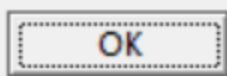
We have provided a working program demonstrating this technique in the VM attached to this room. Look for the vm-detection.exe file in the C:\Malware\ directory and execute it. You will see the result immediately.

Upon execution, a message window will appear:

VM Detected



Proceeding with non-malicious activities...



As you can see, the malware sensed it was running in a VM. It modified its behaviour by continuing a non-malicious activity to throw off analysts snooping around.

Below is the code snippet for this particular program:

```
{
    IWbemClassObject* pclsObj = (IWbemClassObject*)malloc(sizeof(IWbemClassObject));

    ULONG uReturn = 0;

    HRESULT hr = CoInitializeEx(0, COINIT_MULTITHREADED);
    hr = CoInitializeSecurity(NULL, -1, NULL, NULL, RPC_C_AUTHN_LEVEL_DEFAULT, RPC_C_IMP_LEVEL_IMPERSONATE, NULL, EOAC_NONE, NULL);
    hr = CoCreateInstance(CLSID_WbemLocator, NULL, CLSCTX_INPROC_SERVER, IID_IWbemLocator, (LPVOID*)&pLoc);
    hr = pLoc->ConnectServer(L"root\\wmi", NULL, NULL, 0, NULL, 0, 0, &pSvc);
    hr = CoSetProxyBlanket(pSvc, RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE, NULL, RPC_C_AUTHN_LEVEL_CALL, RPC_C_IMP_LEVEL_IMPERSONATE, NULL, EOAC_NONE);
    hr = pSvc->ExecQuery(L"WQL", L"SELECT * FROM MSAcpi_ThermalZoneTemperature", WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY, NULL, &pEnumerator);

    while (pEnumerator)
    {
        hr = pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn);
        if (uReturn == 0)
        {
            return 0;
        }

        VARIANT vtProp;

        hr = pclsObj->Get(L"CurrentTemperature", 0, &vtProp, 0, 0);
        if (SUCCEEDED(hr))
        {
            printf("Thermal Zone Temperature: %d\n", vtProp.intVal);
            return 1;
        }

        VariantClear(&vtProp);
        pclsObj->Release();
    }

    pEnumerator->Release();
    pSvc->Release();
    pLoc->Release();

    CoUninitialize();

    return 0;
}
```

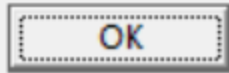
You can try compiling and running this code on your physical machine to see how it changes its behaviour (No actual malicious execution is in the provided code so it is safe to run). The output below shows how the malware behaves when run from a physical machine. The malware receives the correct temperature reading from the WMI class and proceeds with its execution.



Starting malware



Proceeding with malicious activities...



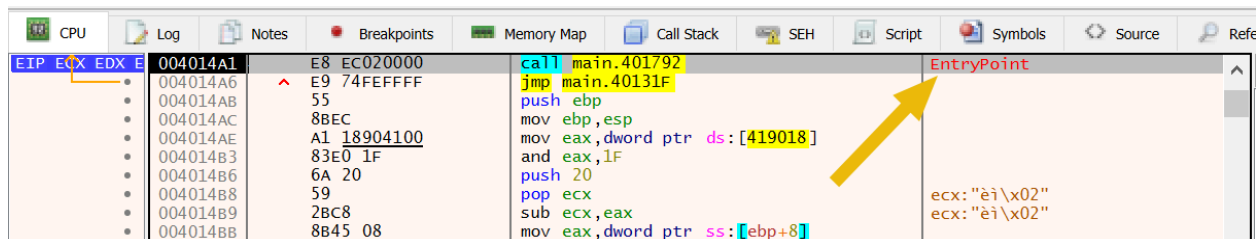
## Preventing Temperature Checking

As mentioned in the previous task, architectural checks like checking the temperature can be challenging to prevent. Fortunately for us, there are still ways around this. For example, we've learned in the last task that we can patch a function with nops to prevent it from executing. We can do the same thing here, but for the sake of giving you more tools to help you with reverse engineering, we'll be covering two other approaches:

- Manipulating memory directly
- Changing the execution flow with EIP

Here are the steps:

- Open x32dbg.exe by double-clicking on the desktop icon
- Go to File > Open, then look for the vm-detection.exe file in the C:\Malware\ directory.
- Press F9 to start running the program in the debugger. Debugging will halt at the code marked as the "EntryPoint".



- Press Ctrl+G on your keyboard to bring up the "Enter expression to follow..." pop-up window. Enter the value 004010E0 and press "OK". This will take you to the memory address. 0x004010E0.
- Press F2 to set up a breakpoint at this memory location, then press F9 once to halt execution here.
- Right now, we are at the memory location 0x004010E0. The block of instructions from this location up until 0x004010F8 corresponds to this part of our C code snippet: `hr = pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn);`. To

help you visualize this, I've added comments on the right side of the screenshot below to highlight the values passed to the function.

- As previously discussed, this function will fail because we run the program in a VM. After executing, the function will write the value 0 to the memory location [ebp-18] (the one marked with uReturn in the screenshot above).
- Press F8 a few times to step through the code until you reach the memory location 004010FD. Here, we can see that the value of [ebp-18] (or uReturn) is being compared to zero. This corresponds to this part in the C code: if (uReturn == 0).
- Since we want this "if" condition to evaluate to False, what we can do is edit the memory location pointed to by [ebp-18] so that it will contain the value 1 instead of 0.
- To do this, right-click on the line that has [ebp-18], click on "Follow in dump", then select [ebp-18].

- You will see at the bottom of the screen, at the "Dump 1" window tab, that our current memory location is 0019FF08. This memory location contains the value for [ebp-18] or uReturn.

Address	Hex	ASCII
0019FEA8	38 49 7F 00	8I.....ÿ.....
0019FEB8	64 1D 7E 00	d.~...»mÊë}D.bD0
0019FEC8	D0 08 5D E6	Ð. ]æx[E6`ÿ...`p
0019FED8	FF FF FF FF	ÿÿÿÿ ÿ..ú.@.ÄI..
0019FEE8	FF FF FF FF	ÿÿÿÿ.....ÿ...ÿ..
0019FEF8	00 60 23 00	.#.p.Éw.ü@.é..
0019FF08	00 00 00 00	... Ä .ÄÄ .0.}
0019FF18	C0 49 7E 00	ÄI.....(ÿ...@.
0019FF28	70 FF 19 00	pÿ....@.....0. .
0019FF38	E8 95 7C 00	è. ...îË .@. .@.
0019FF48	00 60 23 00	.#.....
0019FF58	3C FF 19 00	<ÿ.....îÿ..ð.@.
0019FF68	B0 85 B4 C8	°.É.....ÿ..)ú.v
0019FF78	00 60 23 00	.#..ú.vüÿ...zÇw

- Double-click on the rightmost 00 value. This will open up a "Modify Value" window. Enter the value 01 in the expression field and press the "OK" button. The result should look like this:

Address	Hex	ASCII
0019FEA8	38 49 7E 00	8I.....ÿ.....
0019FEB8	64 1D 7E 00	d.~...»mÊë}D.bd0
0019FEC8	D0 08 5D E6	ð.]æx[E6`ÿ...p
0019FED8	FF FF FF FF	ÿÿÿÿ ÿ..ú.@.ÄI..
0019FEE8	FF FF FF FF	ÿÿÿÿ.....ÿ...ÿ..
0019FEF8	00 60 23 00	..#.p.Éw.ü@.é...
0019FF08	00 00 00 01	... Ä .ÄÄ .0.}
0019FF18	C0 49 7E 00	ÄI.....(ÿ...@.
0019FF28	70 FF 19 00	pÿ....@.....0. .
0019FF38	E8 95 7C 00	è. ...ÿÿj.@.j.@.
0019FF48	00 60 23 00	..#.....
0019FF58	3C FF 19 00	<ÿ.....ÿÿ..ð.@.
0019FF68	B0 85 B4 C8	°.É.....ÿ..)ú.v
0019FF78	00 60 23 00	..#.ú.vÿÿ...zÇw

- Turn your attention back to the main window (CPU tab) and press F8 a few times. You'll notice that the code will not exit anymore as it skips the execution from 00401101 to 0040110A.

What you've done is that you've manipulated the bits of a memory location directly to influence the flow of the program. This method is helpful in many other ways, like for example, you could change the value that will be passed to a function before you execute that function.

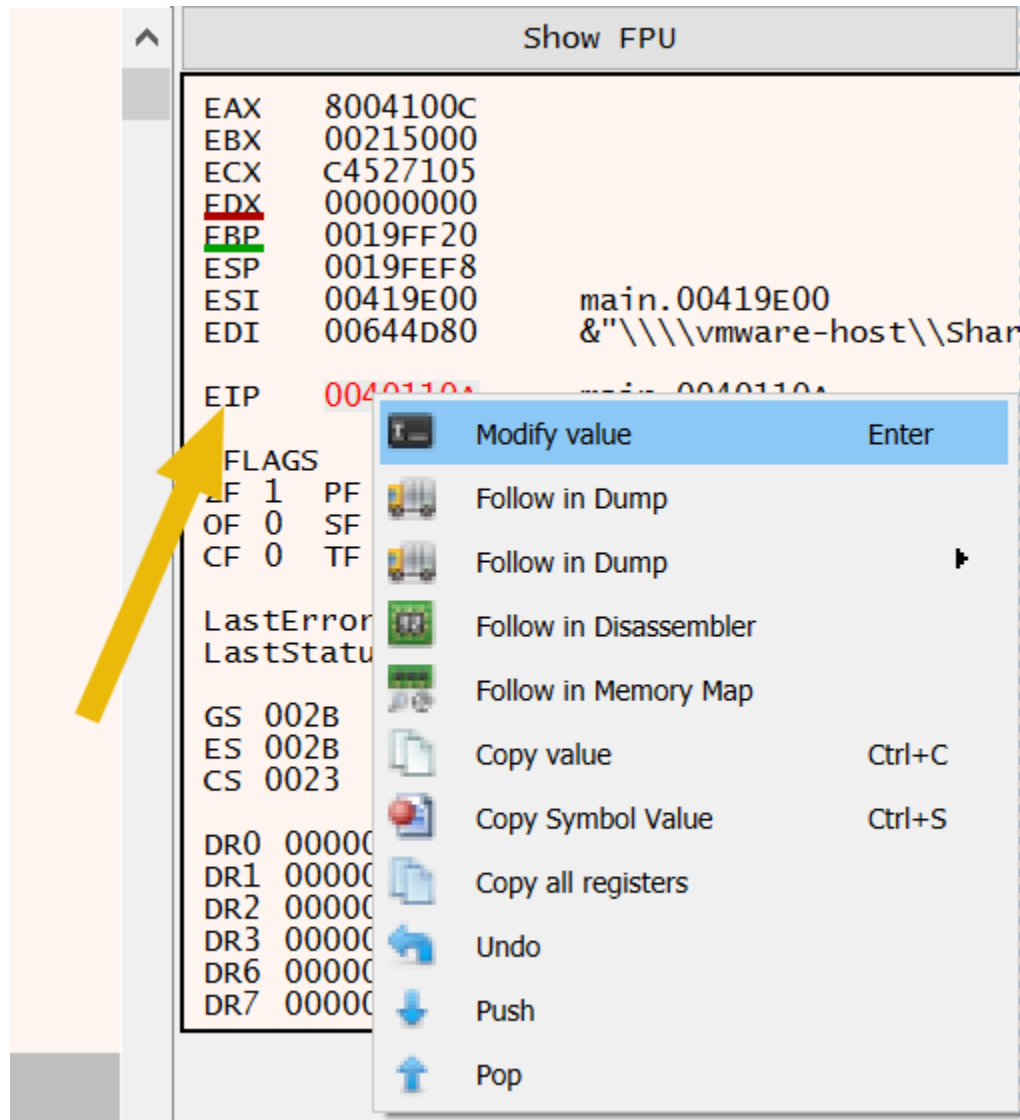
However, the program will now crash if you continue running the program. The program was supposed to exit at the previous function, but we've introduced a bug since we intentionally skipped this. Don't worry; this always happens when messing around in the debugger. So for us to continue to investigate, there is one other method we can use that would allow us to jump around and continue.

- Let us say we want to skip everything and jump to this part of the code right here: printf("Thermal Zone Temperature: %d\n", vtProp.intVal);. In the assembly code, this would be the instructions 00401130 to 00401139, as shown below:

00401105	E9 80000000	jmp main.401119	Exit the function with 0
0040110A	6A 00	push 0	
0040110C	6A 00	push 0	
0040110E	8D45 D8	lea eax,dword ptr ss:[ebp-28]	
00401111	50	push eax	
00401112	6A 00	push 0	
00401114	68 68224100	push main.412268	412268:L"CurrentTemperature"
00401119	8B4D F0	mov ecx,dword ptr ss:[ebp-10]	
0040111C	8B11	mov edx,dword ptr ds:[ecx]	
0040111E	8B45 F0	mov eax,dword ptr ss:[ebp-10]	
00401121	50	push eax	
00401122	8B4A 10	mov ecx,dword ptr ds:[edx+10]	
00401125	FFD1	call ecx	
00401127	8945 FC	mov dword ptr ss:[ebp-4],eax	
0040112A	837D FC 00	cmp dword ptr ss:[ebp-4],0	
0040112E	7C 18	j1 main.401148	
00401130	8B55 E0	mov edx,dword ptr ss:[ebp-20]	vtProp.intVal
00401133	52	push edx	
00401134	68 90224100	push main.412290	412290:"Thermal Zone Temperature: %d\n"
00401139	E8 E2000000	call main.401220	Call to printf()
0040113E	83C4 08	add esp,8	

The great thing about debuggers is that we can change every value related to the program; we can change the values in the memory locations, the values on the stack, and even the values of the registers. The EIP register is the register that holds the memory address that tells the debugger what next instruction to execute next. And yes, this can be edited!

- Continue from the last step. On the "Registers" panel on the right of your screen, look for the one that says EIP. Right-click on the value and select "Modify Value".



- Enter 00401134 as the new value and press OK.
- Now when we press F8 to step through the code, you'll notice that we'll jump directly to that instruction. Notice how the EIP marker now points to the location we wanted.

0040110A	6A 00	push 0	
0040110C	6A 00	push 0	
0040110E	8D45 D8	lea eax,dword ptr ss:[ebp-28]	
00401111	50	push eax	
00401112	6A 00	push 0	
00401114	68 68224100	push main.412268	412268:L"CurrentTemperature"
00401119	8B4D F0	mov ecx,dword ptr ss:[ebp-10]	
0040111C	8B11	mov edx,dword ptr ds:[ecx]	
0040111E	8B45 F0	mov eax,dword ptr ss:[ebp-10]	
00401121	50	push eax	
00401122	8B4A 10	mov ecx,dword ptr ds:[edx+10]	
00401125	FFD1	call ecx	
00401127	8945 FC	mov dword ptr ss:[ebp-4],eax	
0040112A	837D FC 00	cmp dword ptr ss:[ebp-4],0	
0040112E	7C 18	j1 main.401148	
00401130	8B55 E0	mov edx,dword ptr ss:[ebp-20]	vtProp.intVal
00401133	52	push edx	
00401134	68 90224100	push main.412290	412290:"Thermal Zone Temperature: %d\n"
00401139	E8 E2000000	call main.401220	Call to printf()
0040113E	8374 08	add esp,8	

- Pressing F9 again would continue running the program, resulting in the output below:

Good job! With the above steps, you successfully prevented the malware from detecting that it is running on a VM. You can confirm this by also checking the Command Prompt output.

These simple techniques will be helpful in going through the code and influencing the program flow.

But we can't celebrate just yet, for there are other things malware can do to prevent you from discovering more of its capabilities, like the use of Obfuscation and Packers, which is discussed in the next task.

\*\*\*\*\*

**Answer the questions below:**

**In the C code snippet, what is the full WQL query used to get the temperature from the Win32\_TemperatureProbe class?**

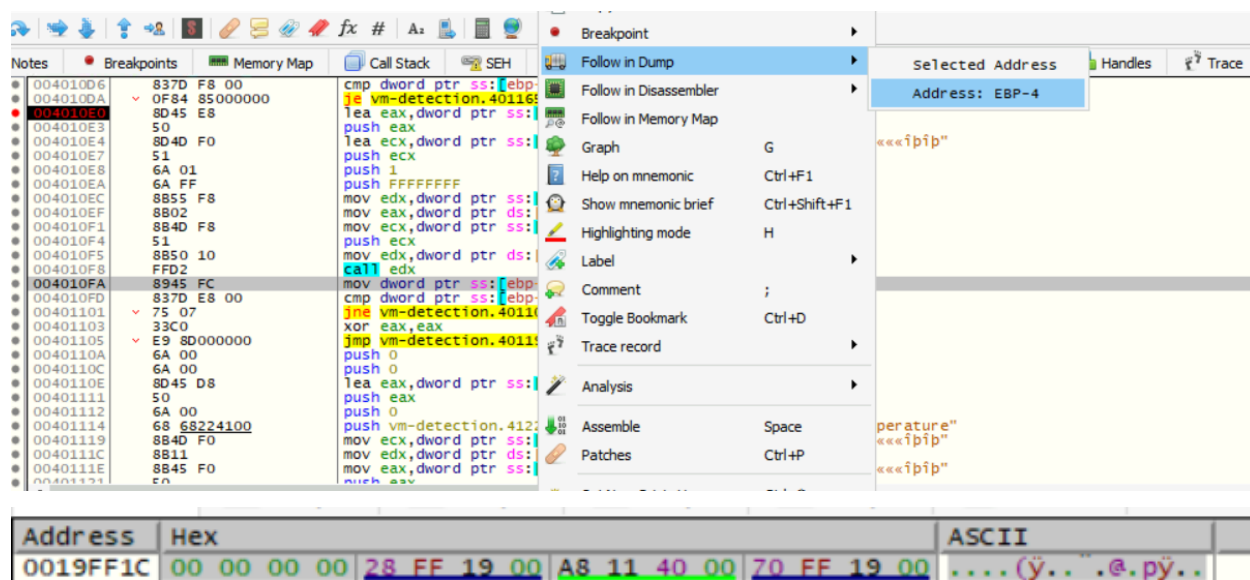
```
hr = pSvc->ExecQuery(L"WQL", L"SELECT * FROM MSAcpi_ThermalZoneTemperature", WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY, NULL, &pEnumerator);
```

Answer: **SELECT \* FROM MSAcpi\_ThermalZoneTemperature**

**What register holds the memory address that tells the debugger what instruction to execute next?**

Answer: **EIP**

**Before uReturn is compared to zero, what is the memory location pointed to by [ebp-4]**



Answer: 0019FF1C

## Packers: Overview

Obfuscation is a technique that aims to intentionally obscure data and code so that it is harder to understand or analyze.

The most common obfuscation techniques used by malware authors include:

- Using encoding techniques: This involves encoding data (i.e. command line strings, domain names, etc.) using popular encoding techniques like XOR or Base64. You may have seen a Base64 encoded strings that look like this VGhpcyBpcyBhbiBCQVNFNjQgZW5jb2RlZCBzdHJpbmcu==.
- Using encryption techniques: This involves encrypting data such as communications to a command and control server, file formats, and network traffic. The most common types used are symmetric key and public key encryption.
- Code obfuscation: This involves various techniques such as manipulating the code to alter its syntax and structure, renaming functions, or splitting code across multiple files or code segments.

There's much to discuss with obfuscation, but we won't delve deep into them in this room. In this task, we want to talk about the wide use of Packers in malware nowadays, a technique that can also be considered to fall under obfuscation.

## Packers

Packers are tools that compress and encrypt executable files. It compresses the target executable and embeds it within a new executable file that serves as a wrapper or container. This dramatically reduces the size of the file, making it ideal for easy distribution and installation. Also, some packers offer additional features, such as code obfuscation, runtime packing, and anti-debugging techniques. And it is because of these features that made Packers a popular tool for malware authors.

There are a lot of Packers available. Each has a unique approach and algorithm for packing. Here is a list of some that were seen used in the wild:

- [Alternate EXE Packer](#)
- [ASPack](#)
- [ExeStealth](#)
- [hXOR-Packer](#)
- [Milfuscator](#)
- [MPress](#)
- [PELock](#)
- [Themida](#)
- [UPX: the Ultimate Packer for eXecutables](#)
- [VMProtect](#)

It is essential to state that not all packed programs are malicious. Packers are also used by legitimate software to protect their programs, like for protecting their intellectual property from theft. For example, the Packer tool "Themida" is widely used to prevent video game cheating.

Because Packers encrypts and obfuscates a program, it would be impossible to know the malware's capabilities without running it. Because of this, we cannot reliably depend on static analysis and signature-based detection techniques to determine its capabilities. The only information we could glean from a malware sample at this state is the Packer tool used. This can still be a good starting point for an investigation, which we will see in the next task.

\*\*\*\*\*

**Answer the questions below:**

**What is the decoded string of the base64 encoded string  
"VGhpcyBpcyBhIEJBU0U2NCBibmNvZGVkIHNoZmluZy4="?**

## Input

```
VGhpcyBpcyBhIEJBU0U2NCBlbmNvZGVkIHNoemluZy4=
```

ABC 44 1

## Output

```
This is a BASE64 encoded string.
```

Answer: **This is a BASE64 encoded string.**

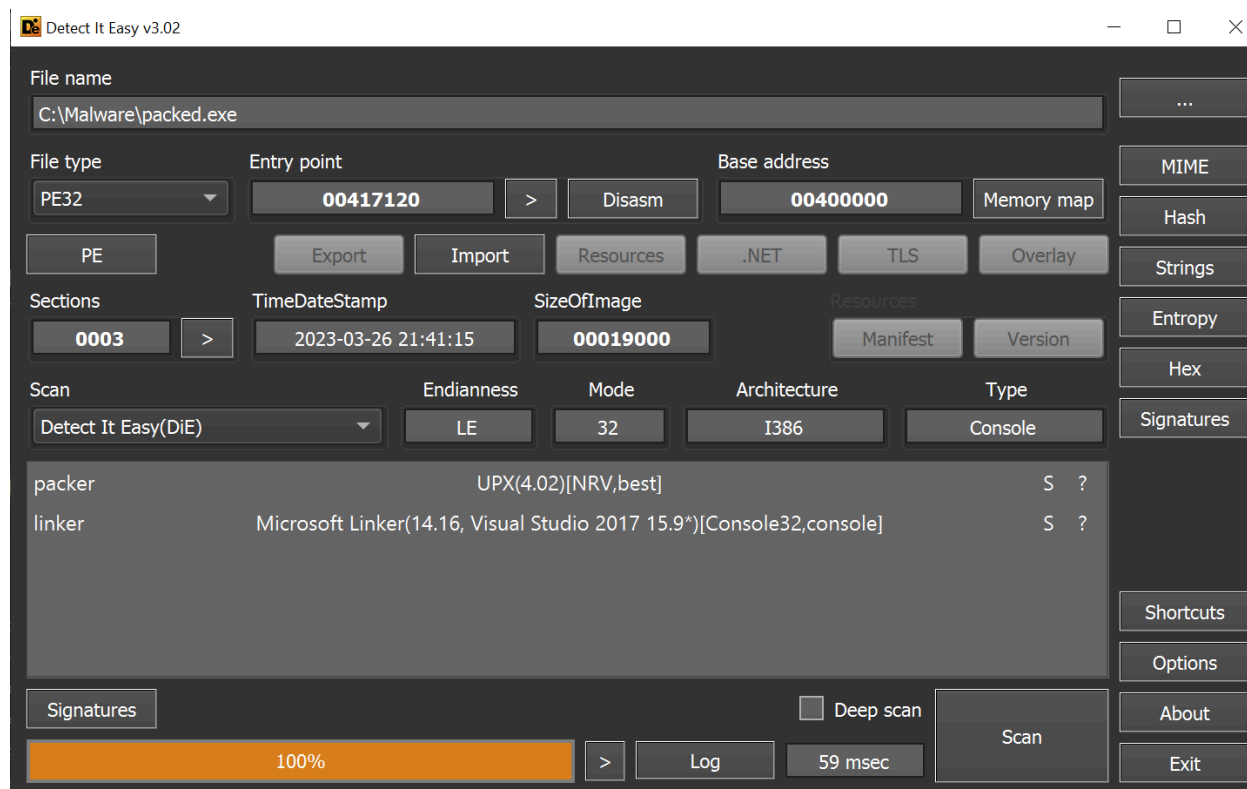
## Identifying and Unpacking

The first step in dealing with packed malware is identifying the Packer used. Thankfully, there are tools that we can use for this. The most accessible tools are DetectItEasy (DIE) and PEStudio, which are already included in the VM attached to this room. You will also find a packed sample program titled packed.exe under the C:\Malware\ directory that we can use to test these tools on.

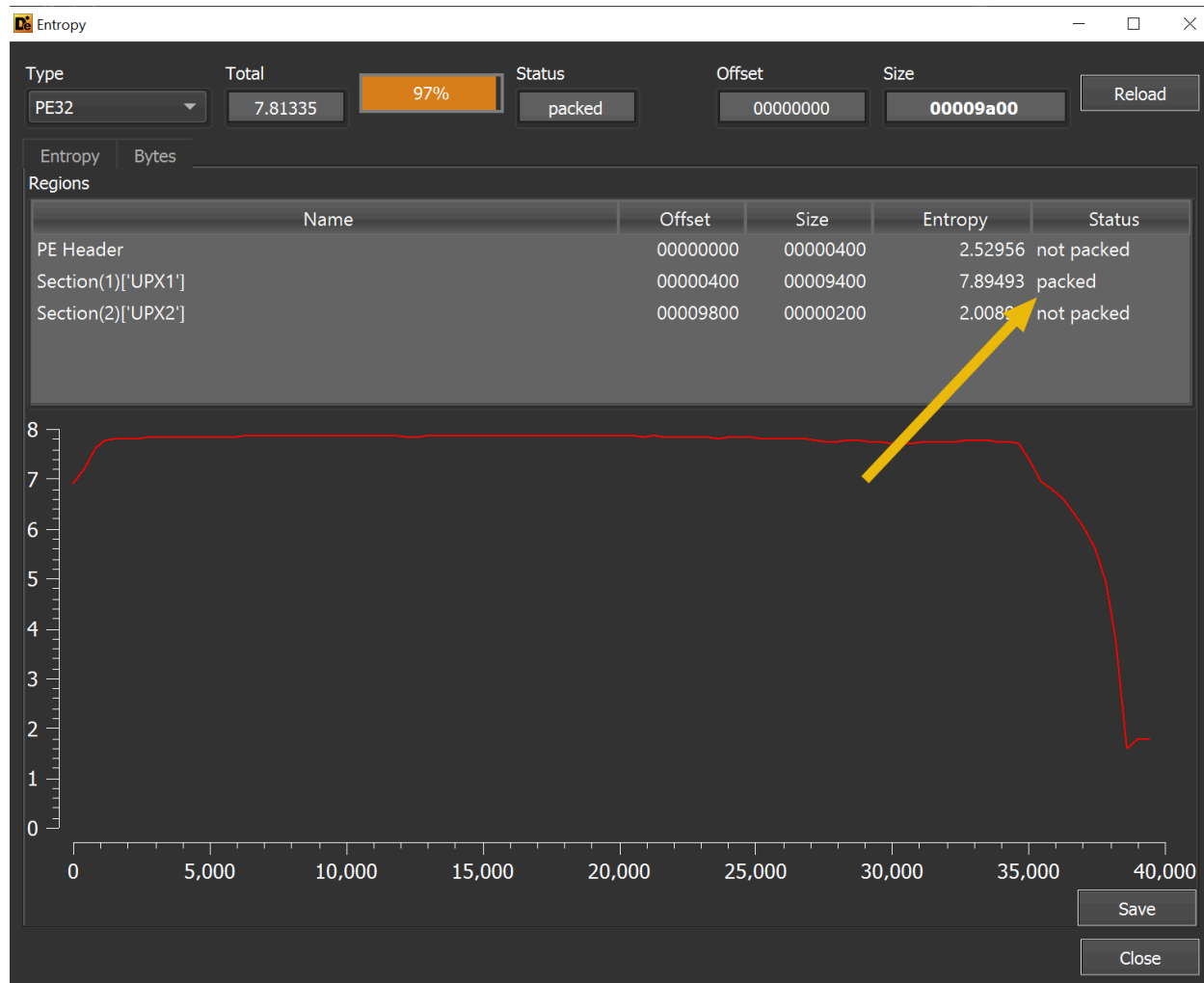


DetectItEasy is a tool that uses a collection of signatures of known packers to identify the one used on the selected file.

1. Right-click on the packed.exe executable file and select "Detect it easy (DIE)".
2. If DetectItEasy can identify the used Packer, it will display its "best guess" under this section.



1. You can also click on the "Entropy" button on the right side of the screen to open the "Entropy" window.
2. The Entropy window looks and determines how much entropy each section has. And as you can see, it has been determined that "Section 1" is "packed".



Entropy measures the level of disorder or uncertainty in a system. Packed malware tends to have high entropy due to the randomness of the packing process.

Another tool is PESTudio which lists information on PE files. This is important because there are specific Packers that will still leave clues.

1. From the Desktop, double-click on the "pestudio" icon.
2. Go through the menus, "File > Open", and open the packed.exe.
3. In the list of items on the left panel window, click on "sections (self-modifying)".

property	value	value
general		
name	UPX0	UPX1
md5	n/a	2039C9896F5F41EF88FE23E7...
entropy	n/a	7.895
file-ratio (97.40%)	n/a	96.10 %
raw-address	0x00000400	0x00000400
raw-size (38400 bytes)	0x00000000 (0 bytes)	0x00009400 (37888 bytes)
virtual-address	0x00001000	0x0000E000
virtual-size (98304 bytes)	0x0000D000 (53248 bytes)	0x0000A000 (40960 bytes)

Usually, the section names of a standard PE file are .text, .data and .rsrc. But in the output above, we can see that this is the section names have been changed. The strings UPX0, UPX1, and UPX2 are one of those identifiable pieces of information that Packers like UPX leave behind. Not all Packers change this value, but usually, this is the most accessible first place to look.

Because different Packer tools use various methodologies in packing executables, sometimes tools like DetectItEasy could not identify all types of Packers.

### Automated Unpacking

Once the Packer for a packed malware is identified, it is possible to use an unpacker to get back the original file. Some are readily available, like in the case of UPX, where you can use the same packing program for unpacking. For other commercial tools like Themida, you may have to rely on unpacker scripts made by 3rd parties.

Here is a short list of scripts that you could use for specific Unpacker tools:

- [Themida](#)
- [Enigma Protector](#)
- [Mpress unpacker](#)

If you chance upon malware packed with an obscure tool or modified to thwart available scripts, things might be more difficult.

Thankfully, there are services like [unpac.me](#) where you can upload a sample. It will try to identify and unpack the malware for you using custom unpacking and artefact extraction processes. While this service is excellent, it can still fail.

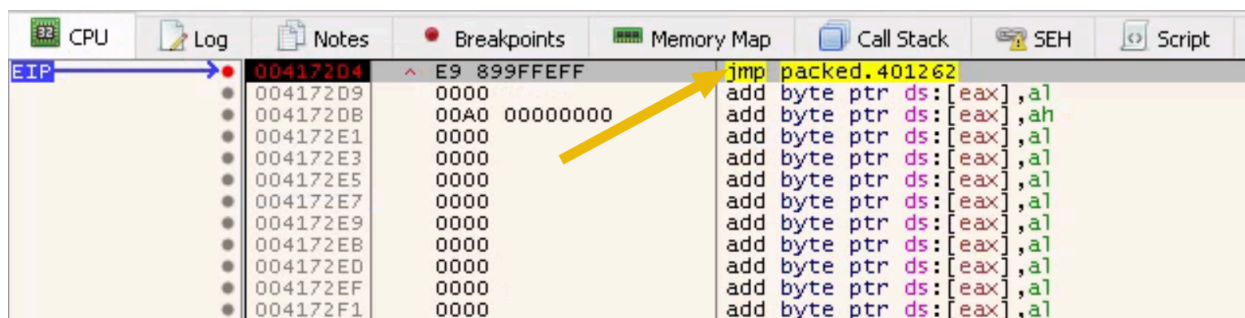
### Manual Unpacking and Dumping

Ultimately, the best way to unpack malware is to execute it.

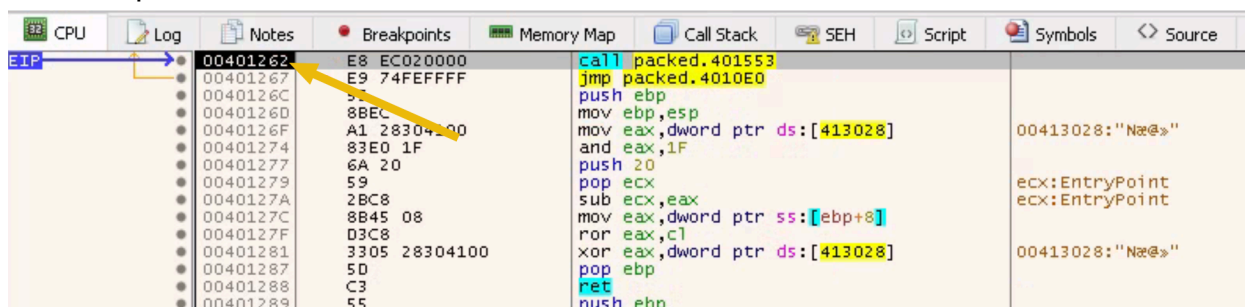
When a packed malware is executed, the wrapper or container code performs decryption and deobfuscation. Once fully unpacked, the malware can proceed with its true intentions. At this point, the Packer is useless, and we can thoroughly analyze the malware by debugging it while it's in memory using a debugger.

We can also dump this unpacked version as a separate executable.

- Open the packed.exe using x32dbg.
- Press CTRL+G to open the "Enter expression to follow..." window, input 004172D4 then press "OK".
- You will now be taken to the memory location 004172D4. This memory location is before the entry point to the legitimate part of the program.
- Press "F2" to set a breakpoint here, then press "F9" twice to move execution to this location.



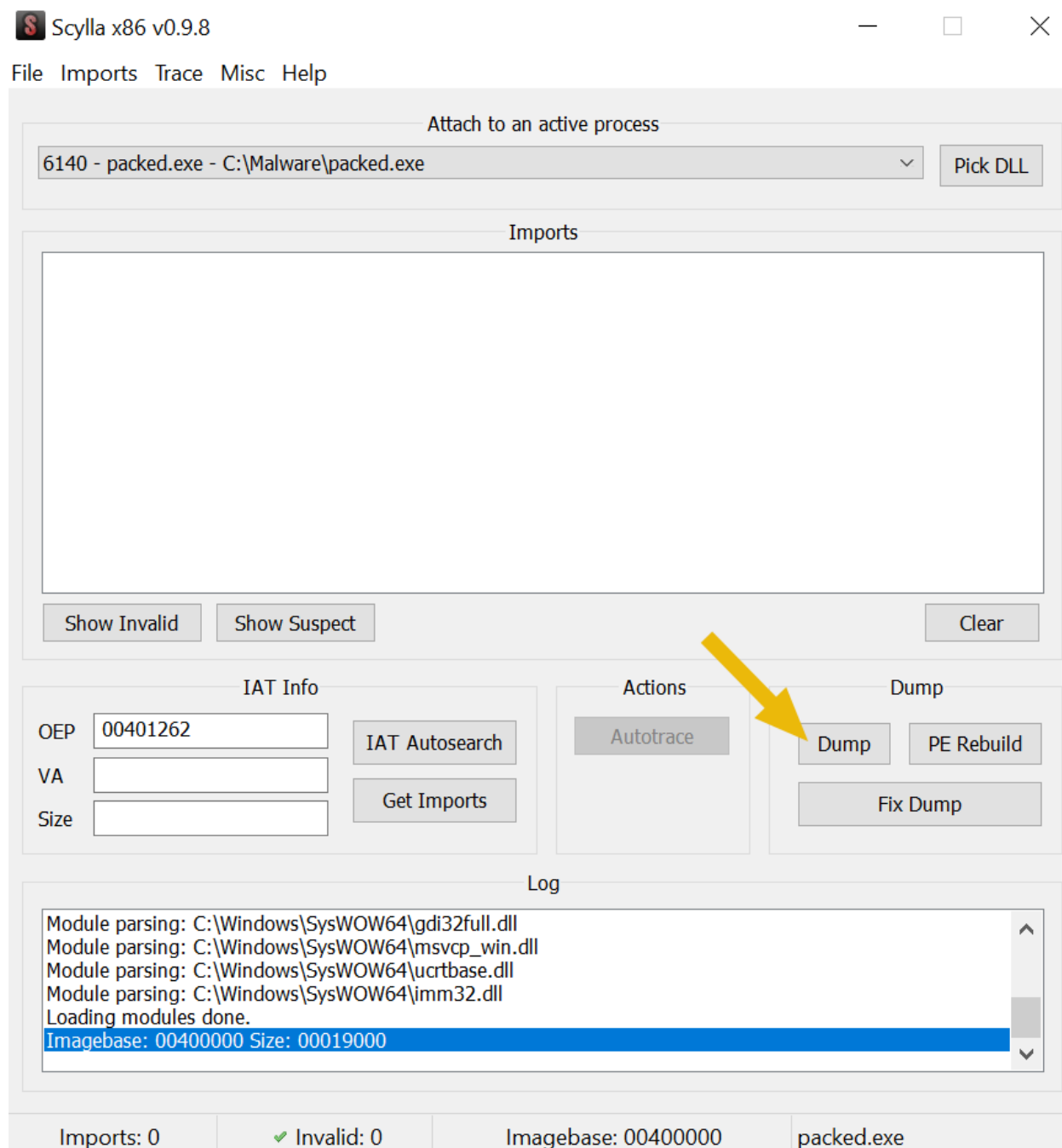
- Finally, press "F7" to jmp to the memory location 00401262. This location is the starting location of this legitimate program that the Packer goes to once it's fully unpacked.



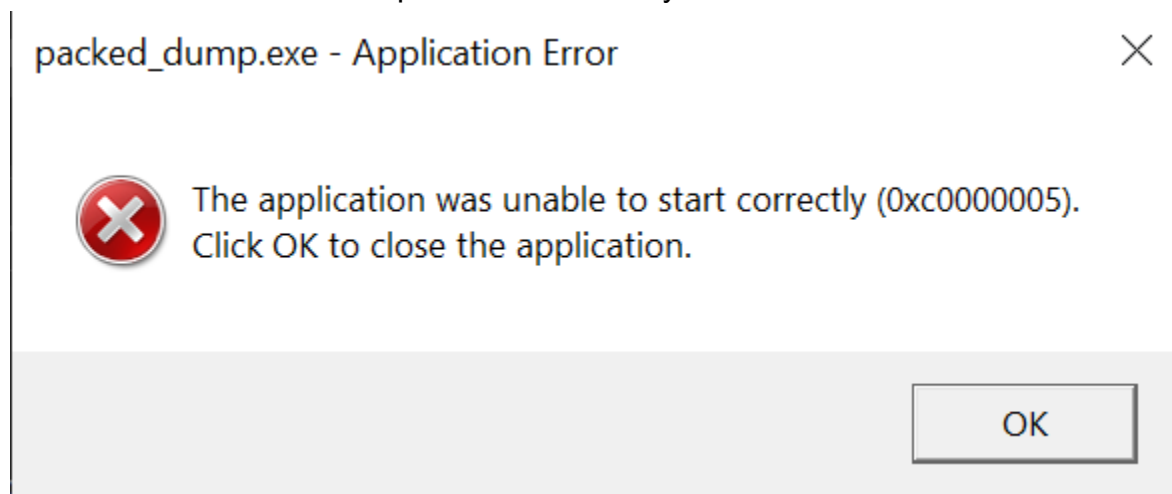
By now, you may be curious about the memory locations 004172D4 and 00401262, and their relevance to what we are doing. When this UPX-packed malware is executed and the execution location reaches 004172D4, it indicates that the legitimate part of the program has already been successfully unpacked and is now copied in memory at 00401262. As outlined in the next steps, we can now begin dumping from this location.

Note: Other packers have different approaches to unpacking the legitimate part of the program, so the steps above would not work on them.

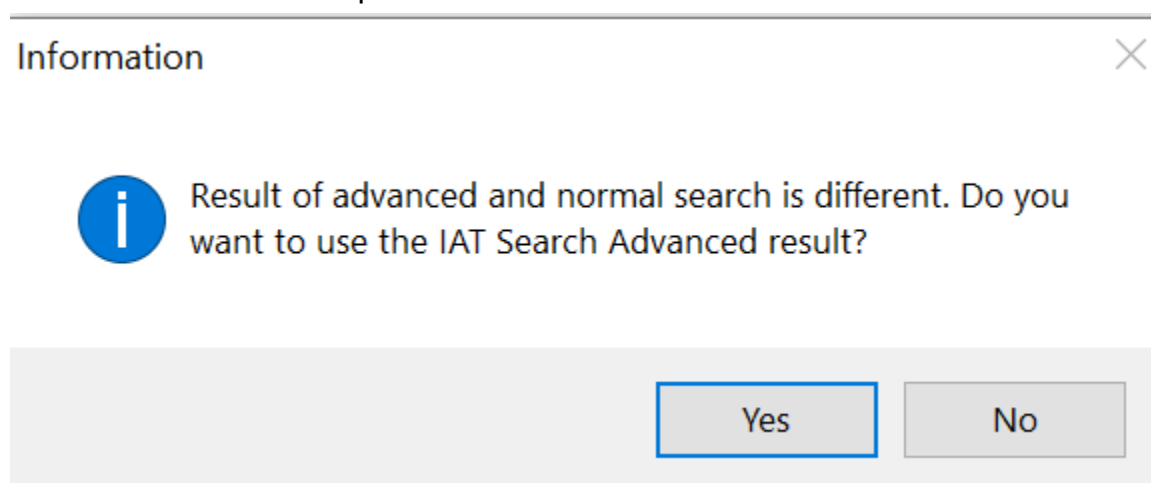
- To start dumping, go to the top bar menu and click "Plugins > Scylla".
- Scylla is a tool that can dump process memory to disk and fix and rebuild the Import Address Table (IAT). We'll use this to dump the unpacked legitimate part of the program to memory and fix it so it will have the updated memory locations from its DLL imports.
- Once the Scylla plugin window is open, click the "Dump" button.



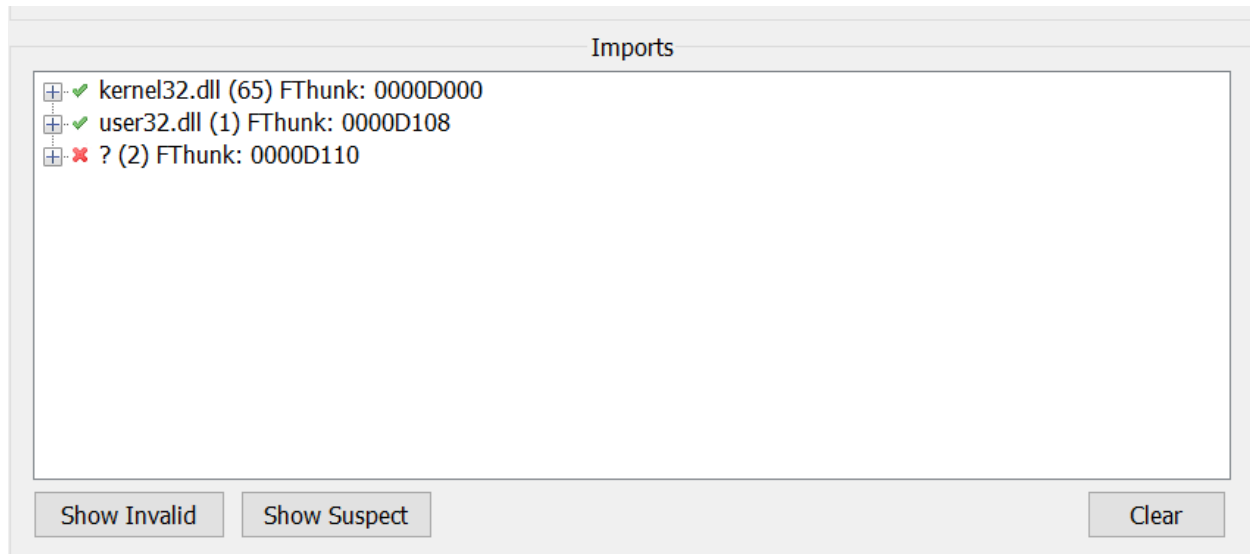
- Save this dumped version to the exact location as the original file.
- If you try to run this dumped version, an error will appear. This is because we have not fixed the import address table yet.



- Back in the Scylla plugin window, and click the "IAT Autosearch" button to start. This scans the memory of the process to locate the import address table.
- If a pop-up window asks if you want to use the IAT Search Advanced result, select "No". Then press "OK".

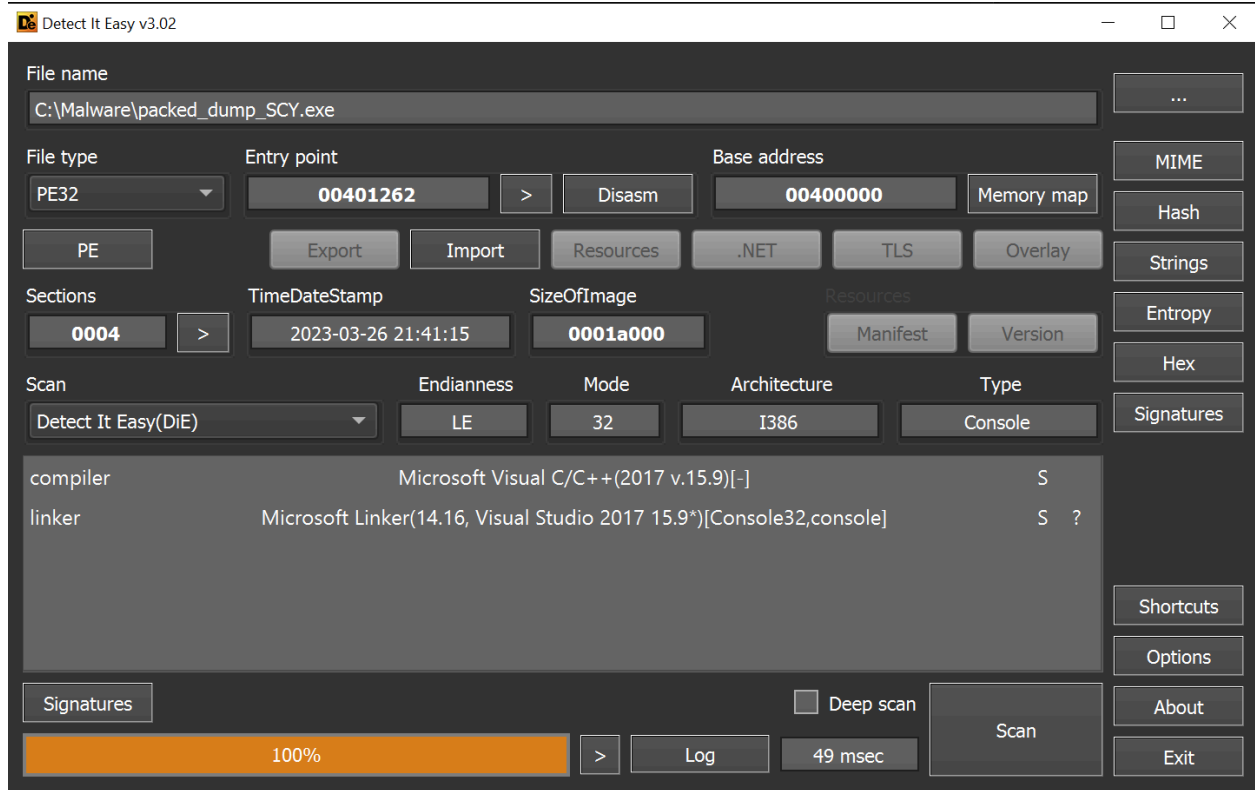


- At this point, the appropriate values related to the IAT are now computed. We can now click on the "Get Imports" button to update the Imports list section.



- This list will show all DLLs that are used by the program. The list is short because the program is simple. An actual malware will have a longer list.
- Sometimes, Scylla will find invalid entries; this is marked with the "X" icon next to it. These are safe to delete because the list already covers every DLL that the program uses. To delete these, go through each invalid entry, right-click, and select "Cut thunk". Once deleted, you should be left with valid entries with check mark icons.
- Click on the "Fix Dump" button, and select the previously generated dump file you created a few steps ago.
- A new file will be created in the same directory as the dumped file with the string "\_SCY" appended to its filename. If you execute this, the program will now work correctly.

You can confirm if unpacking was successful by inspecting the generated SCY file using DetectItEasy or PEStudio. Here's the output of DetectItEasy; you'll notice that it does not detect the Packer anymore:



With the malware unpacked, you can now continue investigating the malware. However, don't be fooled into thinking that unpacking packed malware is easy. UPX is the most basic packer tool and the easiest to unpack. Other Packers will be more difficult, primarily if it employs other anti-reverse engineering techniques.

\*\*\*\*\*

**Answer the questions below:**

**According to DetectItEasy, what is the version of the Microsoft Linker used for linking packed.exe?**

linker Microsoft Linker(14.16, Visual Studio 2017 15.9\*)[Console32,console]

Answer: 14.16

**According to pestudio, what is the entropy of the UPX2 section of packed.exe?**



property	value	value	value
general			
name	UPX0	UPX1	UPX2
md5	n/a	2039C9896F5F41EF88FE23E7...	B16AFED5950BEA8532F7C66...
entropy	n/a	7.895	2.006

Answer: 2.006

## Conclusion

In this room, we learned about the different anti-reverse engineering techniques used by malware today. However, it is crucial to recognize that this topic is expansive, and we've only scratched the surface of what could be explored. Still, we hope this has given you an idea about the techniques and inspired you to explore more.

In this room, we learned the following:

- Why malware authors use anti-reverse engineering techniques
- About different anti-reverse engineering techniques
- The methods on how to circumvent anti-reverse engineering using various tools
- How anti-reverse engineering techniques are implemented by reading the source code