Debugging

Debugging adds a dynamic, interactive layer to code analysis, offering a real-time view of malware behavior. It empowers analysts to confirm their discoveries, witness runtime impacts, and deepen their comprehension of the program execution. Uniting code analysis and debugging allows for a comprehensive understanding of the malware, leading to the effective exposure of harmful behavior.

We could deploy a debugger like x64dbg, a user-friendly tool tailored for analyzing and debugging 64-bit Windows executables. It comes equipped with a graphical interface for visualizing disassembled code, implementing breakpoints, examining memory and registers, and controlling the execution of programs.

Let's now navigate to the bottom of this section and click on "Click here to spawn the target system!". Then, let's RDP into the Target IP using the provided credentials. The vast majority of the actions/commands covered from this point up to end of this section can be replicated inside the target, offering a more comprehensive grasp of the topics presented.

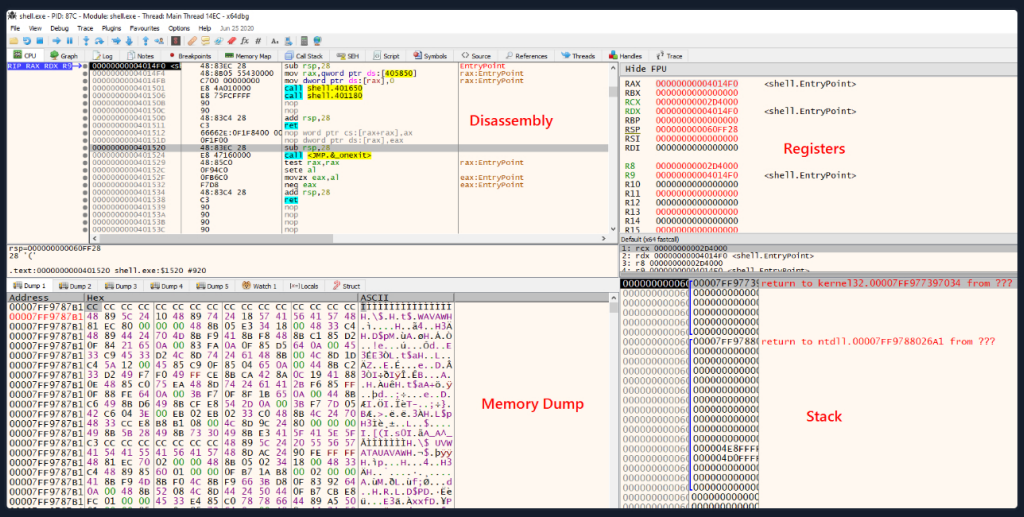
Debugging

```
MisaelMacias@htb[/htb]$ xfreerdp /u:htb-student /p:'HTB_@cademy_stdnt!' /v:[Target IP] /dynamic-res
```

Here's how to run a sample within x64dbg to familiarize with its operations.

- Launch x64dbg.
- At the top of the x64dbg interface, click the File menu.
- Select Open to choose the executable file we wish to debug.
- Browse to the directory containing the executable and select it.
- Optionally, command-line arguments or the working directory can be specified in the dialog box that appears.
- Click OK to load the executable into x64dbg.

Upon opening, the default window halts at a default breakpoint at the program's entry point.



Loading an executable into x64dbg reveals the disassembly view, showcasing the assembly instructions of the program, thereby aiding in understanding the code flow. To the right, the register window divulges the values of CPU registers,

Resources
Go to Questions

Table of Contents

Introduction To Malware & Malware Analysis
Prerequisites
Windows Internals
Static Analysis
Static Analysis On Linux
Static Analysis On Windows
Dynamic Analysis
Dynamic Analysis
Code Analysis
Code Analysis
Debugging
Creating Detection Rules
Creating Detection Rules
Skills Assessment
Skills Assessment

My Workstation

OFFLINE
Start Instance
∞ / 1 spawns left

shedding light on the program's state. Beneath the register window, the stack view displays the current stack frame, enabling the inspection of function calls and local variables. Lastly, on the bottom left corner, we find the memory dump view, providing a pictorial representation of the program's memory, facilitating the analysis of data structures and variables.

Simulating Internet Services

The role of `INetSim` in simulating typical internet services in our restricted testing environment is pivotal. It offers support for a multitude of services, encompassing `DNS`, `HTTP`, `FTP`, `SMTP`, among others. We can fine-tune it to reproduce specific responses, thereby enabling a more tailored examination of the malware's behavior. Our approach will involve keeping `InetSim` operational so that it can intercept any `DNS`, `HTTP`, or other requests emanating from the malware sample (`shell.exe`), thereby providing it with controlled, synthetic responses.

Note: It is highly recommended that we use your own VM/machine for running `InetSim`. Our VM/machine should be connected to VPN using the provided VPN config file that resides at the end of this section.

We should configure `INetSim` as follows.

Debugging

```
MisaelMacias@htb[/htb]$ sudo nano /etc/inetsim/inetsim.conf
```

The below need to be uncommented and specified.

Debugging

```
service_bind_address <Our machine's/VM's TUN IP>
dns_default_ip <Our machine's/VM's TUN IP>
dns_default_hostname www
dns_default_domainname iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
```

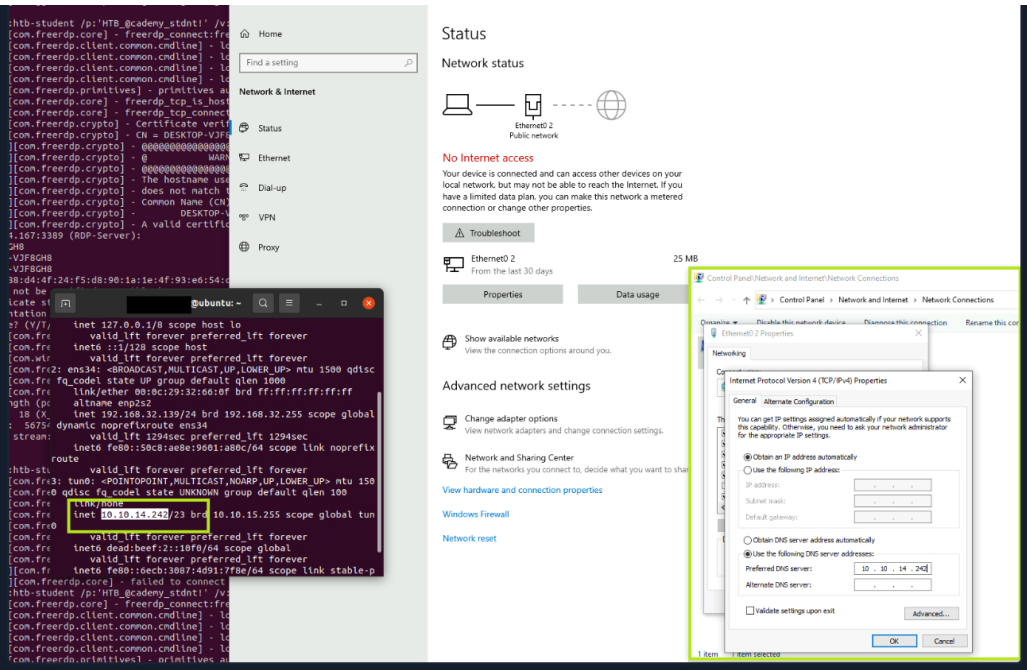
Initiating `INetSim` involves executing the following command.

Debugging

```
MisaelMacias@htb[/htb]$ sudo inetsim
INetSim 1.3.2 (2020-05-19) by Matthias Eckert & Thomas Hungenberg
Using log directory:      /var/log/inetsim/
Using data directory:     /var/lib/inetsim/
Using report directory:   /var/log/inetsim/report/
Using configuration file: /etc/inetsim/inetsim.conf
Parsing configuration file.
Configuration file parsed successfully.
=== INetSim main process started (PID 34711) ===
Session ID:     34711
Listening on:   0.0.0.0
Real Date/Time: 2023-06-11 00:18:44
Fake Date/Time: 2023-06-11 00:18:44 (Delta: 0 seconds)
 Forking services...
  * dns_53_tcp_udp - started (PID 34715)
  * smtps_465_tcp - started (PID 34719)
  * pop3_110_tcp - started (PID 34720)
  * smtp_25_tcp - started (PID 34718)
  * http_80_tcp - started (PID 34716)
  * ftp_21_tcp - started (PID 34722)
  * https_443_tcp - started (PID 34717)
  * pop3s_995_tcp - started (PID 34721)
  * ftps_990_tcp - started (PID 34723)
 done.
Simulation running.
```

A more elaborate resource on configuring `INetSim` is the following: https://medium.com/@xNymia/malware-analysis-first-steps-creating-your-lab-21b769fb2a64

Finally, the spawned target's DNS should be pointed to the machine/VM where `INetSim` is running.



Applying the Patches to Bypass Sandbox Checks

Given that sandbox checks hinder the malware's direct execution on the machine, we need to patch these checks to circumvent the sandbox detection. Here's how we can dodge sandbox detection checks while debugging with `x64dbg`. Several methods can lead us to the instructions where sandbox detection is performed. We will discuss a few of these.

By Copying the Address from IDA

During code analysis, we observed the sandbox detection check related to the registry key. We can extract the address of the first `cmp` instruction directly from `IDA`.

To find the address, let's revert to the IDA windows, open the first function we had renamed as `assumed_Main`, and look for the `cmp` instruction. To view the addresses, we can transition from graph view to text view by pressing the spacebar button.

This exposes the address (as highlighted in the below screenshot)

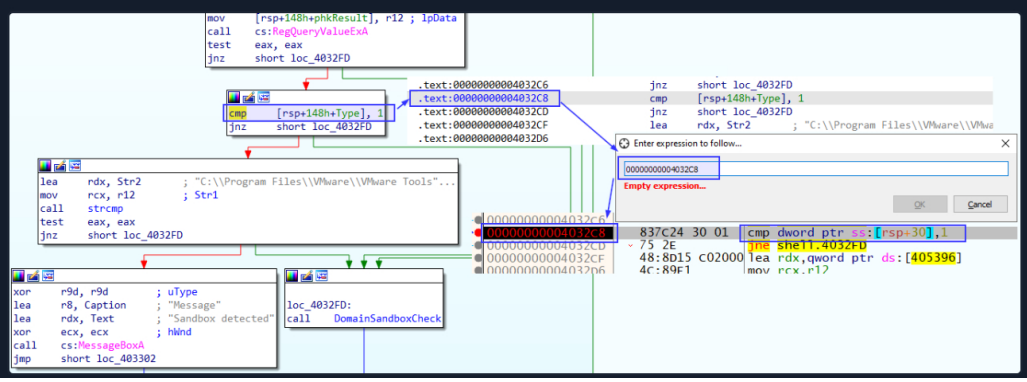
We can copy the address `00000000004032C8` from `IDA`.

Code: ida

```
.text:00000000004032C8                cmp     [rsp+148h+Type], 1
```

In `x64dbg`, we can right-click anywhere on the disassembly view (CPU) and select `Go to` > `Expression`. Alternatively, we can press `Ctrl+G` (go to expression) as a shortcut.

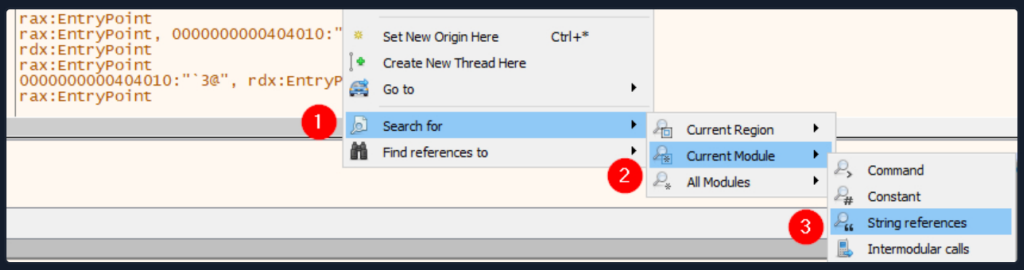
We can enter the copied address here, as shown in the screenshot. This navigates us to the comparison instruction where we can implement changes.



By Searching Through the Strings.

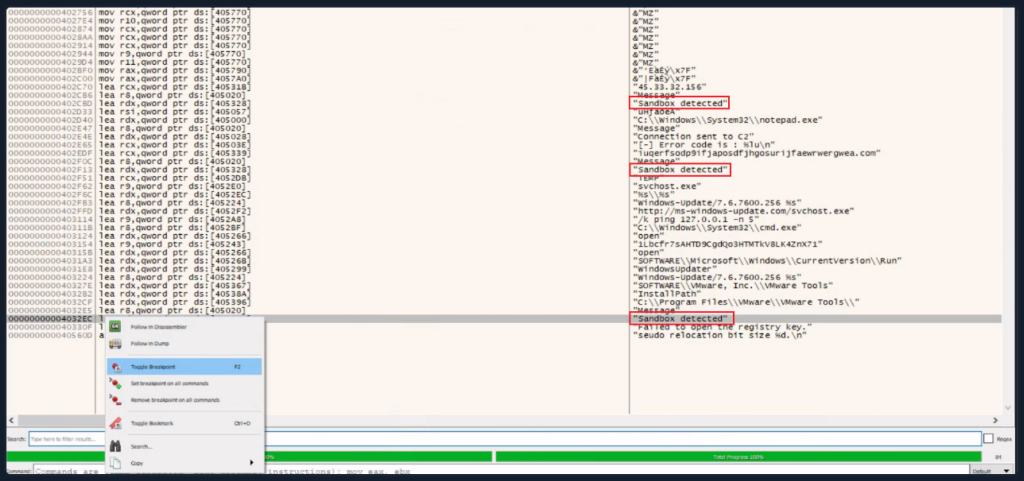
Let's look for `Sandbox detected` in the `String references`, and set a `breakpoint`, so that when we hit run, the execution should pause at this point.

To do this, first click on the `Run` button once and then right-click anywhere on the disassembly view, and choose `Search for` > `Current Module` > `String references`.

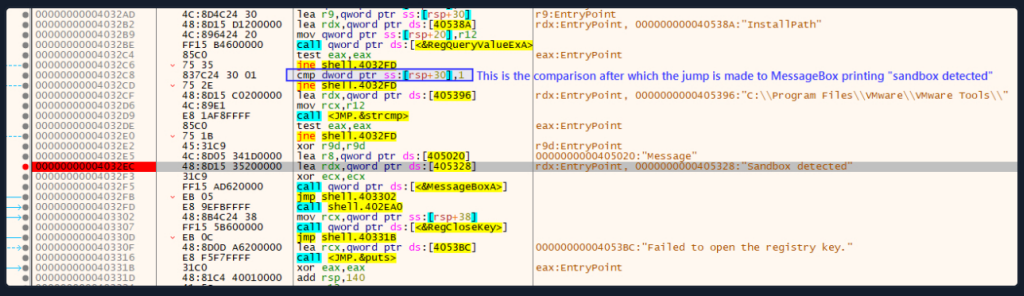


Next, we can add a breakpoint to mark the location, then study the instructions before this Sandbox `MessageBox` to discern how the jump was made to the instruction printing `Sandbox detected`.

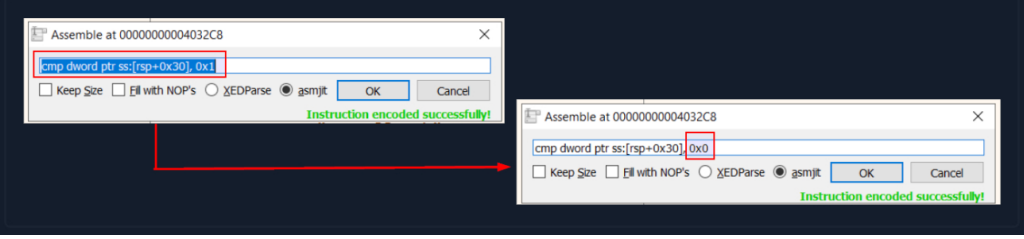
Let's start by adding a breakpoint at the last `Sandbox detected` string as follows.



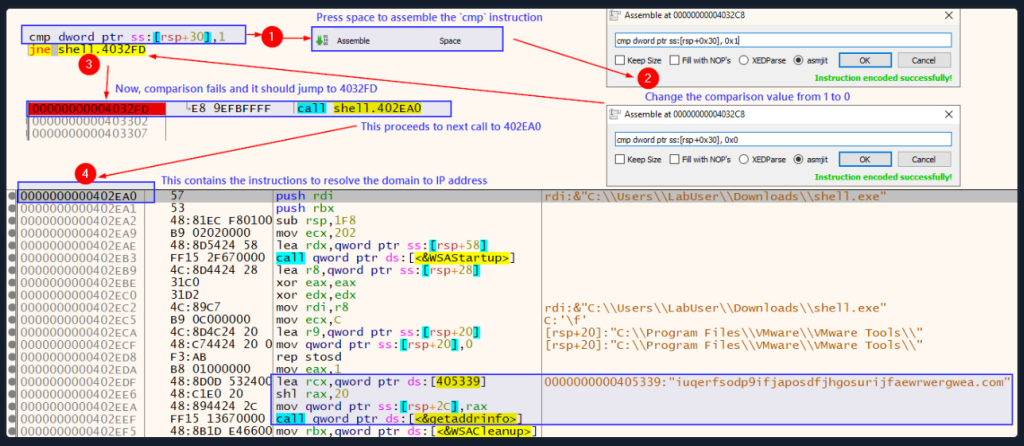
We can then double-click on the string to go to the address where the instructions to print `Sandbox detected` are located.



As observed, a `cmp` instruction is present above this `MessageBox` which compares the value with `1` after a registry path comparison has been performed. Let's modify this comparison value to match with `0` instead. This can be done by placing the cursor on that instruction and pressing `Spacebar` on the keyboard. This allows us to edit the assembly code instructions.

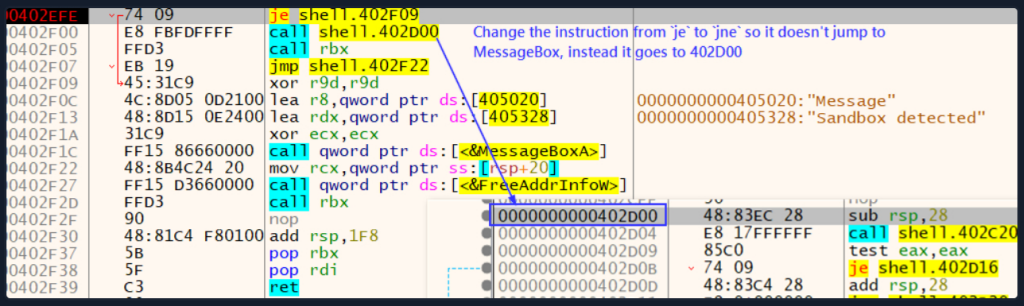


We can change the comparison value of `0x1` to `0x0`. Changing the comparison to `0` may shift the control flow of the code, and it should not jump to the address where `MessageBox` is displayed.

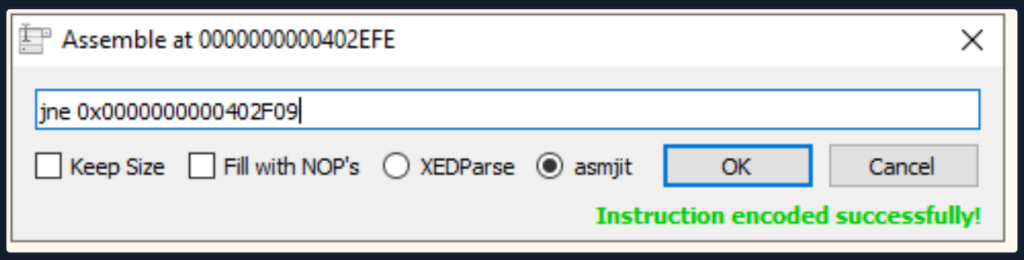


Upon clicking on `Run` in `x64dbg` or pressing `F9`, it won't hit the breakpoint for the first sandbox detection message code. This means that we successfully patched the instructions.

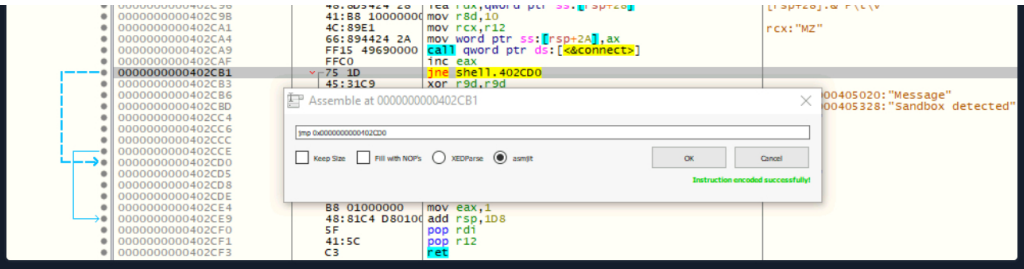
In a similar manner, we can add a breakpoint on the next sandbox detection function before it prints a `MessageBox` as well. To do that, the breakpoint should be placed at the `second to last Sandbox detected` string (`0000000000402F13`). If we double-click this string we will notice there's a `jump` instruction which we can skip, directing the execution flow to the next instruction that calls another function. That's exactly what we need – instead of the sandbox detection `MessageBox`, it jumps to another function.



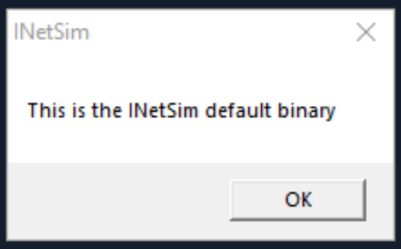
We can alter the instruction from `je shell.402F09` to `jne shell.402F09`.

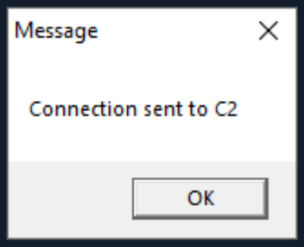


`shell.exe` performs sandbox detection by checking for internet connectivity. This section's target doesn't have internet connectivity. For this reason we should patch this sandbox detection method as well. We can do that by clicking on the first `Sandbox detected` string (`0000000000402CBD`) and patching the following instruction.

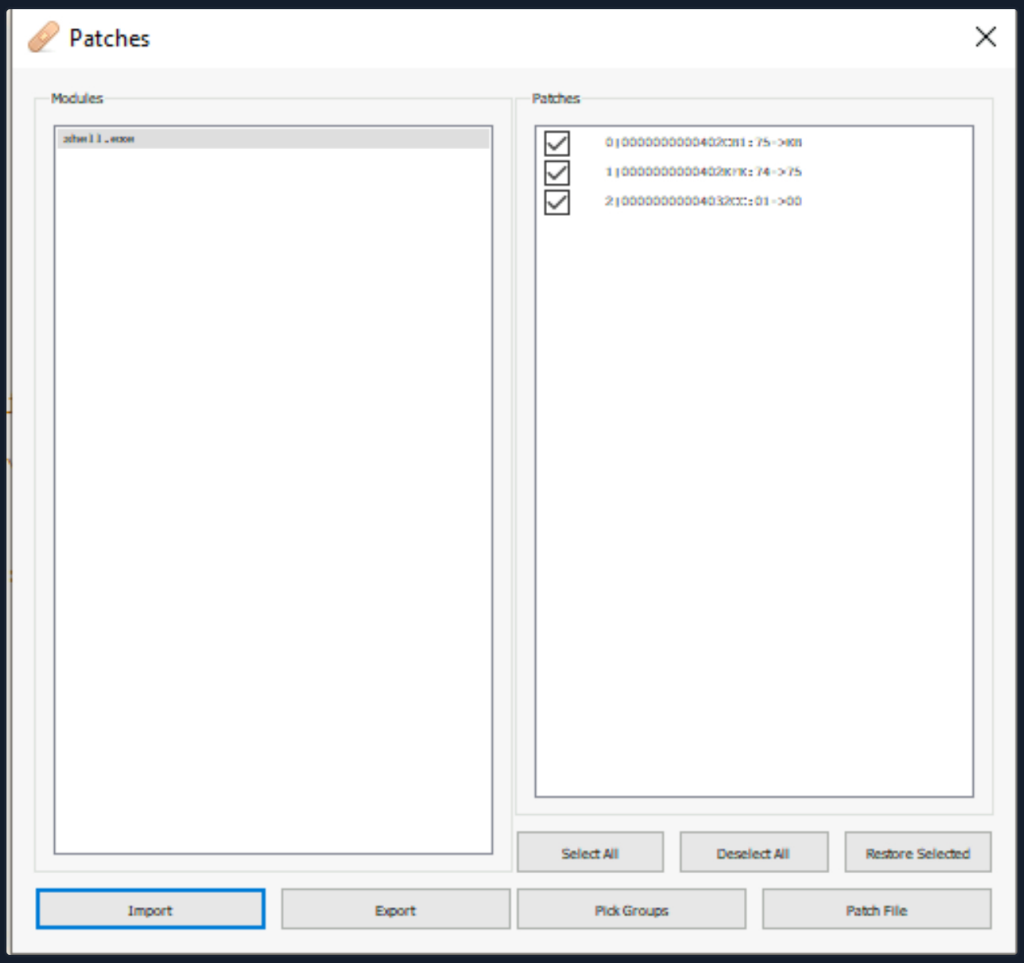


Now, when we press `Run`, the patched `shell.exe` proceeds further, downloads the default executable from `INetSim`, and executes it.





With the sandbox checks bypassed, the actual functionality is unveiled. We can save the patched executable by pressing `Ctrl+P` and clicking on `Patch File`. This action stores the patched file, which skips the sandbox checks.



We undertake this process to ensure that the next time we run the saved patched file, it executes directly without the

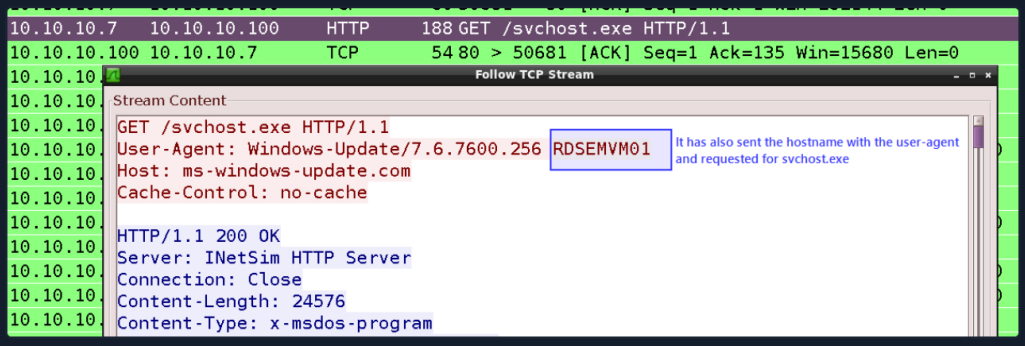
sandbox checks, and we can observe all the events in `ProcessMonitor`.

Analyzing Malware Traffic

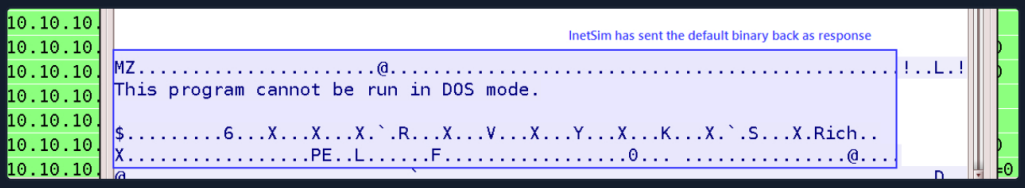
> Keep in mind that traffic analysis not only can be, but ideally should be incorporated as an integral part of Dynamic Analysis.

Let's now employ `Wireshark`, to capture and examine the network traffic generated by the malware. Be mindful of the color-coded traffic: red corresponds to client-to-server traffic, while blue denotes the server-to-client exchanges.

Examining the HTTP Request reveals that the malware sample appends the computer hostname to the user agent field (in this case it was `RDSEMVM01`).



When inspecting the HTTP Response, it becomes evident that `InetSim` has returned its default binary as a response to the malware.



The malware's request for `svchost.exe` solicits the default binary from `InetSim`. This binary responds with a `MessageBox` featuring the message: `This is the INetSim default binary`.

Additionally, DNS requests for a random domain and the address `ms-windows-update[.]com` were sent by the malware, with `INetSim` responding with fake responses (in this case `INetSim` was running on `10.10.10.100`).

Source	Destination	Protocol	Length	Info
10.10.10.7	10.10.10.100	DNS	105	Standard query A iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
10.10.10.100	10.10.10.7	DNS	121	Standard query response A 10.10.10.100
10.10.10.7	10.10.10.100	DNS	81	Standard query A ms-windows-update.com
10.10.10.100	10.10.10.7	DNS	97	Standard query response A 10.10.10.100

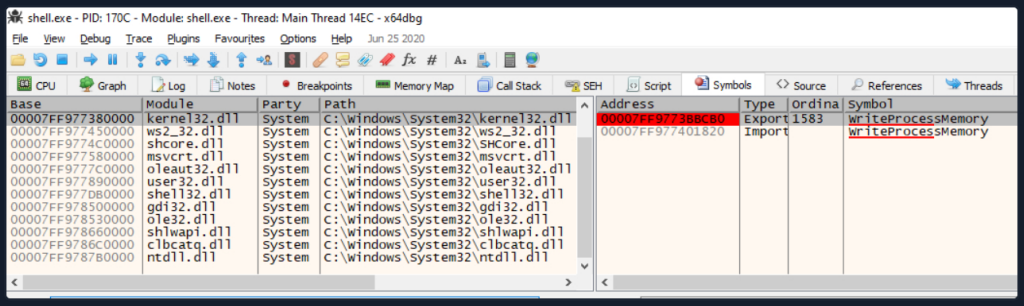
Analyzing Process Injection & Memory Region

On the journey of code analysis, we discovered that our executable performs process injection on `notepad.exe` and displays a `MessageBox` stating `Connection sent to C2`.

To probe deeper into the process injection, we propose setting breakpoints at WINAPI functions `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. These breakpoints will allow us to scrutinize the content held in the registers during the process injection. Here's the procedure to set these breakpoints:

- Access the `x64dbg` interface and navigate to the `Symbols` tab, located at the top.
- In the symbol search box, search for the desired DLL name on the left and function names, such as `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`, on the right within the `Kernel32.dll` DLL.
- As the function names materialize in the search results, right-click and select `Toggle breakpoint` from the context menu for each function. An alternative shortcut is to press `F2`.

Executing these steps sets a breakpoint at each function's entry point. We'll replicate these steps for all the functions we intend to scrutinize.



After setting breakpoints, we press `F9` or select `Run` from the toolbar until we reach the breakpoint for `WriteProcessMemory`. Up until this moment, `notepad` has been launched, but the `shellcode` has not yet been written into notepad's memory.

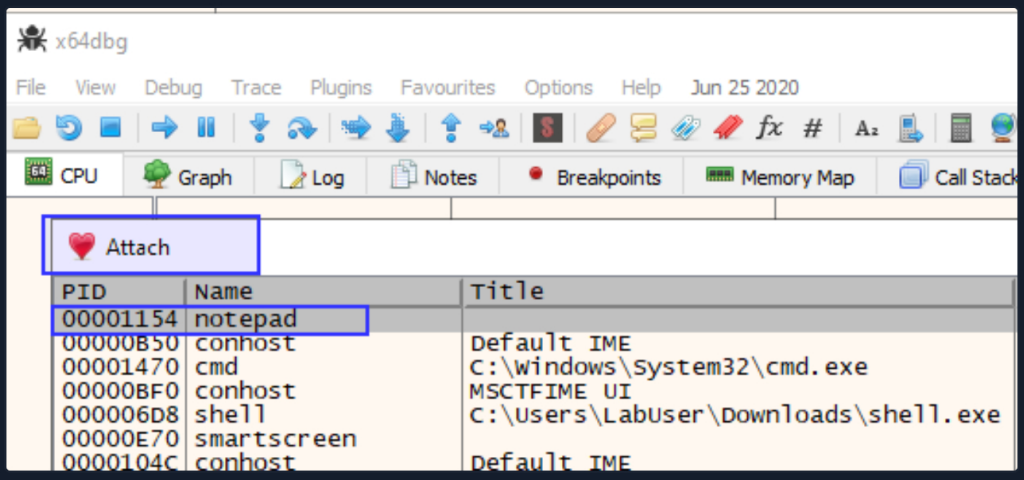
Attaching Another Running Process In x64dbg

In order to delve further, let's open another instance of `x64dbg` and attach it to `notepad.exe`.

- Start a new instance of `x64dbg`.
- Navigate to the `File` menu and select `Attach` or use the `Alt + A` keyboard shortcut.
- In the `Attach` dialog box, a list of running processes will appear. Choose `notepad.exe` from the list.
- Click the `Attach` button to begin the attachment process.

Once the attachment is successful, `x64dbg` initiates the debugging of the target process, and the main window displays the assembly code along with other debugging information.

Now, we can establish breakpoints, step through the code, inspect registers and memory, and study the behavior of the attached notepad.exe process using `x64dbg`.



The 2nd argument of `WriteProcessMemory` is `lpBaseAddress` which contains a pointer to the base address in the specified process to which data is written. In our case, it should be in the `RDX` register.

C++ Copy

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

Parameters

`[in] hProcess`

A handle to the process memory to be modified. The handle must have PROCESS_VM_WRITE and PROCESS_VM_OPERATION access to the process.

`[in] lpBaseAddress`

A pointer to the base address in the specified process to which data is written. Before data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for write access, and if it is not accessible, the function fails.

When invoking the `WriteProcessMemory` function, the `rdx` register holds the `lpBaseAddress` parameter. This parameter represents the address within the target process's address space where the data will be written.

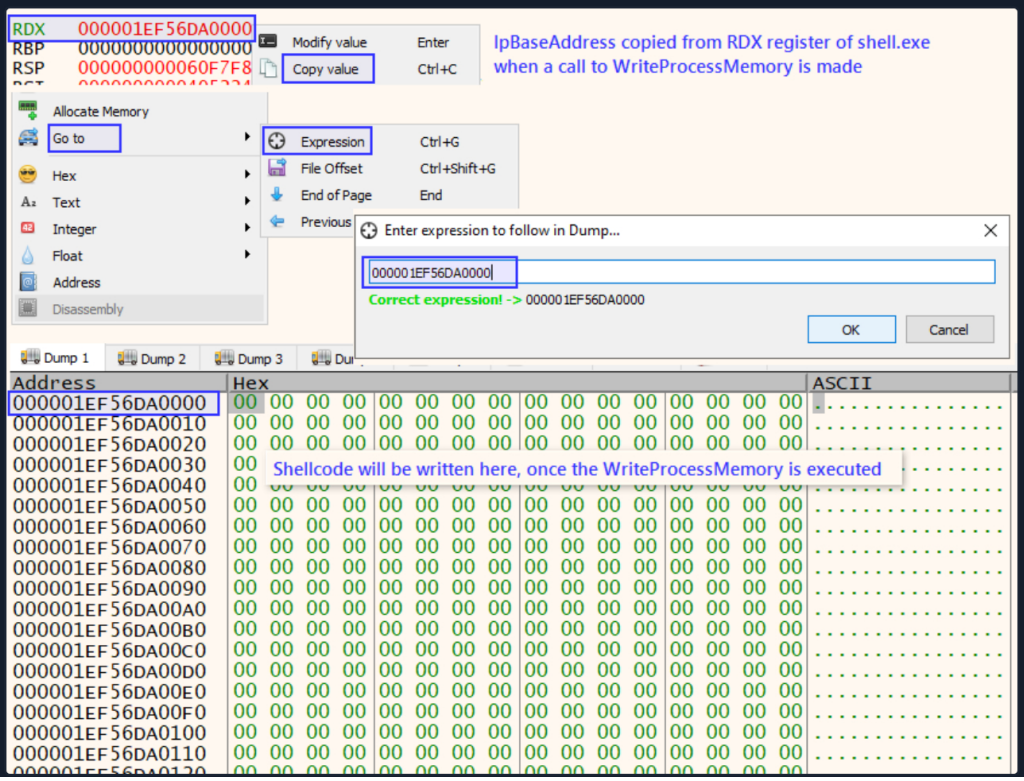
We aim to examine the registers when the `WriteProcessMemory` function is invoked in the `x64dbg` instance running the `shell.exe` process. This will reveal the address within `notepad.exe` where the shellcode will be written.

```
Hide FPU
RAX   000001EF56DA0000
RBX   000001EF56DA0000
RCX   0000000000000648      L'ٍ'
RDX   000001EF56DA0000
RBP   0000000000000000
RSP   000000000060F7F8
RSI   0000000000405224      "Windows-Update/7.6.7600.256 %s"
RDI   000000000060FAA0

R8    000000000060F8D3
R9    00000000000001CD      L'Ă'
R10   0000000000000000
R11   0000000000000246      L'Ɇ'
R12   0000000000000648      L'ٍ'
R13   0000000000B617A0      &"C:\\Users\\LabUser\\Downloads\\shell.exe"
R14   0000000000000000
R15   0000000000000000

RIP   00007FF9773BBCB0      <kernel32.WriteProcessMemory>
```

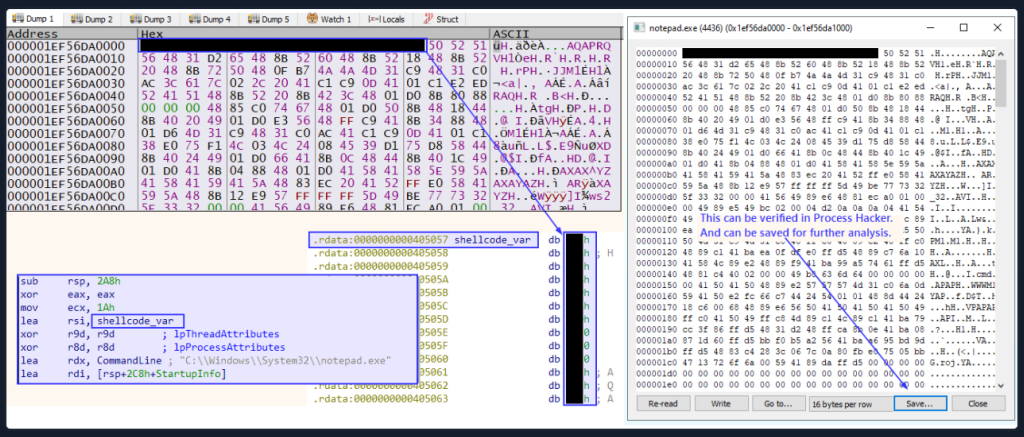
We copy this address to examine its content in the memory dump of the attached `notepad.exe` process in the second `x64dbg` instance.

We now select `Go to` > `Expression` by right-clicking anywhere on the memory dump in the second `x64dbg` instance running `notepad.exe`.

With the copied address entered, the content at this address is displayed (by right-clicking on the address and choosing `Follow in Dump` > `Selected Address`), which currently is empty.



Next, we execute shell.exe in the first `x64dbg` instance by clicking on the `Run` button. We observe what is inscribed into this memory region of `notepad.exe`.



Following its execution, we identify the injected `shellcode`, which aligns with what we discovered earlier during code analysis. We can verify this in `Process Hacker` and save it to a file for subsequent examination.

Waiting to start...

Enable step-by-step solutions for all questions

Questions

Answer the question(s) below to complete this Section and earn cubes!

Download VPN Connection File

Target(s): Click here to spawn the target system!

RDP to with user "htb-student" and password "HTB_@cademy_stdnt!"

+ 2 Reproduce all the debugging procedures mentioned in this section and provide the hidden shellcode-related hex values from the final screenshot as your answer. Remove all spaces.

FC4883E4F0E8C0000000415141

Submit

Previous Next

Mark Complete & Next

Powered by HACKTHEBOX