Module 1 - Introduction to Anti-VM and Anti-Sandbox Techniques

Module Overview

In Module 1, we'll introduce the concept of anti-VM (anti- Virtual Machine) and anti-sandbox techniques. We'll discuss how malware uses these techniques, and most importantly, how you (the reverse-engineer) can identify and circumvent these techniques.

Part 1: Enumerating File and Registry Artifacts

Tip: Before we begin running malware on the VM, make sure to take a VM snapshot! You can do this in both VMWare Workstation and VirtualBox by right-clicking your VM in the main menu of your hypervisor and selecting the "take snapshot" option or similar.

Registry

Virtual machines and sandboxes often leave behind telltale artifacts - like specific files, directories, and registry keys/values. Malware commonly checks for the presence of these artifacts as a way to detect whether it's running in a virtualized or sandboxed environment. Here's how you can observe this technique in practice:

1. Start procmon.exe in your VM. *(Tip: ProcMon automatically starts a trace as soon as its started.)*
2. Execute the malware executable.
3. Once the malware has completed execution (or has crashed, in this case), stop the ProcMon trace.
4. In ProcMon, filter on the malware process(es). You can do this with the **Tools -> Process Tree** function. Select the target process, right click on it, and select "**Add process to Include filter**". Optionally, select "**Add process and children to Include filter**".
5. Examine the **RegOpenKey** events (you can optionally filter on these if this is easier). Notice how the malware seems to be searching for an interesting VMware or VirtualBox registry keys. *This will look different depending on if you are running VirtualBox or VMware hypervisor.*



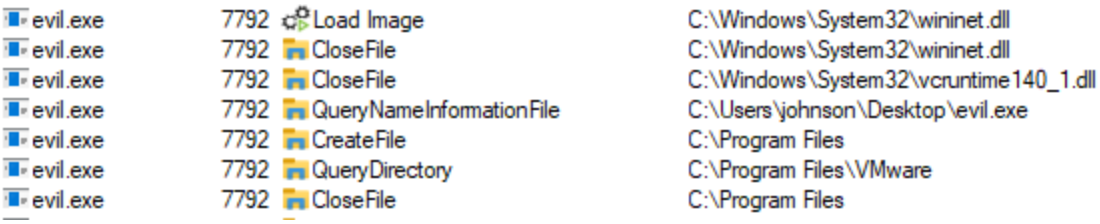
This malware sample seems to be attempting to identify if it is executing in a VM. Let's see if we can "trick" the malware into executing further:

1. Modify the suspect registry key/value. Rename it to something innocuous.
    - *Tip: You can modify this registry key by right-clicking the key in Procmon, and selecting **Jump To**. This will open up the Windows registry editor. Then you can modify the **VMware, Inc.** key name.*
    - *Note that when you rename hypervisor-related registry keys and files on the system, it may affect your VM in minor ways. For example, you may not be able to suspend your VM. This doesn't matter much as you can simply revert the VM snapshot later.*
2. Re-run the malware. Did the malware get any further in execution?

Files & Directories

Similarly, malware often attempts to enumerate files and directories on disk for hypervisor-related artifacts. Let's take a closer look at this:

1. Launch procmon.exe in your VM. *(Tip: Make sure you have no active filters set!)*.
2. Execute the malware executable in your VM.
3. Once the malware has completed execution (or has crashed, in this case), stop the ProcMon trace.
4. In ProcMon, filter on the malware process.
5. Examine the **CreateFile** events (you can optionally filter on these if this is easier). Notice how the malware seems to be searching for an interesting directory on disk. *Note that this will look different depending on if you are running VirtualBox or VMWare hypervisor.*



Again, let's try to convince the malware to continue execution:

1. Modify the suspect directory paths on your VM. You can simply rename these directories to something innocuous.
2. Execute the malware again. Did you get any further in execution?

Finally, let's open the malware executable in IDA and see what this behavior looks like as assembly code. This is good practice as you'll often be using a disassembler to analyze the behaviors of malware, especially malware that uses evasion techniques.

1. Open the malware file in IDA.
2. Navigate to the Imports tab, and search for the **FindFirstFileW** function.
3. Double click on that function to identify where it is used in the code.
4. Find the cross-references to the function *(Tip: Click on the function name and press the X key on your keyboard).*
5. Examine the code and investigate how the malware is enumerating the registry keys and files previously discussed.

Tip: The tools VBoxCloak and VMwareCloak (for VirtualBox and VMware Workstation, respectively) automate a lot of what we accomplished above. You can find these scripts on Github:

https://github.com/d4rksystem/VBoxCloak
https://github.com/d4rksystem/VMwareCloak

Part 2: Enumerating System Hardware

Malware often queries hardware and system specs to detect if it's running in a VM or sandbox. Common targets include CPU details, available RAM, network adapters, and other indicators that suggest it's being analyzed in a controlled lab environment. Let’s dive into some of these detection techniques.

After completion of the previous steps, you may have noticed that the malware terminated itself, even after you modified the hypervisor-related directories and registry keys. Let's see if we can figure out what's happening here:

1. Open the malware sample in IDA.
2. Examine the code after the **FindFirstFileW** calls and registry-related functions (such as **RegOpenKeyExW**) we just discussed. You should spot some interesting functions such as: **GlobalMemoryStatusEx**, **GetSysInfo**, and **GetDiskFreeSpaceEx**.

```
.text:0000000140001662                lea     rcx, [rsp+310h+Buffer] ; lpBuffer
.text:0000000140001667                mov     [rsp+310h+Buffer.dwLength], 40h ; '@'
.text:000000014000166F                call    cs:GlobalMemoryStatusEx
.text:0000000140001675                test    eax, eax
.text:0000000140001677                jz      short loc_1400016A7
.text:0000000140001679                mov     rdx, [rsp+310h+Buffer.ullTotalPhys]
.text:000000014000167E                shr     rdx, 1Eh
.text:0000000140001682                cmp     rdx, 4
.text:0000000140001686                jnb     short loc_140001699
```

3. Take special note of the **cmp** instructions, which compare two values. These instructions are usually followed by some sort of jump instruction, such as **jnz** (**jump if not zero**). For example, after the call to **GlobalMemoryStatusEx**, there is a **cmp** instruction which

compares the value **4** to the value in **rdx**. **rdx** is currently storing the returned value from **GlobalMemoryStatusEx**:

```
mov     rdx, [rsp+310h+Buffer.ullTotalPhys]
shr     rdx, 30
cmp     rdx, 4
jnb     short loc_140001699
```

3. Can you identify what the malware is looking for here? What is required for the malware to continue executing?
4. Inspect the calls to **GetSysInfo** and **GetDiskFreeSpaceEx** as well. What is the malware looking for?

The malware requires the following system hardware conditions to be met:

- At least 4GB of RAM:

```
mov     rdx, [rsp+310h+Buffer.ullTotalPhys]
shr     rdx, 30
cmp     rdx, 4
jnb     short loc_140001699
```

- At least 2 processor (CPU) cores:

```
loc_1400016B3:          ; lpSystemInfo
lea     rcx, [rbp+210h+SystemInfo]
call    cs:GetSystemInfo
mov     edx, [rbp+210h+SystemInfo.dwNumberOfProcessors]
cmp     edx, 2
jnb     short loc_1400016D3
```

- A hard disk size of at least 80GB:

```
mov     rdx, qword ptr [rsp+310h+TotalNumberOfBytes]
shr     rdx, 1Eh
cmp     rdx, 50h
jnb     short loc_14000171A
```

To circumvent these checks, the easiest method is to simply modify your virtual machine settings to ensure you meet the malware's requirements. This can all be configured in you VM settings menu.

- For VMWare Workstation, the settings can be accessed in **VM -> Settings**.
- For VirtualBox, the settings can be accessed in **VirtualBox Manager -> Settings**.

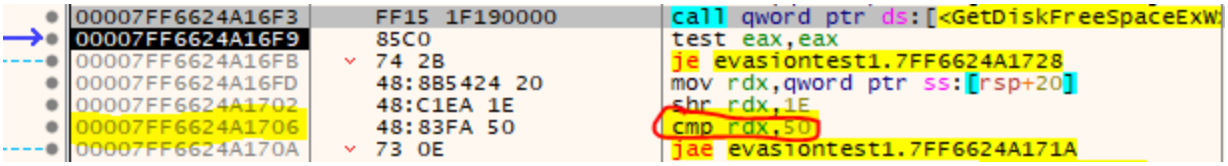
Instead of modifying your VM settings, you can also modify (or, patch) certain instructions or return values to "fake" these hardware configurations, which is what we'll do in this workshop. Let's talk about this.

Binary Patching in x64Dbg

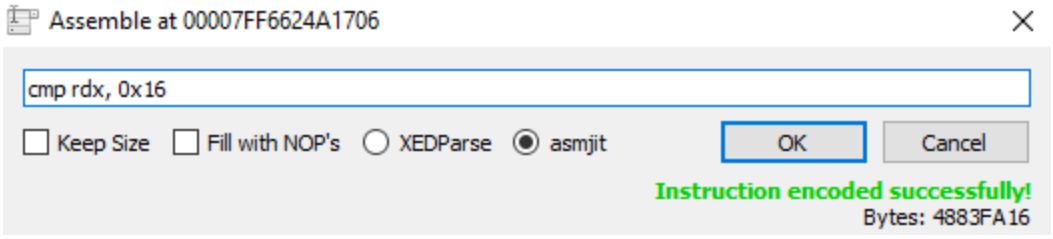
Knowing how to patch a malware binary in a disassembler or debugger is very important, especially when dealing with evasion checks. You're basically modifying the executable code at runtime or statically so that it skips over or neutralizes anti-analysis logic (like VM detection, debugger checks, etc.).

To bypass the hardware detection techniques in this sample, we're going to modify the malware's code dynamically in x64dbg.

1. Open the malware executable in x64dbg *(Tip: Drag the malware exe file to the x64dbg icon in your VM.)*
2. Select "**Run to user code**" to skip to the entry point of the malware's code.
3. For our VM, we'll need to fake the hard disk size (right now, our disk is less than 80GB - unless you manually changed it!)
4. Place a breakpoint on **GetDiskFreeSpaceExW**
   - Hint: Type "*bp <function_name>*" in the command bar to set a breakpoint on a function. (Example: bp **GetDiskFreeSpaceExW**).
5. Select **Debug -> Run** button to run the malware until the breakpoint.
6. After the breakpoint is hit, select **Debug -> Run to user code** to allow the function to return.
7. Find the **cmp** (comparison) operation that compares the return value from **GetDiskFreeSpaceExW** with **50h**:



8. **Right-click** this instruction and select **Assemble** to modify it.
9. Change the 0x50 to 0x16 (or, 10 GB):



10. Finally, let's save our patched binary to disk. This can be done with **File -> Patch File -> Patch File** in x64dbg. Save the newly patched file as something like "malware-patched.exe".
11. Close the malware in x64dbg and open the newly patched file in x64dbg and **Run** it. What happens?

You should have bypassed the hardware checks and now see something like "Process X found!".

Part 3: Enumerating Process Name Artifacts

Malware often enumerates the list of running processes on the host to identify processes that could be related to hypervisors or a sandbox environment. It is also common for malware to try to identity if malware analysis tools are running on the host.

A good example of this is the process check functionality of the Bumblebee malware family. If any of the below processes are running on the victim system, Bumblebee terminates itself:

```
__int64 check_processes_against_hardcoded_malware_tools()
{
  __int64 v0; // rbx
  __int64 result; // rax
  PCWSTR psz2[32]; // [rsp+20h] [rbp-E0h]

  v0 = 0i64;
  psz2[0] = L"ollydbg.exe";
  psz2[1] = L"ProcessHacker.exe";
  psz2[2] = L"tcpview.exe";
  psz2[3] = L"autoruns.exe";
  psz2[4] = L"autorunsc.exe";
  psz2[5] = L"filemon.exe";
  psz2[6] = L"procmon.exe";
  psz2[7] = L"regmon.exe";
  psz2[8] = L"procexp.exe";
  psz2[9] = L"idaq.exe";
  psz2[10] = L"idaq64.exe";
  psz2[11] = L"ImmunityDebugger.exe";
  psz2[12] = L"Wireshark.exe";
  psz2[13] = L"dumpcap.exe";
  psz2[14] = L"HookExplorer.exe";
  psz2[15] = L"ImportREC.exe";
  psz2[16] = L"PETools.exe";
  psz2[17] = L"LordPE.exe";
  psz2[18] = L"SysInspector.exe";
  psz2[19] = L"proc_analyzer.exe";
  psz2[20] = L"sysAnalyzer.exe";
  psz2[21] = L"sniff_hit.exe";
  psz2[22] = L"windbg.exe";
  psz2[23] = L"joeboxcontrol.exe";
  psz2[24] = L"joeboxserver.exe";
  psz2[25] = L"joeboxserver.exe";
  psz2[26] = L"ResourceHacker.exe";
  psz2[27] = L"x32dbg.exe";
  psz2[28] = L"x64dbg.exe";
  psz2[29] = L"Fiddler.exe";
  psz2[30] = L"httpdebugger.exe";
  while ( 1 )
  {
    result = compare_proc_names(psz2[v0]);
    if ( result )
      break;
    if ( ++v0 >= 31 )
      return result;
  }
  return 1i64;
}
```

Source: https://www.proofpoint.com/us/blog/threat-insight/bumblebee-is-still-transforming

Is your VM running any of these processes?

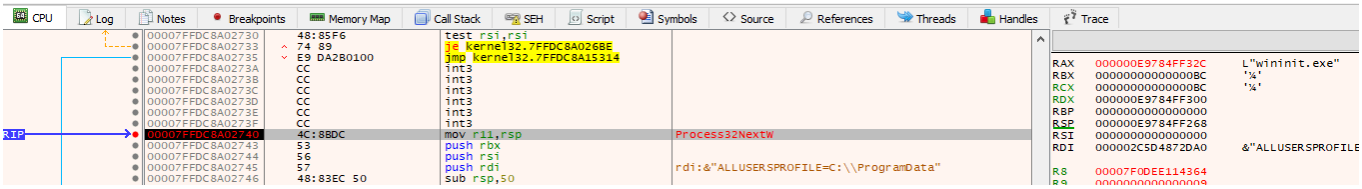
The malware we are investigating today is trying to identify suspect processes running and may terminate itself if any of the processes are detected. Let's investigate this further:

1. You should still have your malware sample opened in IDA. If not, re-open it.
2. Find the call to the **CreateToolhelp32Snapshot** function.
3. Notice anything interesting before and after this function call?

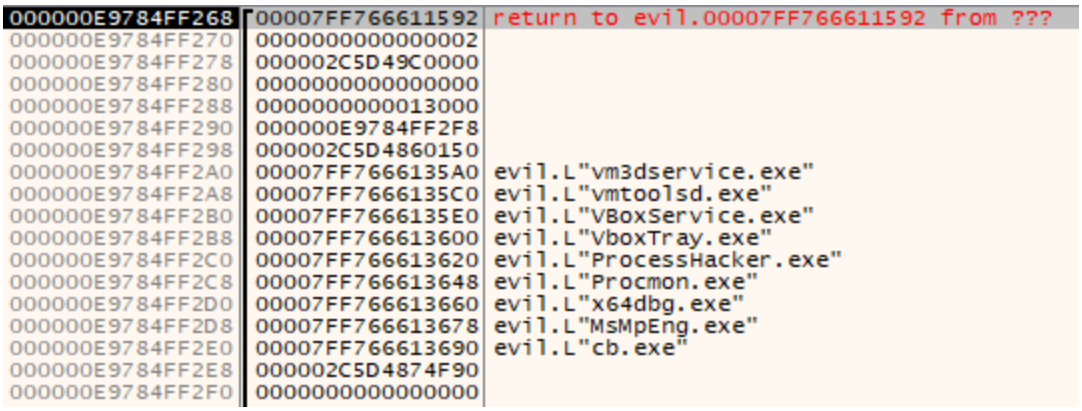
To enumerate processes, malware normally calls the windows API functions **CreateToolhelp32Snapshot**, followed by **Process32First** and **Process32Next**. **CreateToolhelp32Snapshot** creates a "snapshot" of the list of names of running process on the host. **Process32First** reads the first process in the list, and **Process32Next** reads the next in the list (and loops until all process names have been read).

ProcMon is great for tracing many behaviors of malware, but we are unable to see malware enumerating processes in a ProcMon trace. We'll need to use a debugger (x64dbg) to inspect this behavior:

1. Open the malware executable in x64dbg.
2. Select "Run to user code" to skip to the entry point of the malware's code.
3. Place a breakpoint on **Process32First**W, and **Process32NextW**.
    1. Hint: Type "***bp <function_name>***" in the command bar to set a breakpoint on a function. (Example: **bp Process32FirstW**)
    2. Hint 2: Make sure you are typing the "***W***" after **Process32First** and **Next**. This is important when creating a breakpoint, bit not important generally when discussing **Process32First** and **Next**.
4. Execute the malware (**F9**). The debugger should not be "paused" at the execution of **Process32First**.
5. Select "Run" again (**F9**) to execute until the next breakpoint. The debugger should break on **Process32Next**.
6. Continue to the next breakpoint (**F9**) a few times. You'll see the processes being iterated (the process names are in the RAX register):



7. If you examine stack memory, you'll also spot an interesting list of processes:



This looks like it could be the list of processes the malware is looking for. Let's investigate this in a disassembler. We'll use IDA for this.

Examining the Executable in IDA

1. Open the executable in IDA.
2. Navigate to the Imports tab, and search for the **CreateProcess32Snapshot** function.
3. Double click on that function to identify where it is used in the code.
4. Find the cross-references to the function (click on the function name and press the **X** key on your keyboard).
5. You'll see an interesting block of code:



```
loc_140001483:
lea     rax, aVm3dserviceExe ; "vm3dservice.exe"
xor     edx, edx        ; th32ProcessID
mov     [rsp+788h+var_758], rax
lea     rax, aVmtoolsdExe ; "vmtoolsd.exe"
mov     [rsp+788h+var_750], rax
lea     rax, aVboxserviceExe ; "VBoxService.exe"
mov     [rsp+788h+var_748], rax
lea     rax, aVboxtrayExe ; "VboxTray.exe"
mov     [rsp+788h+var_740], rax
lea     ecx, [rdx+2]    ; dwFlags
lea     rax, aProcesshackerE ; "ProcessHacker.exe"
mov     [rsp+788h+var_738], rax
lea     rax, aProcmonExe ; "Procmon.exe"
mov     [rsp+788h+var_730], rax
lea     rax, aX64dbgExe ; "x64dbg.exe"
mov     [rsp+788h+var_728], rax
lea     rax, aMsmpengExe ; "MsMpEng.exe"
mov     [rsp+788h+var_720], rax
lea     rax, aCbExe     ; "cb.exe"
mov     [rsp+788h+var_718], rax
call    cs:CreateToolhelp32Snapshot
mov     rbx, rax
cmp     rax, 0FFFFFFFFFFFFFFFFh
jz      loc_1400015BD
```

If you examine the code, as well as the subsequent calls to **Process32First** and **Proces32Next**, you'll see that this malware is querying the process list to see if any of these suspect processes are running. The malware terminates itself if any of these processes are running.

Let's see if we can bypass this. There are a number of ways we can approach this:

- We can simply terminate all suspect processes and then execute the malware sample again (the easiest way)
- We can modify the list of suspect processes in stack memory and change the process names.

- We can modify the code in the binary file on disk to patch out this functionality.
- We can modify the code dynamically inside the debugger.
- We can hook and modify the return values from the function **Process32Next** or other function calls.

For the purposes of this lab, we'll just patch the malware code similar to how we did it previously. We'll use IDA to determine the code we need to modify in order to bypass this process check, and then we'll modify the code in the debugger. Finally, we'll patch the binary file on disk and run the malware again.

1. In IDA, find the call to **Process32NextW**. Notice that there is a **jnz** (**jump-if-not-zero**) instruction. This instruction is responsible for looping through the list of running processes.
2. Open the sample in x64dbg and find this instruction. (Hint: You can place a breakpoint on **Process32NextW** and then **Execute to user code**).
3. Find the proceeding **jnz** instruction in x64dbg (it may be a **jne** instruction in x64dbg, which is equivalent in this case to **jnz**.).
4. Modify the instruction to be **je** instead of **jne** (which will bypass the process iteration loop). *(Hint: You can modify an instruction by **right-clicking** the instruction and selecting **Assemble**).*
5. Finally, let's save our patched binary to disk. This can be done with **File -> Patch File -> Patch File** in x64dbg. Save the newly patched file as something like "malware-patched-2.exe".
6. Close the malware in x64dbg and open the newly patched file in x64dbg and **Run** it. What happens?

Now, you should see a console message suggesting an Internet connection error. Let's take a look at our next evasion technique.

Part 4: Checking for Internet Connectivity

A common sandbox evasion trick used by malware is checking for Internet connectivity. This can be as simple as pinging a known IP or as involved as sending a request to a popular website and inspecting the response. For example, the WikiLoader malware checks connectivity by reaching out to wikipedia.com and examining the server's reply. If the server response doesn't start with a specific string (in the case of WikiLoader, "The Free"), the malware assumes it's in a sandbox and refuses to run.

WikiLoader reference: https://www.proofpoint.com/us/blog/threat-insight/out-sandbox-wikiloader-digs-sophisticated-evasion

You may have noticed that the malware produces an error that it can't connect to some server. It seems like this malware may be trying to contact a remote server, but since our VM is not

connected to the Internet, it can't reach this server.

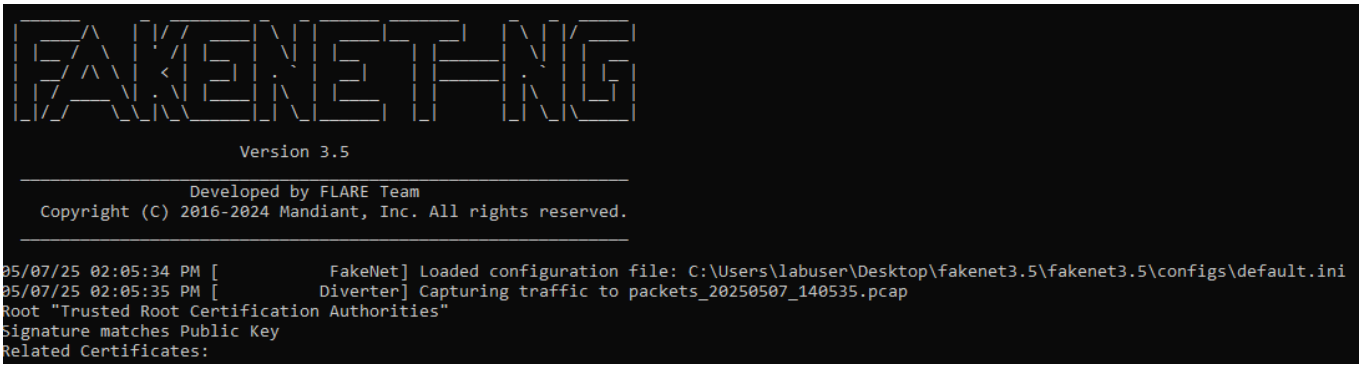
Malware may attempt to communicate with a remote server for many reasons, such as command and control or downloading of additional payloads. When analyzing malware in an isolated VM, its important to remember that malware may behave differently when it doesn't have Internet connectivity. In these circumstances, it can be helpful to "fake" an Internet connection.

There are a number of ways to do this. For this lab, I'll discuss two methods: FakeNet and the Remnux gateway.

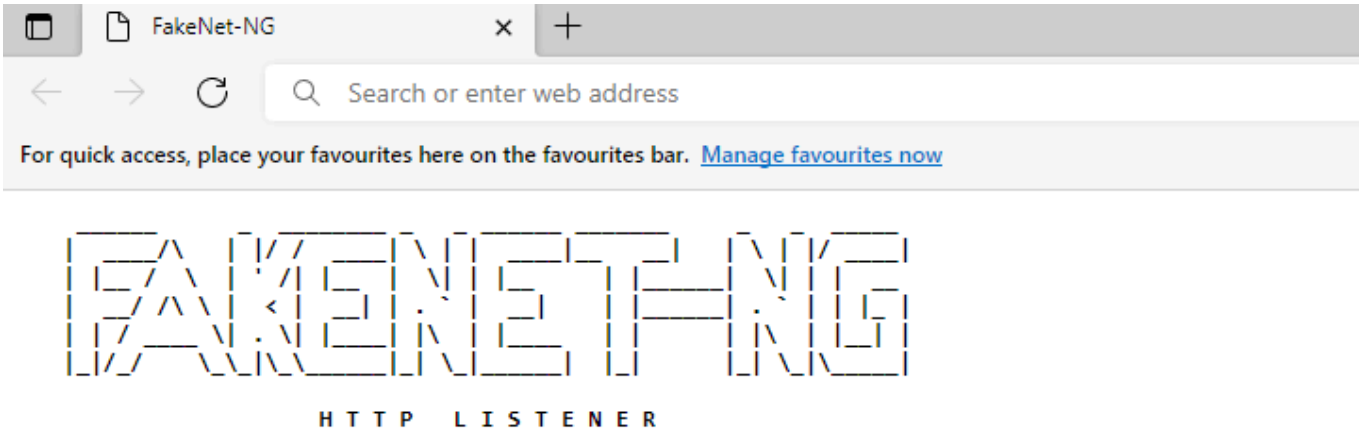
FakeNet

FakeNet (https://github.com/mandiant/flare-fakenet-ng) is a tool that simulates network and Internet connectivity. It essentially proxies and handles network traffic and is a great way to "fake" an network connection. This comes in very handy when analyzing malware in lab environment. Let's try this out now:

1. Start the FakeNet client on your VM:



2. Test to ensure FakeNet is running correctly. Open a browser (such as Edge), and if FakeNet is running, you'll see something like the following:



3. Execute the malware. The malware should have "connected" to the C2, presenting you with a flag.

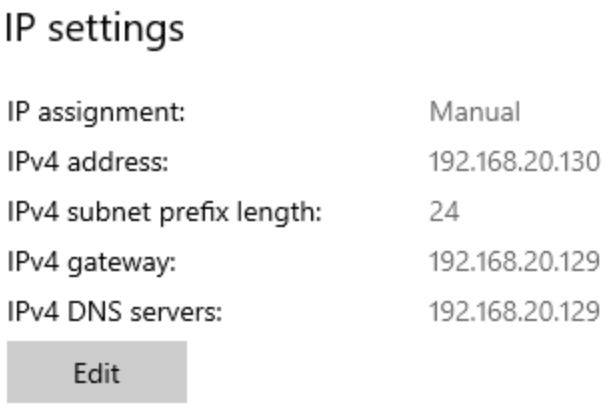
Remnux Gateway VM

Another way to simulating network connectivity is with the Remnux VM acting as a gateway. Remnux is a Linux malware analysis environment with many useful tools. Notably, Remnux makes a great network gateway for "faking" network communication and Internet connectivity. Let's test this out:

1. Make sure your Windows VM is in **Host-Only** mode. You can check this in the VMware or Virtualbox network adapter settings.
2. Start the Remnux VM (this should have been provided by the workshop leaders).

You'll first need to configure the IP of your Windows 10 VM to match Remnux.

1. In Remnux, issue the **ifconfig** command.
2. Set the IP of your Windows 10 machine to match Remnux. For example, if your Remnux IP is 192.168.20.129, you should set the following settings in Windows 10:



Important: Ensure to set the IPv4 gateway and DNS server to the IP of your Remnux host!

Once the IP configuration is completed, issue the following commands in Remnux:

1. **accept-all-ips start** (To force the Remnux firewall to accept all IPs)
2. **inetsim** (to start the Inetsim services)
3. And in a new terminal window, **fakedns** (to start the fake DNS services)
4. In windows 10, test connectivity by opening a browser and navigating to a website (such as "notmalware.com"). You should see an HTML page stating that you are connected to the inetsim server.

Now we have a fake network for our malware. Execute the malware again. Does it fully detonate? If you were successful, you should see a flag!

Congratulations! You now know how to circumvent some common anti-VM tricks that malware uses :)