

**Assignment Report:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Malware Analysis**

Authored By:

**Jon Murray**

8-30-23

Table of Contents

Overview \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 1

Sources \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 1

Anti-Cheating Notice \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 1

Machine Setup \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 2-7

PeStudio \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 8,9

Olevba and WinSCP \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 9-15

Fiddler \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 16-18

Process Monitor \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 18-20

CyberChef \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 20-22

De-Obfuscation \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 23-31

Conclusion \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pg 31

**Overview:**

In lab, we are going to be looking at Malware Analysis. We’ll start with the machine setup procedures. Then, we are going to look at a number of different tools to help us with the process. Then we are going to go through the total process of de-obfuscating a piece of code.

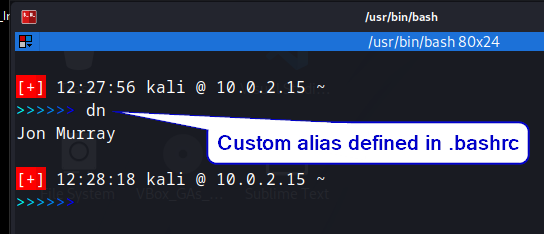
I will explain the usage of these tools and show examples. I will take you through the process of de-obfuscation and explain differences between Clean Code, Dirty Code, and Dead code. At the end, we will have a totally de-obfuscated file.

**Sources:**

Google.com

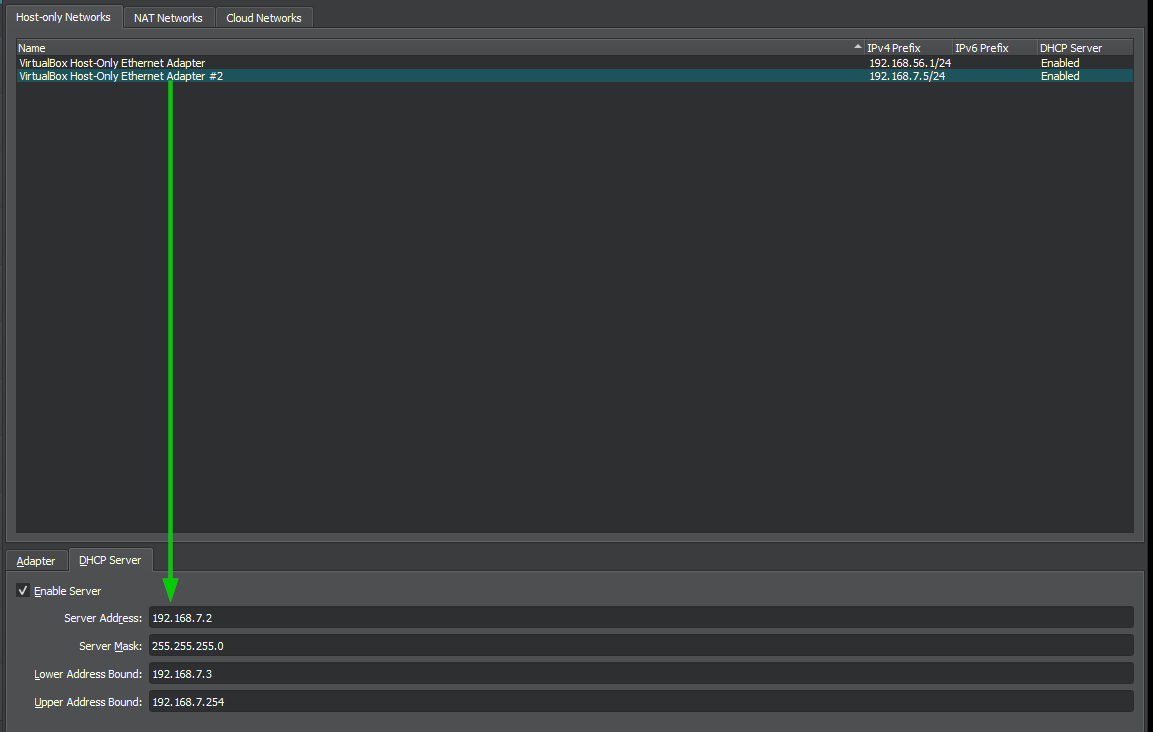
**Anti-Cheating Notice:**

I am running a custom shell on my Kali Linux machine with a custom command prompt as shown. I have also written an alias into my bashrc file to display my name:

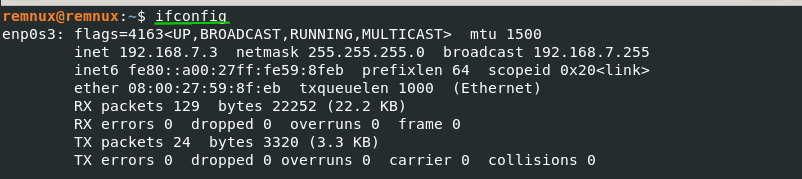


**Machine Setup:**

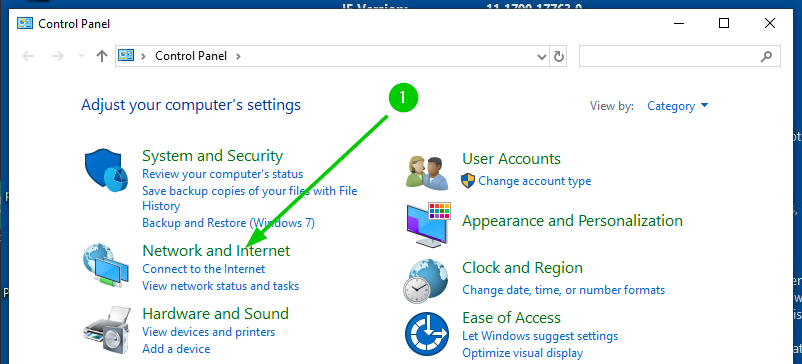
In order to begin this lab, we need to get the machines setup. The Windows machine we are going to be working with has malware on it, and it needs to stay on that machine, and only that machine. We are going to be working with an Ubuntu version of Linux called Remnux as well. Once both of these machines are booted and running, we can start with the network setup in Virtual Box. Here, I went to tools, then created a new network adapter. DHCP was enabled and the following IP address are selected, 192.168.7.3 through 192.168.7.254. This is the range of IP addresses available on this network. Remnux and the Windows machine will get one of these addresses:

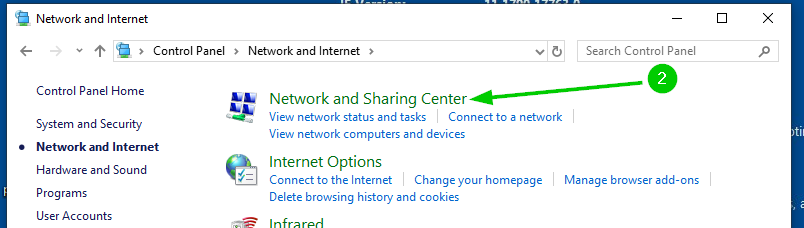


Now I run <**ifconfig**> on the Remnux Linux machine to get the IP address. It has been assigned 192.168.7.3:

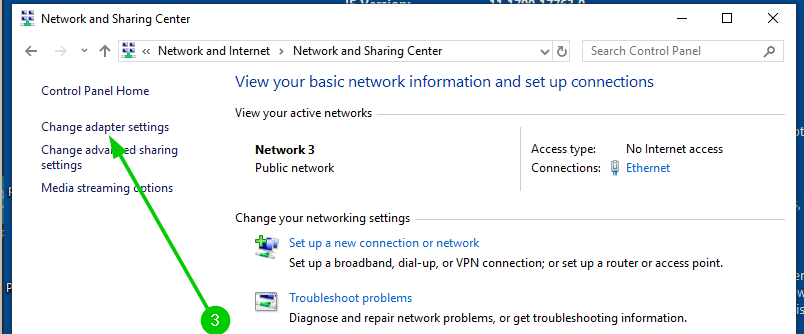


Next, we going to go to the Windows machine, and navigate to the settings menu, under Network and Internet, then select Network and Sharing:

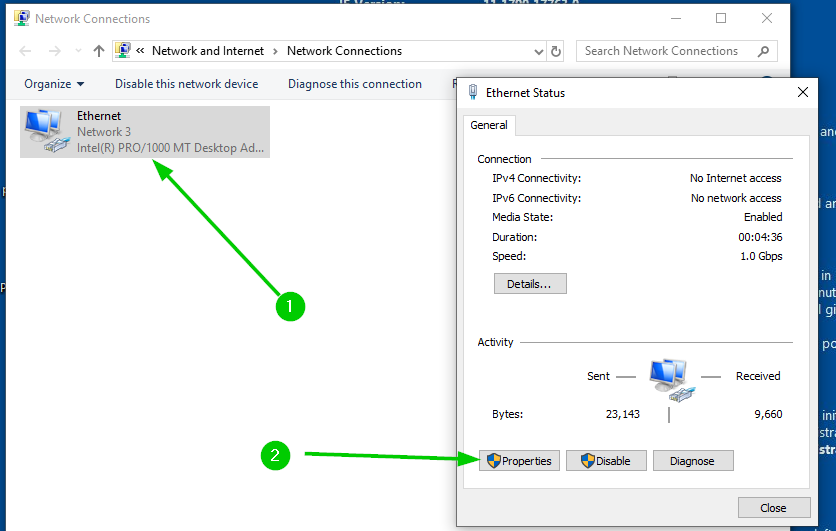




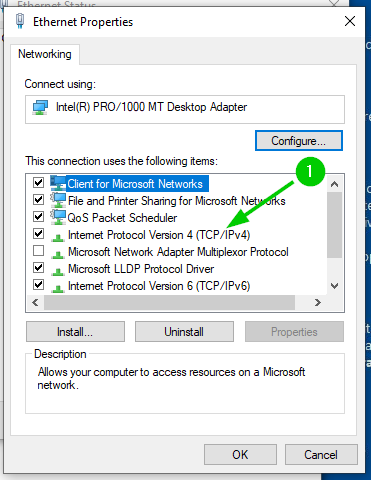
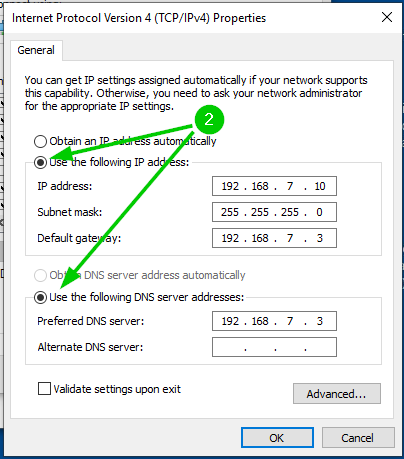
Next, we will select Change Adapter Settings:



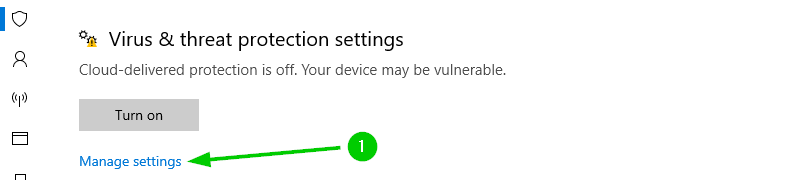
Then, we select the Ethernet adapter, and then Properties from the popup screen:



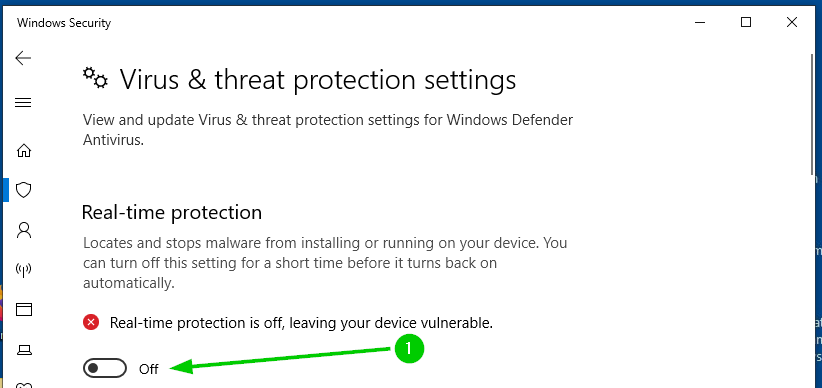
On the next screen, we select IPv4 option, and then select the radio buttons for Use The Following IP address:

In the previous screenshot, we need to manually assign the IP address, which needs to be in the range of 192.168.7.4 to 192.168.7.254, so I’m choosing 192.168.7.10. The subnet mask is set to 255.255.255.0 and the default gateway and preferred DNS server are set to the IP address of the Remnux machine, 192.168.7.3. Now we need to go back to the Window’s settings, and search for Virus and Threat Protection, then select Manage Settings:

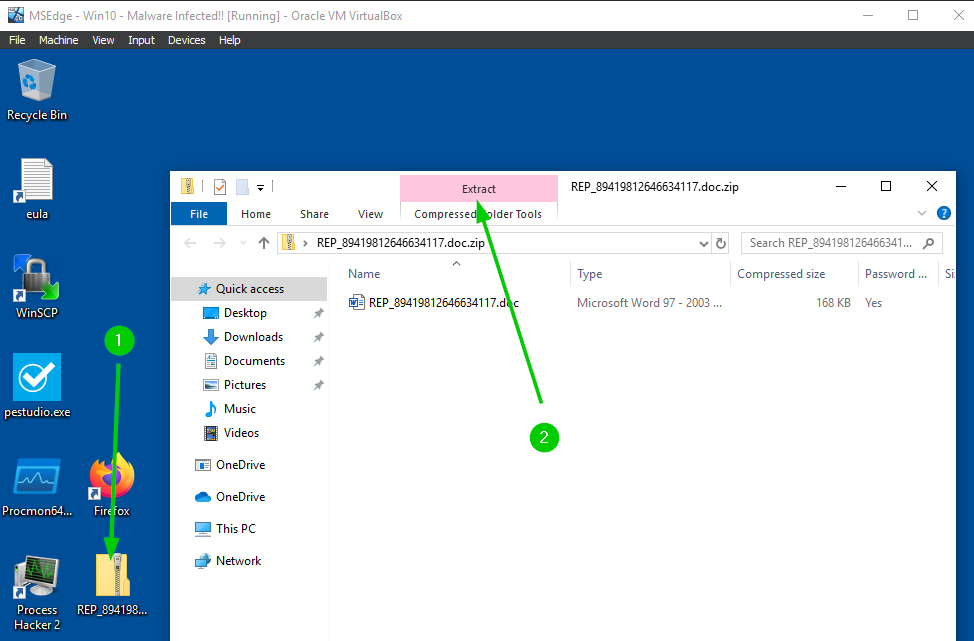


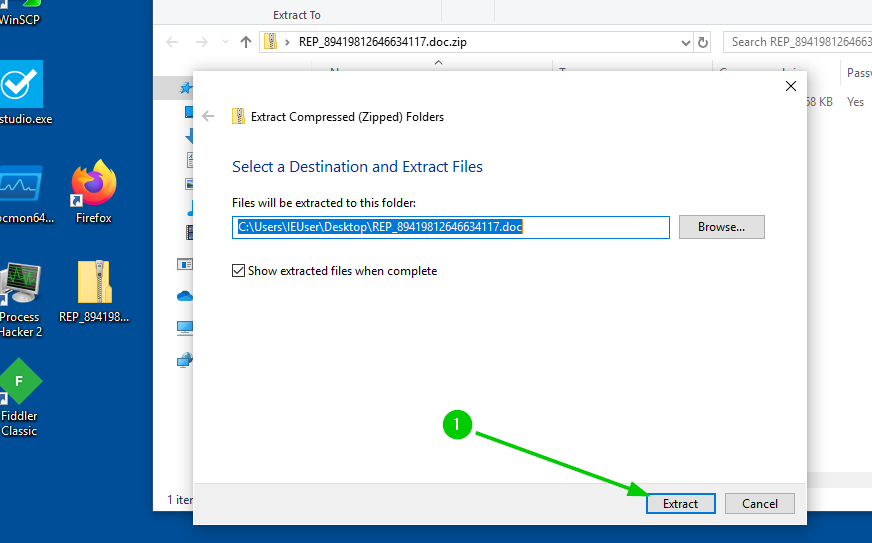
Then, we will toggle the Real Time Protection to off:



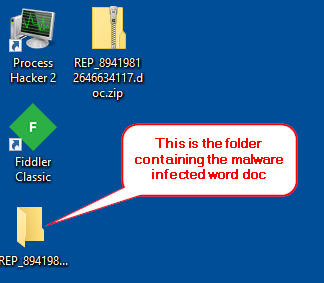
With our network configuration complete, now it’s time to turn to setting up the malware. I have checked to make sure that the Windows machine DOES NOT have access to the internet.

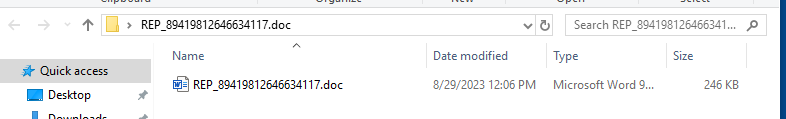
Back on the desktop, I’m opening the zipped folder and selecting Extract, and then Extract from the next window:



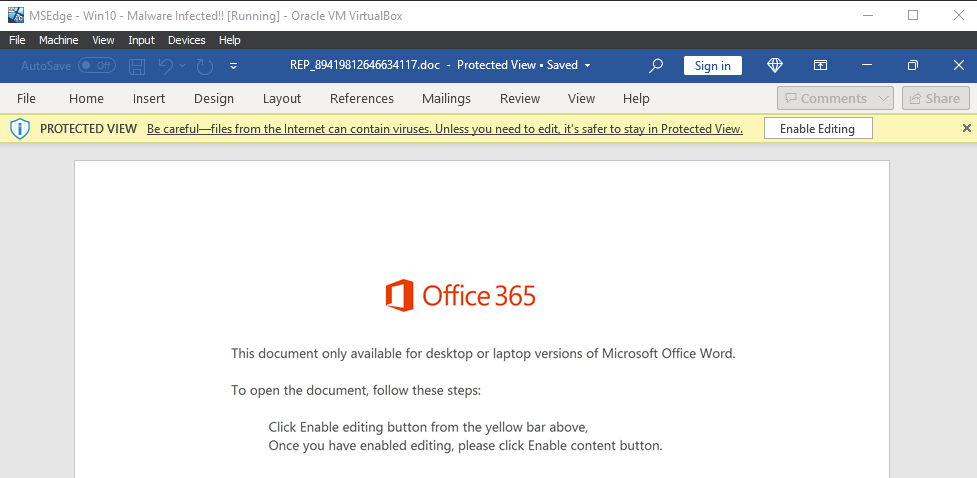


Then I open the folder on the desktop and select the word document inside to open:



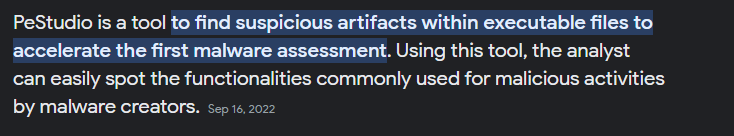


Here is a screenshot of what that document looks like. It is an Officer 365 Word document that has malicious code attached to it. Following the directions here will literally install the malware on your computer system:



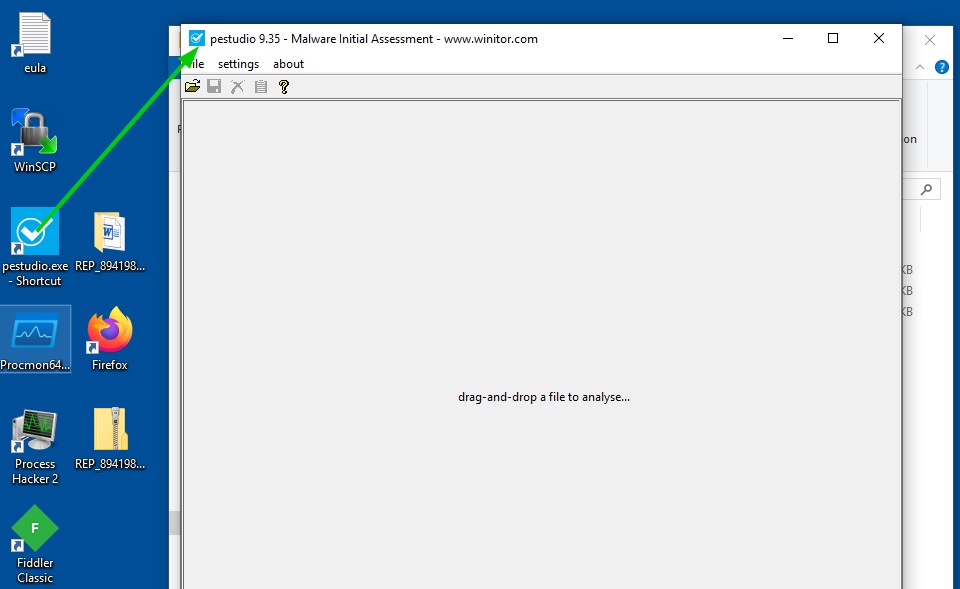
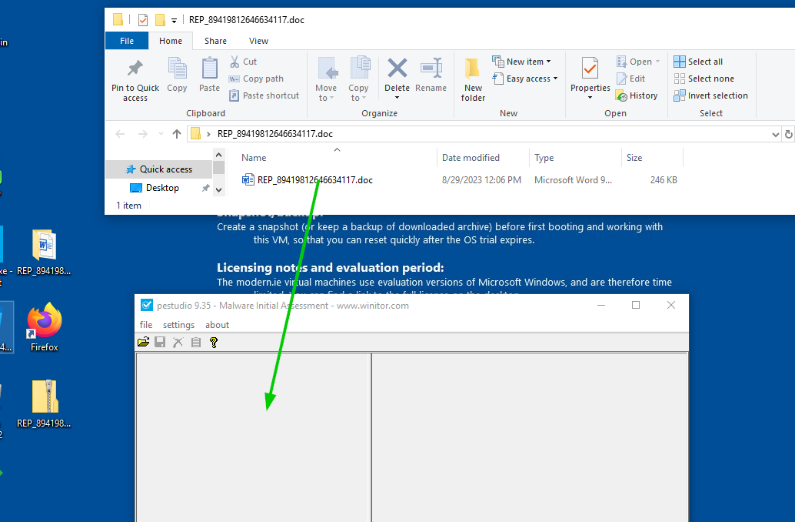
**PeStudio:**

The first tool we are going to look at is PeStudio. This is a tool that can scan a file for malicious content.

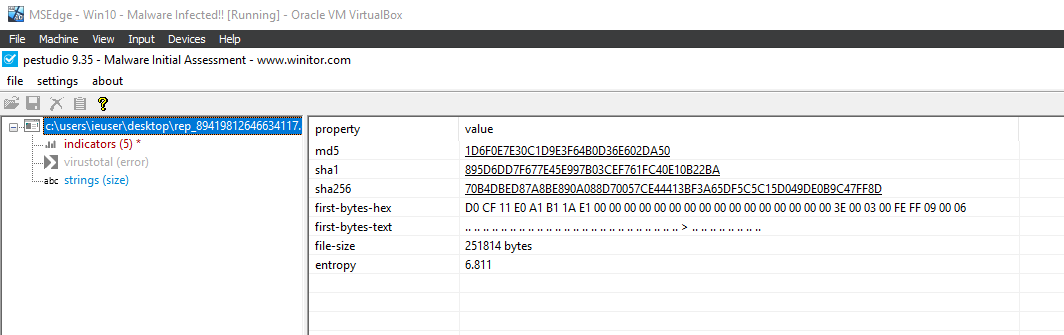


[ Google.com ]

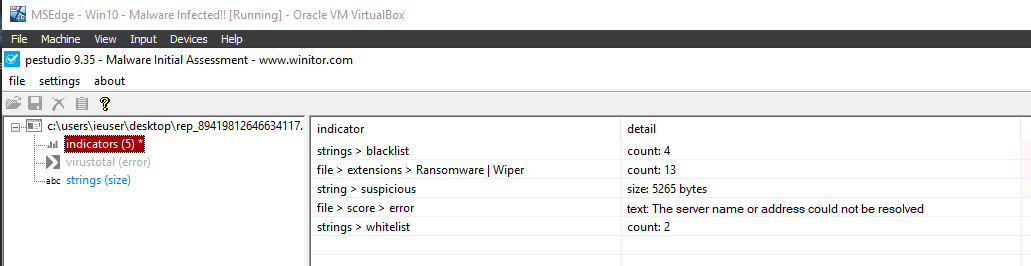
The first step is to open the program and then drag and drop the malicious file into the window:

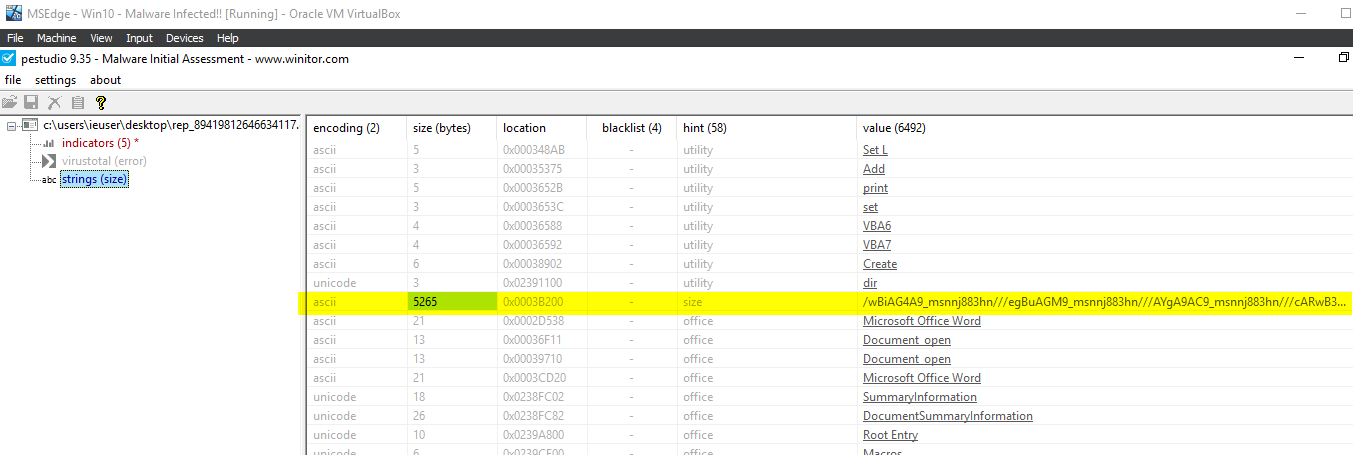
The file will then load into PeStudio and we can analyze the results. PeStudio advises that there are 5 indicators of Malware:



Let’s take a look at those indicators. Here, we can see that there is a suspicious strings file of 5265 bytes, along with 4 blacklisted strings that are likely to be malicious, including a file called Ransomware | Wiper:

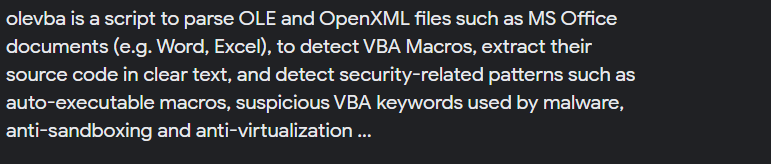


If we click on strings, we can see all the strings associated with the file. The string in question appears to be obfuscated, a red flag indicating malicious content:



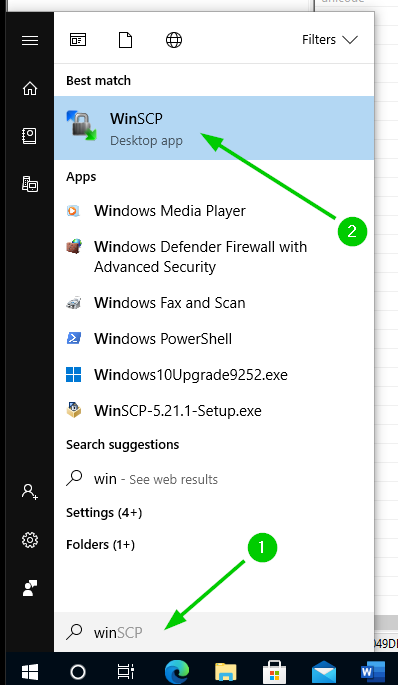
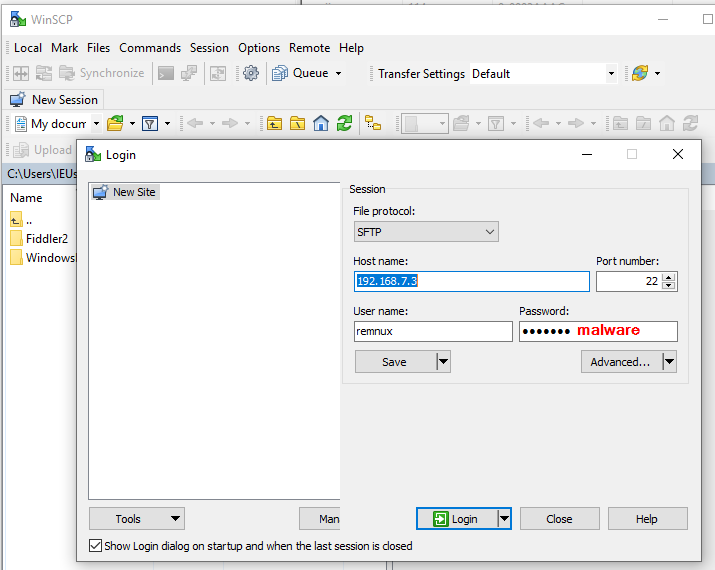
**Olevba and WinSCP:**

Now that we have done a quick analysis on the word file, it’s time to use Remnux. We are going to use a tool called Olevba.py which is a tool for extracting macros and source code inside documents:

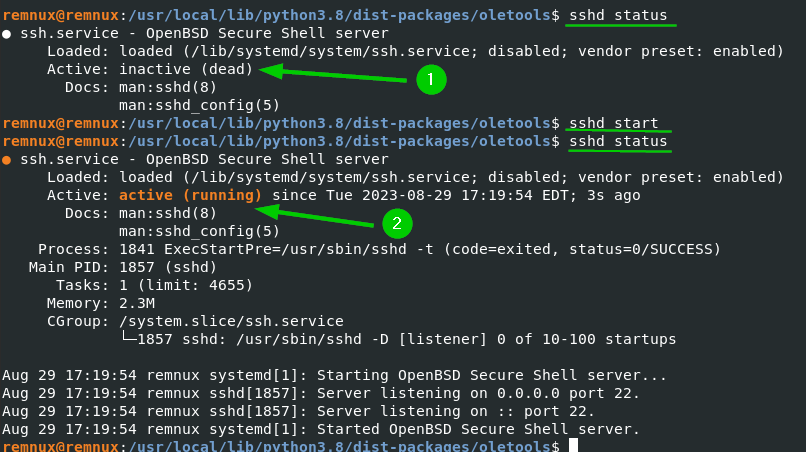


[ Google.com ]

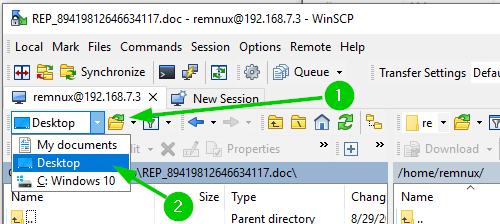
In order to use Olevba, we first need to move our malicious file to our Remnux machine. We are going to do that using WinSCP. I click on the Windows start menu and then type WinSCP, and open the program. Then, in the login window that opens, I’m placing the IP address of the Remnux machine in the host name box, the username as Remnux, and the password as malware:

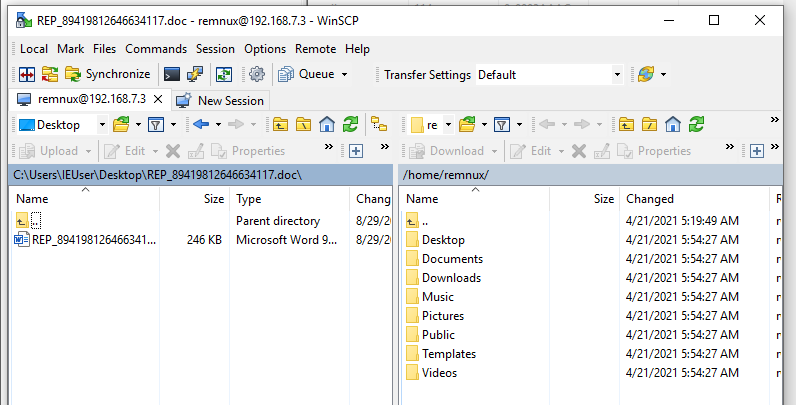
Before clicking login, we need to go back to the Remnux machine and start the sshd service, so that WinSCP can connect to it. The command in the terminal on Remnux is <**sshd status**> to check the status of the service. The service shows as inactive so the command to start is is <**sshd start**>, and then <**sshd status**> again to confirm that it is now running:



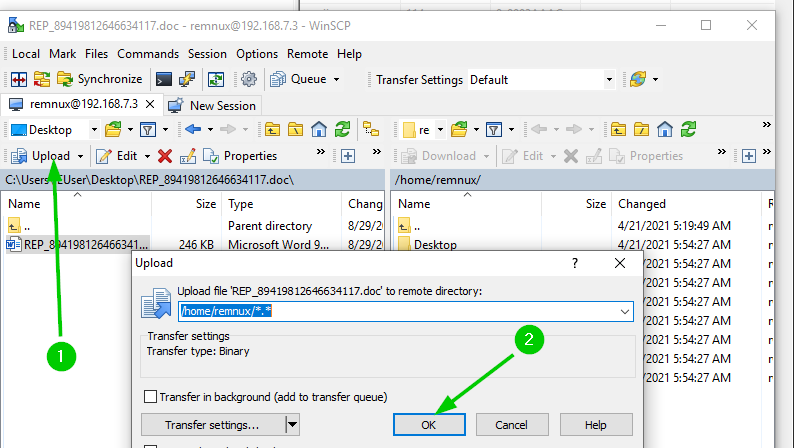
We then head over to WinSCP and click login, then select yes from the following screen. Now we need to load in the word file that we are going to transfer. Click the folder icon, then desktop:



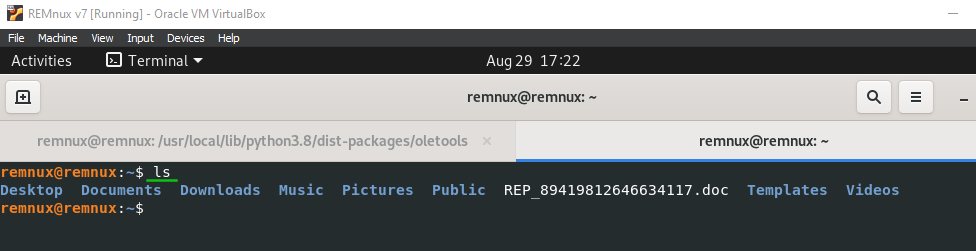
And here we can see the document is loaded and ready to be transferred:



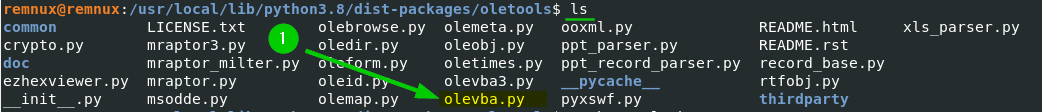
Now we will click upload, and then ok in the popup window:



Back on Remnux, we type <ls> and we can see the file transferred successfully:



Now we will work with Olevba. Olevba is located in the /usr/local/lib/python3.8/dist-packages/oletools directory, so I navigate to that location and <**ls**>. Here is the tools itself:

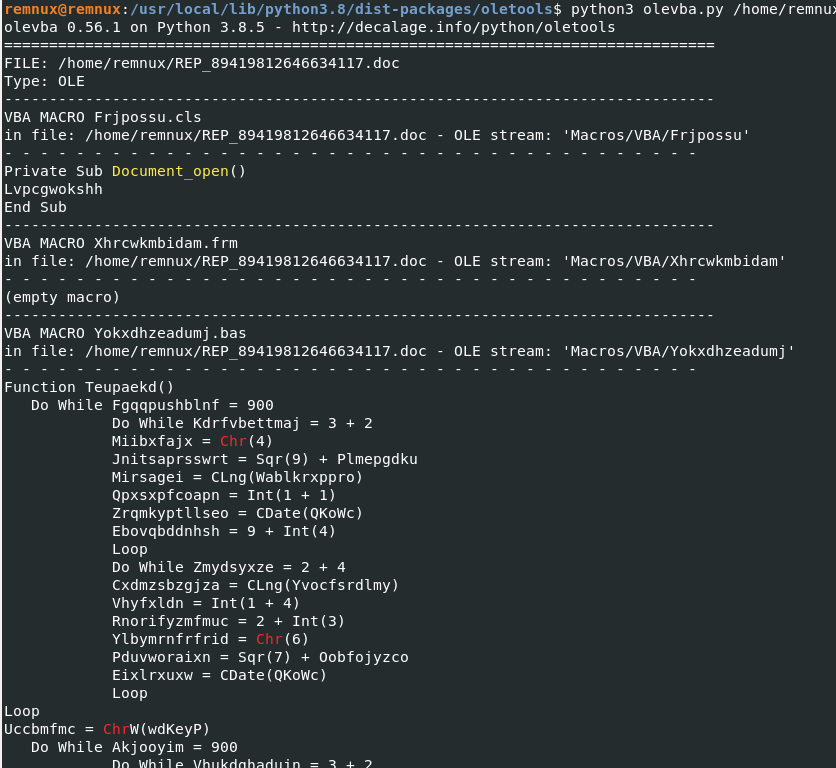


To use the tool, the command is:

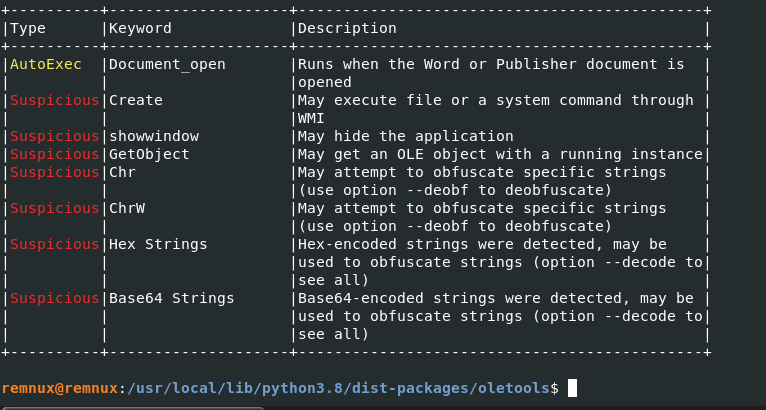
<**python3 olevba.py /home/remnux/(name of file)**>



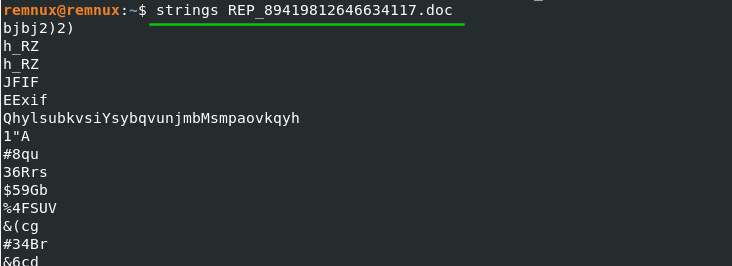
And now we can see the source code of the macro inside the word document. It is highly obfuscated. More indication of malware:



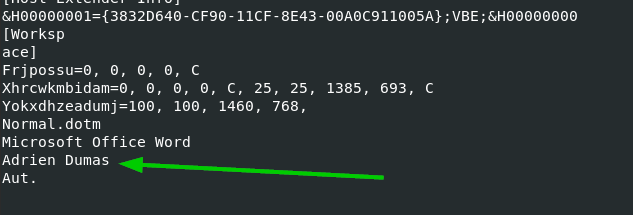
We also see suspicious activity that occurs:



We can further run a <stings> command on the document to see more obfuscation of the strings embedded in the document. Towards the bottom of this output, we can see someone named Adrian Dumas as the potential author:

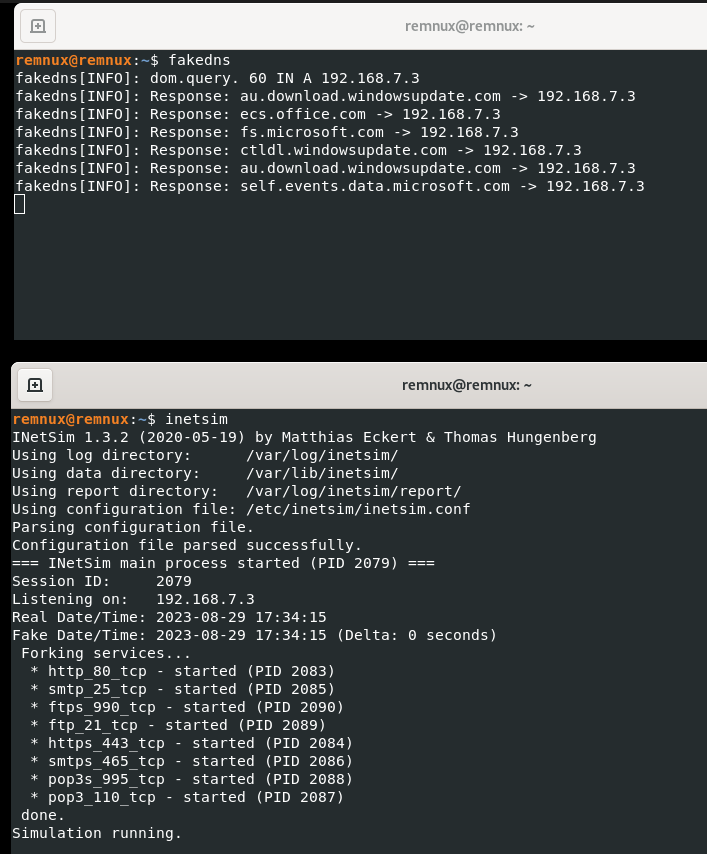




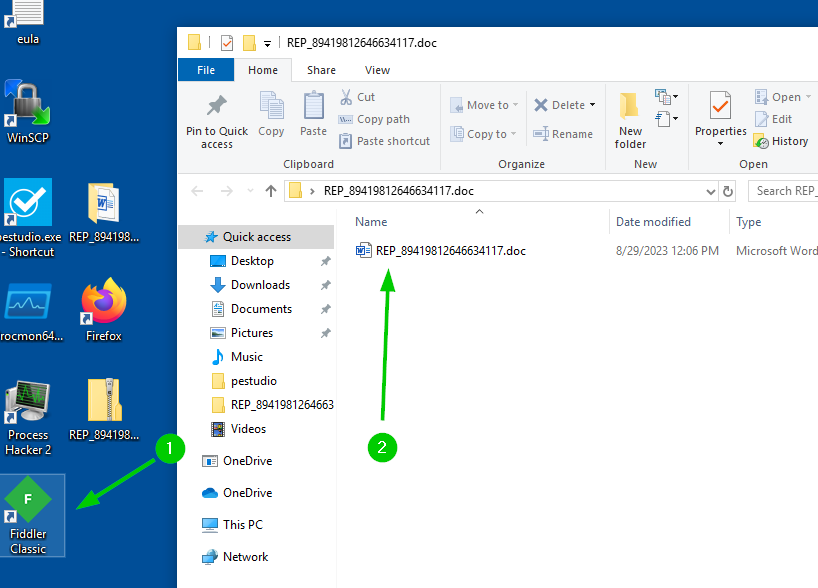


**Fiddler:**

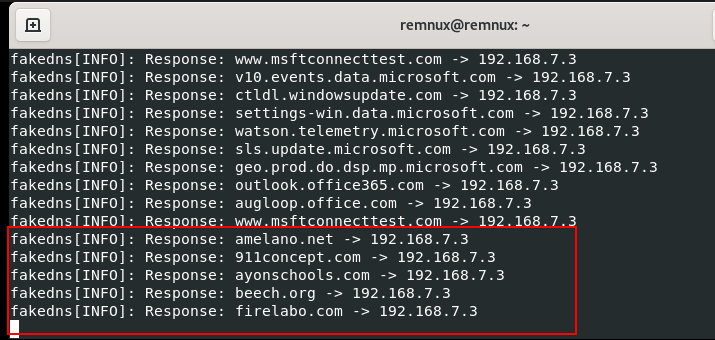
Now we are going to look at Fiddler on the Windows machine. This is a malware inspection tool that can show us exactly what is happening when the malware is running. There are 2 additional tools we are going to run in the Remnux machine, alongside Fiddler, FakeDNS, and Inetsim. FakeDNS will simulate a DNS server for out malware to connect to, and Inetsim keep track of the services being used, as well output a nice report for us. In the following screen shot, I have opened two terminals, one for each machine:



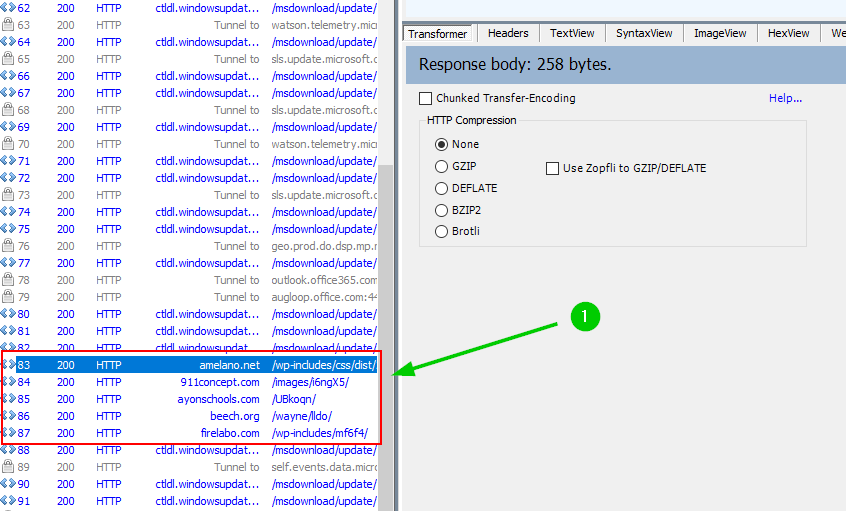
Now I’m going to open Fiddler and then run the word document with the malware inside it:



Back in Remnux we can see what is happening. There have been DNS requests for a number of websites that we did not navigate to ourselves:



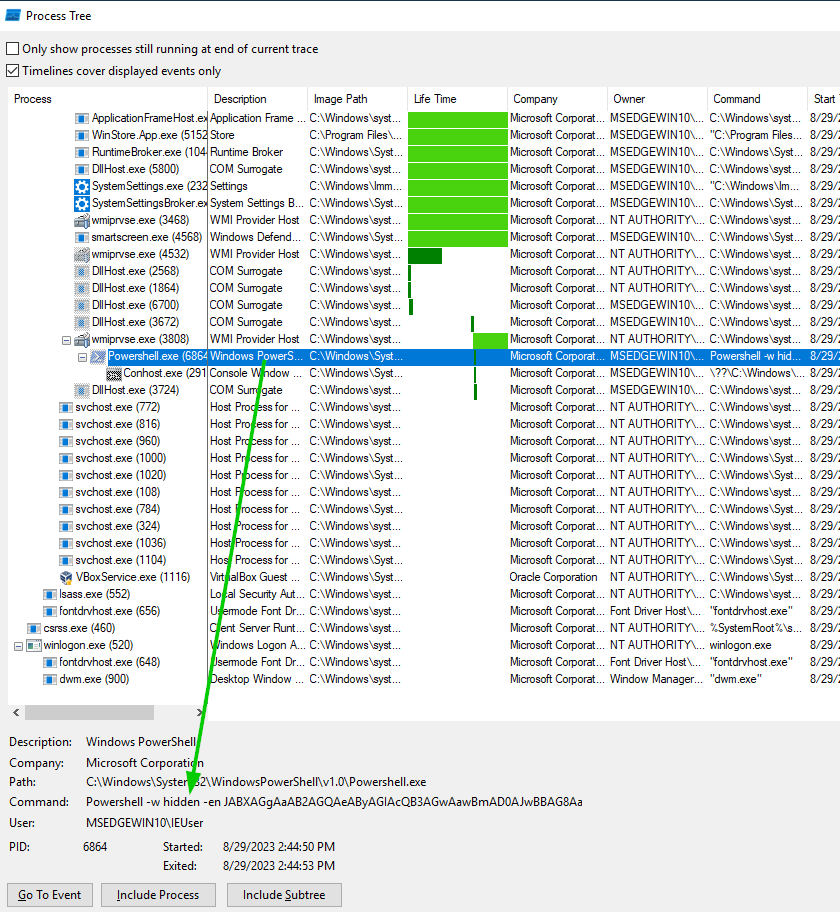
And here in Fiddler, we can see those same requests, in much the same way as we intercept traffic with Wireshark:



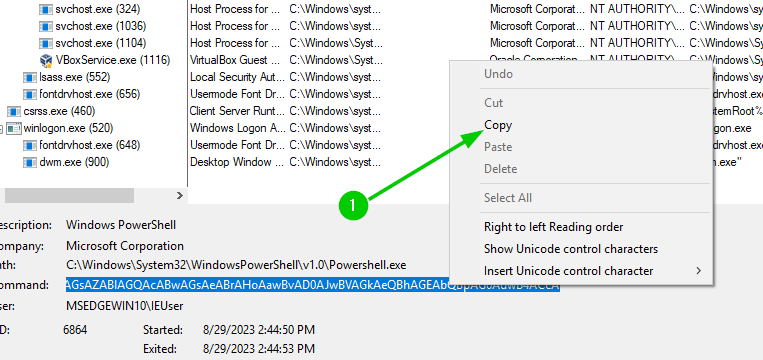
**Process Monitor:**

Let’s review this activity in a program called Process Monitor. This is a process manager similar to the Windows version, but gives some more detailed information as to what is happening on the system. Let’s take a look at the process tree for this application. Here we can see that it lists all the processes running on the system. Special note is the Life Time column. The green bar here indicates the time a given process is running. Most of these processes appear to be normal processes, however, there are a few processes here that suddenly started and then abruptly ended. Since we did not take that activity ourselves on the Windows system, we need to look further at these processes. The very short green bar indicates the start and stop of the process or function. One of these processes is wmiprvse, which, when expanded appears to have run a Powershell code. We know we did not run any Powershell so let’s examine this process:

Here, we click on the process, we can see the actual Powershell command that was executed. A hidden file was opened and then obfuscated. This is a huge red flag that something malicious occurred:

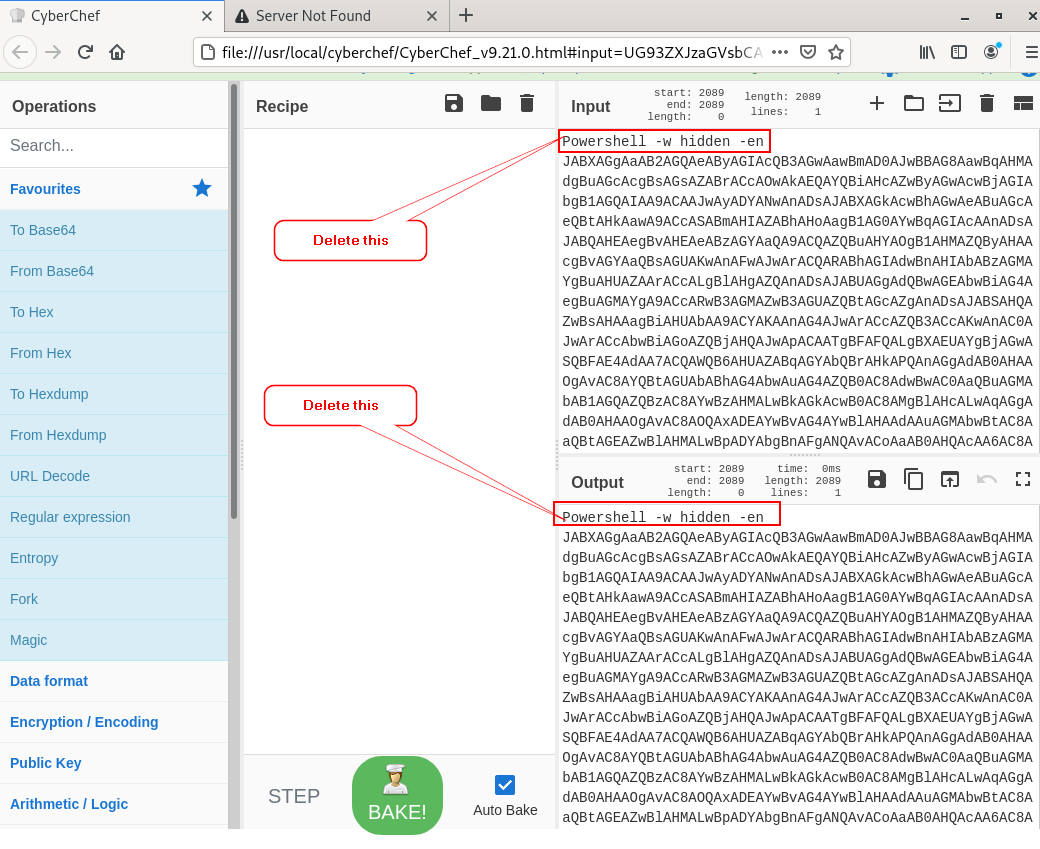


Let’s see if we can work with this to try to understand what happened. I’m going to copy the command in preparation for our next tool:

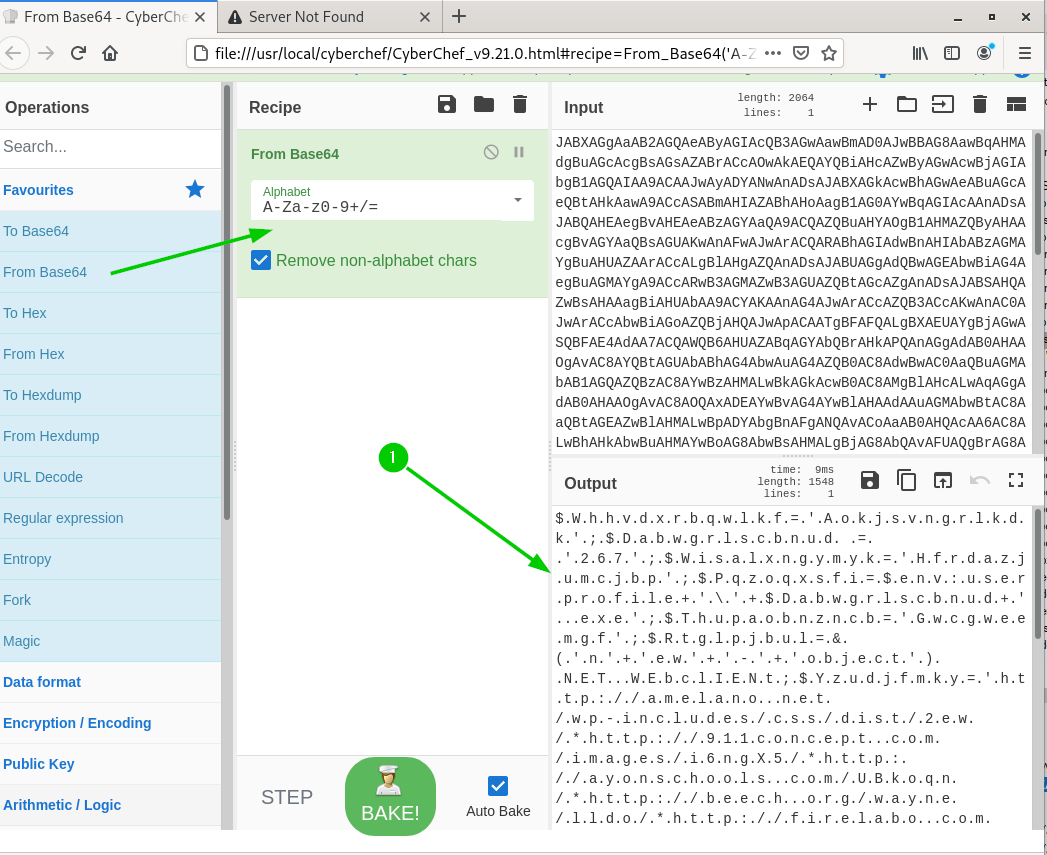


**CyberChef:**

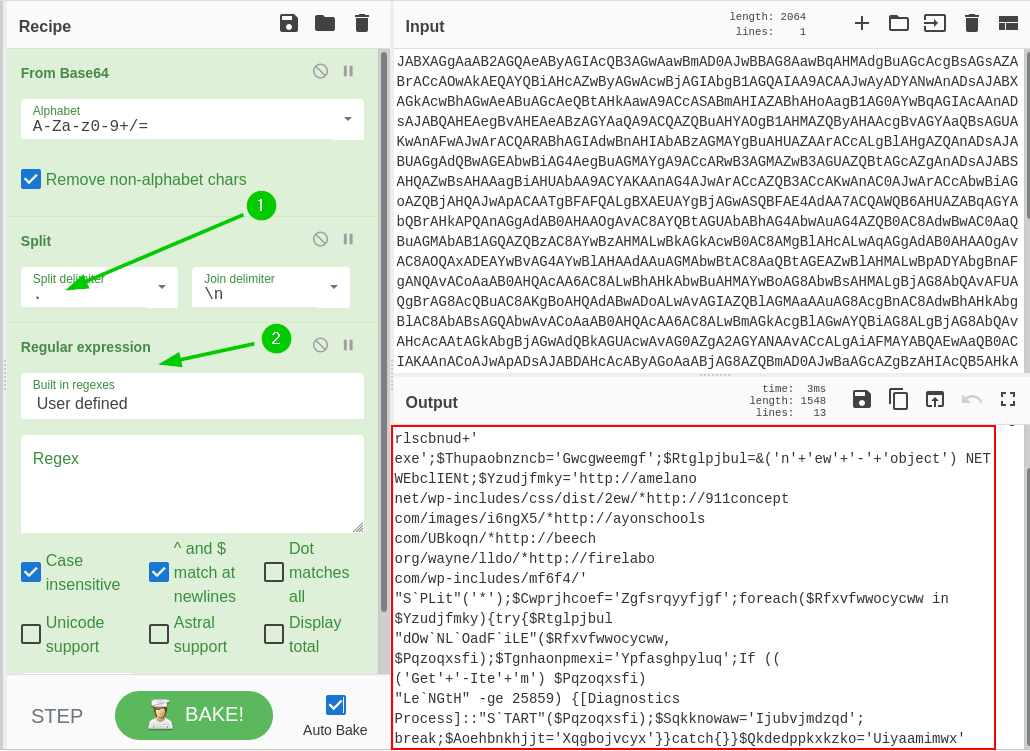
CyberChef is an awesome application that can help convert obfuscated or encrypted code into human readable code, or vice versa. I start up that application and then paste the Powershell command in. Since the first part of the Powershell command is in plain text form, we don’t need to work with it, so we can go ahead and delete it:



If we refer back to PeStudio, we saw that some of the files indicated that there was Base64 encoding going on. Base64 is a way of encoding text. It is recognizable as having padded “=” at the end of it. So, let’s start with apply a “from Base64” filter to the code by simply dragging that filter to the Recipe section. And now we can begin to see some of the code being displayed in a somewhat readable format. We can some plain text, and what looks like website addresses being display, although they are still hard to read:



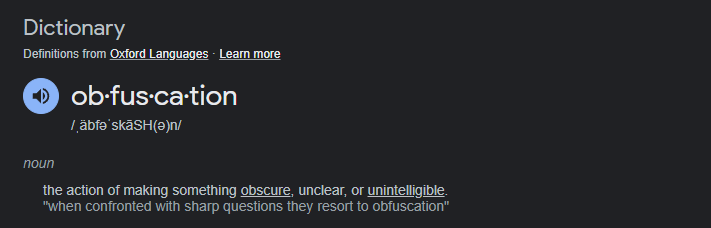
Let’s add a couple more filters to our recipe. This time we are going to add a “split” filter. Since I’m seeing a lot of the code separated by the “ . ”, I’m to select that as the split delimiter. I’m also going to add a REGEX filter:



Now our code is looking much cleaner, although, it is still obfuscated. Let’s try to take care of that next.

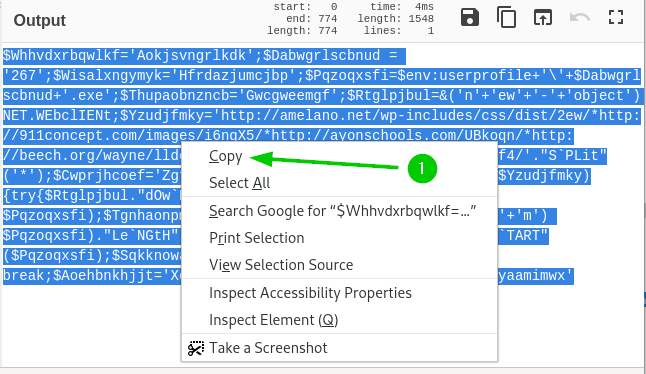
**De-Obfuscation:**

We’ve been throwing this term around a lot, but what does it actually mean? Obfuscation, is a technique that is used to change plain text into unreadable text. It is a form of encryption.



[ Google.com ]

This is a technique that threat actors have long used as a means of hiding their code. While we as humans may not be able to easily read it, a computer has no problem understanding the code and executing it. De-Obfuscation is the act of changing the obfuscated code, back to a human readable form. This is what we are going to do in the next part of this lab. Before we do this, let’s copy the code and place it in a code editor so we can work with it, in this case, I’m going to use VS Code as my preferred code editor:



Here I have pasted the code into the VS Code editor. I then went and added some spaces between the lines of code, and separated out the websites based on the \* character I found in the code. The reason for this is just trying to make the code easier to work with when we de-obfuscate it:

Before we start with the de-obfuscation, let’s talk about some of my personal coding ideals. I separate code into 3 broad categories; Clean Code, Dirty Code, and Dead Code.

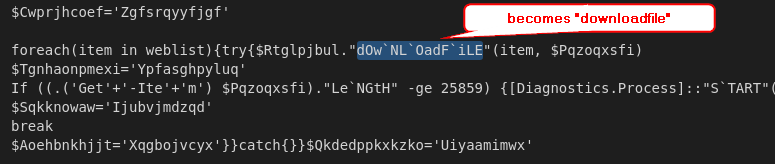
Let’s begin with Clean Code. This is code that I imagine is professional, production level code. It follows all the styling rules of the language it’s written in, including spacing, indentation, naming conventions, capitalization, etc. This code is likely written by a professional coder, and has a specific purpose which is accomplished in the fewest lines of code needed. It is also well commented, but not overly commented.

Dirty Code is what I imagine an everyday coder might use. It likely does not follow specific styling rules like Clean Code does. It is likely not written by a professional coder, and probably uses more lines of code than is necessary to accomplish its function, but it does have a specific function and use, and does accomplish that. It may be overly commented, or not commented at all. Dirty Code is what I use most commonly.

Dead Code does not fall into either of these two categories. Dead Code is what I call code that has absolutely no function in a script or program. It serves no purpose whatsoever and has no specific function or use. As it turns out, threat actors often use this type of code to “pad” dirty code in their scripts, and, we will see some of this as we go through the process of de-obfuscating the code we are working with.

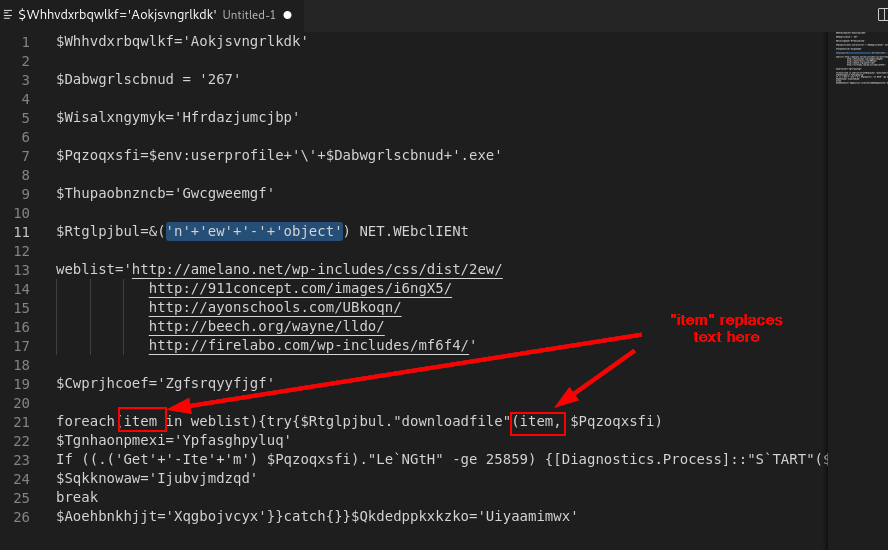
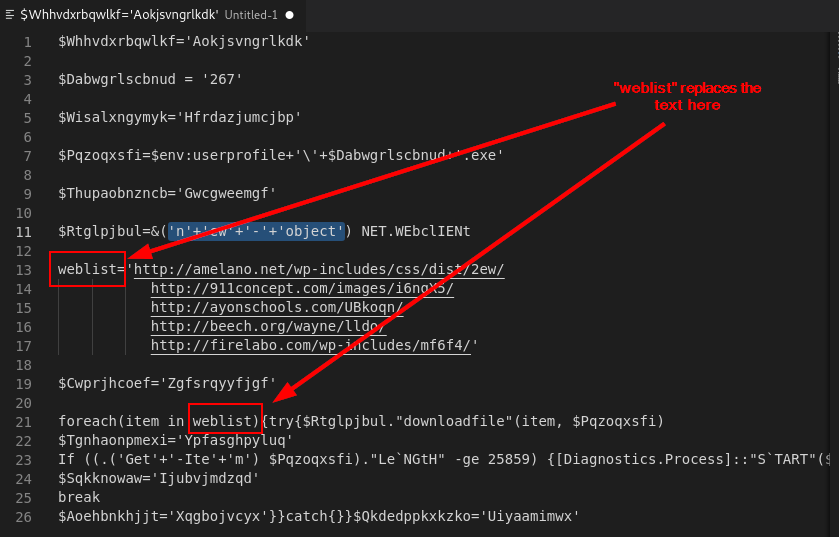
Threat actors use this as another means of obfuscating their code. This can be a very deep rabbit hole for a malware analyst to go down if not recognized for what it is. Luckily, VS Code, and other code editors, have a way of helping us identify this type of code, which I will showcase in the de-obfuscation.

Let’s begin this de-ofuscation with concatenating some of the readable code we are working with. Here, we can see what looks like it should read “downloadfile” so let’s change it:

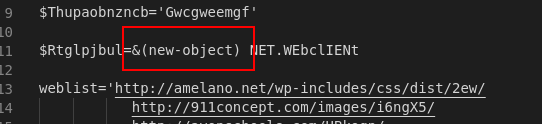


“Weblist” is what I replaced the gobly gook with in the list of websites. To do this, I highlighted the original unreadable variable and hit Ctrl + F2 to select all occurrences of that variable, and replace all occurrences at the same time. This change then also had an effect in the For Loop seen just below the list of web addresses.

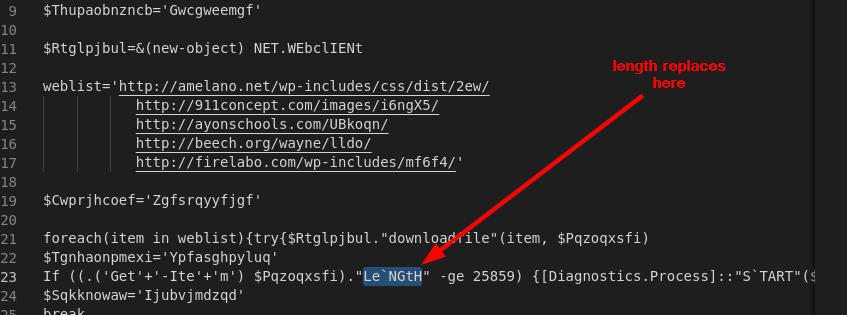
Speaking of For Loops, these typically formed in the manner of: for item in list, do something. So I then also changed the unreadable variable in the For Loop to “item” using Ctrl + F2 to select all occurrences of the variable, which then also had an effect in the code, just after “downloadfile”:

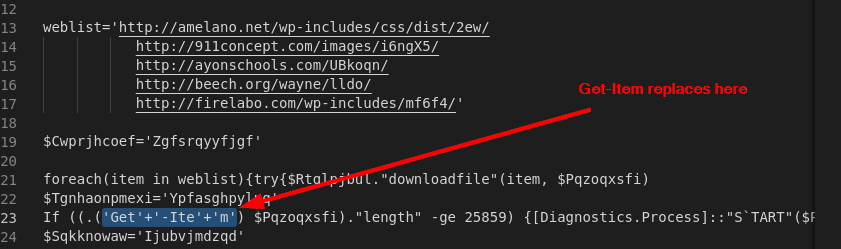


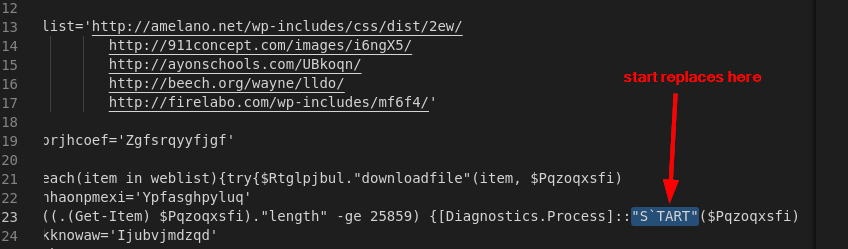
Good start so far, now let’s focus on that blue highlighted text. This is string concatenation going on here. If just add it together as the + signs indicate, we get: new-object. So, let’s replace that next:



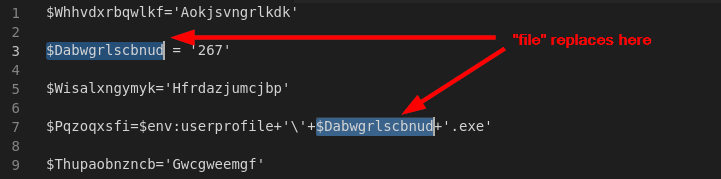
Looking better. We have some more string concatenation going on in the If statement. “Le’NGtH” becomes “length”, and ‘Get’+’-Ite’+’m’ becomes “Get-Item”, and “S’TART” becomes “start”:







OK, that takes care of the easy stuff. Now let’s move on to these gobly gook variables we are seeing throughout the code. We have a line: $Dabwgrlscbund = ‘267’ and then 2 lines down from that, we see the same variable. I located this by highlighting the first occurrence and using Ctrl + F2 again to select all occurrences of the variable. If we look at that second occurrence, it looks like string concatenation again, but this time, we are building out the name of an executable file. So, let’s just replace this unreadable variable with the word “267”:

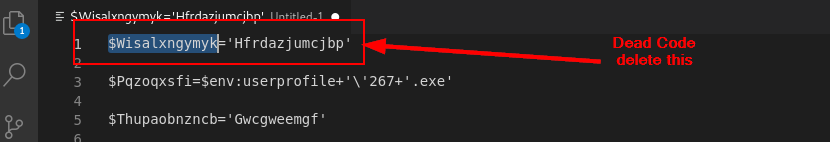




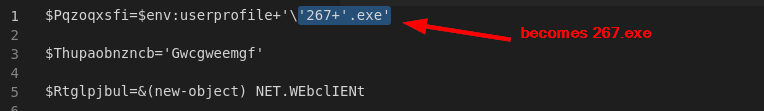
Now, remember what I was saying about dead code, let’s see a real example of that now. Here in the code, we have the line: $Whhvdxrbqwlkf = more unreadable mess. Again, with the Ctrl +F2 shows us that this is the only occurrence of this variable. It is not used anywhere else in the script. That makes it dead code. And code on one side of an “=” must be the same as the other side, so we know that the following variable is also dead code. We simply delete this from the script. It has no purpose and is just a rabbit hole meant to confuse us. IGNORE IT!!



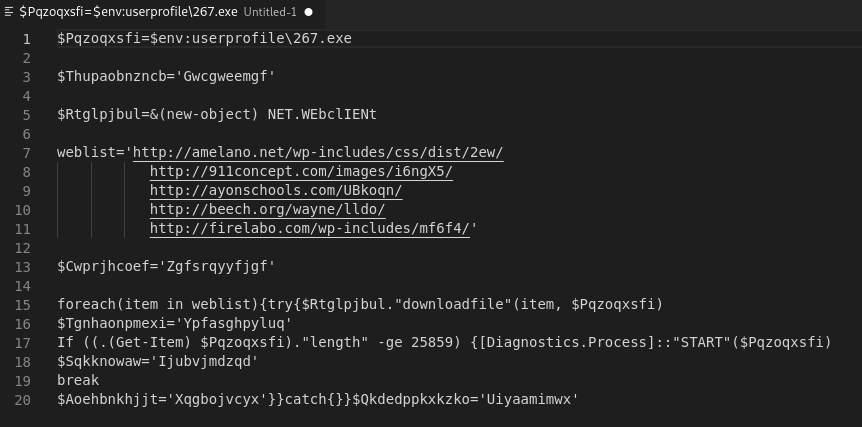
Here’s more of that same mess:



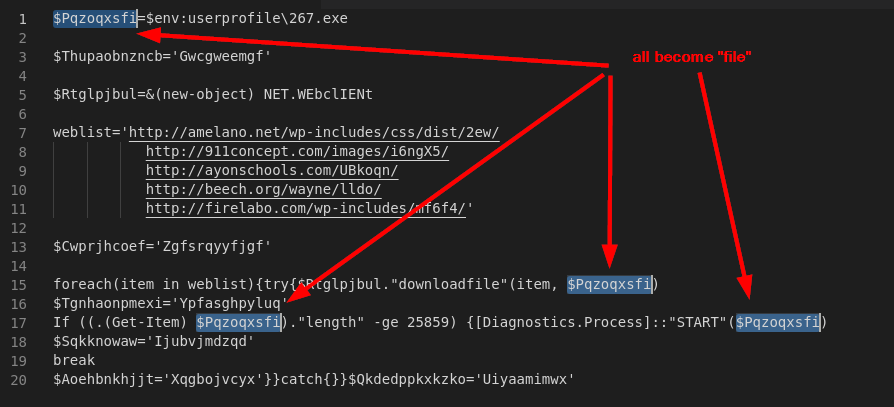
Ok so that leaves us with the first real line of code that seems to be creating some kind of .exe file in the current user’s profile. Let’ deal with the easy string concatenation before we mess with that variable:



Let’s take a look at what we have so far. As we can see, this is already looking much much cleaner, I went ahead and restructured the for loop so we can get it all on the same screen without having to scroll to the side:



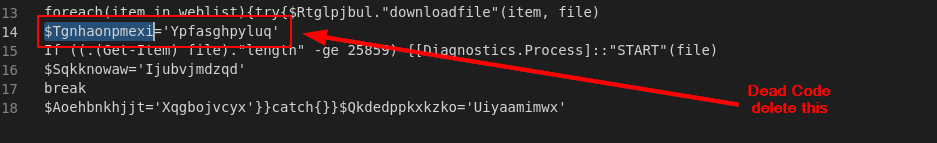
Let’s keep cleaning this up. Let’s work on that $Pqzoqxsfi variable in line 1. Ctrl + F2 selects all occurrences. Let’s just call this variable “file” as that’s what it seems to be creating. And look at all the occurrences of this variable that get changed in the process:



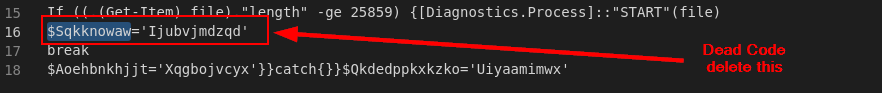
On to line 3, I bet you know what this is. One single occurrence of a variable, yep, more dead code, let’s get rid of it:



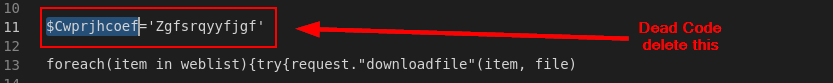
Let’s look at line 14 now, ugh, more dead code, you know what to do by now:



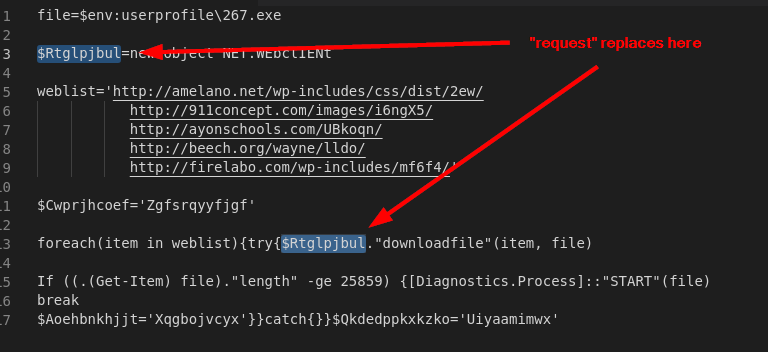
And on line 16, yep, annoying right? That’s what rabbit holes do though:



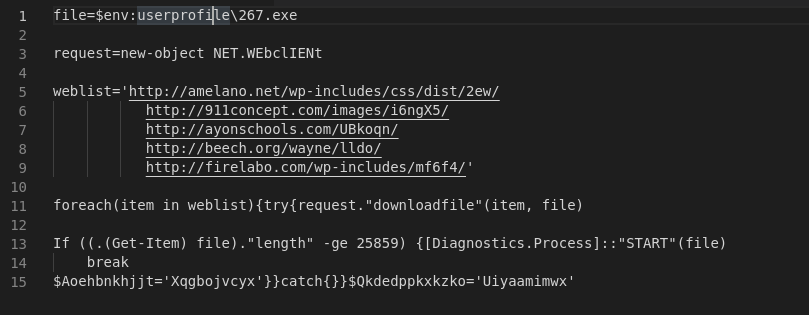
Sigh... 11 too:



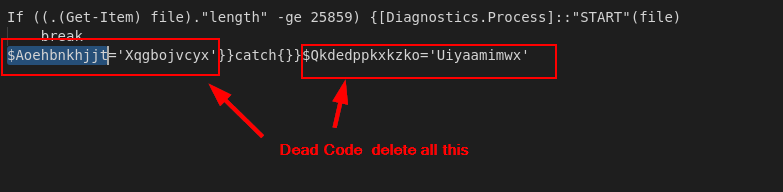
Alright, that brings us to the following code. Let’s look at this “new-object net.webclient”. This is what is called a class in coding. We are creating a new instance of this class and then assigning that instance to a variable for use later in the code. Looking down through the code with Ctrl + F2 again, we see another occurrence of this variable. I called this variable “request” as it seems to making a request for a certain website. Basically, for each item in weblist, try to “request” the website, then download:



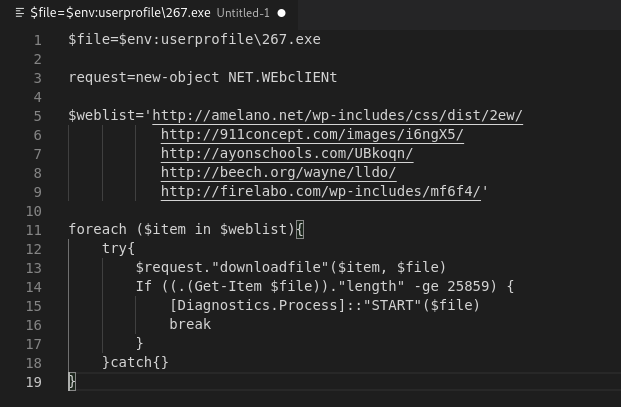
Alright, almost there, let’s take a look at our code now. We got some more junk going on at the bottom so let’s try to de-obfuscate that mess:



This looks like its part of a try/except statement in python, but since we are dealing with powershell here, it is formed as a try/catch statement. But, what’s this, a single occurrence of a variable again? Yea you know it, more dead code!! And this occurs both before and after the “catch”, so let’s get rid of all that:



And all that’s left is to do a little restructuring so we can read the code a bit better and viola, a totally de-obfuscated piece of code!!!:



We could go through this same process with every piece of code we found in this malware, totally revealing what the malware does.

**Conclusion:**

In this lab we got to take a look at a number of different malware analysis tools. We quarantine a malware infected file and then analyzed on the Remnux machine. We then located a powershell script and went through a fun process of de-obfuscation to learn what the code really did.

Lesson learned here is never open a file that is flagged as possible malware or infected by a virus. Always make sure you are using the most up-to-date antivirus protection and that it is turned on.