Practical Malware Analysis and Triage Malware Analysis Report

Sample: Malware.javaupdate.cs
Source Code Analysis

November 14, 2023

Muhfat Alam

Summary

MD5 and SHA256 hash value for **Malware.javaupdate.cs**

MD5: d825ce85cc6866a3a64486d461758280
SHA256: ea63f7eb9e3716fa620125689cfef1d5fed278ded90810e7c97db3b66b178a89

URL: burn.ec2-13-7-109-121-ubuntu-2004.local

Malware.javaupdate.cs is a malicious shell code that is trying to download something from a URL (**burn.ec2-13-7-109-121-ubuntu-2004.local**) and executed via the WinINet API.

Static Analysis

Get the File Hash
Get the hash value for the source code of **Malware.javaupdate.cs** on the cmder by running the following command (Figure 01),

```
sha256sum.exe Malware.javaupdate.cs //in sha256 hash
md5sum.exe Malware.javaupdate.cs //in md5 hash
```

Figure 01

OSINT Tool: As we have the hash value from the command line, we can use some OSINT Tools, like VirusTotal and MetaDefender for any flagged by security vendors.

VirusTotal Verdict: There are 1/59 security vendors flagged this **Malware.javaupdate.cs** / **ea63f7eb9e3716fa620125689cfef1d5fed278ded90810e7c97db3b66b178a89** file. Does not seem to be a malicious code. (Figure 02)

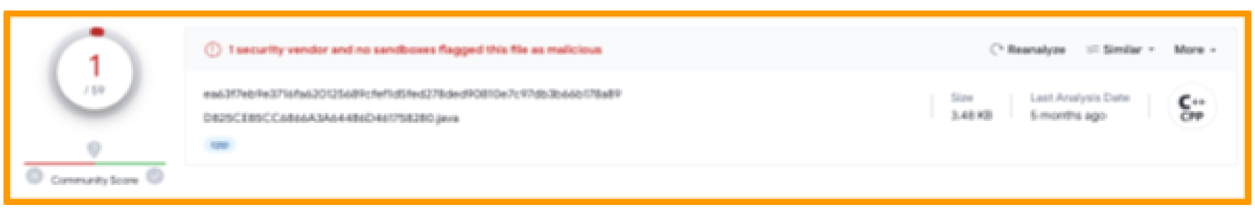

Figure 02

MetaDefender Result: No thread was found on the MetaDefender. (Figure 03)

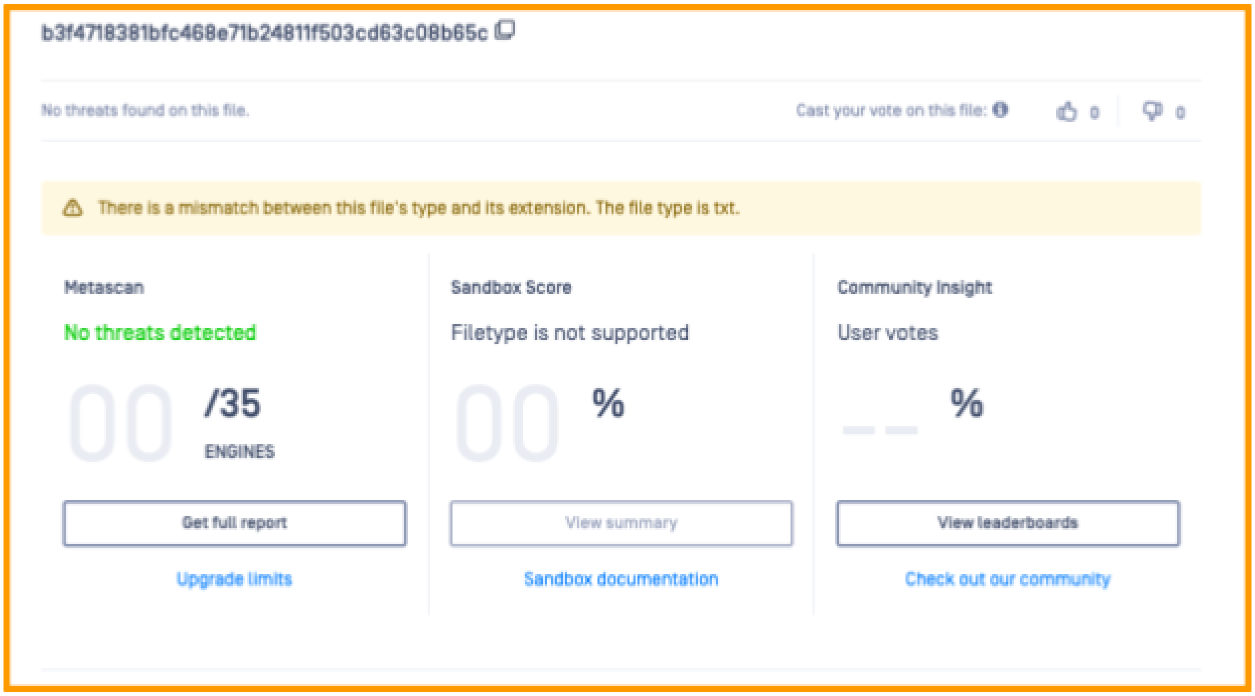

Figure 03

Code Analysis:
As this is a source code and written in C#, we can open up the **Malware.javaupdate.cs** file with **Microsoft Visual Code** and analyze the process. (Figure 04)

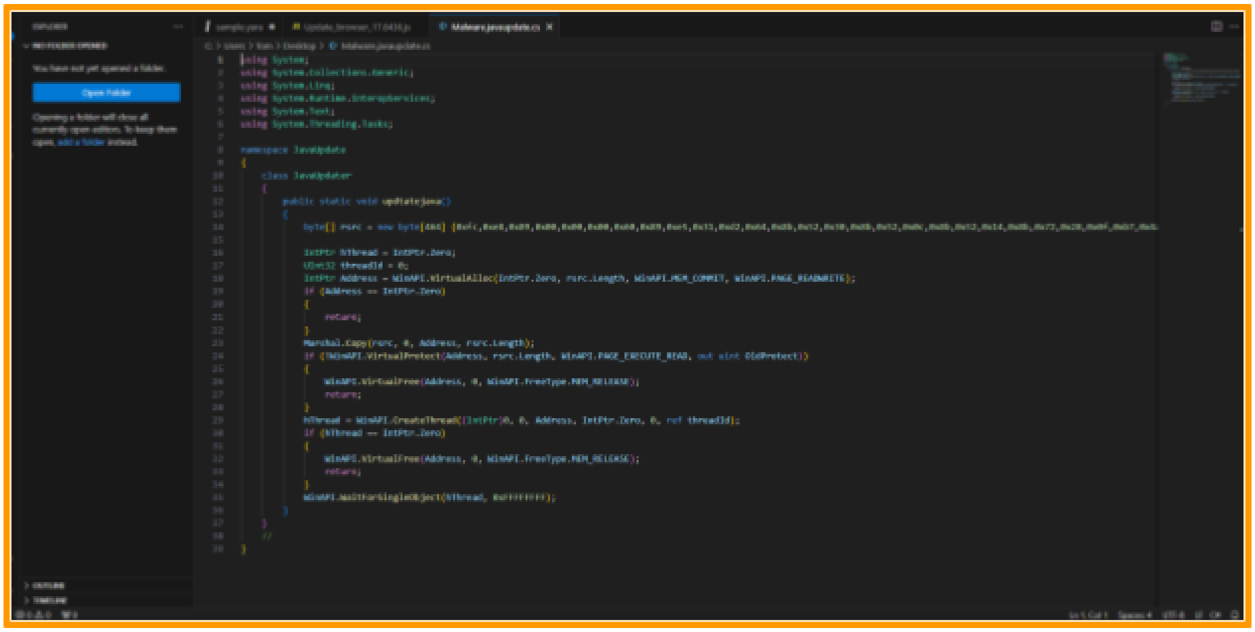

Figure 04

So this is a sample code in the C# class called “JavaUpdater” with a single method “updatejava()”.

The first thing that’s defined in this method is a **byte[ ]** array called “**rsrc”,** which may be our resource and it is a wide array of size 464.

So if we inspect the API calls that are being made by this particular snippet of code. We start with a “**VirtualAlloc**” which takes a section of memory and allocates it to a particular size. It then proceeds to copy the bytes from the resource into the “**Address**” that’s been allocated. It will check to see the memory protection on this section of memory and change it if necessary.

In the bottom, it will call “**CreateThread**” to execute a thread that is pointing to the address space and when the thread executes, it will execute whatever is in the byte array of resources.

Then, in the end, there is a call **“WaitForSingleObject”**, which puts the thread in a waiting state for the object for an indefinite amount of time, which is signified by the **“0x”** and then FFFFFFFF that is the end of the register.

So far, after investigation, it seems, this calls some API but nothing malicious unless the byte array has some malicious code inside of it. If these bytes do some kind of malicious activity, then this is definitely malware and it needs to be handled accordingly.

Separate the bytes into text file:

In the C# syntax, **0x** is going to denote as a hex byte and then the two characters after that are actual data of the hex byte.

To get the raw value of the data, copy the byte array into a text file (**ShellCode.txt**) and move it over to the **Remnux Machine**. Inside the Remnux Machine, we create a sample Python code (Figure 05) that removes everything except after "0x" which are two alphanumeric or numeric characters that are represented for the Hex Bytes.

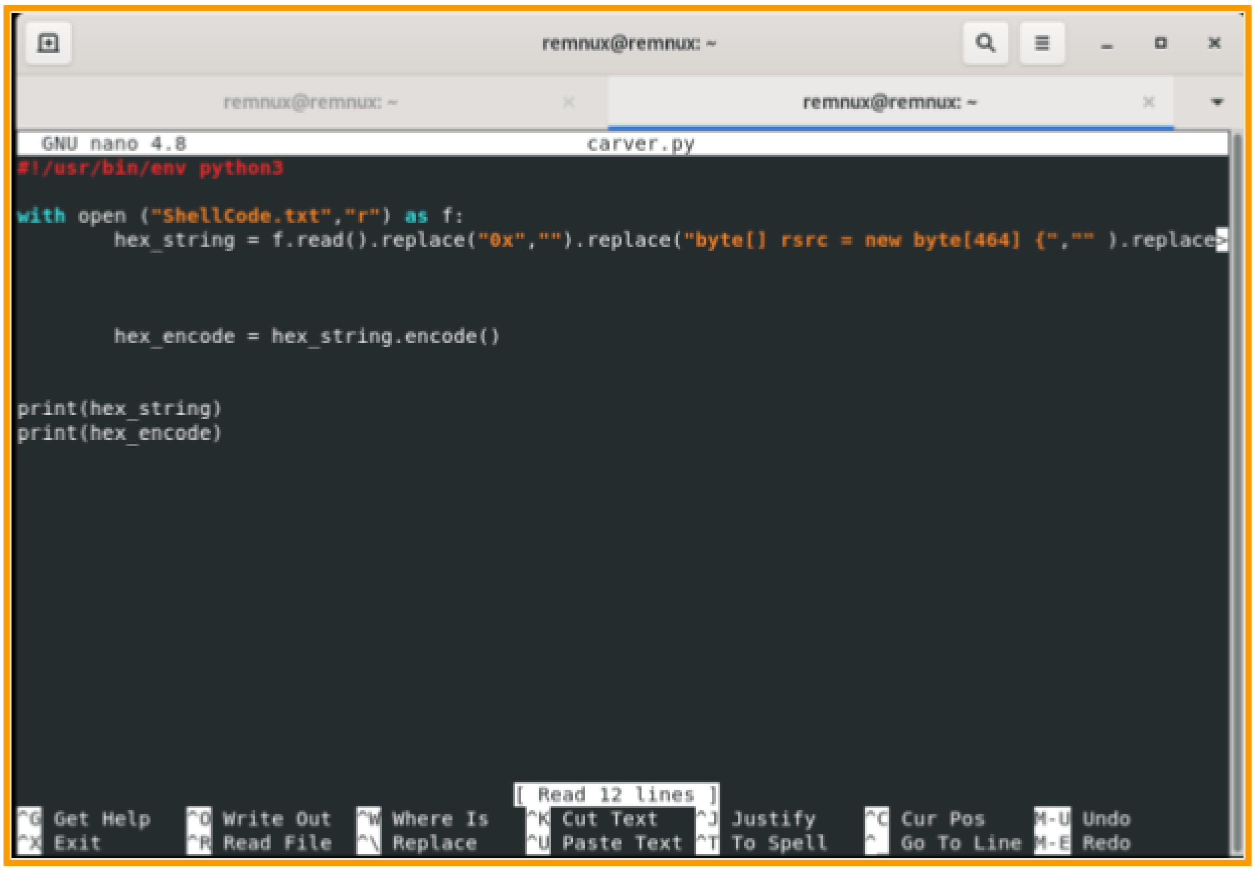

Figure 05

Inside the Python code, there is a print function that is written for test purposes, if it works or not. Now, if we run this code (Figure 06) we can see the result

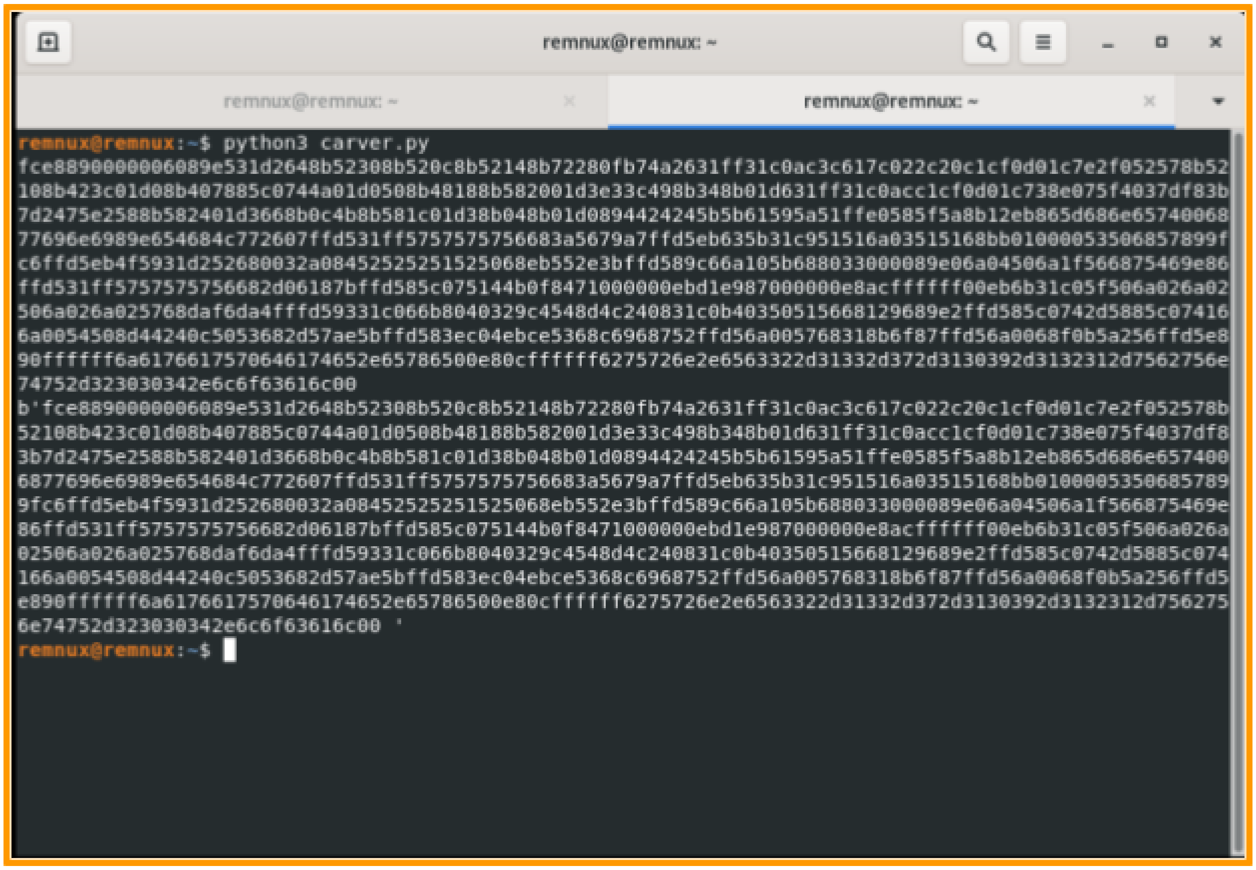

Figure 06

Now we can modify the code that will give us a file “**out.bin**” in **.bin** format which will store the data of the byte everything except raw hex value. (Figure 07)

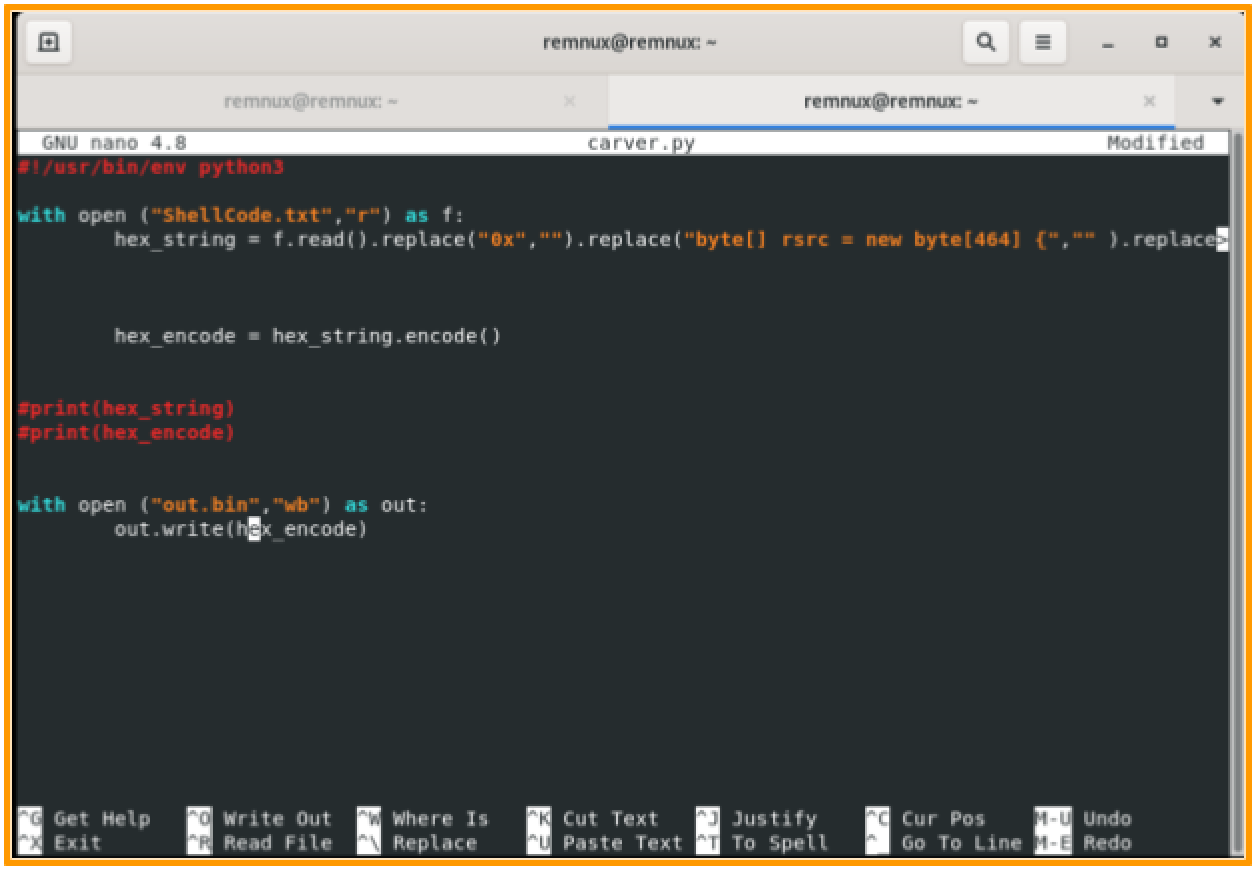

Figure 07

Now, we can run this file and make sure the file is written as we expected. (Figure 08)

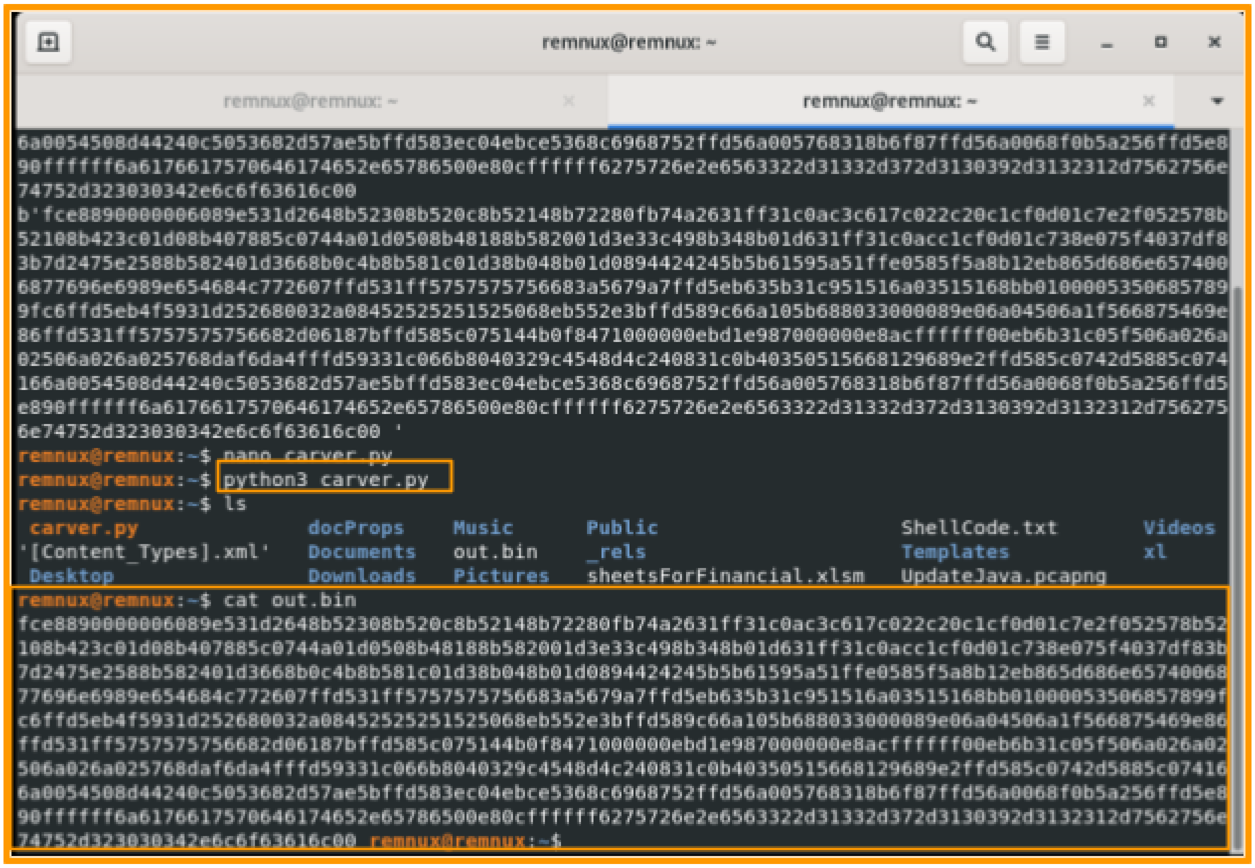

Figure 08

After that, we can save this file into our **Windows Machine** from PowerShell and open up an http server port in the Remnux Machine. (Figure 09)

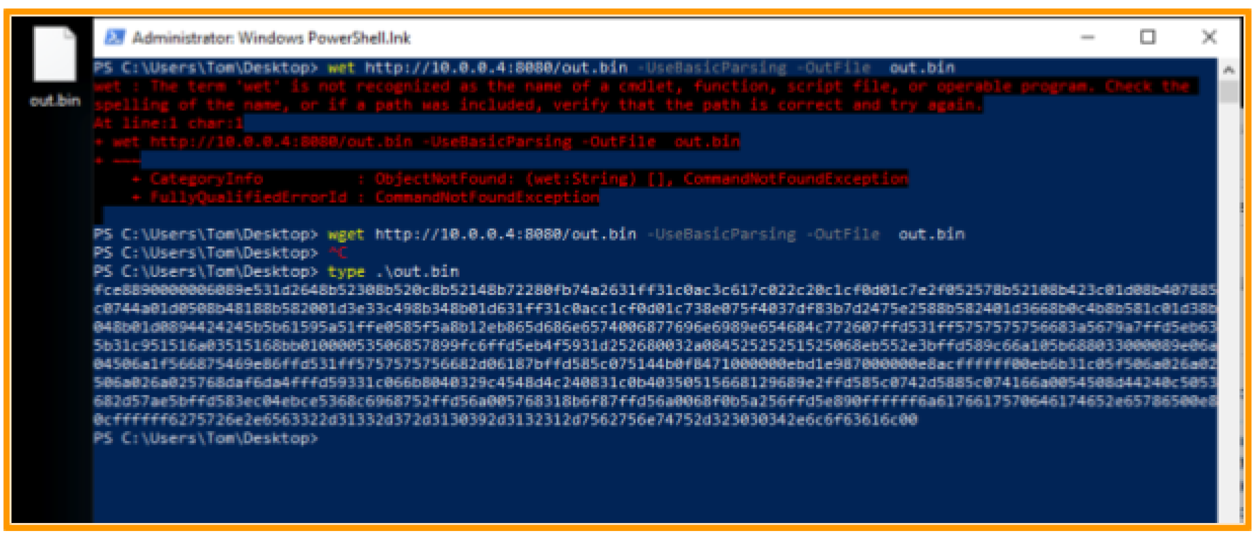

Figure 09

Dynamic Analysis

Using “scdbg” Tool for ShellCode Analyse:

ShellCode Debug (scdbg) is a tool that will interpret the bytes of the shell code and step through the program to resolve API calls and see what the shell code is doing ()

Note: scdbg does not run the shell code, but will resolve all of the things that the shell code is trying to do. (Figure 10)

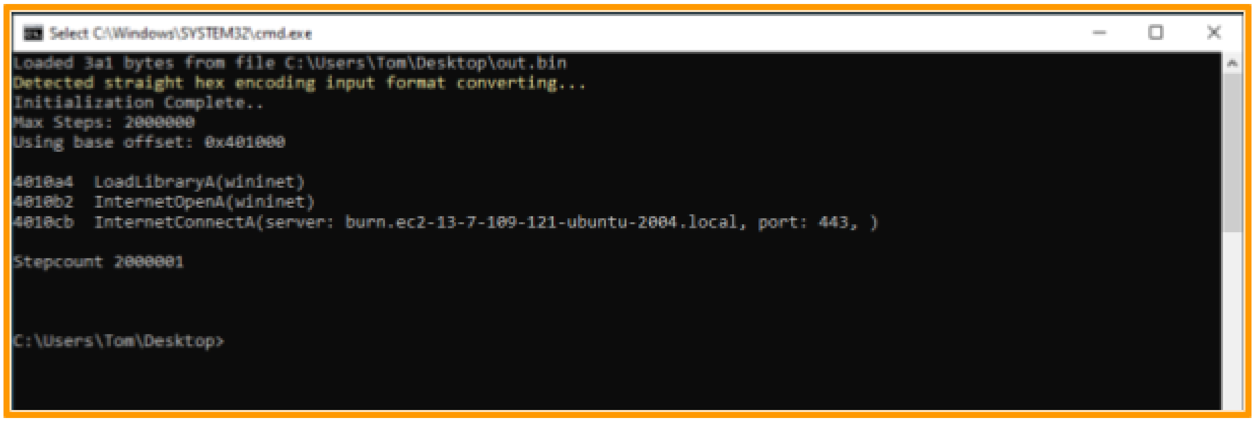

Figure 10

From the bottom of the static analysis section, we see several API calls.

The API “LoadLibraryA(wininet)” is attempting to load the WinINet dynamically. WinINet is the Windows Internet API, and loading it dynamically suggests that the malware intends to use functions from this library for internet-related operations. It may be involved in communication with external servers or fetching additional payloads.

API “InternetOpenA(wininet)” call opens a handle to the WinINet service. The malware is initializing its interaction with the WinINet service. This is a common step in malware that communicates over the internet. The opened handle is likely used in subsequent internet-related operations.

API “InternetConnectA(server: burn.ec2-13-7-109-121-ubuntu-2004.local, port: 443)” call attempts to establish a connection to a remote server. The malware is trying to connect to the server "burn.ec2-13-7-109-121-ubuntu-2004.local" on port 443. Port 443 is commonly associated with HTTPS, indicating that the malware might be attempting to communicate with the server over a secure channel. This connection may be for command and control (C2) purposes or for downloading additional payloads.

In summary, based on the API calls, the malware appears to be setting up communication with a remote server, potentially for command and control purposes. The use of WinINet functions indicates that internet-related operations are a key aspect of its behavior.