# Malware Analysis – Infostealer.Dyre

With the start of a new year, I've decided to start writing about something new – malware analysis.

Every now and then, we hear about a cyber attack on an organization that led to network outages and massive leaks of data. How do the attackers achieve this? How can attackers send commands remotely from their C&C center to a computer within an organization? The answer is – malware.

According to Verizon's Data Breach Report in 2015, 70-90% of malware samples were unique to an organization. This means that malware signatures are usually not available in Anti-Virus (AV) databases and no one really knows about the malware. The only option is to reverse engineer it and understand its purpose.

In this post, we will look at the analysis procedure of a malware sample called Infostealer.Dyre. The malware sample can be found in theZoo. Read this article for one of Dyre's real world exploits.

## Setting up the Test Lab

It is common knowledge that executing malware is not a good idea. So, we need an environment where we can safely analyse malware without the risk that it may spread across the network or damage our host system.

The test lab consists of the following systems:

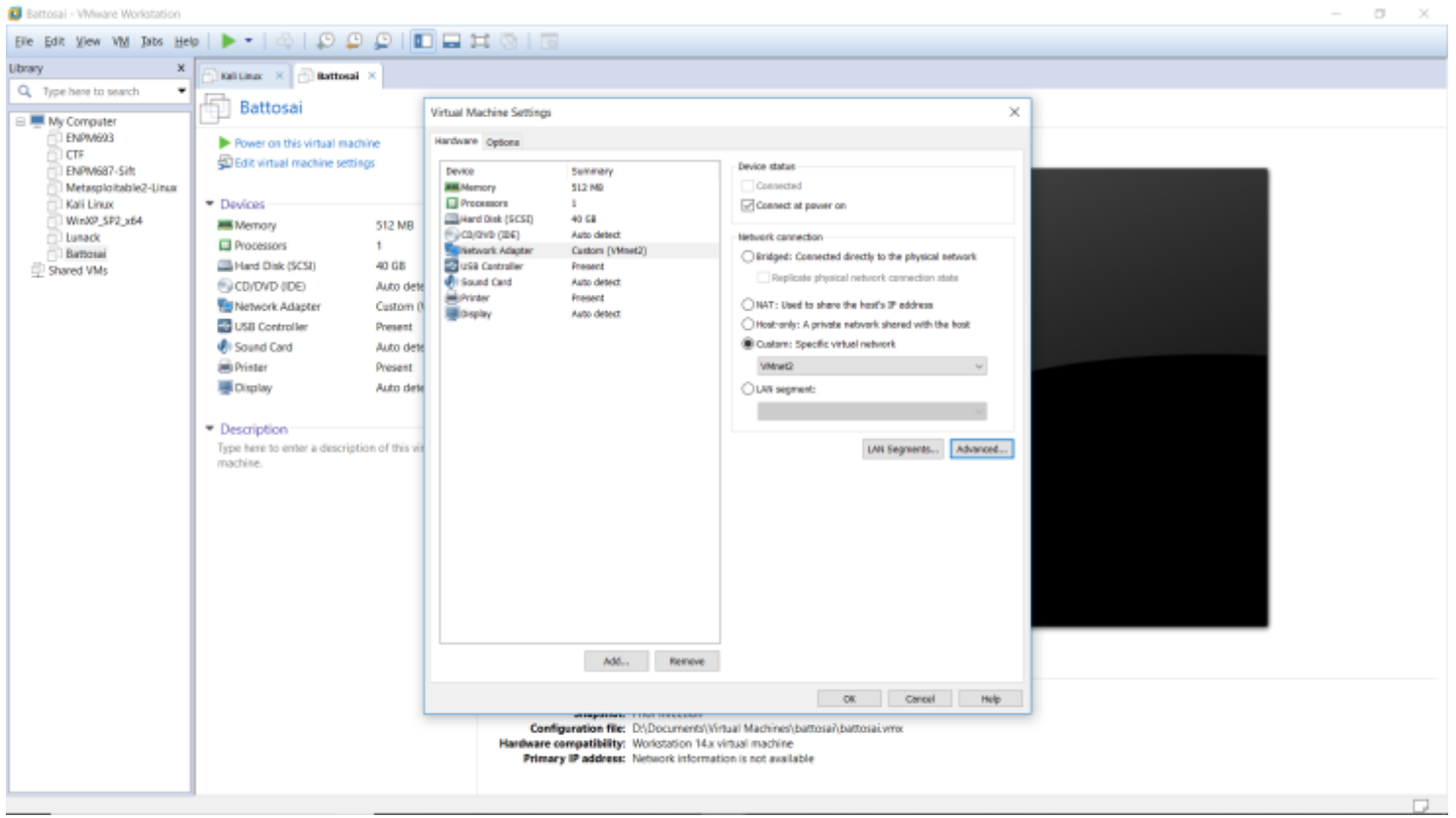
1. A 32-bit Windows XP VM on which the malware will be executed. It is to be set up in such a way that it has no access to the Internet but has a network interface facing another Kali Linux VM.

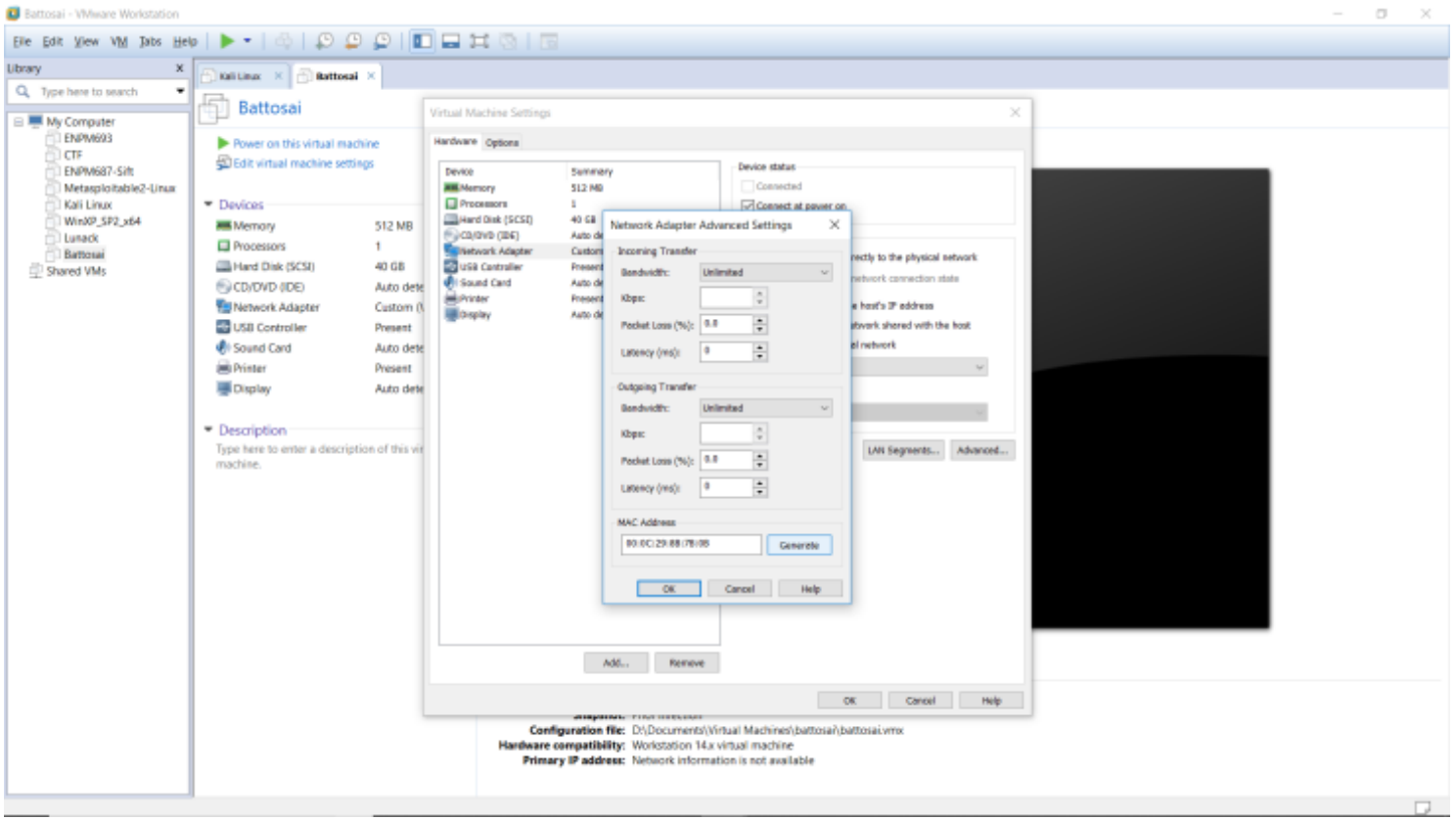
2. A Kali Linux VM which has one network interface connected to the Internet and another interface facing the Windows XP VM.

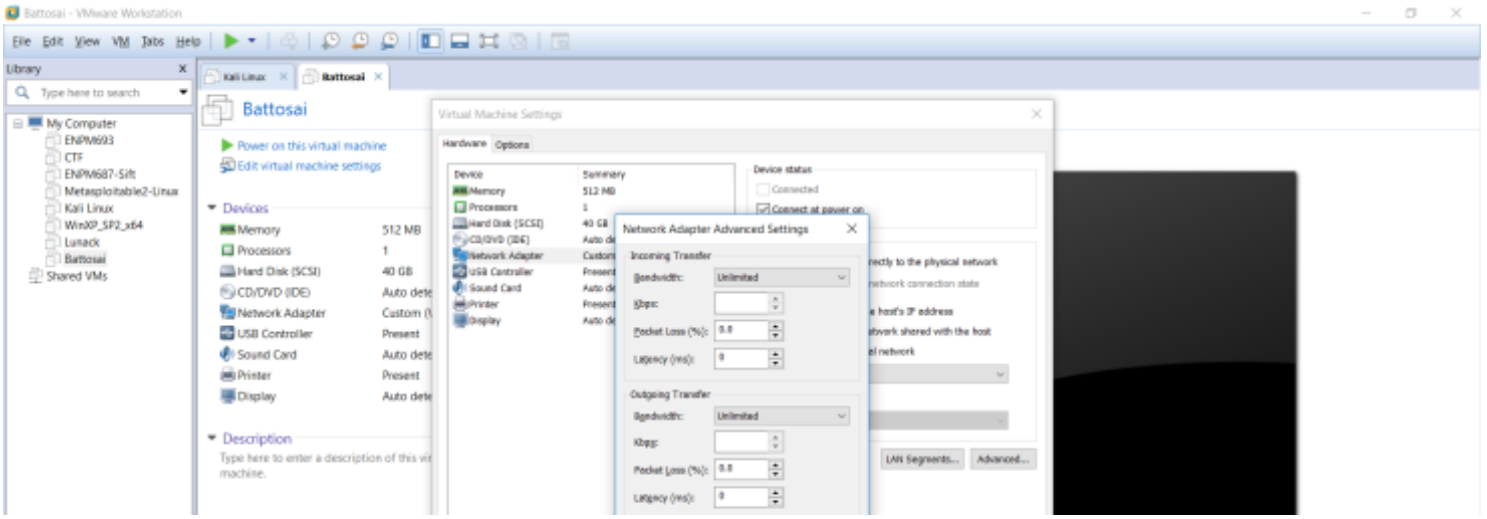
I use a Windows XP victim system because most malware are compatible with it.

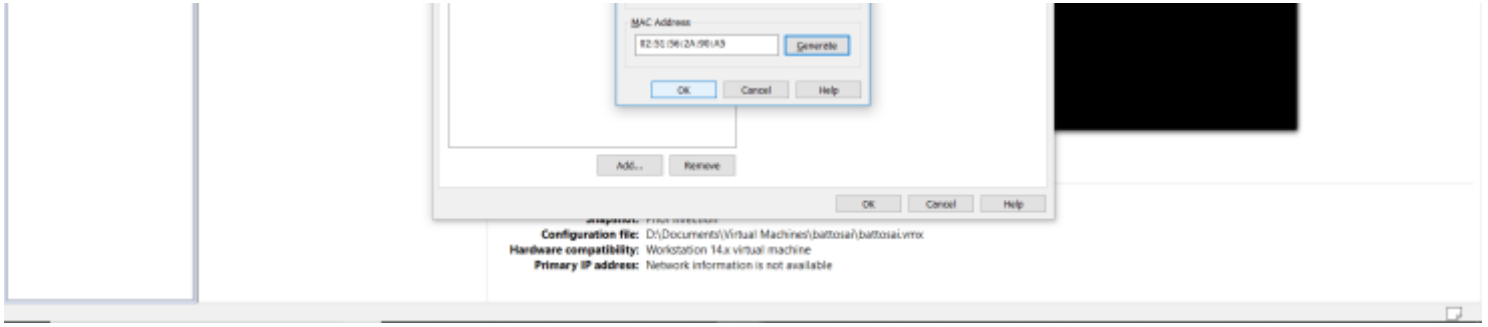
## Windows XP Setup

We need to ensure that the Windows XP VM is present in a separate VLAN and not connected to the Internet. Also, I'm going to randomize its MAC address (*Advanced* options) so that malware cannot identify (anti-VM technique) that the system it is running in is actually a VMWare VM.

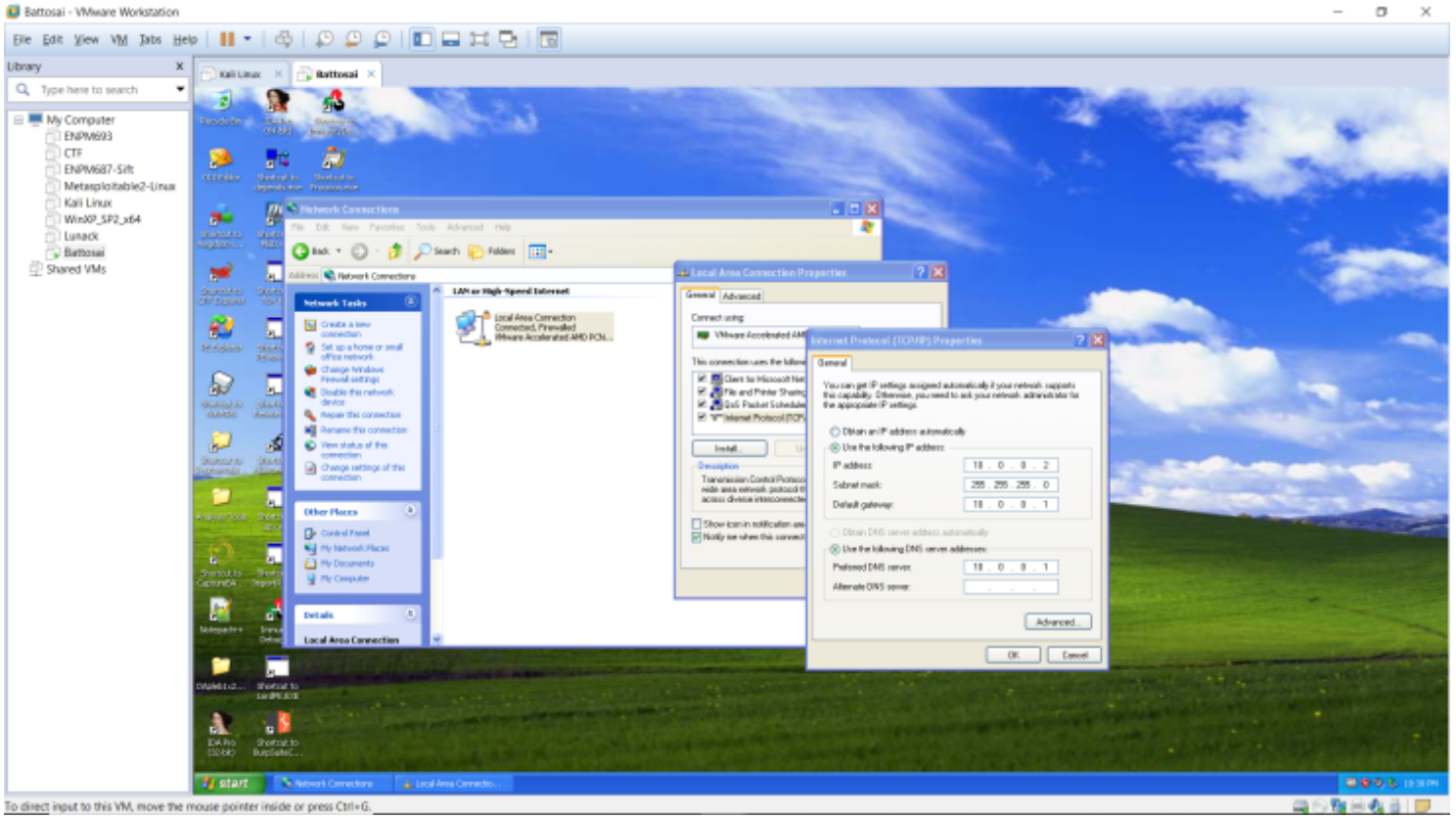




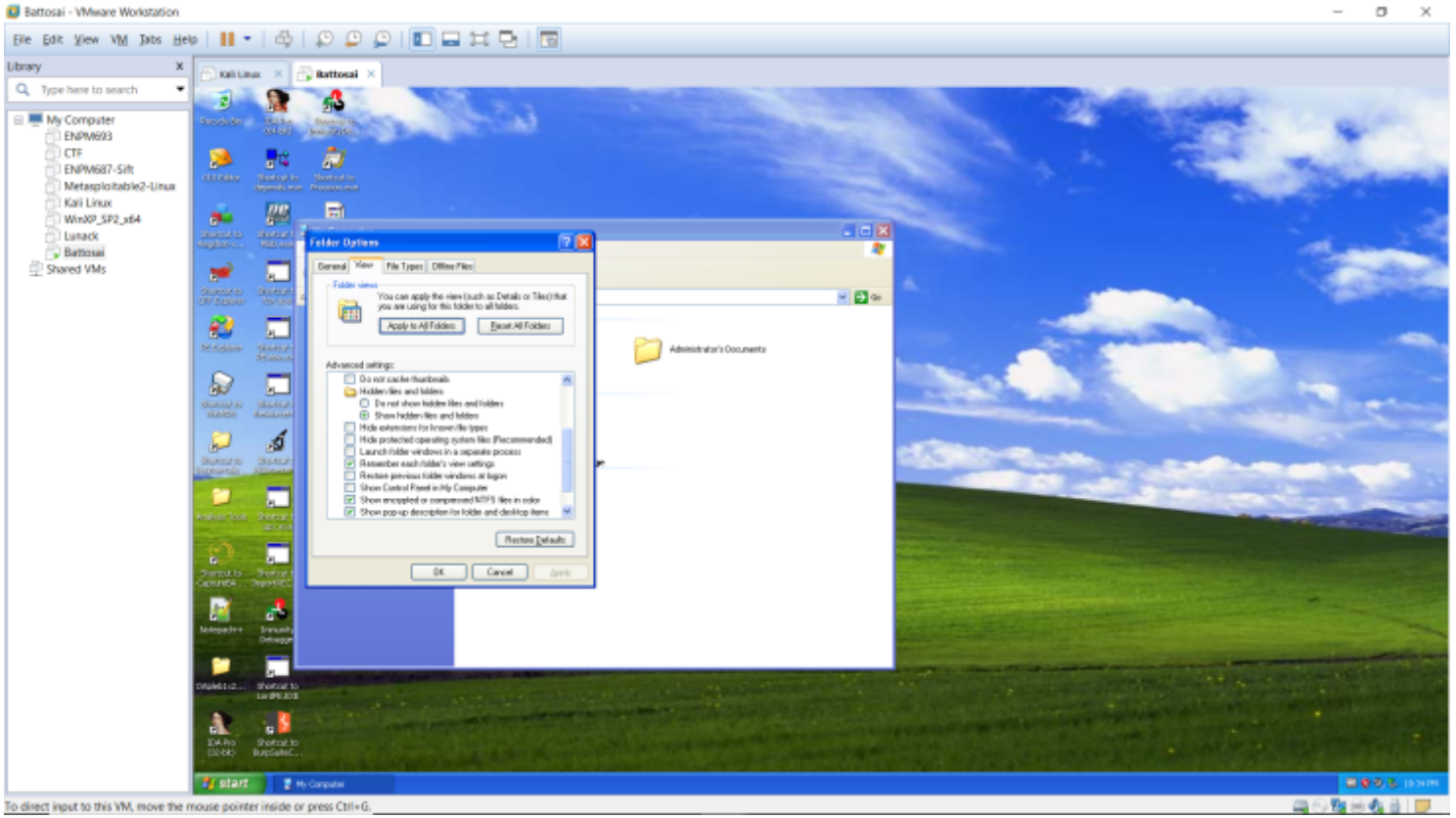



The next step is to assign an IP address to the Windows XP VM. This network interface will face the Kali Linux VM.



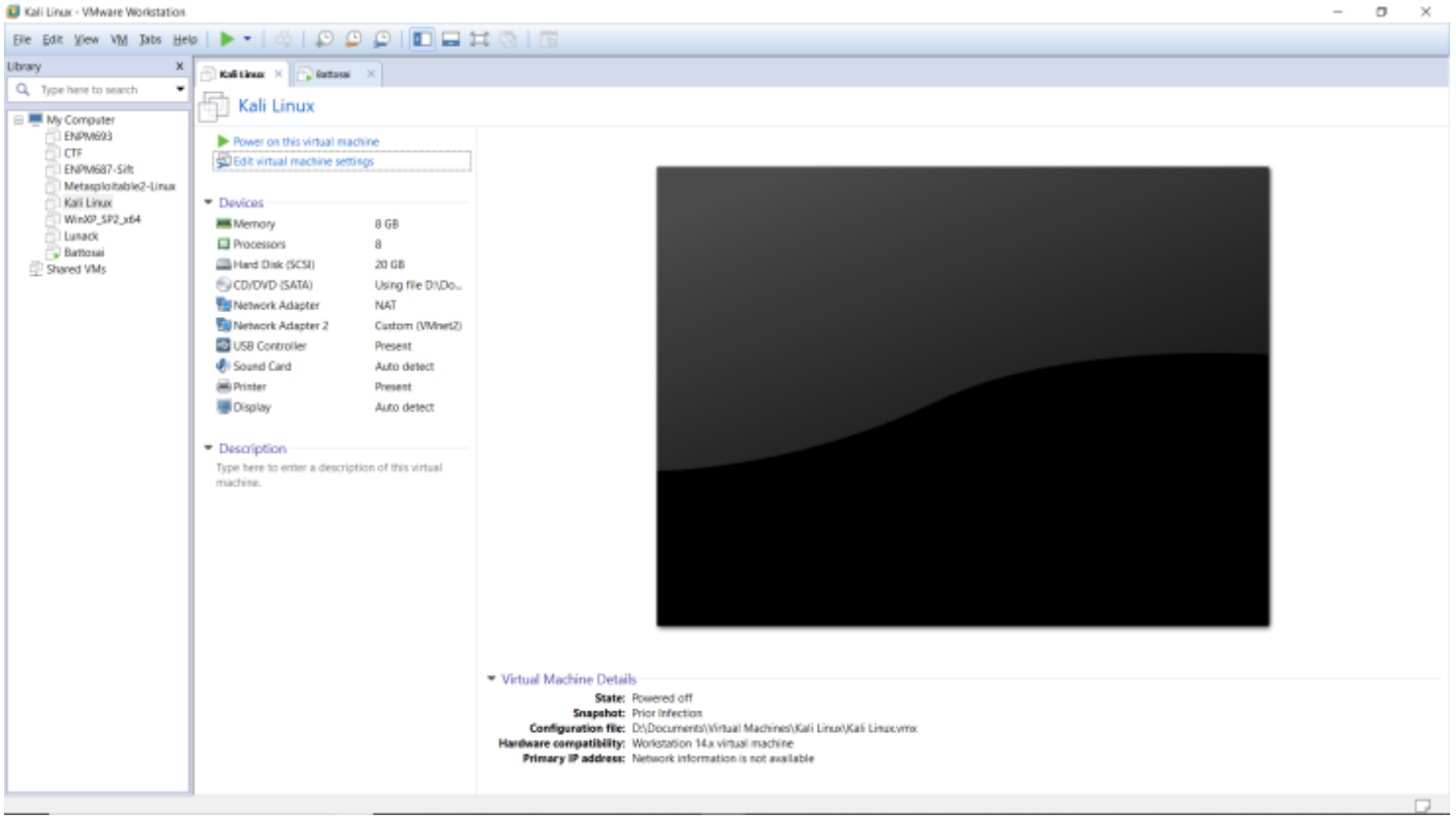

We will also make sure that no files are hidden. So, make sure to un-check the following in *Folder Options*:

1. Hide extensions for known file types
2. Hide protected operating system files (Recommended)
3. Do not show hidden files and folders

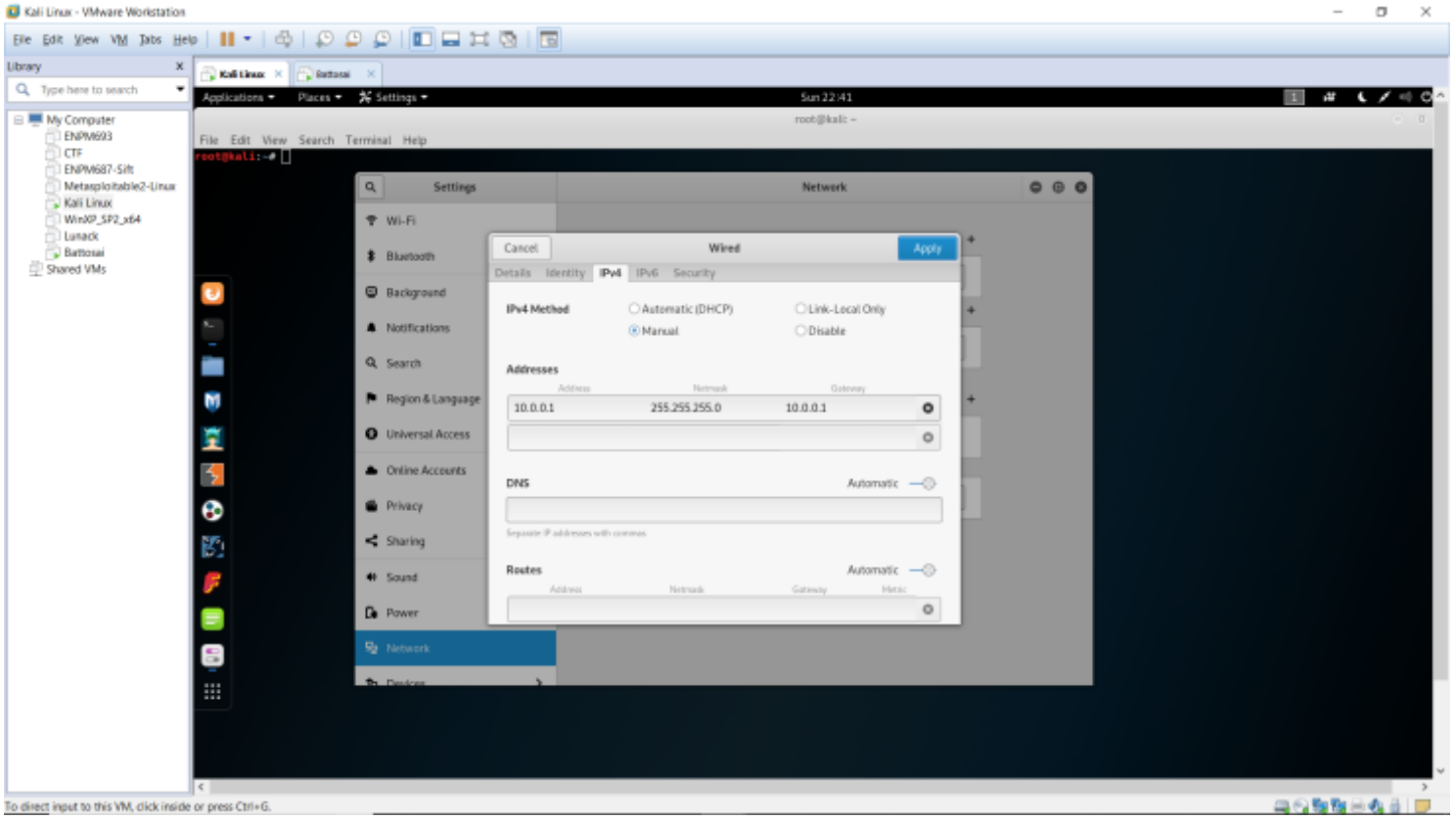



# Kali Linux Setup

Since we want two interfaces for the Kali Linux VM, we'll have to add an additional network interface which will be present on the same VLAN as the Windows XP VM.

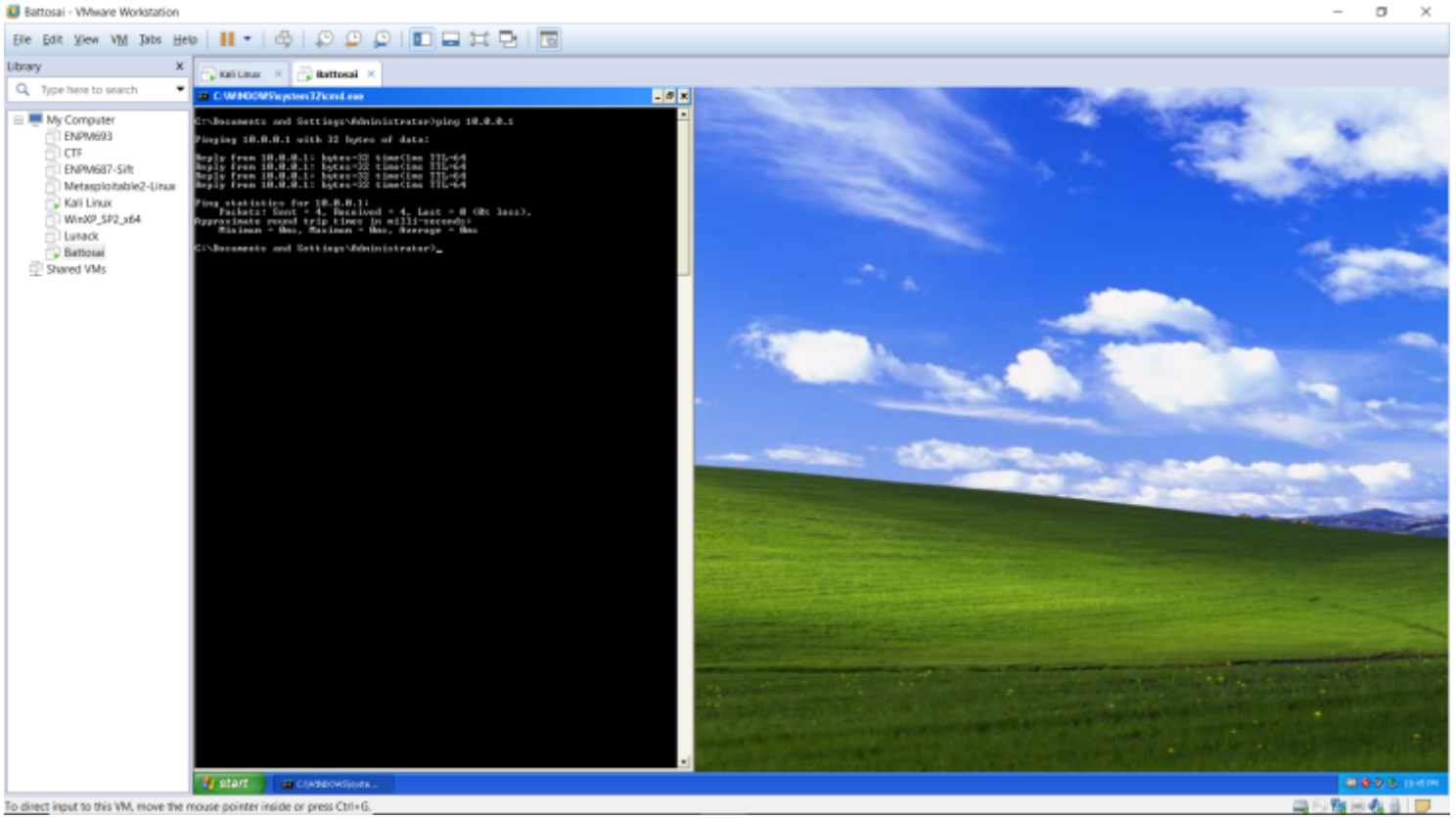



The next step is to set the IP address of the Kali Linux VM which will be in the same subnet as the Windows XP VM.




# Connection Verification

Ensure that the Windows XP VM is able to ping the Kali Linux VM.




# Malware Analysis Tools Installation

Analyzing a malware requires many tools, each used to answer a specific question. In this post, we'll require the following tools to be installed on the Windows XP machine:

1. 7Zip

2. 010 Editor
3. PEiD
4. PEView
5. ResourceHacker
6. CaptureBAT
7. Regshot
8. Autoruns
9. Process Explorer
10. IDA Pro

It is important to state that the choice of tools depends on the questions to be answered.

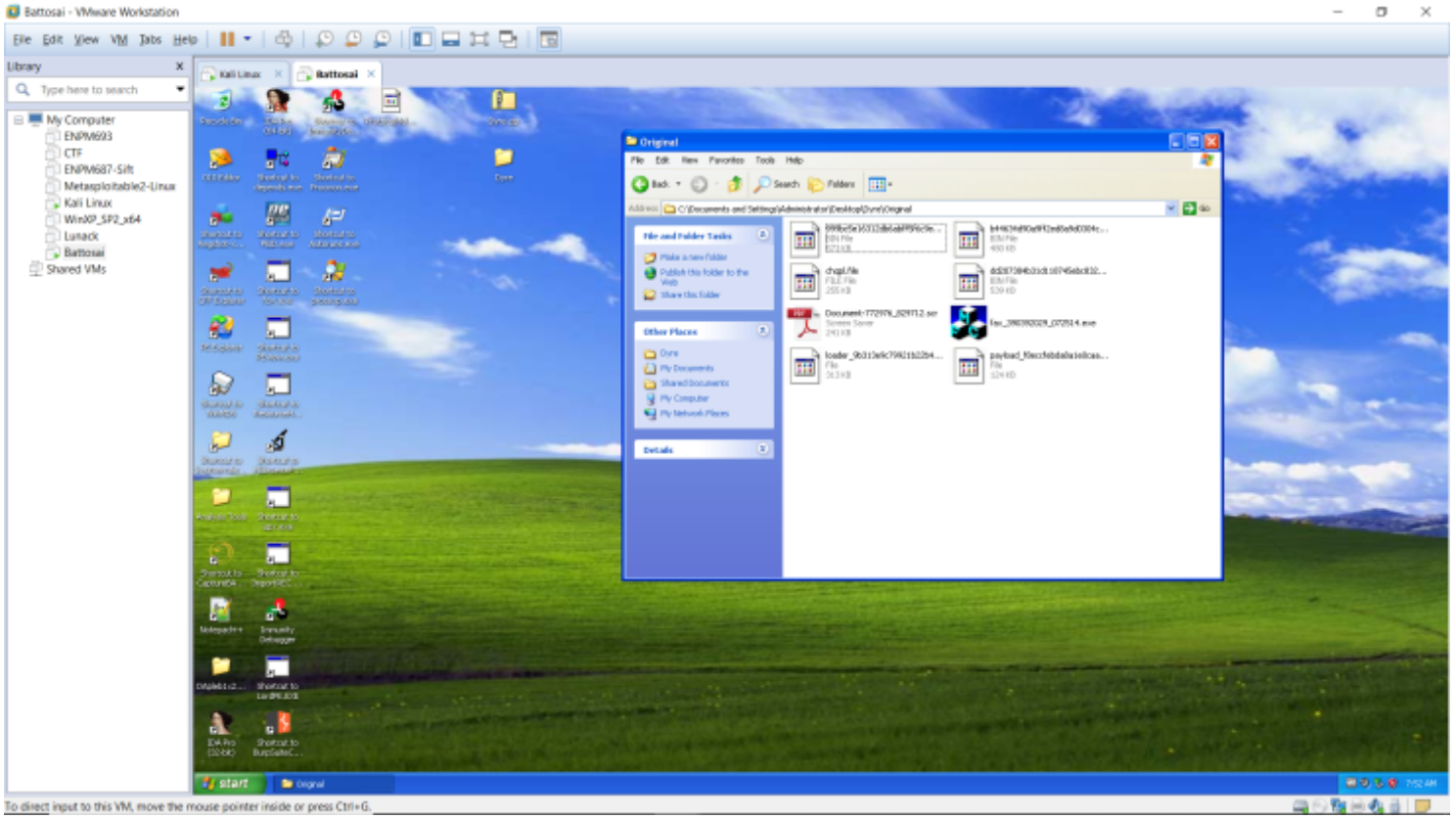
## Notes

After installation, take a snapshot of both the VMs. Snapshots are essential when analyzing malware because we'll need fresh images after every malware execution.

Note the MD5 hash of our malware sample – *c2d73485095efdbd7ab625e469affb11*.

# Analysis

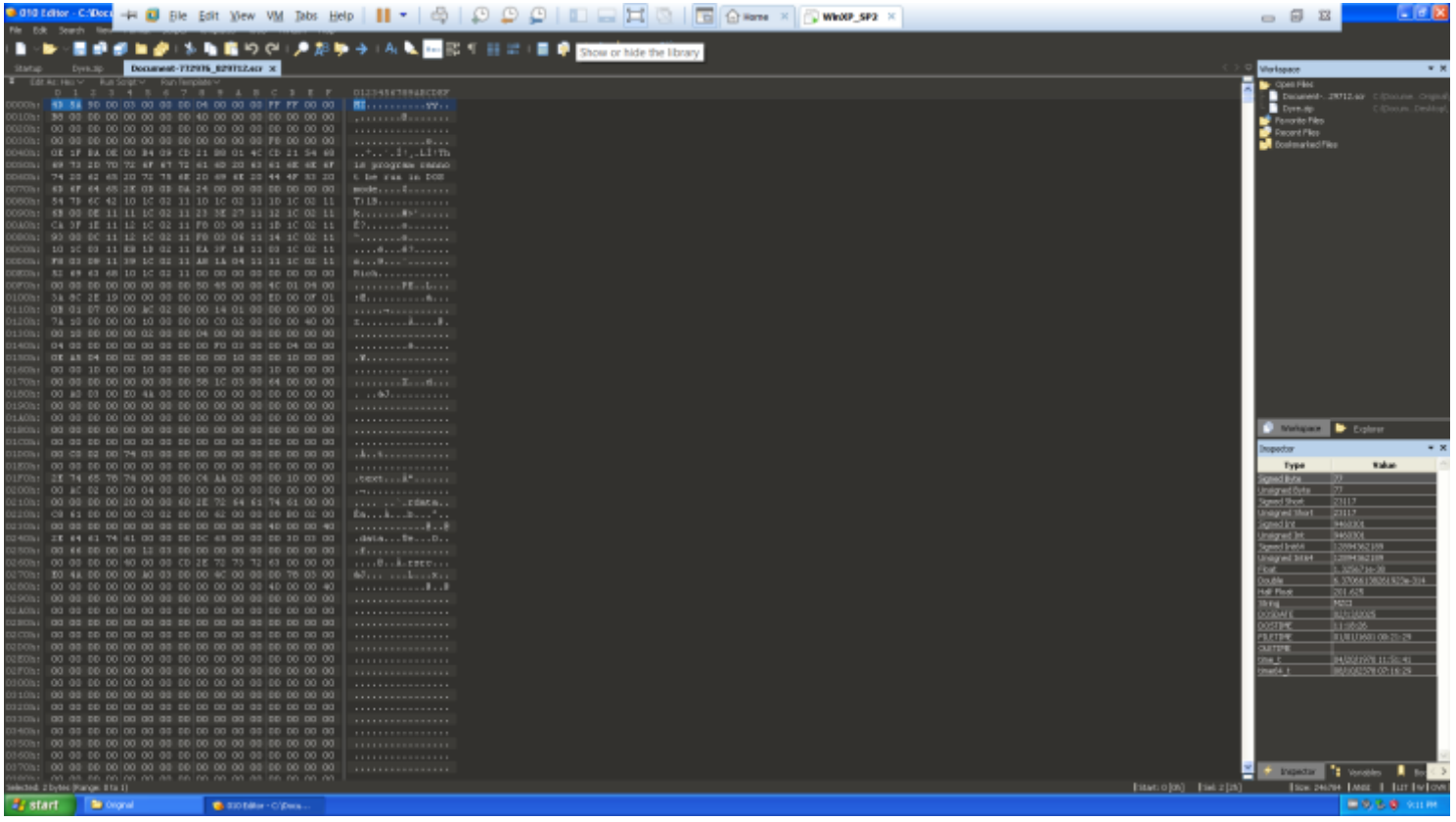
Place the malware sample on the Windows XP VM desktop (drag and drop) and unzip it using *7Zip* (password – *infected*). In the *Original* sub-folder, there is a file named *Document-772976_829712.scr* which is the malware sample that we'll analyze.



# Is the sample executable?

This question is quite unnecessary since the file extension is *.scr* and we know that it is a Windows executable. However, we will confirm it using *010 Editor*.

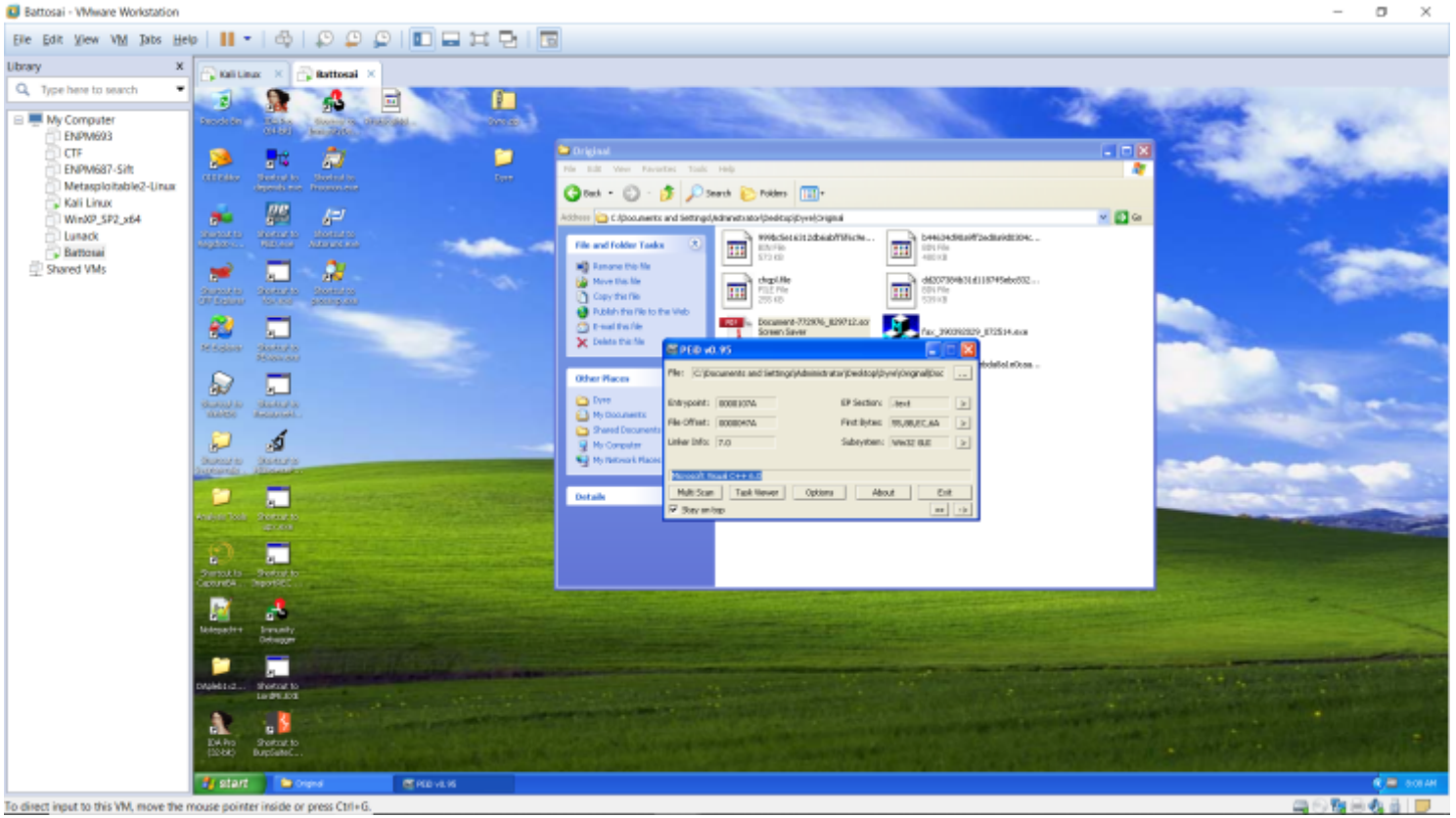
On opening the executable in *010* Editor, we notice that the first 2 bytes of the file is *MZ* in ASCII. This is the *magic number* that identifies a Windows executable.



# Is the sample packed?

Many malware authors obfuscate their code to make it difficult for malware analysts. Obfuscation techniques include packing, encryption, etc. It is common to find malwares that are packed using UPX. In such cases, the malware sample needs to be unpacked (using the UPX program) and then analyzed.

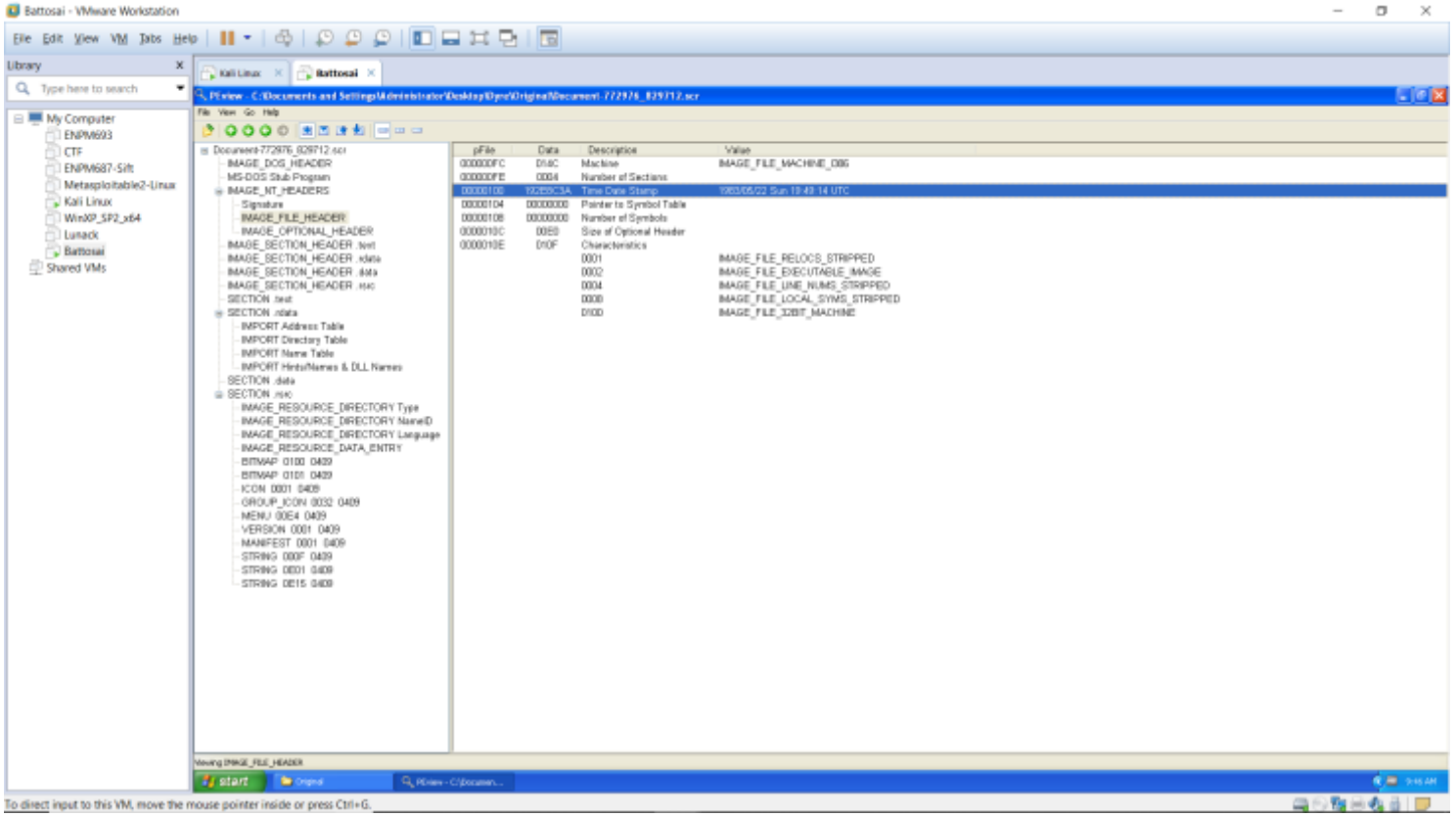
To check whether our malware sample is packed, we'll use the *PEiD* program. On opening the malware sample in *PEiD*, we notice that it recognizes the malware sample as a *Microsoft Visual C++ 6.0* executable. Since *PEiD* recognizes this, we can confirm that the malware sample is not packed.




## When was the sample compiled?

Knowledge of the malware sample's compile time can help us know if AV databases might contain its signatures. An old compile time suggests that the malware is not new and AV databases probably may contain its signature, whereas a newer compile time suggests that the malware may be part of a zero-day attack.

We can determine the compile time using *PEView*. On opening the sample in *PEView*, navigate to the *IMAGE_FILE_HEADER* section. Notice that the *Time Date Stamp* is *1983/05/22 Sun 19:49:14 UTC*.



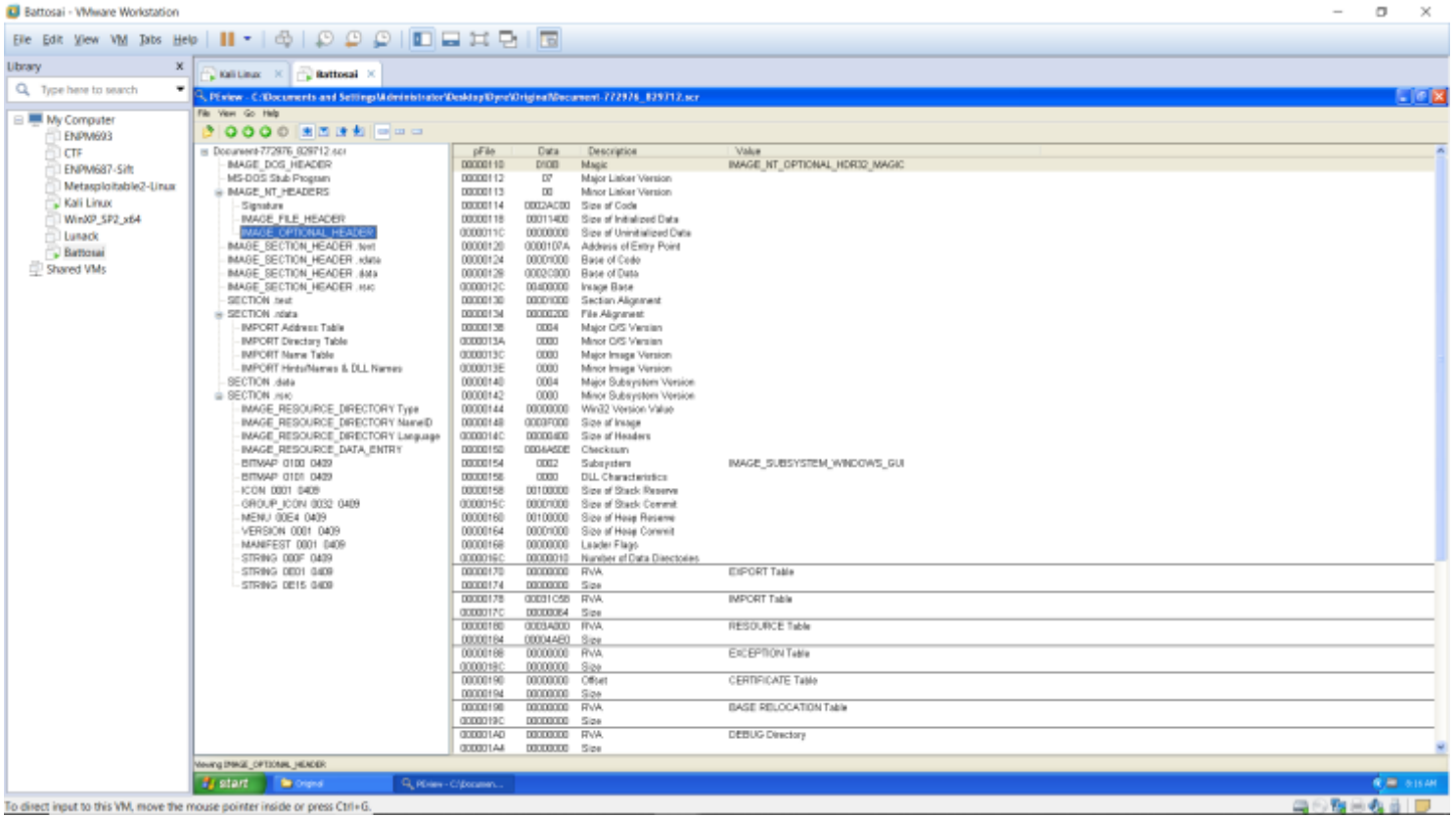

It seems like our malware sample is old and AV databases would likely contain its signatures. However, the crux of this post is to reverse engineer our malware sample, so we will ignore this information.

Keep in mind that malware authors can fake the compile time. So, if you don't find signatures of a supposedly old malware, it is possible that the compile time was faked.

# What sub-system does the sample operate in?

Some malwares are solely command-line based and do not have a GUI component. When such samples are executed, they run in the background and will not interact with any windows. For malwares with GUI components, they may interact with one or more windows during their execution.

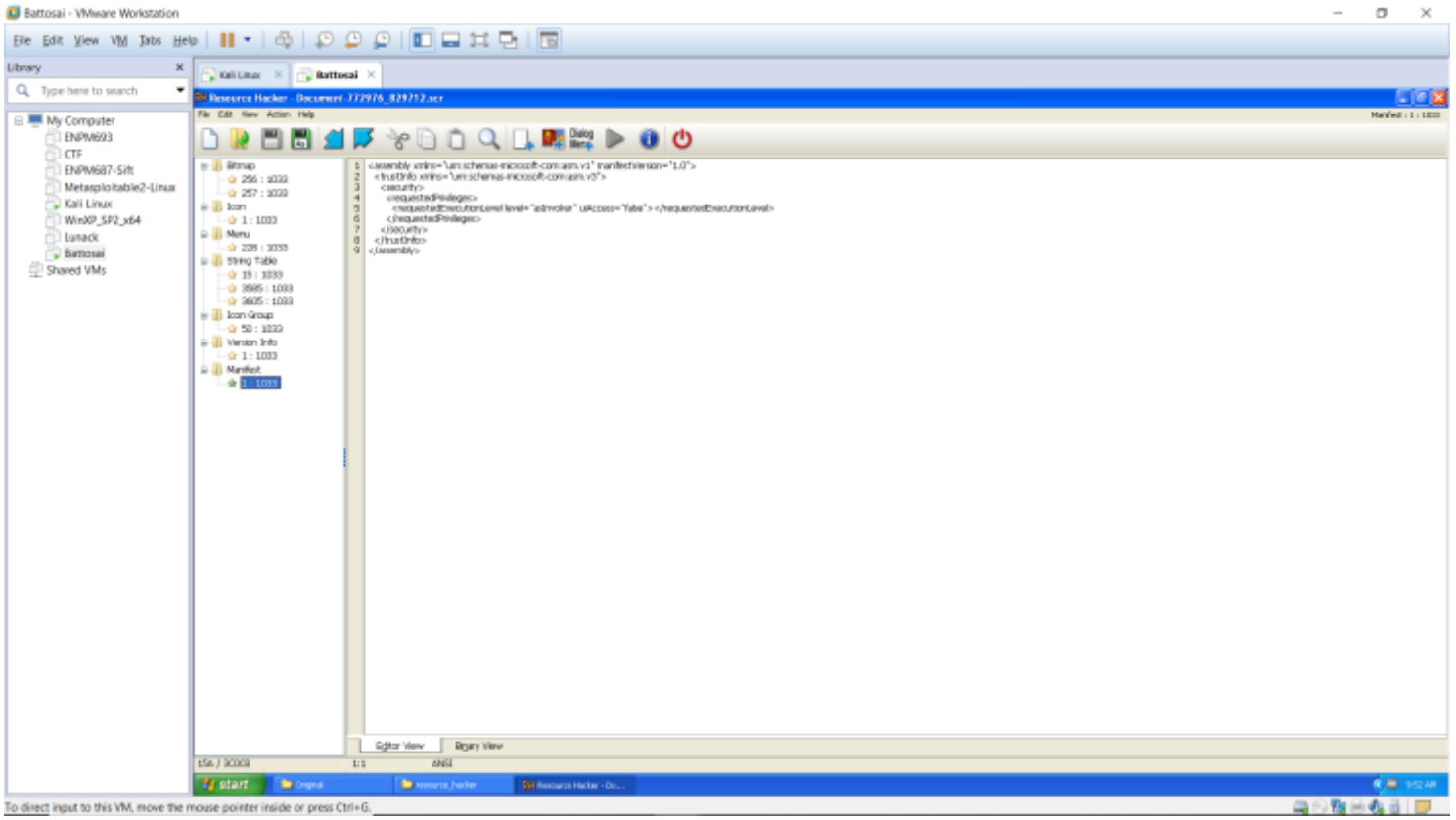
We can confirm the sub-system of our malware sample in *PEView*. On opening the sample in *PEView*, navigate to the *IMAGE_OPTIONAL_HEADER* section. Notice that the sub-system is *IMAGE_SUBSYSTEM_WINDOWS_GUI*. Therefore, our malware sample has a GUI component to it and will open one or more windows on execution.



## Does the sample have other embedded executables?

Some malwares have other embedded executables in them which are extracted during execution. In the *PEView* window, notice that our malware sample has a *.rsrc* section. It is not unheard of that some malware embed executables in the *.rsrc* section.

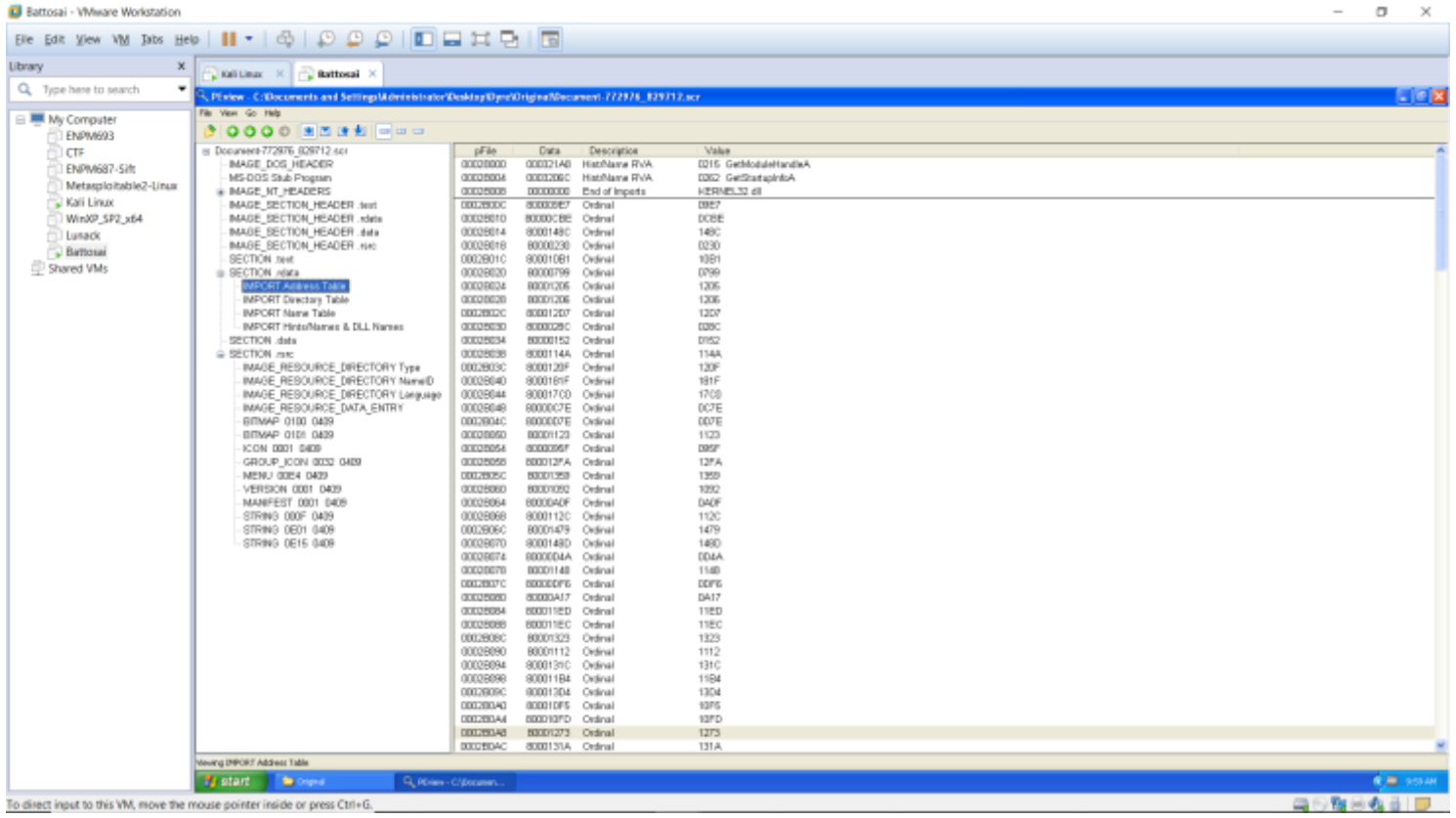
To determine if executables are embedded in the *.rsrc* section, we use *ResourceHacker*. On opening the sample in *ResourceHacker*, we can confirm that there are no other embedded executables in our malware sample.




## What functions does the sample import / export?

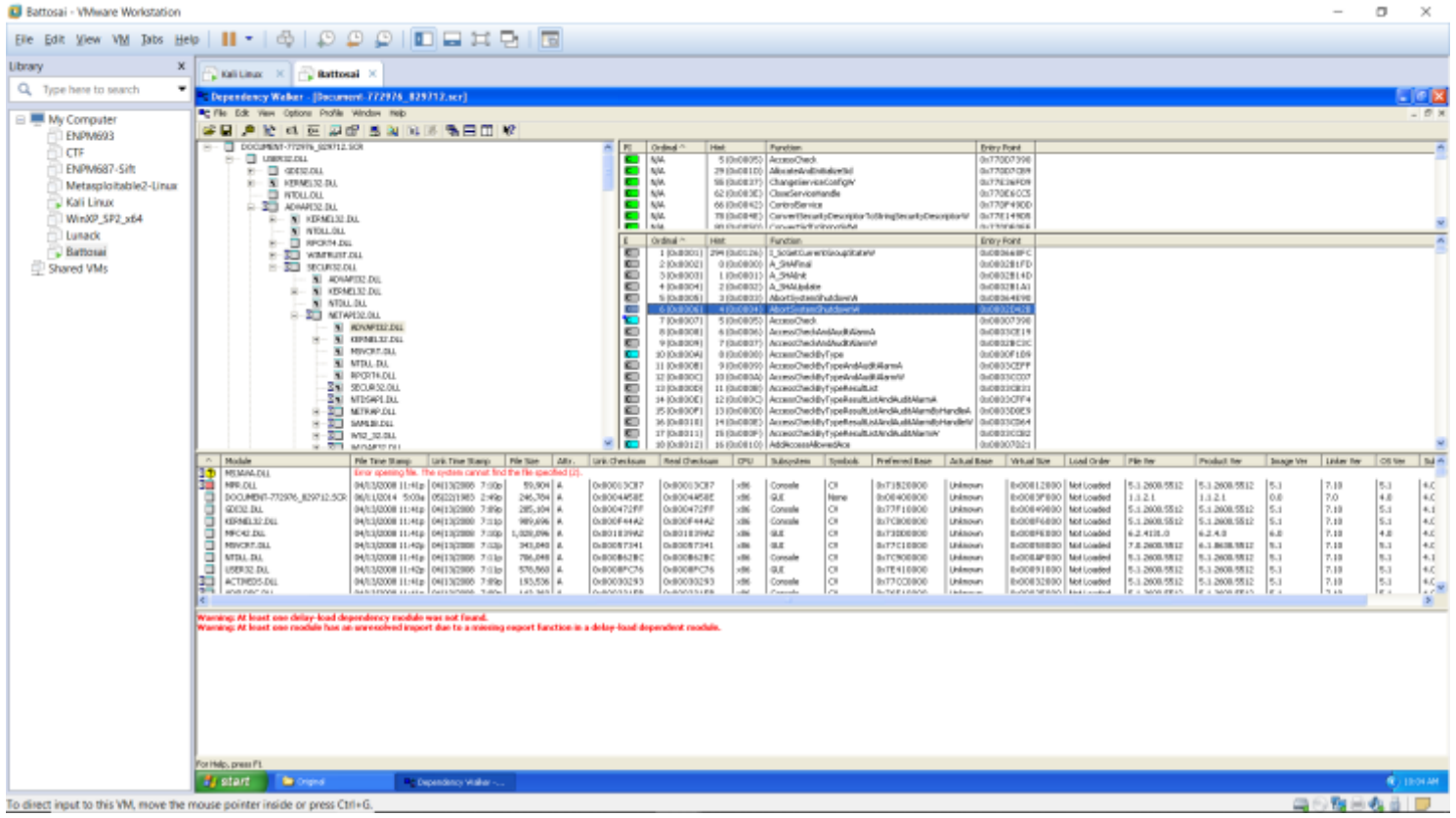
Knowing the functions that the malware imports / exports can help us understand the purpose of the malware. For example, if a malware sample uses the function *CreateMutexA()*, it suggests that the malware probably uses mutexes in its operation. With this knowledge, we can specifically look for strings that look like a mutex name.

We can determine which functions are imported / exported using *PEView* under the *.rdata* section.



Hmm… we cannot see any function names, but we can see hexadecimal characters. This is another way of importing functions in a program – through ordinal numbers. At this point, *PEView* is at its limits. To determine the function names, we need to use another application, *Dependency Walker*.

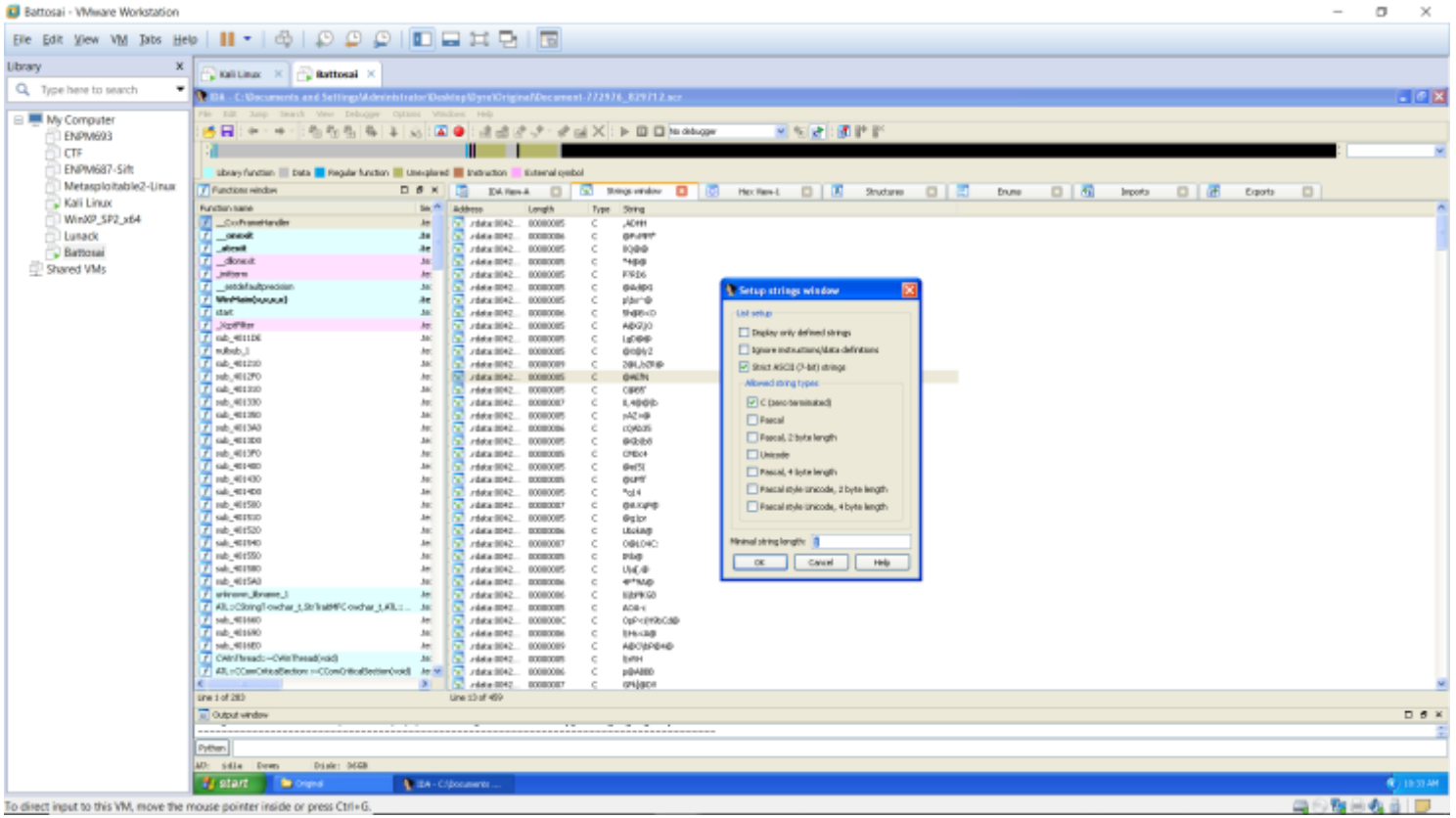
On opening the malware sample in *Dependency Walker*, we can see the function names and associated ordinal numbers in the top right window.

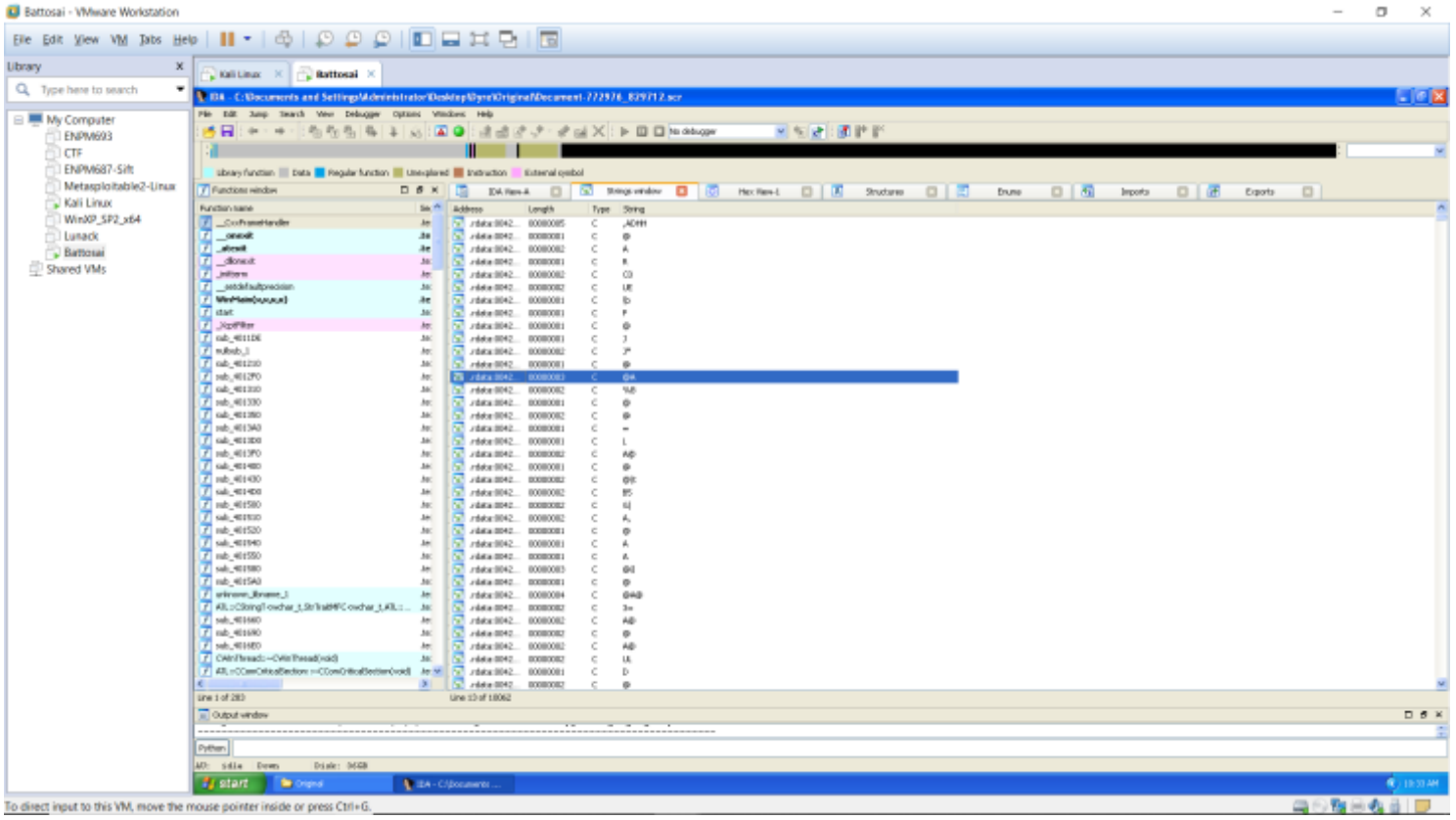


We can now crosscheck the ordinal numbers from *PEView* and determine the function names from *Dependency Walker*. As you can imagine, this is quite laborious and in my opinion the outcome is not as rewarding. So, I'm going to skip this step. At this point, we have no idea what functions are being imported / exported.

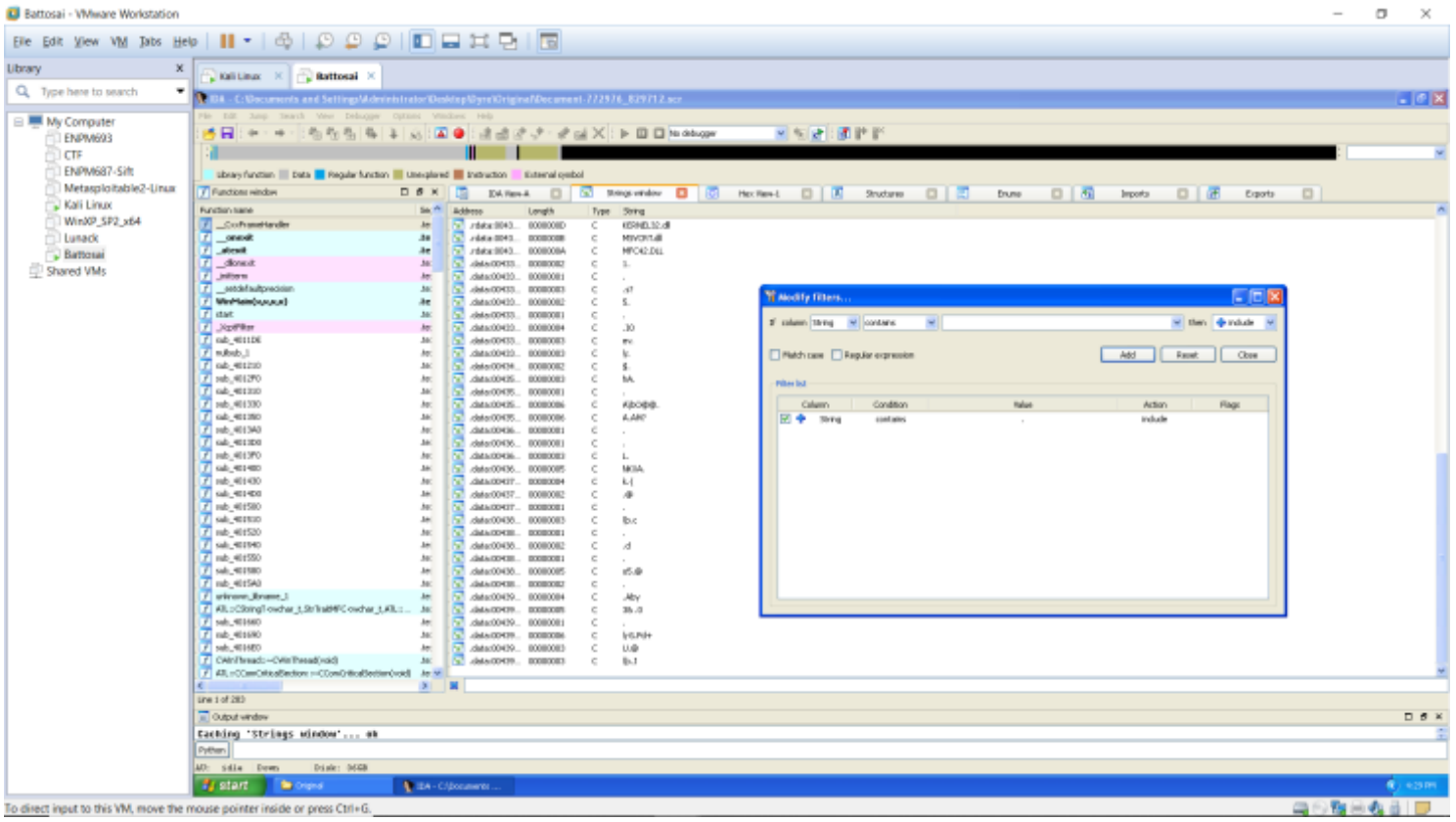
## Does the sample have any clear text strings?

Clear text strings are great for finding IP addresses, domain names, mutex names, etc. We can check for clear text strings using *IDA Pro* under the *Strings Sub View*. Ensure that the *Minimal String Length* is set to 1. This allows *IDA Pro* to display strings which may be short command line arguments.





We can see that there are loads of strings in the window, but they are incomprehensible. We can put a filter to parse through the results. I filtered the *String* column to contain a period character, but as we can see there are no interesting strings.



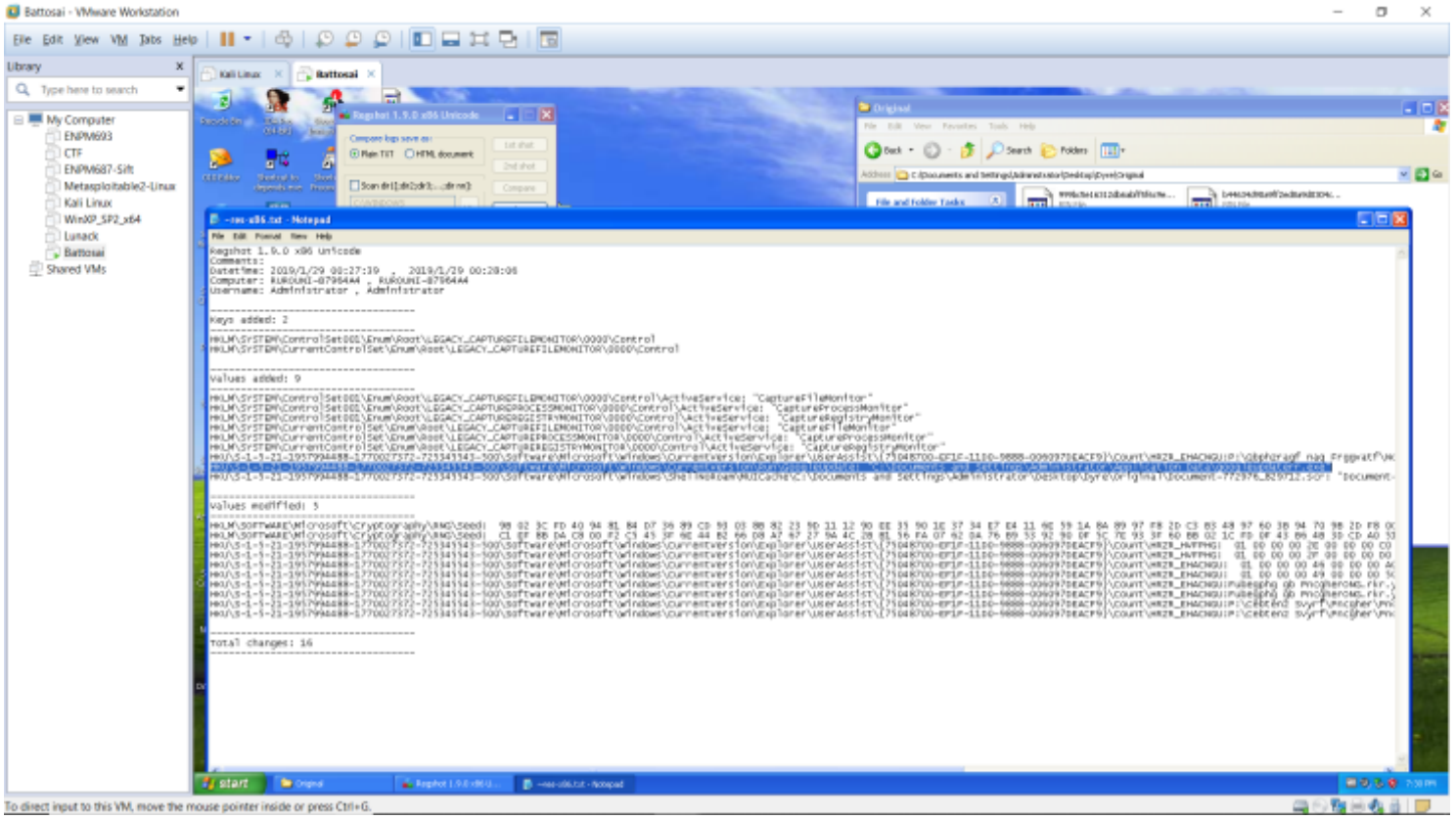
This suggests that critical strings in the malware sample may be encrypted and are probably decrypted at runtime.

**Note:** At this point, we have reached a dead end as far as basic static analysis goes. After this point, basic dynamic analysis (malware will be executed) would be able to give us more information. Ensure that you have a snapshot of the Windows XP VM ready.

# What kind of changes are introduced in the Windows Registry?

Many malwares are built to persist on the victim system. One of the techniques to achieve persistence is through the registry.

We can track the changes to the registry through *Regshot*. The first snap of the registry is taken prior to malware execution and the next snap is taken after malware execution. On comparing the two snaps, we can determine the changes to the registry.

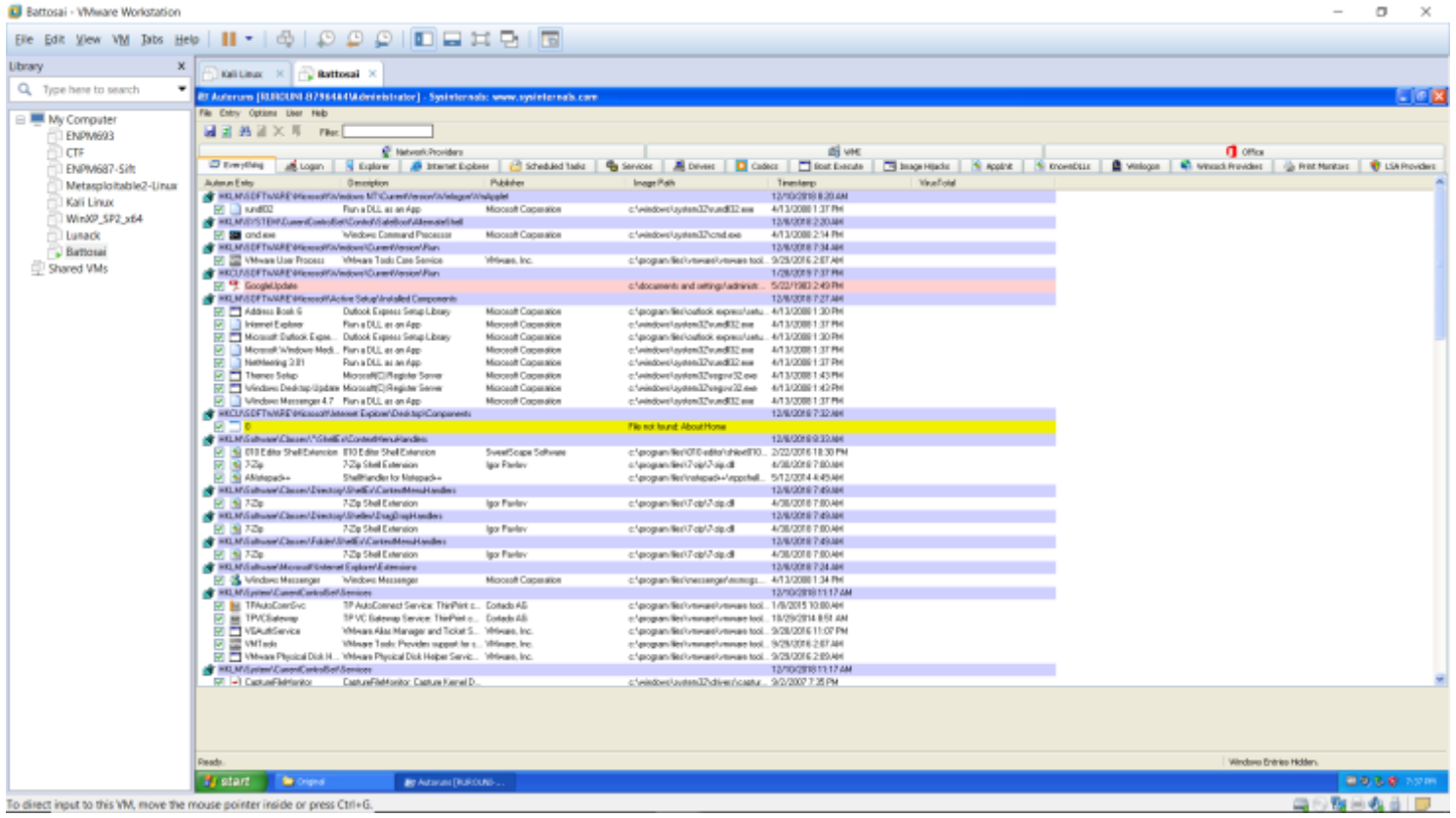


Most of the changes are irrelevant, but there is one entry which is VERY important and is proof that the malware achieves persistence.

HKU\S-1-5-21-1957994488-1770027372-725345543-500\Software\Microsoft\Windows\CurrentVersion\Run\GoogleUpdate: "C:\Documents and Settings\Administrator\Application Data\googleupdaterr.exe"

# Are there any startup services that are installed by the sample?

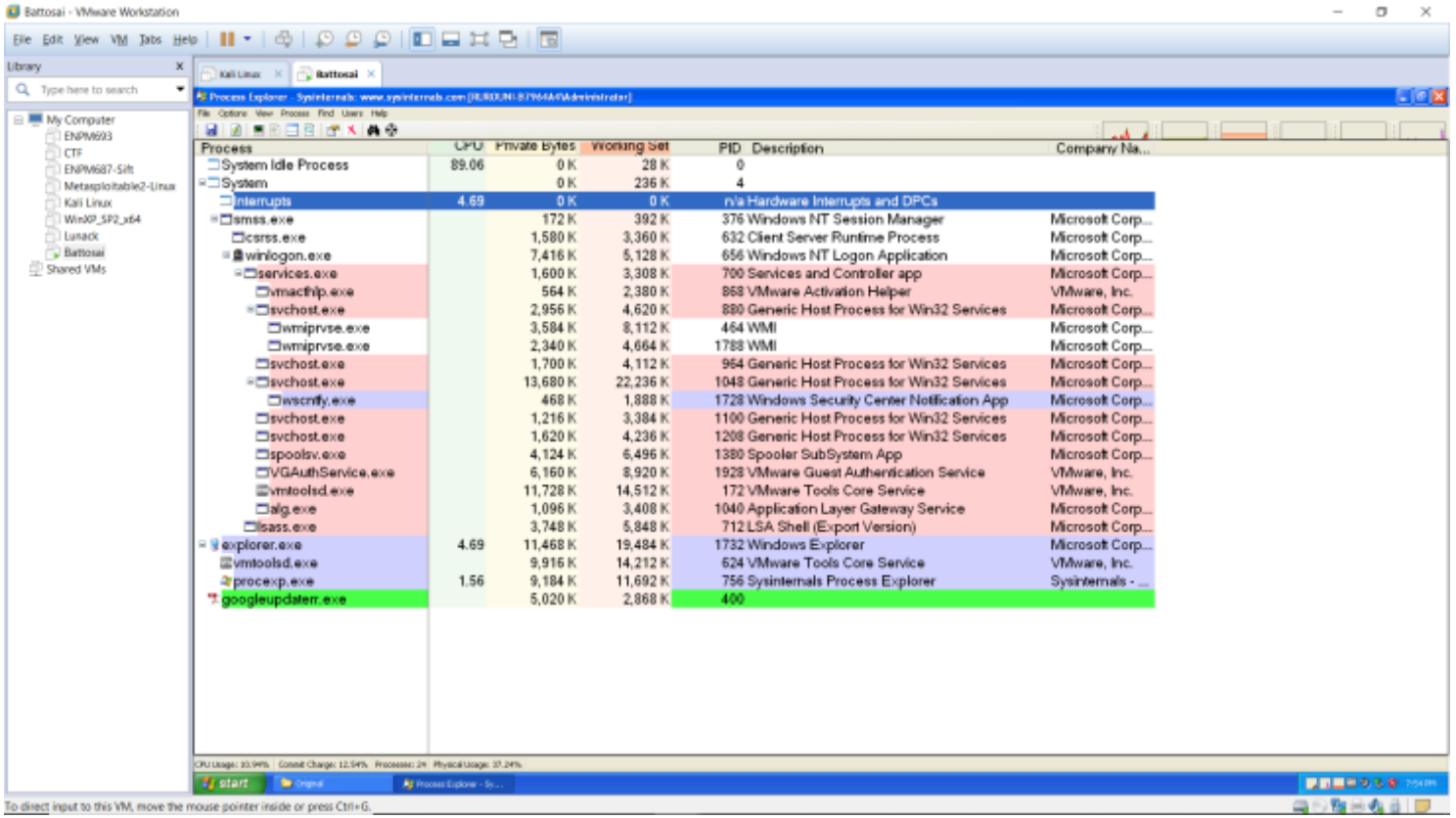
Many malwares achieve persistence and may install themselves as a startup service. This can be checked using *Autoruns*. If there are any malicious services that are installed after malware execution, *Autoruns* highlights them in red.



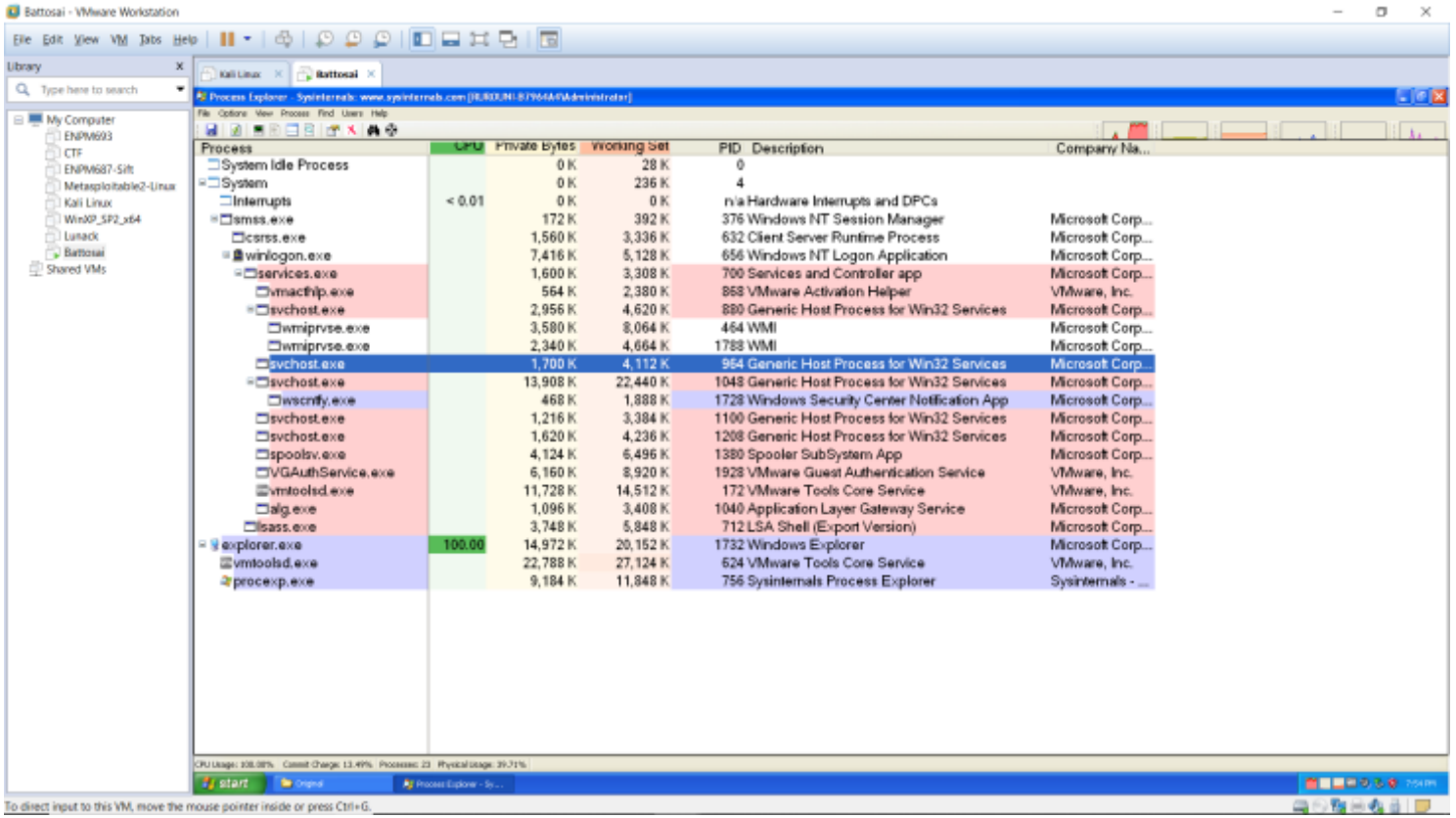
We can see that there was a service, *GoogleUpdate* that was installed as a startup service and runs from *C:\Documents and Settings\Administrator\Application Data\googleupdaterr.exe*. This is the same application that was introduced into the Windows Registry as we noted before.

## What processes are running when the sample is executed?

When a malware sample is executed, it can spawn multiple sub-processes in the background or may also inject itself into other legitimate processes. Processes running on the system can be tracked through *Process Explorer*.




We can see that *googleupdaterr.exe* was spawned when the malware sample was executed. However, it terminates after about a minute.

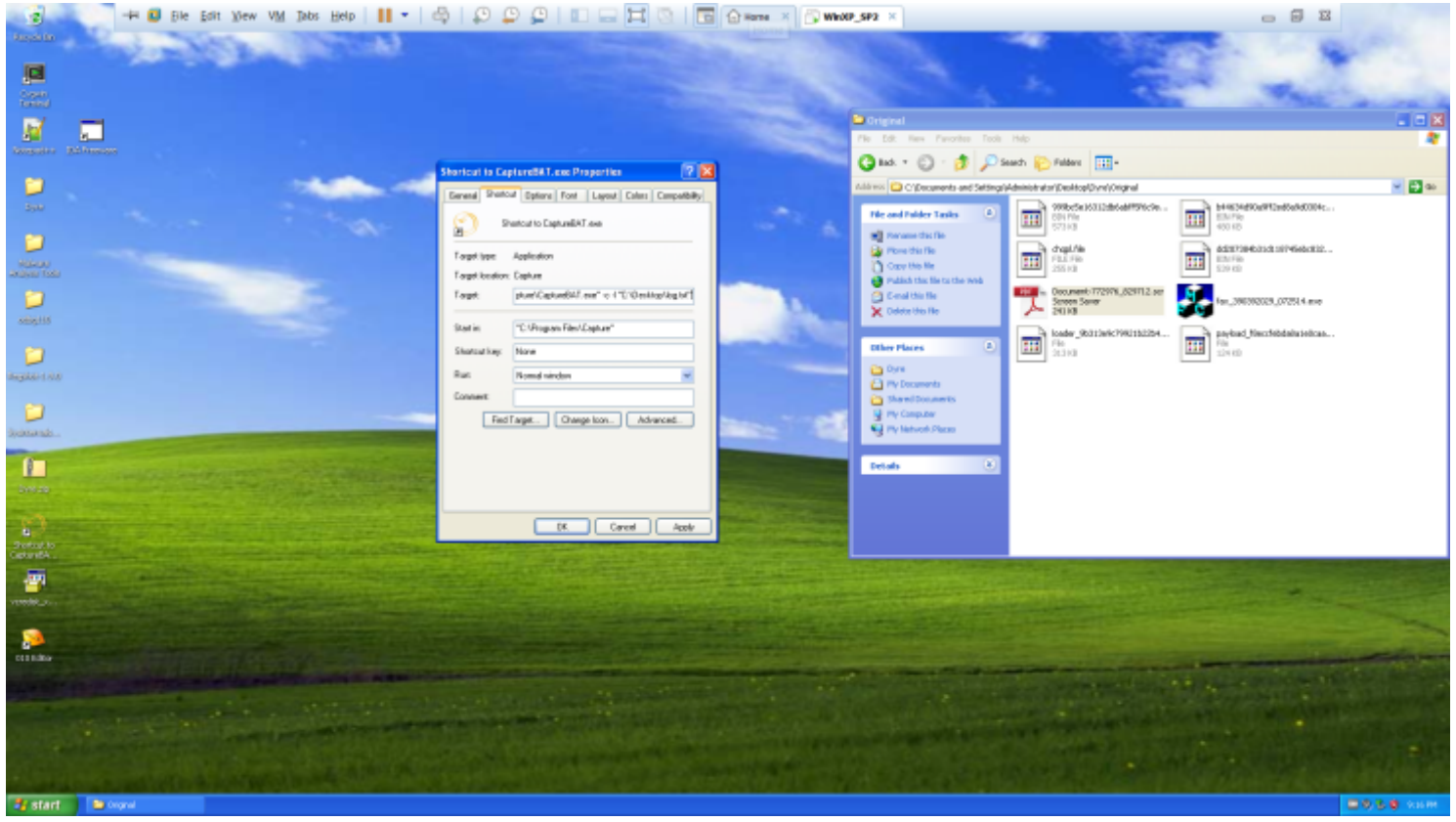



## What changes are made to the file system?

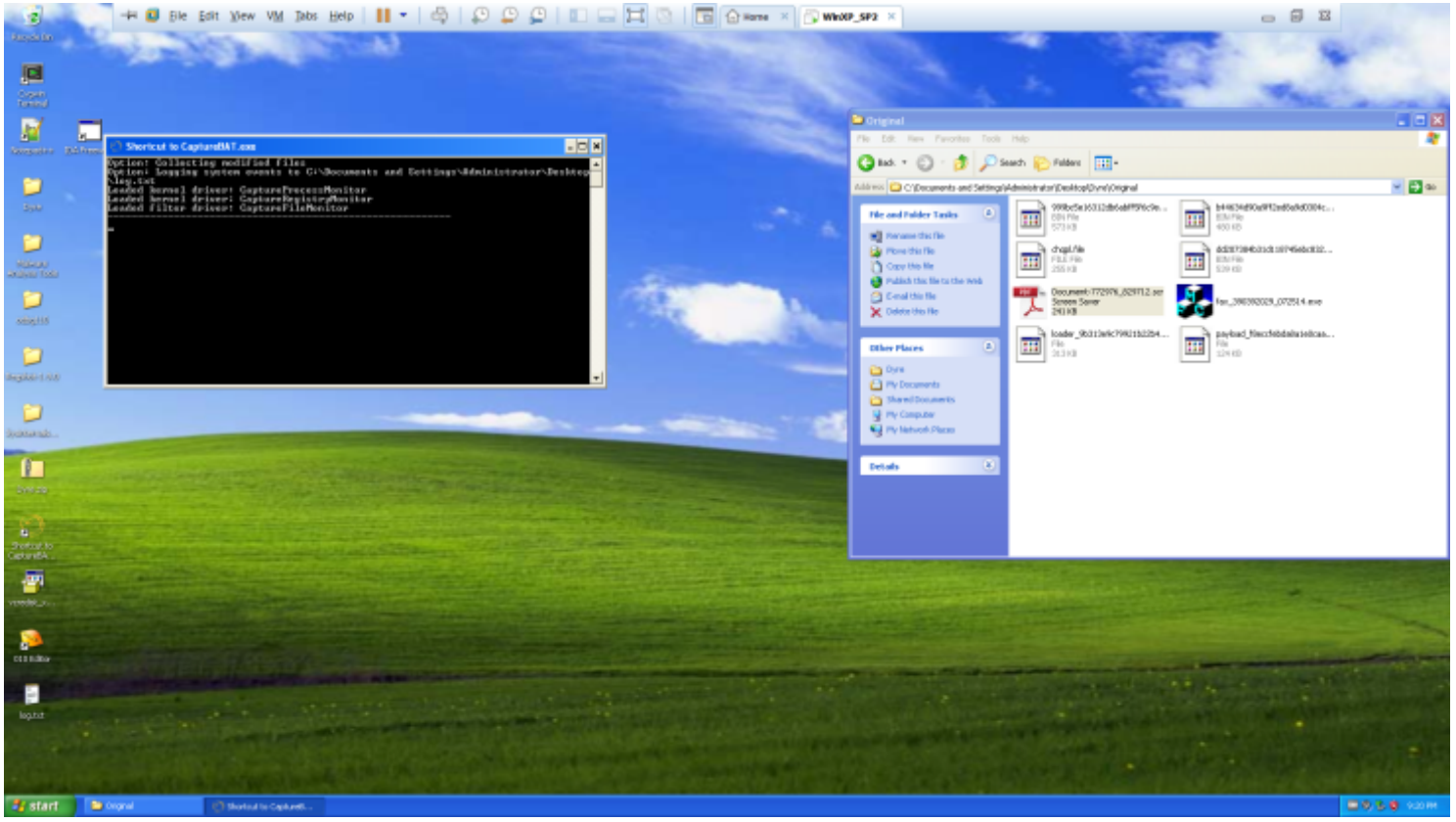
On execution, a malware sample may create, modify or delete files on the victim host and these changes can be tracked through *CaptureBAT*.

We will add two command line arguments to *CaptureBAT* for effective analysis. Add the following command line arguments to *CaptureBAT*:

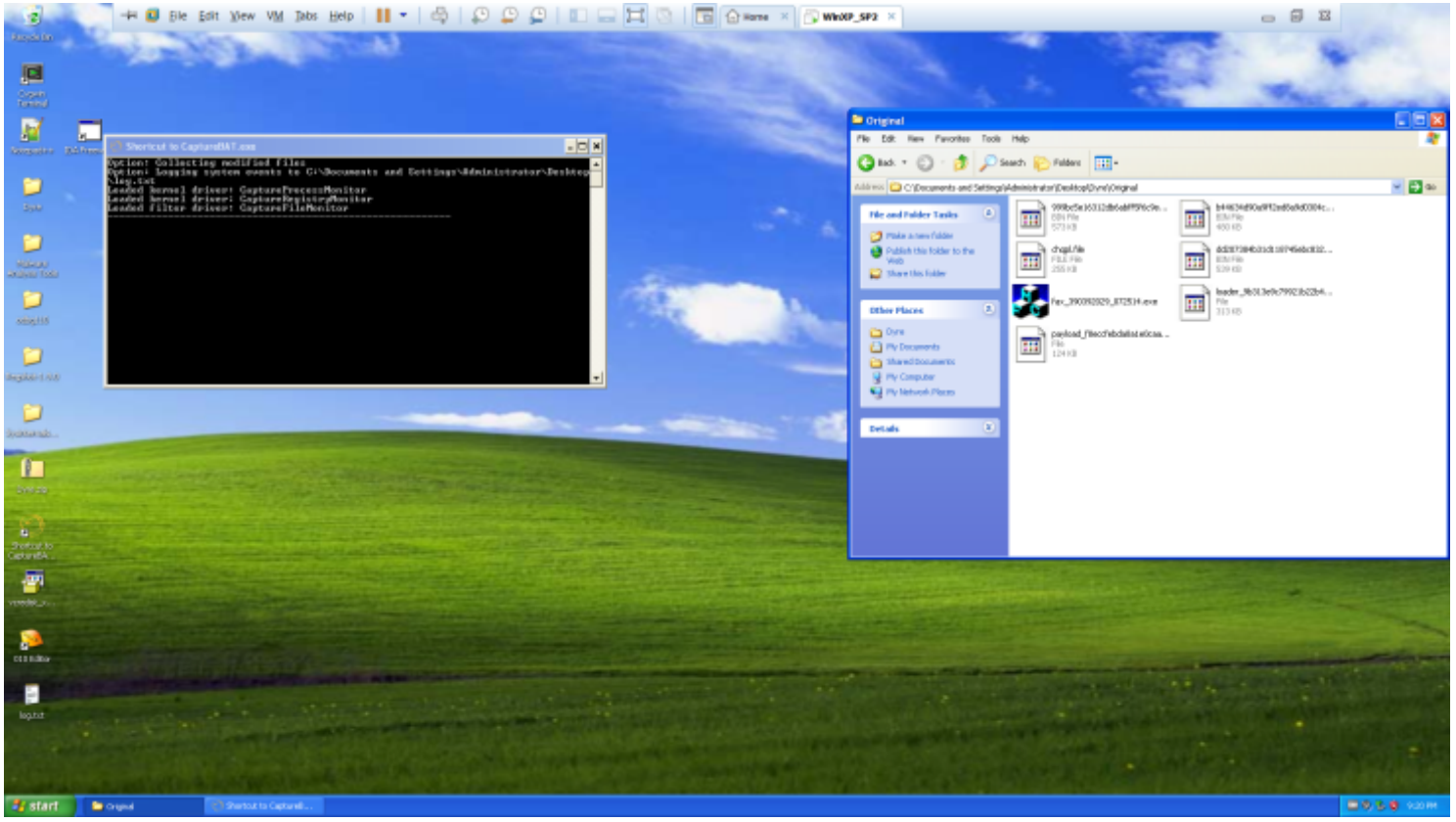
```
-c -l "/some/path/log.txt"
```



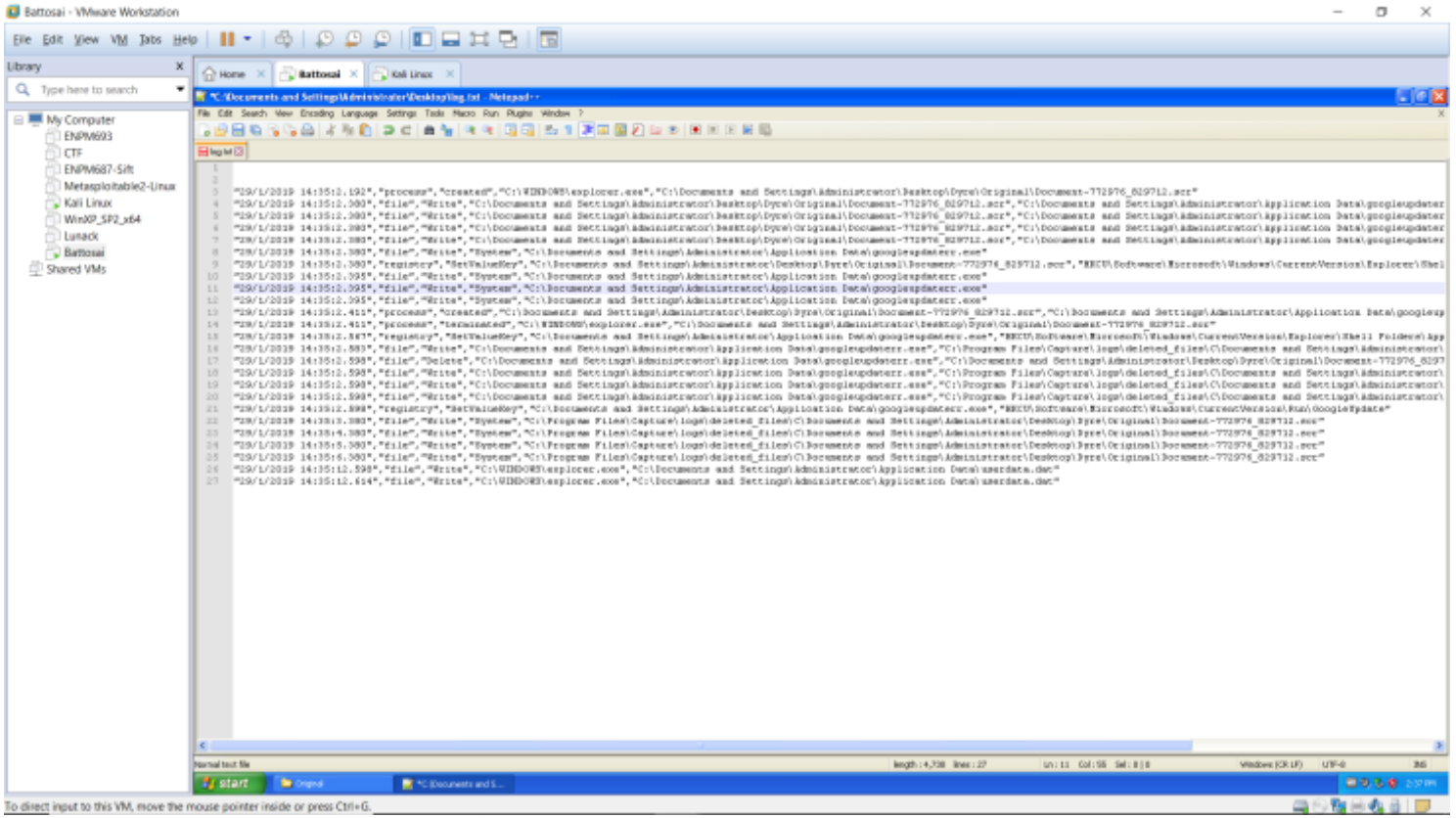
The *-c* argument stores any deleted files during the time when *captureBAT* was running and the *-l* flag stores the output of *captureBAT* in a log file at the specified path.



After launching the the malware sample, wait for about 10 seconds. Notice that the malware sample deletes itself.



Enter *Ctrl∧C* to exit *CaptureBAT*. Open *log.txt* in *Notepad++* and arrange the entries in time-ascending order.



Some entries suggest that the malware sample copies itself to a different location. We already know that *googleupdaterr.exe* is a malicious executable.

> "29/1/2019 14:35:2.380","file","Write","C:\Documents and Settings\Administrator\Desktop\Dyre\Original\Document-772976_829712.scr","C:\Documents and Settings\Administrator\Application Data\googleupdaterr.exe"

To verify this hypothesis, we'll compare the MD5 hashes of the two executables. The MD5 hash of *googleupdaterr.exe* is *c2d73485095efdbd7ab625e469affb11* which is the same as the MD5 hash of our original

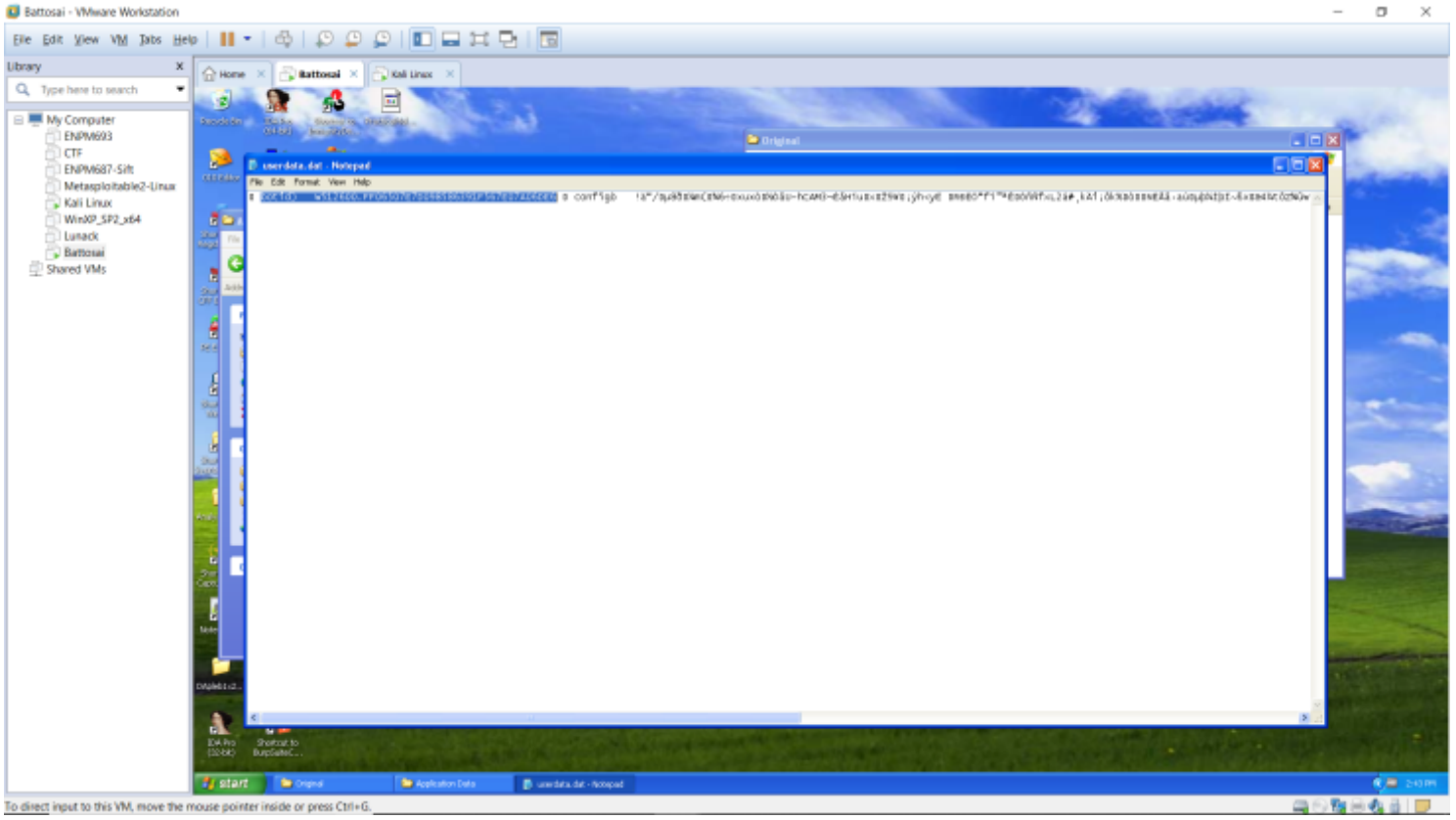
malware sample.

The malware sample also created a file, *userdata.dat.*

```
"29/1/2019 14:35:12.598","file","Write","C:\WINDOWS\explorer.exe","C:\Documents and Settings\Administrator\Application Data\userdata.dat"
```

On opening *userdata.dat*, we can see that it contains an interesting entry. Most of the file contents are incomprehensible and may be encrypted.

```
botid) W512600.FF06507E7D098586391F567E07A06DE6
```

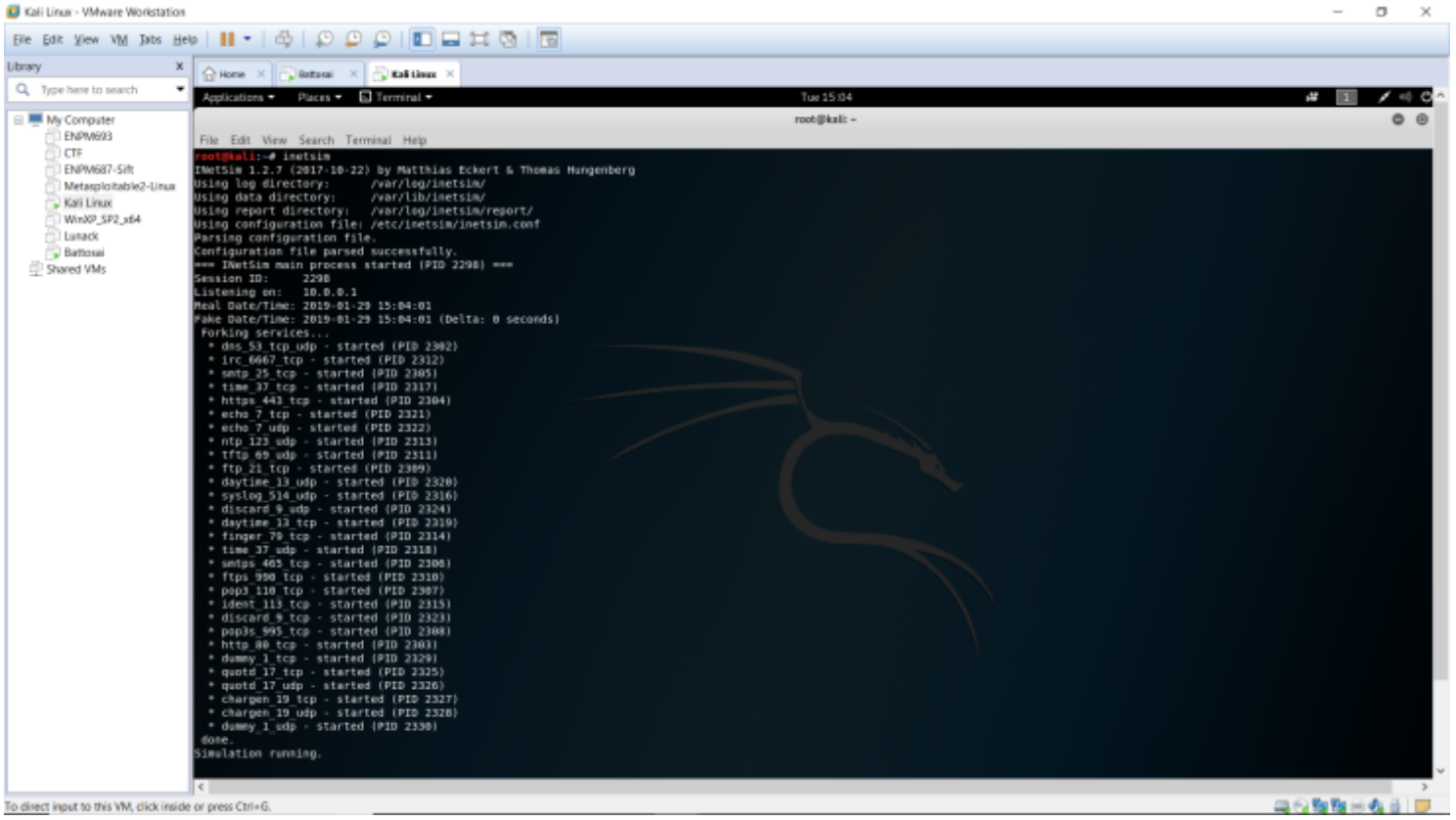



The string *W512600.FF06507E7D098586391F567E07A06DE6* may be a bot identifier which the malware author can use to track the location a specific malware sample (and thus, the host).

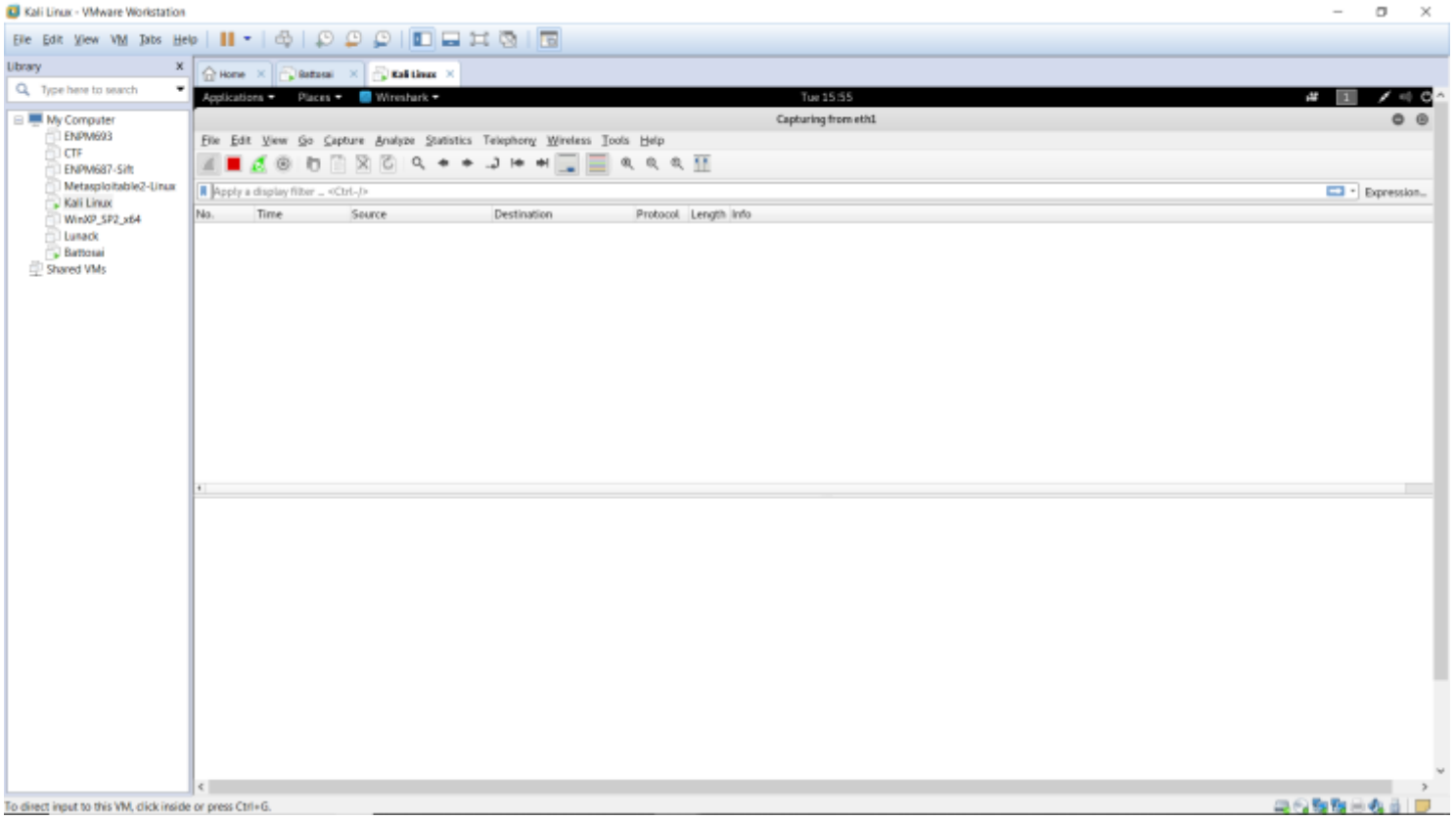
## Is there any network activity?

Most malwares have network activity on some level and it usually means that the malware is contacting command and control (C2) server to wait for further commands or exfiltrate data from the infected system.

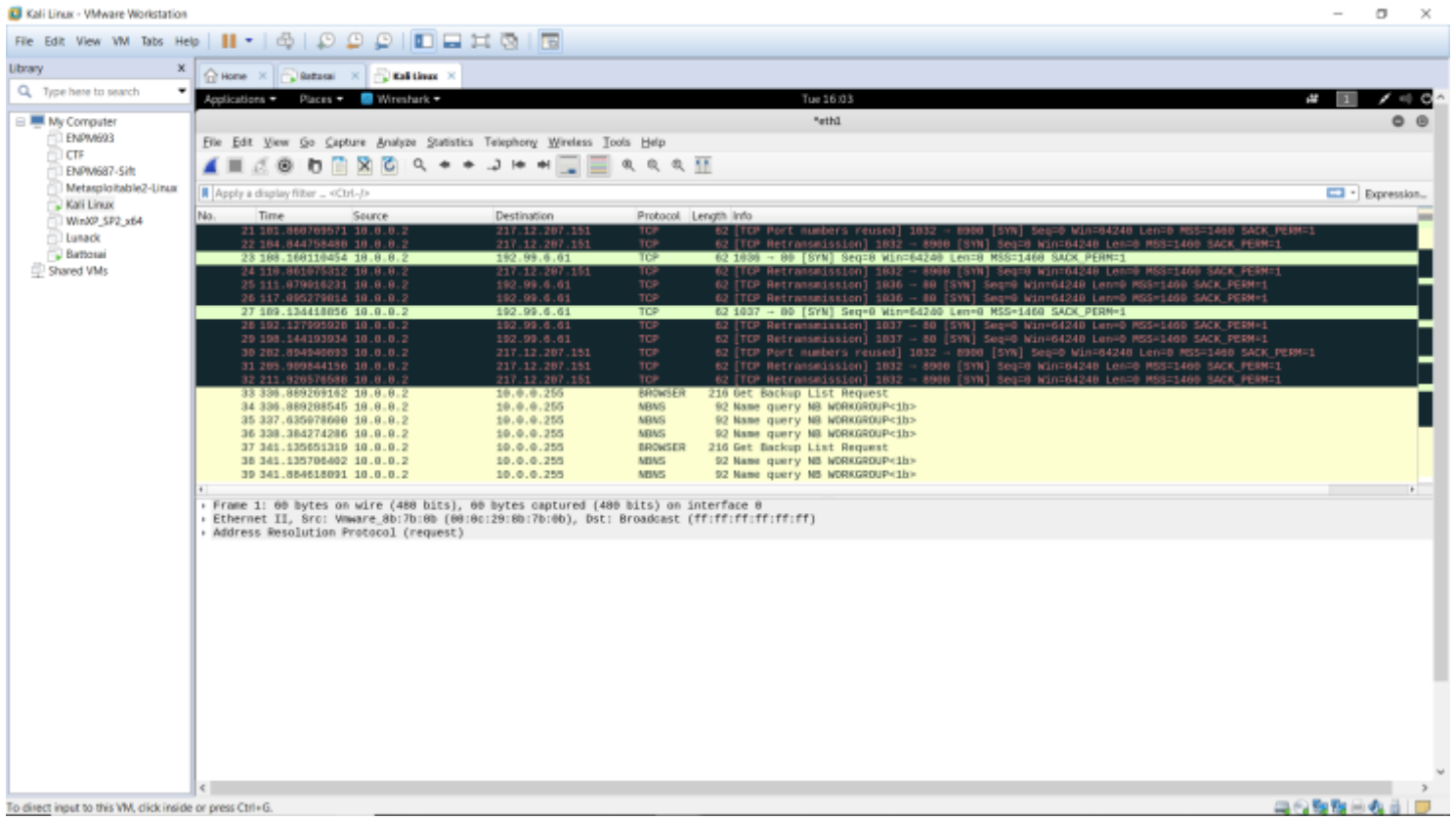
The Windows XP VM is connected to the Kali Linux VM and that's where the network packets are going to reach when the malware executes. The first step is to simulate network services on the Kali Linux VM so that the malware sample believes that it is actually connected to its C2 server. We can simulate network services using *inetsim* which is builtin on Kali Linux.




The second step is to start *Wireshark* on the Kali Linux VM. We'll use *Wireshark* to capture the packets going to and coming from the Windows XP VM. Packet captures can help us understand what the malware is looking for in the remote server that it is trying to connect with.

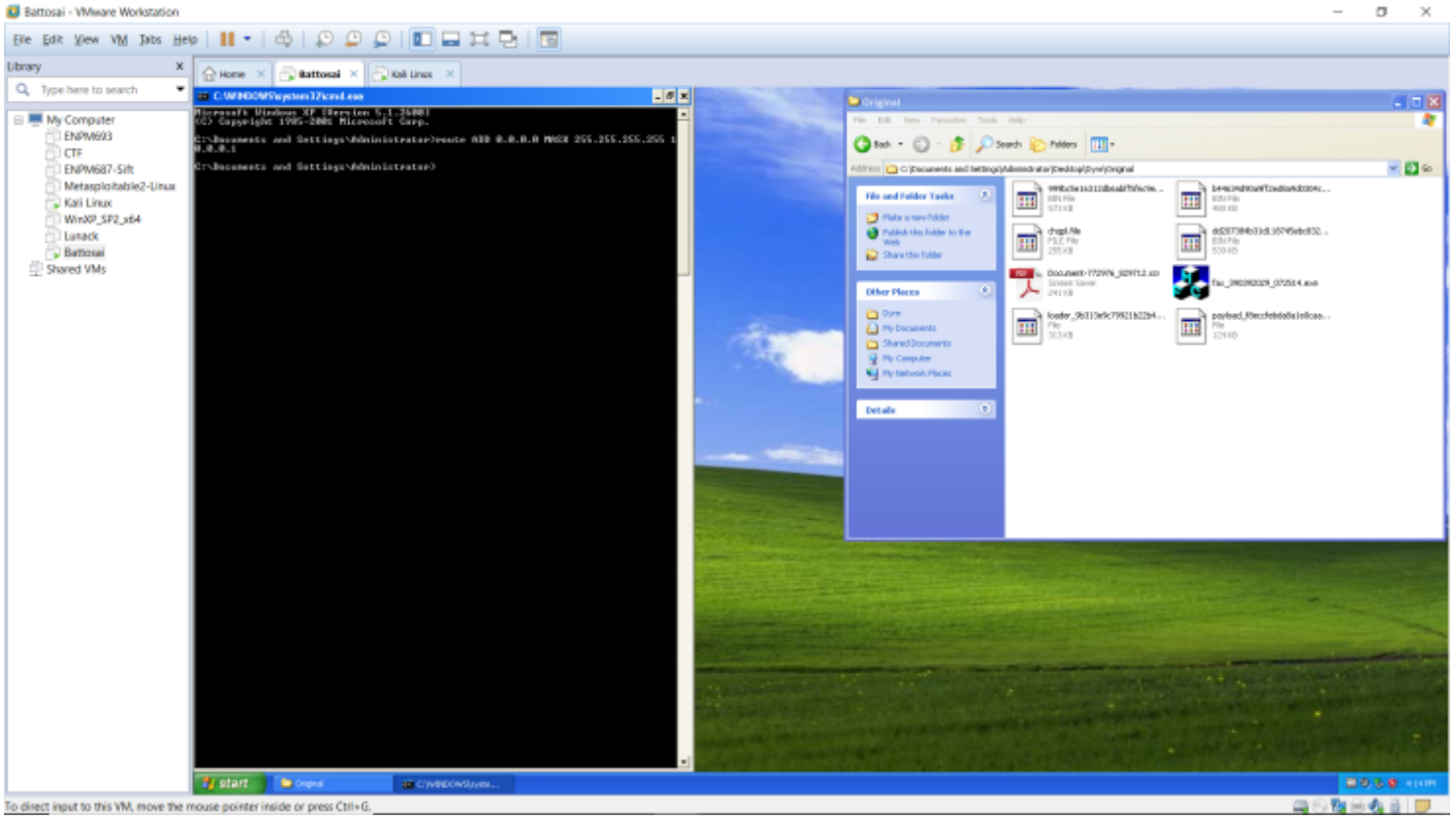



On executing the malware, notice that the packet capture window starts to populate.



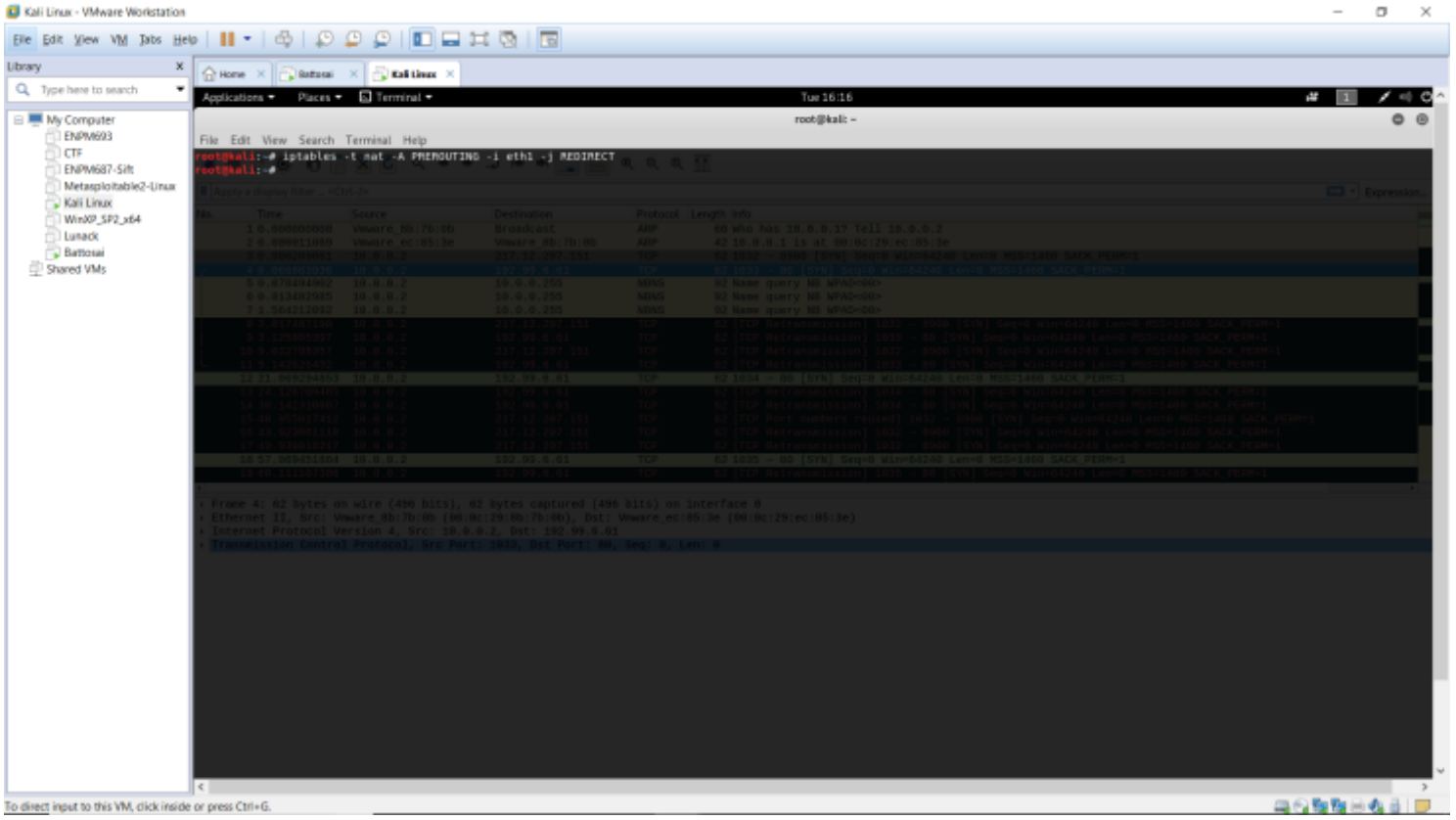

We can see that the malware sample is trying to connect to *217.12.207.151* and *192.99.6.61*, but obviously it is not able to. But there is a method which can be used to redirect all packets destined for an arbitrary IP address to our Kali Linux VM.

The first step is to add a route in the routing table of the Windows XP VM.



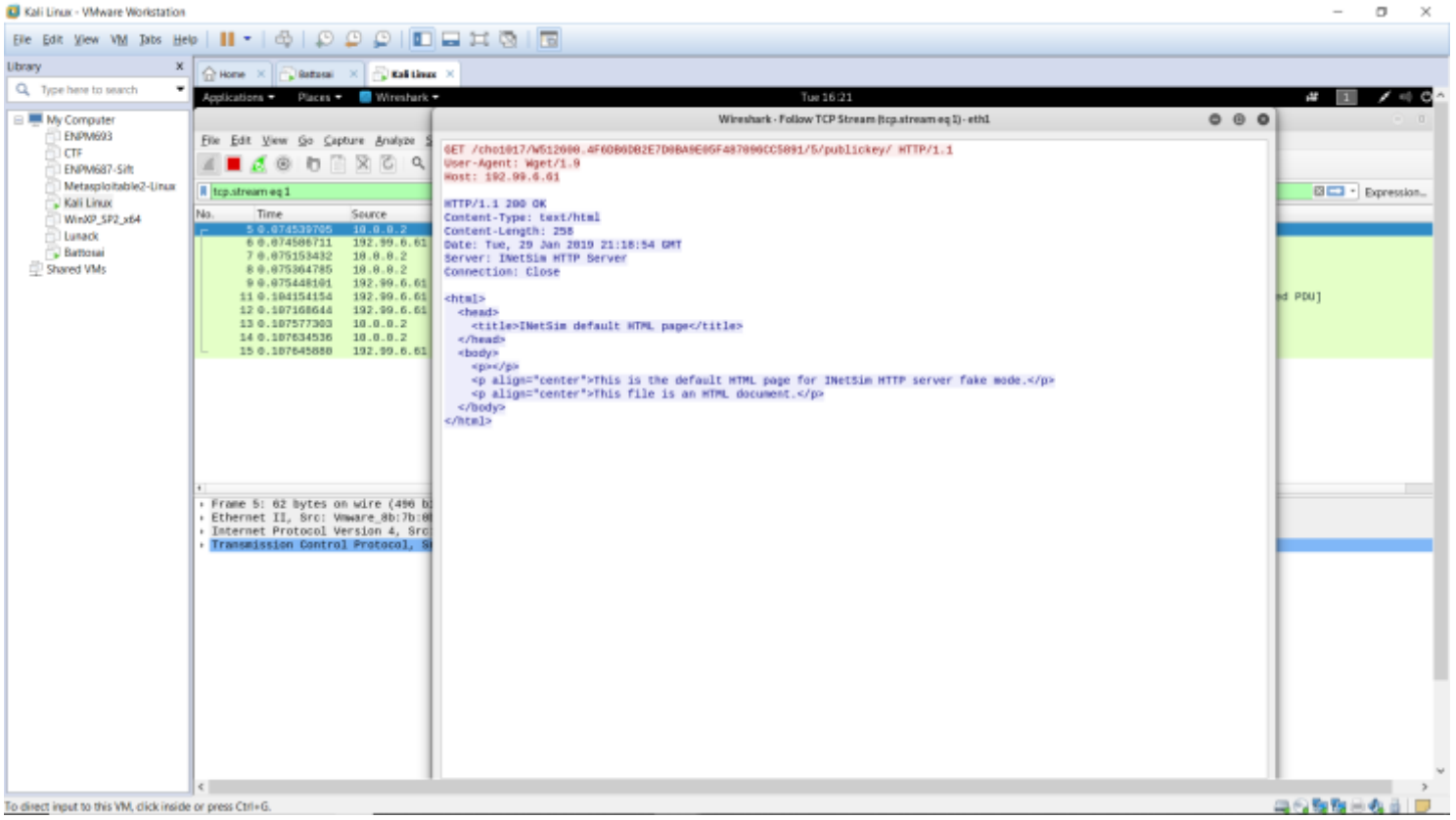

This route forces all IP packets outgoing from the Windows XP VM to be directed to our Kali Linux VM.

The next step is to force the Kali Linux VM to accept packets which are not destined for it.




Revert the Windows XP VM image to its snapshot and we will use the *inetsim/Wireshark* combination again to track the network packets.

Notice that now there are more greens in the packet capture window which means that our malware sample was successfully deceived. The malware sample was (maybe) trying to get a public key from the C2 server.

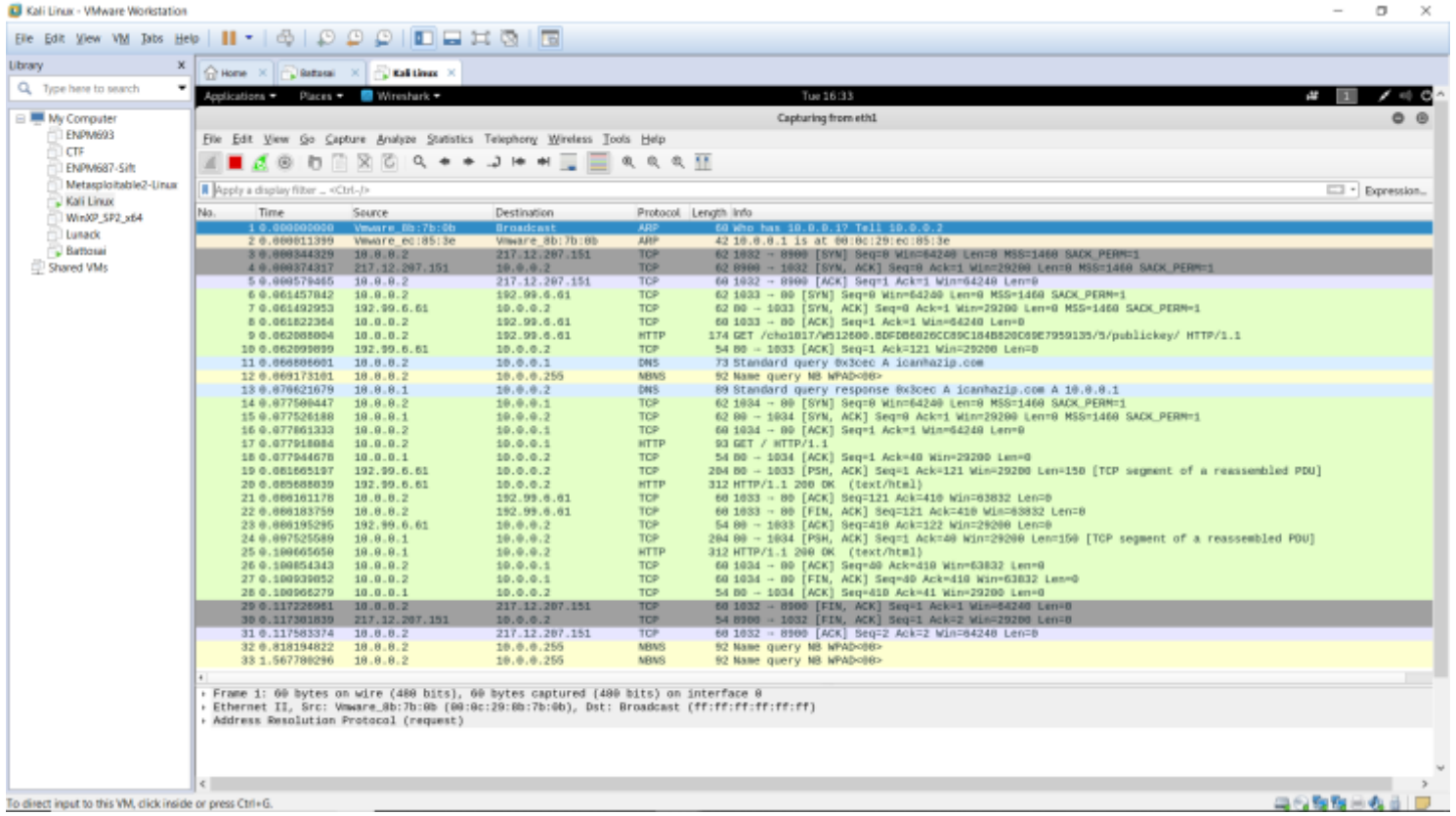



The malware sample wasn't able to connect with the other IP, *217.12.207.151* because the port *8900* is closed on our Kali Linux VM. To get around this, we'll launch *netcat* to listen on that port on the Kali Linux VM.

```
root@kali:~# nc -l -p 8900
```

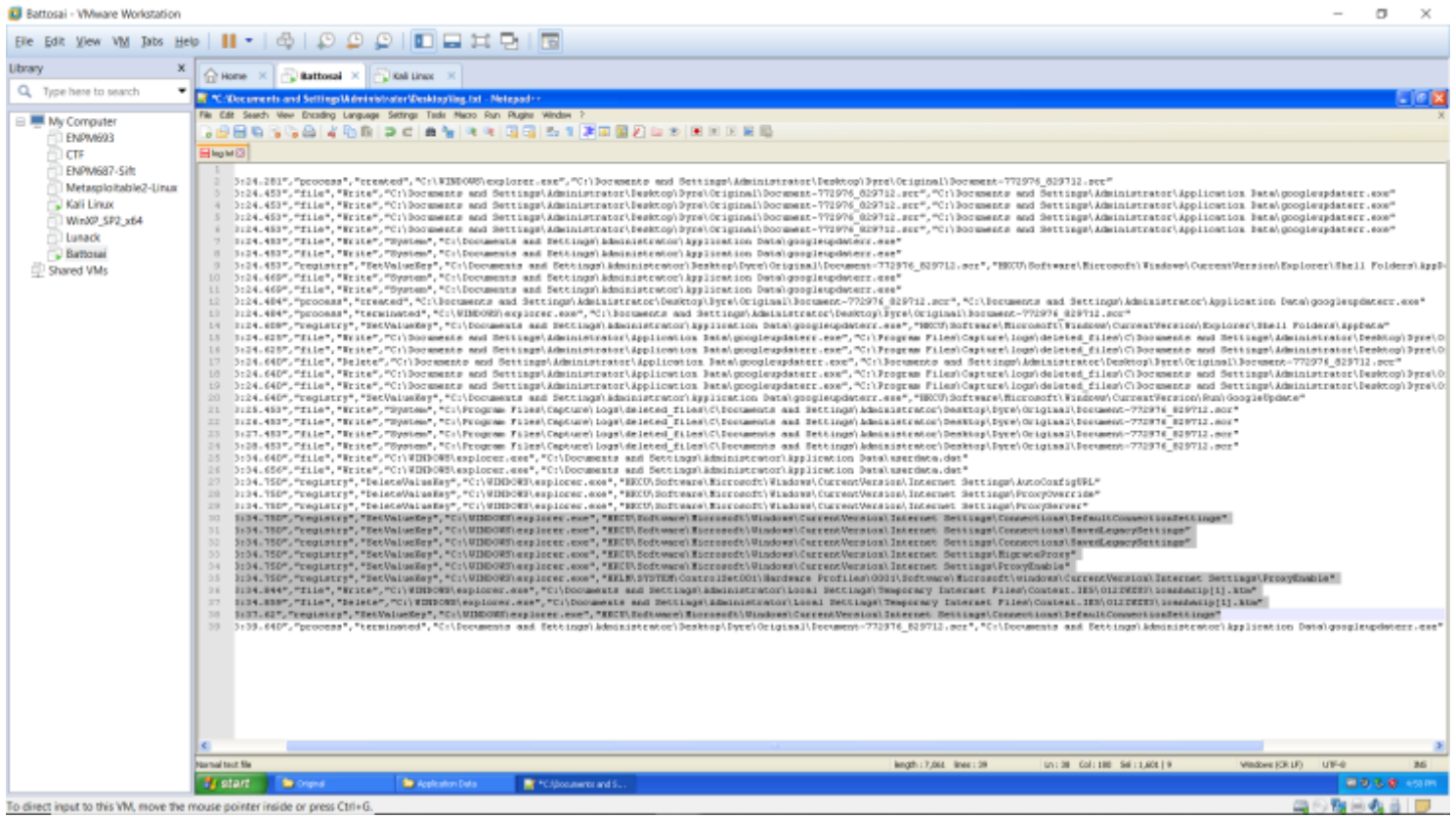
**Note:** When you find traffic on weird unpopular ports (like 8900), it might be an indicator of compromise.

Now, we have a very good picture of the packet captures.




From the packets, it seems that the malware sample establishes a connection with *217.12.207.151*, asks for a resource from *192.99.6.61* and then terminates its connection with both the remote servers. It is possible that the malware sample terminates the connection because it DID NOT actually get that resource; it only got a standard *inetsim* response.

Also, notice that there are more entries in the *CaptureBAT* logs. The new entries are the ones which are Internet Explorer (IE) related registry changes.



**Note:** At this point we have the following findings:

1. The malware achieves persistence through a registry edit and with the installation of a startup service, *googleupdaterr.exe*
2. It interacts with two remote servers, and terminates soon after.

3. It requests for a resource, the path of which is dependent on the *botid* present in the *userdata.dat* file.
4. There were a few IE related registry key changes which suggests that the malware sample may have some interaction with IE on the host system.

So far, we've used techniques for both static and dynamic analysis. However, they've provided us with only a very basic overview of what the malware is capable of. We don't know if the malware employed anti-VM techniques and thus, did not utilize itself completely in our test lab. We don't know if the malware would behave differently in a different environment.

To answer such questions, we need to look at the malware's assembly code. Writing in detail about the IDA Pro analysis of our malware sample is probably going to take a lot of time and being a graduate student, I don't have that much time to spare.

## Done!

Malware analysis is a procedural task, but frequently requires the analyst to ask out-of-the-box questions. In this post, we looked at basic static and dynamic analysis techniques and inferred a few observations about the malware sample, *Infostealer.Dyre*.

Thank you for reading! If you have any questions, leave them in the comments section below and I'll get back to you as soon as I can!

SHARE THIS:

Twitter Facebook

Like

Be the first to like this.

RELATED


Malware Analysis - NanoCore + MITRE ATT&CK Mapping
In "malware analysis"


Malware Analysis - WannaCry
In "malware analysis"


Malware Analysis - Gozi/Ursnif Downloader
In "malware analysis"

## Published by Nikhil

A budding front-line cyber soldier!

View all posts by Nikhil

January 30, 2019

malware analysis

analysis, malware, reverseengineering, windows

Return-Oriented Programming – ROP Chaining

Malware Analysis – Illusion Bot

## Leave a Reply

Enter your comment here...

This site uses Akismet to reduce spam. Learn how your comment data is processed.

### FOLLOW SECUREHUB VIA EMAIL

Follow SecureHub to receive notifications by email. One email per week max.

Enter your email address

Follow SecureHub

Search ...

POSTS BY CATEGORY

- » android (1)
- » cryptography (2)
- » CTF solutions (3)
- » exploits (11)
- » forensics (2)
- » fuzzing (1)
- » git (1)
- » incident response (1)
- » linux (1)
- » malware analysis (8)
- » pentesting (3)
- » playtime (1)
- » programming (4)
- » reverse engineering (3)
- » social engineering (2)
- » splunk (1)
- » stories (2)
- » threat intel (5)
- » Uncategorized (1)

CATEGORIES

Select Category

ARCHIVES

Select Month

CATEGORIES

Select Category

Follow SecureHub



UP ↑