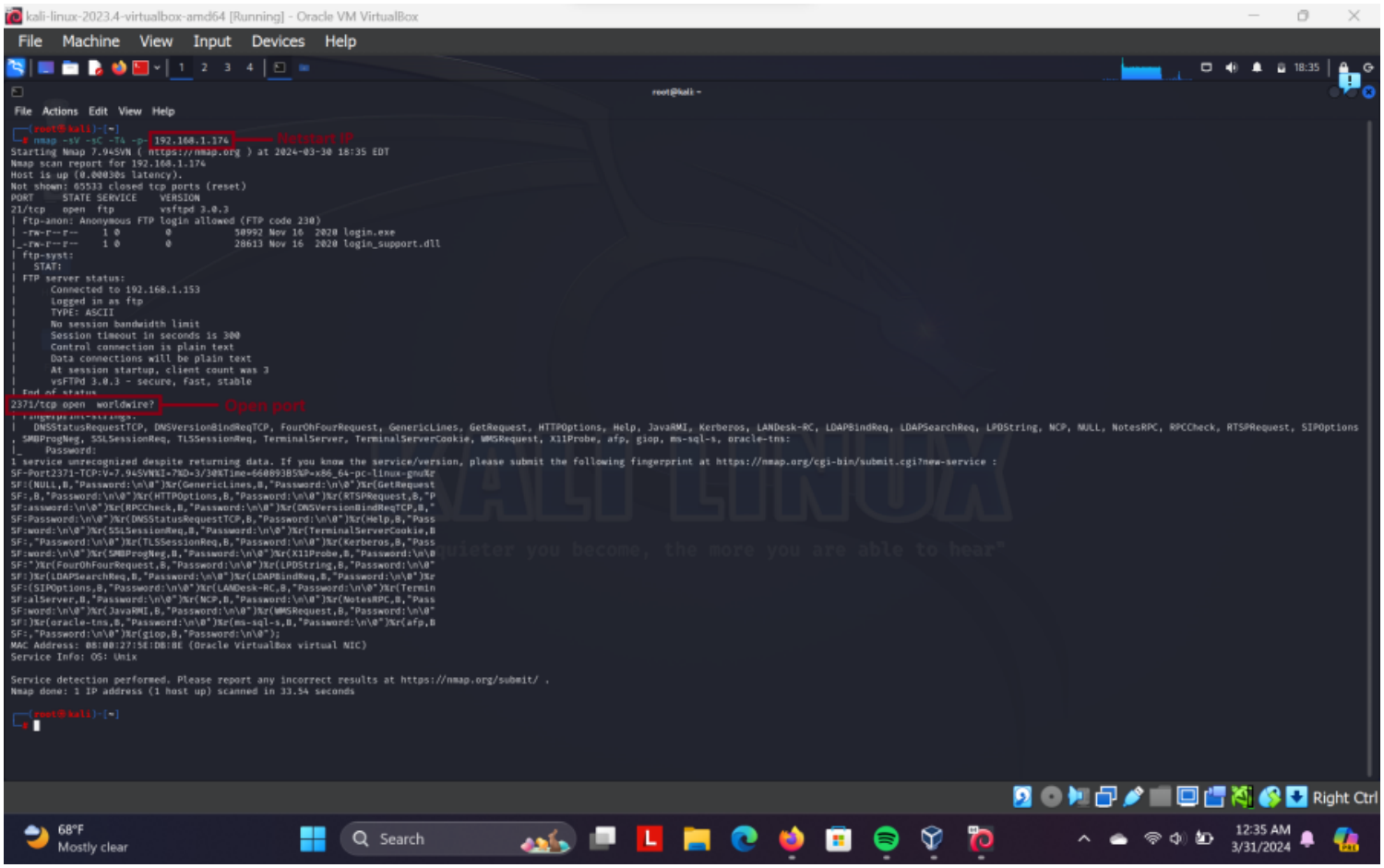
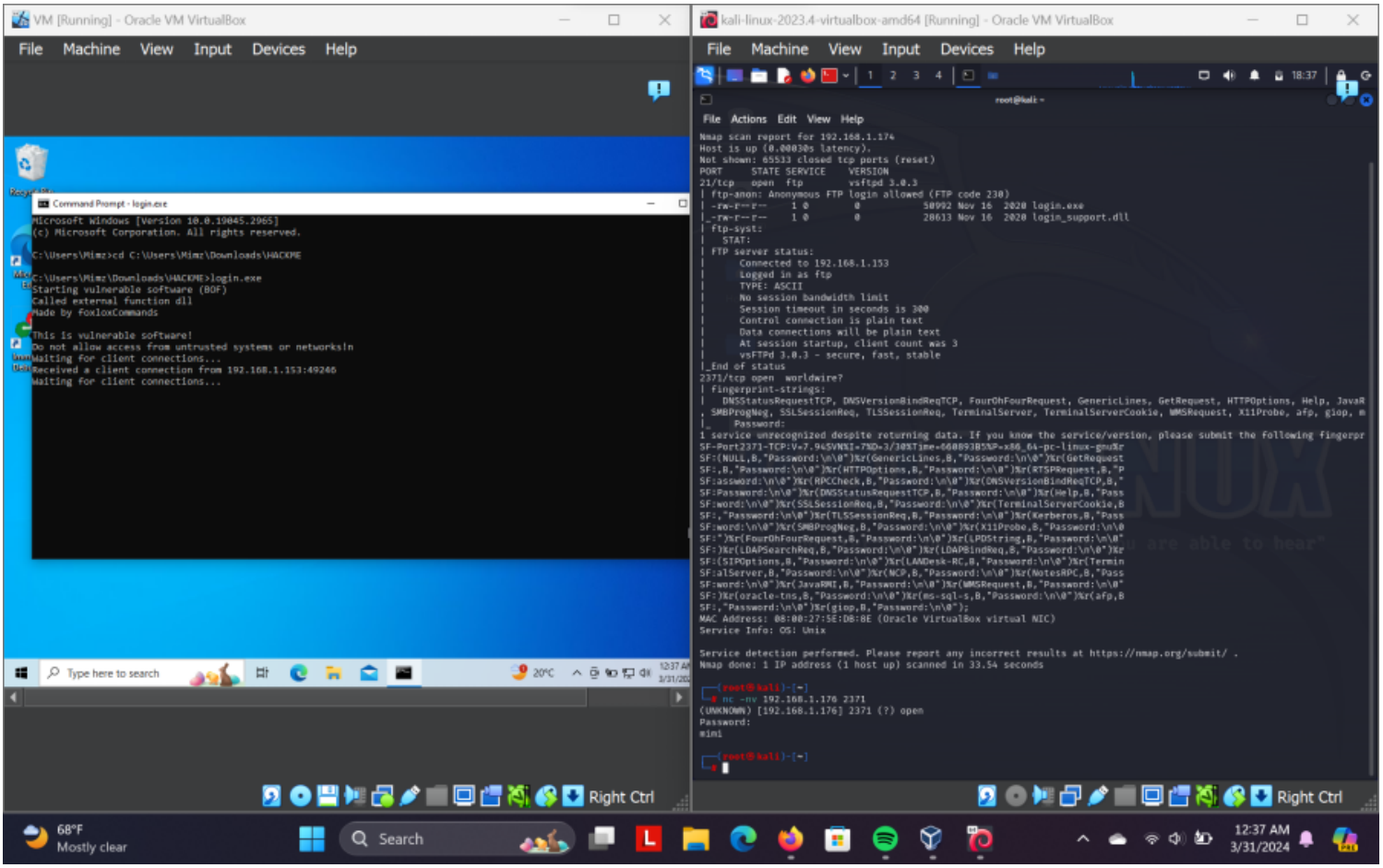
Spiking

Mariam Khaled
7004693 T-08

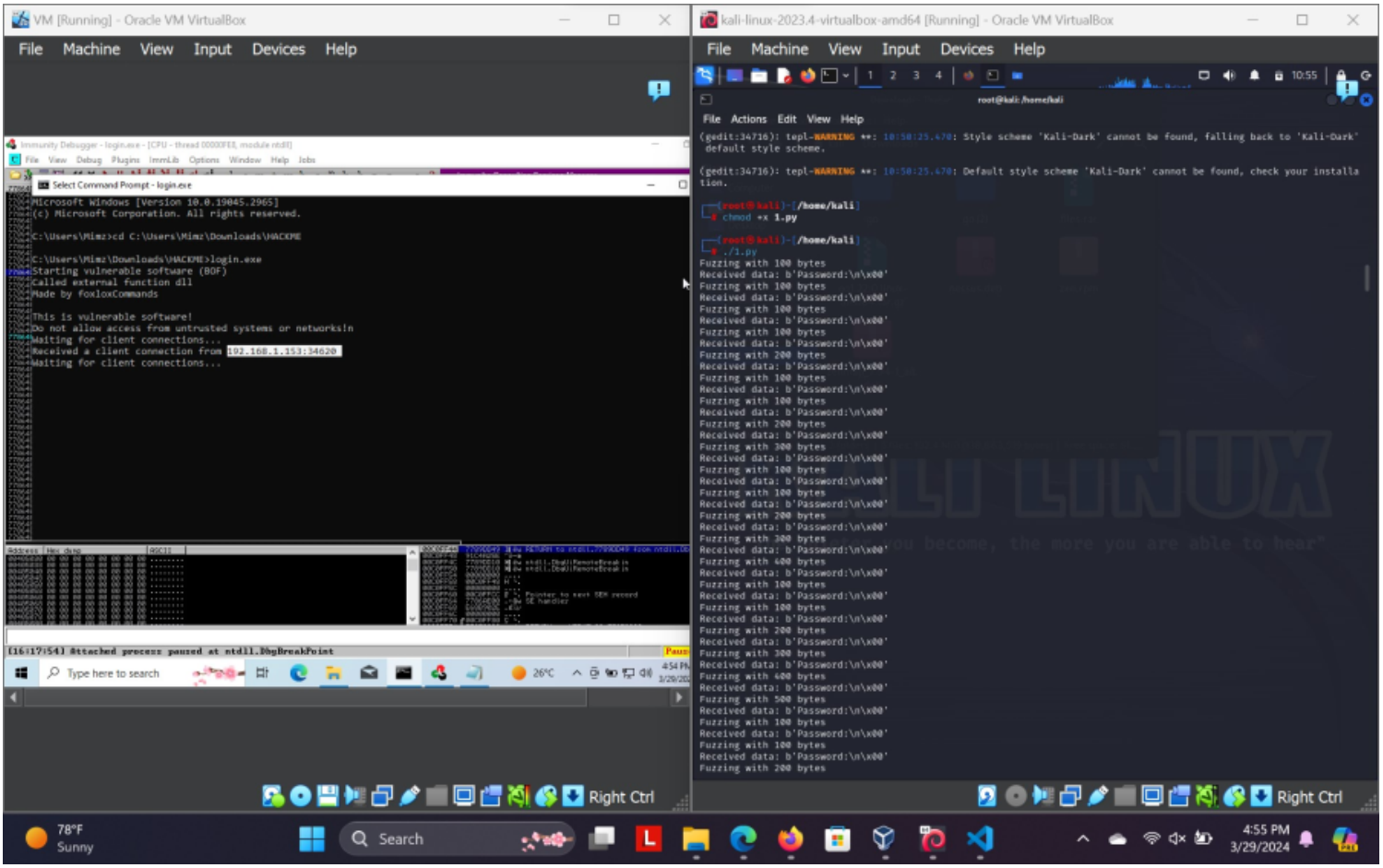


"ⓘNmap -sV -T4 -p- 192.168.1.174" is a command used for network scanning. It aggressively scans all ports on the target IP address (192.168.1.174), aiming to identify the services and their versions running on those ports.

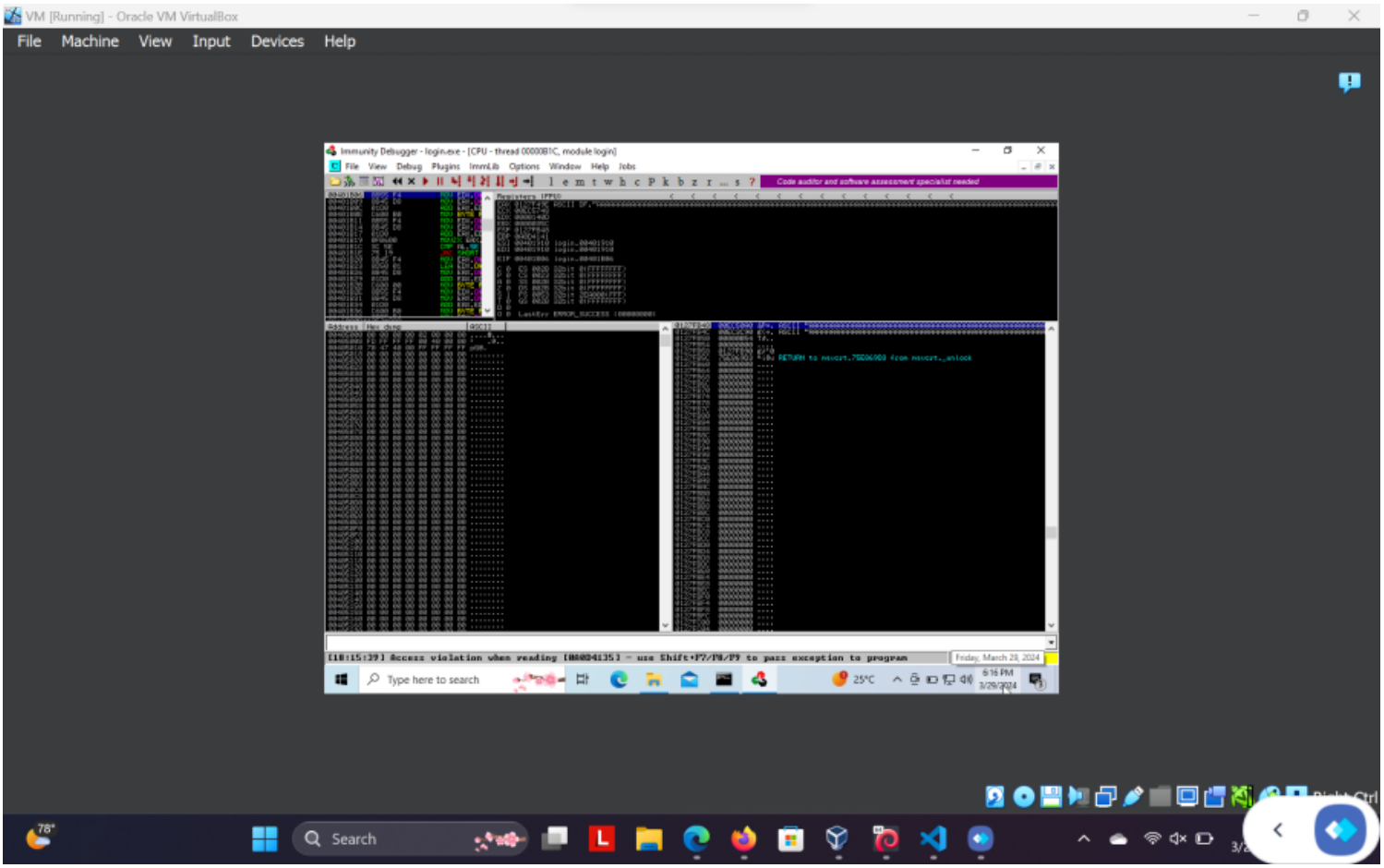
After running the scan, I discovered two open ports on the target IP (192.168.1.174). Port 21/tcp requires a password for access, indicating it might be an FTP service. Additionally, port 2371/tcp was identified as a worldwide port



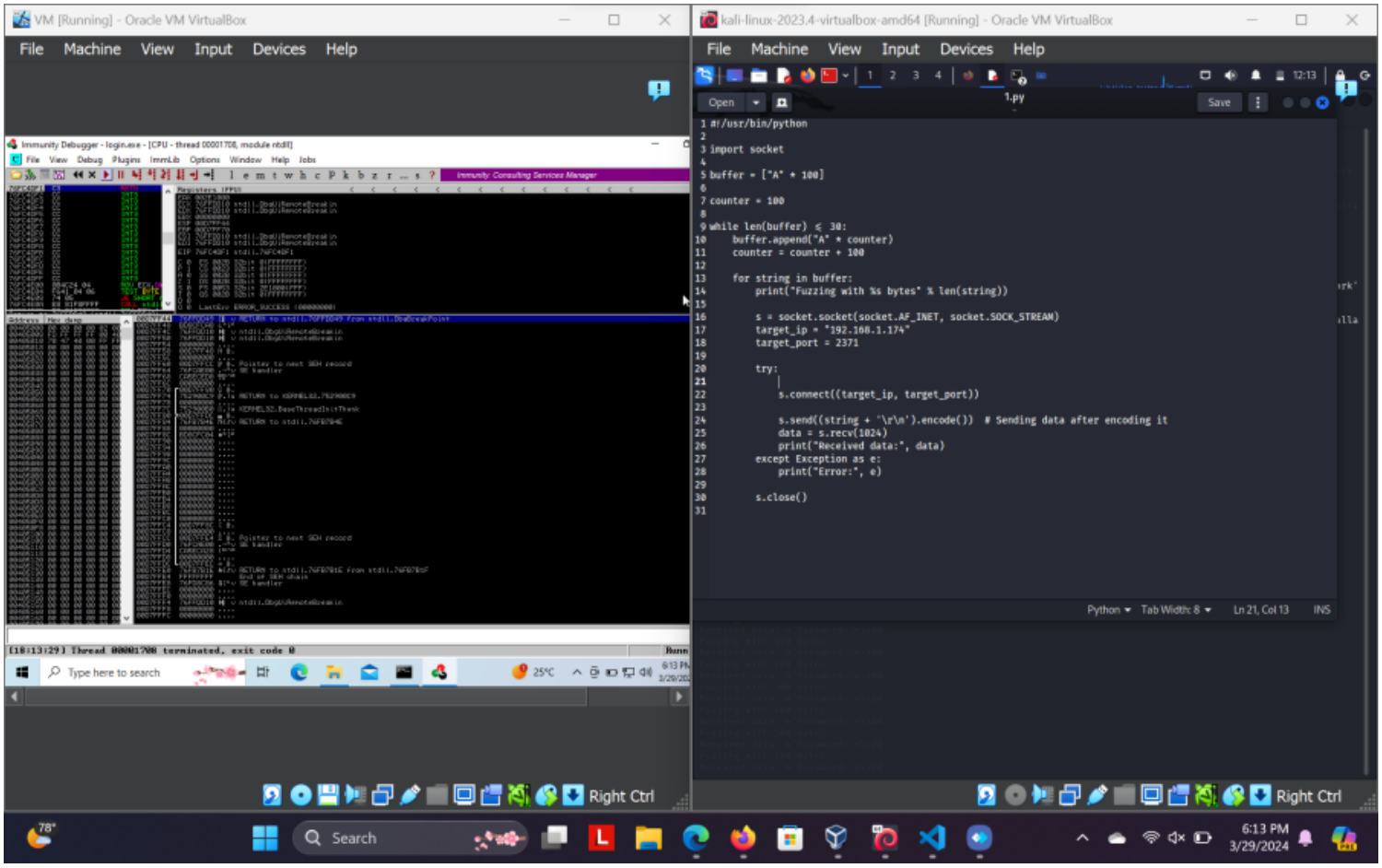
ⓘ I proceeded to download two files, 'login.exe' and 'login_support.dll', onto a virtual machine (VM) from the network.
After executing 'login.exe' as an administrator, I initiated a connection from my Kali Linux machine using the command 'nc -nv 192.168.1.176 2371' to establish communication with the 'login.exe' process running on the target system.



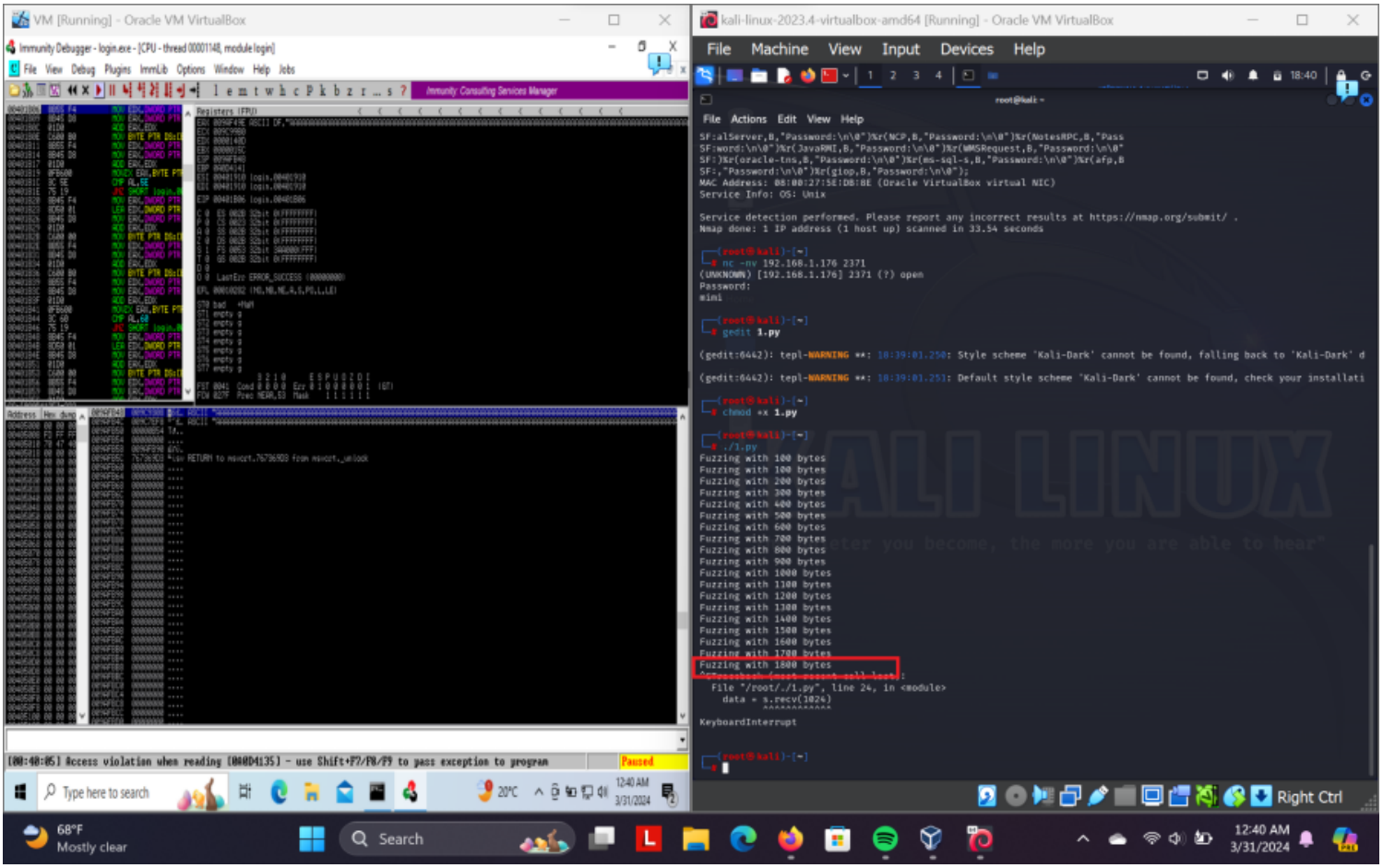
I continuously sent A's script, to the target system in order to test its susceptibility to buffer overflow vulnerabilities.



Fuzzing

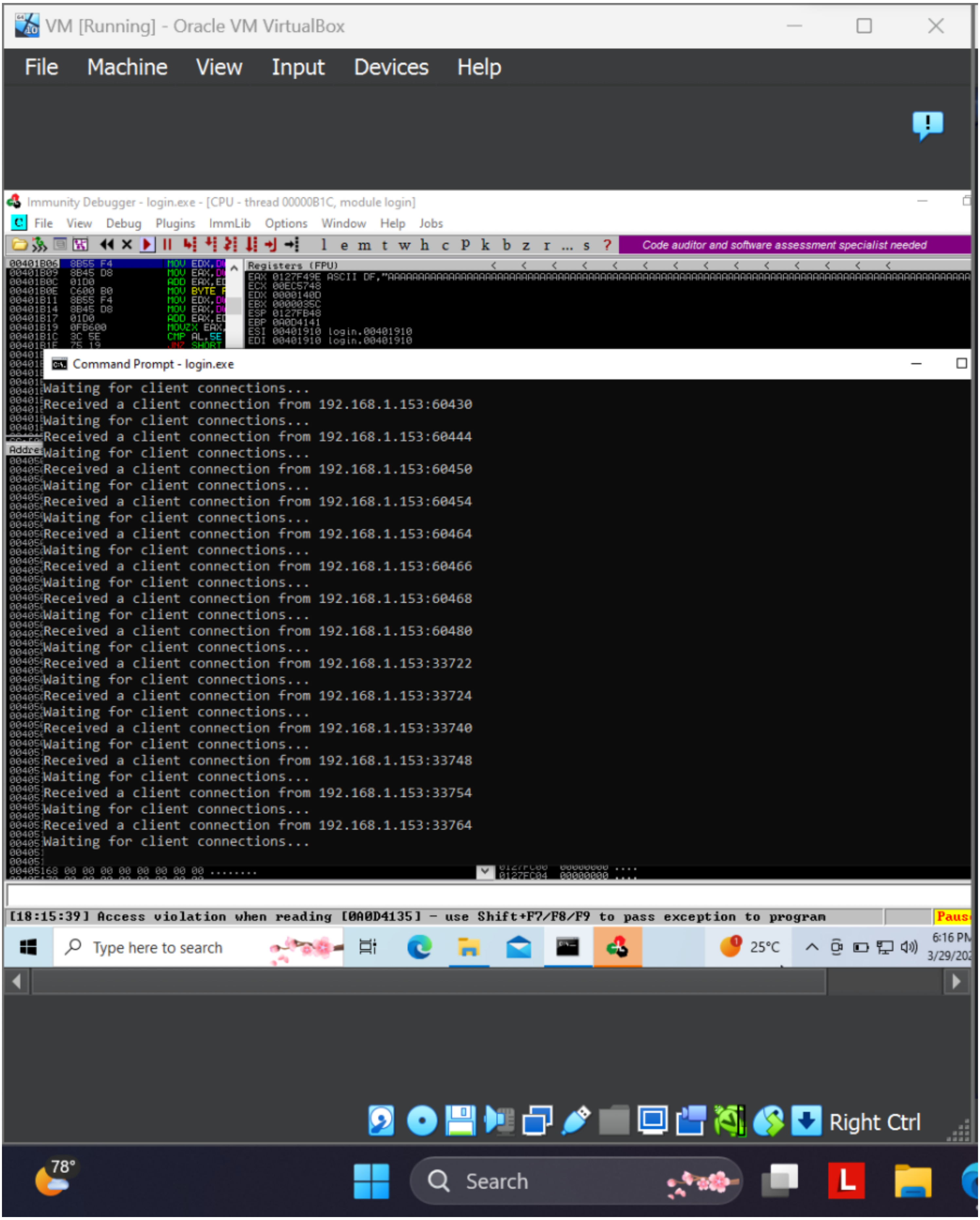


I attempted to fuzz the program to determine whether it crashes or remains stable. The Python script generates strings filled with 'A's of increasing length, up to 3000 characters.

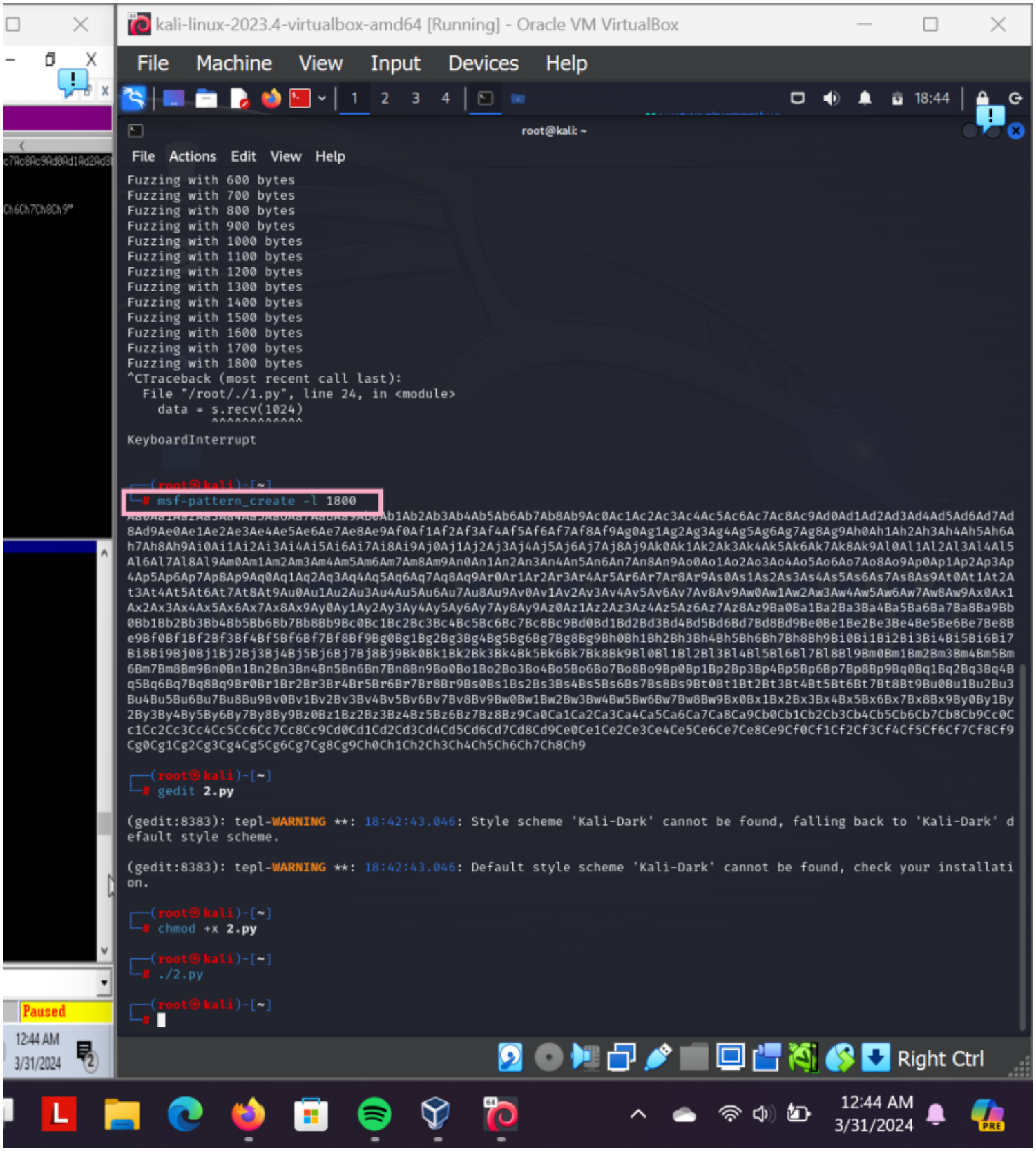


The program crashed when fuzzing with a payload of 1800 bytes.

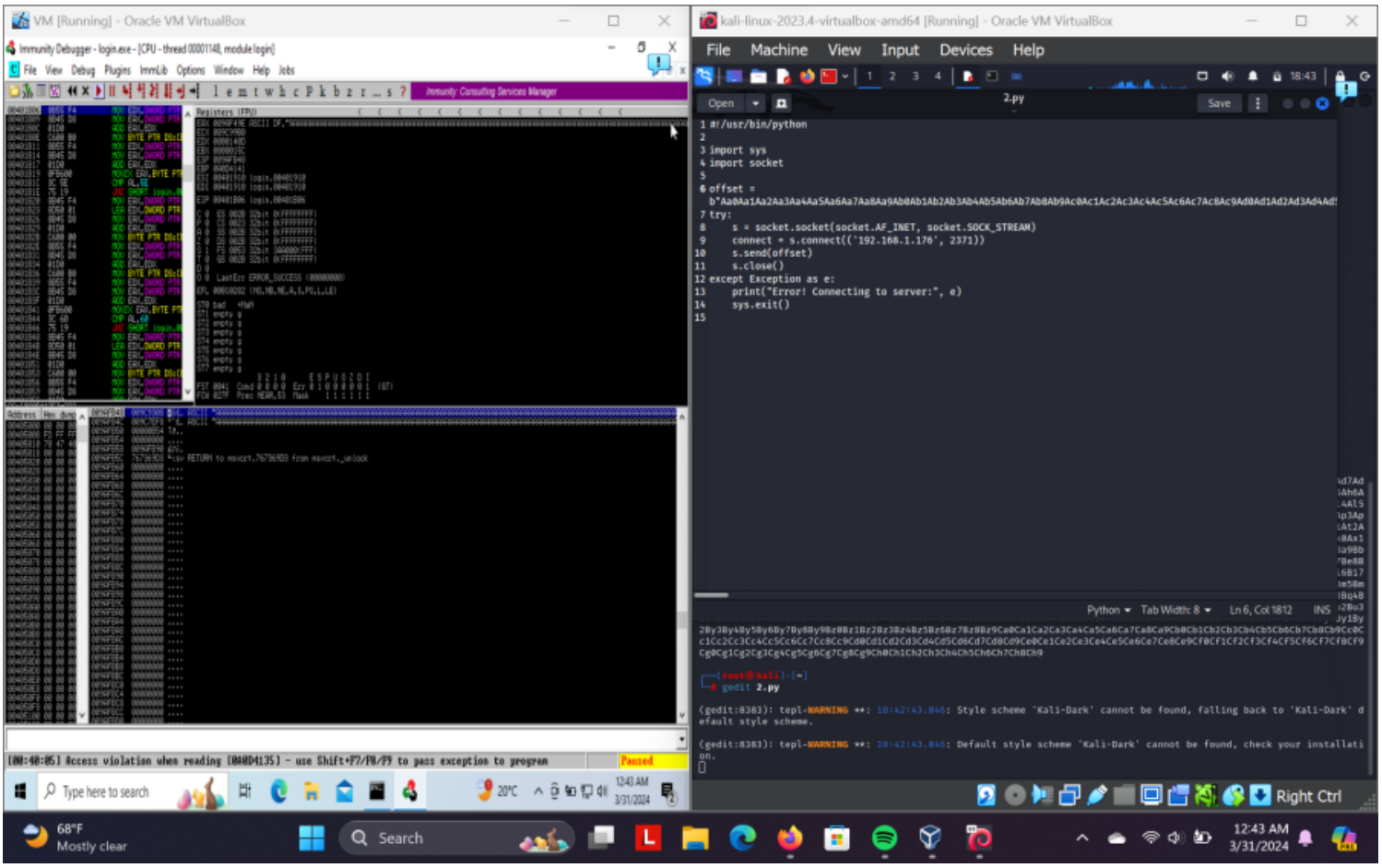
ⓘ I indicates that the program being tested failed to handle the input data properly. Specifically, when a payload of 1800 bytes was sent to the program during the fuzzing process


VM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Immunity Debugger - login.exe - [CPU - thread 00000B1C, module login]
File View Debug Plugins ImmLib Options Window Help Jobs
Code auditor and software assessment specialist needed
Registers (FPU)
Command Prompt - login.exe
Waiting for client connections...
Received a client connection from 192.168.1.153:60430
Waiting for client connections...
Received a client connection from 192.168.1.153:60444
Waiting for client connections...
Received a client connection from 192.168.1.153:60450
Waiting for client connections...
Received a client connection from 192.168.1.153:60454
Waiting for client connections...
Received a client connection from 192.168.1.153:60464
Waiting for client connections...
Received a client connection from 192.168.1.153:60466
Waiting for client connections...
Received a client connection from 192.168.1.153:60468
Waiting for client connections...
Received a client connection from 192.168.1.153:60480
Waiting for client connections...
Received a client connection from 192.168.1.153:33722
Waiting for client connections...
Received a client connection from 192.168.1.153:33724
Waiting for client connections...
Received a client connection from 192.168.1.153:33740
Waiting for client connections...
Received a client connection from 192.168.1.153:33748
Waiting for client connections...
Received a client connection from 192.168.1.153:33754
Waiting for client connections...
Received a client connection from 192.168.1.153:33764
Waiting for client connections...
[18:15:39] Access violation when reading [0A0D4135] - use Shift+F7/F8/F9 to pass exception to program
Type here to search
25°C
6:16 PM
Right Ctrl
78°
Search

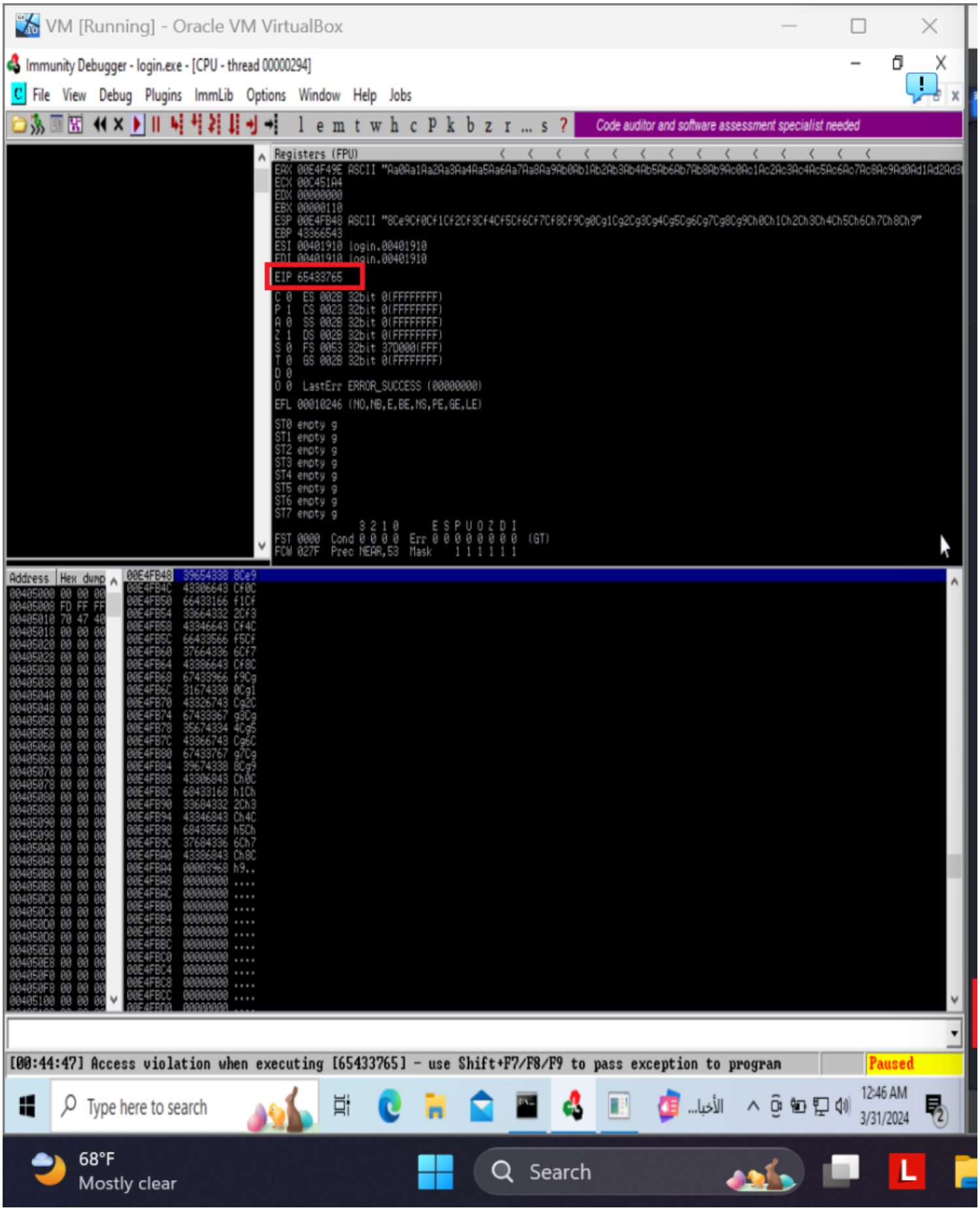
Finging the Offset



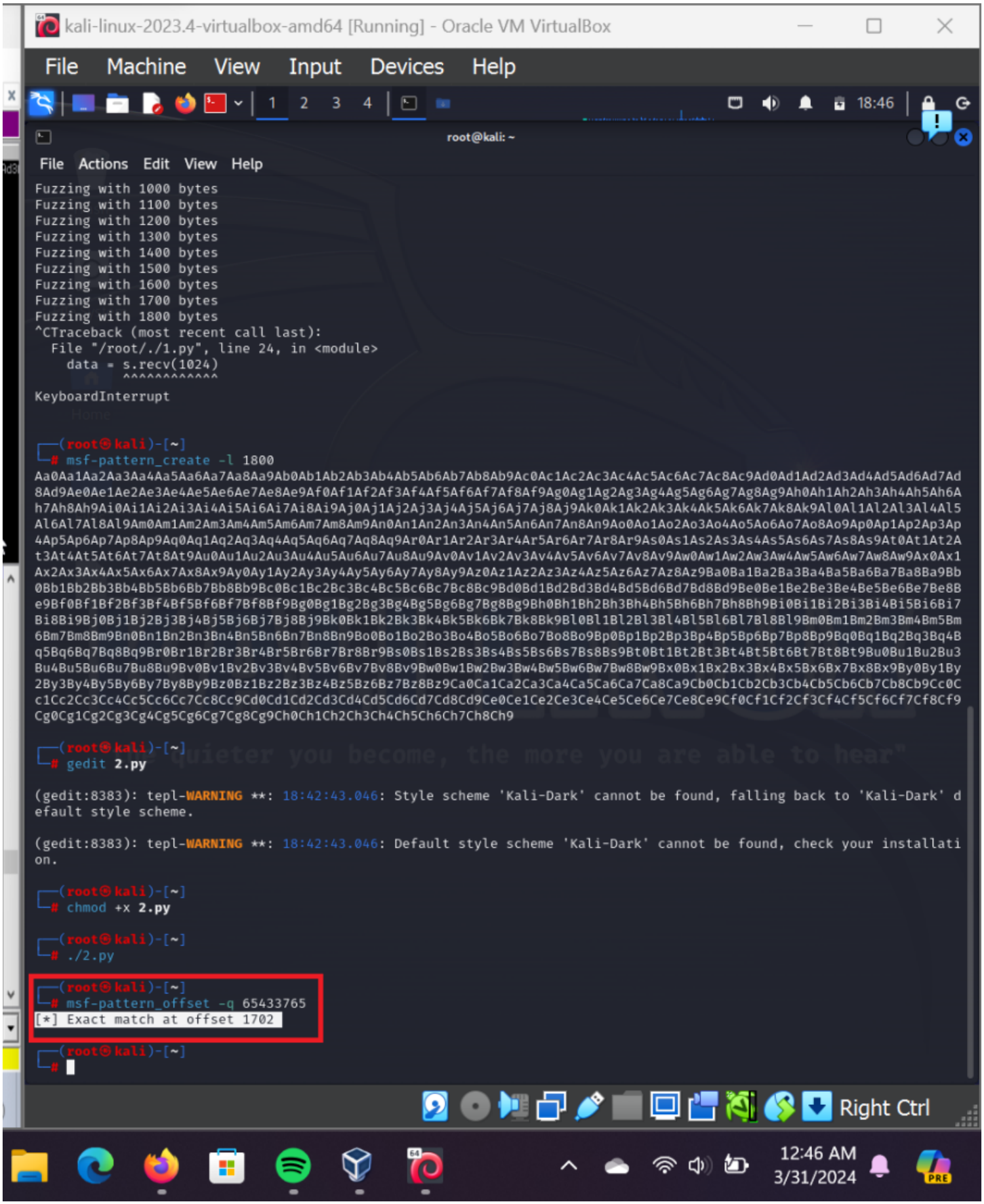
After discovering that the system crashed when a payload of 1800 bytes was used, I employed 'msf-pattern_create -l 1800' to generate a pattern of characters with the same length for further analysis and testing.



This is the code I used.

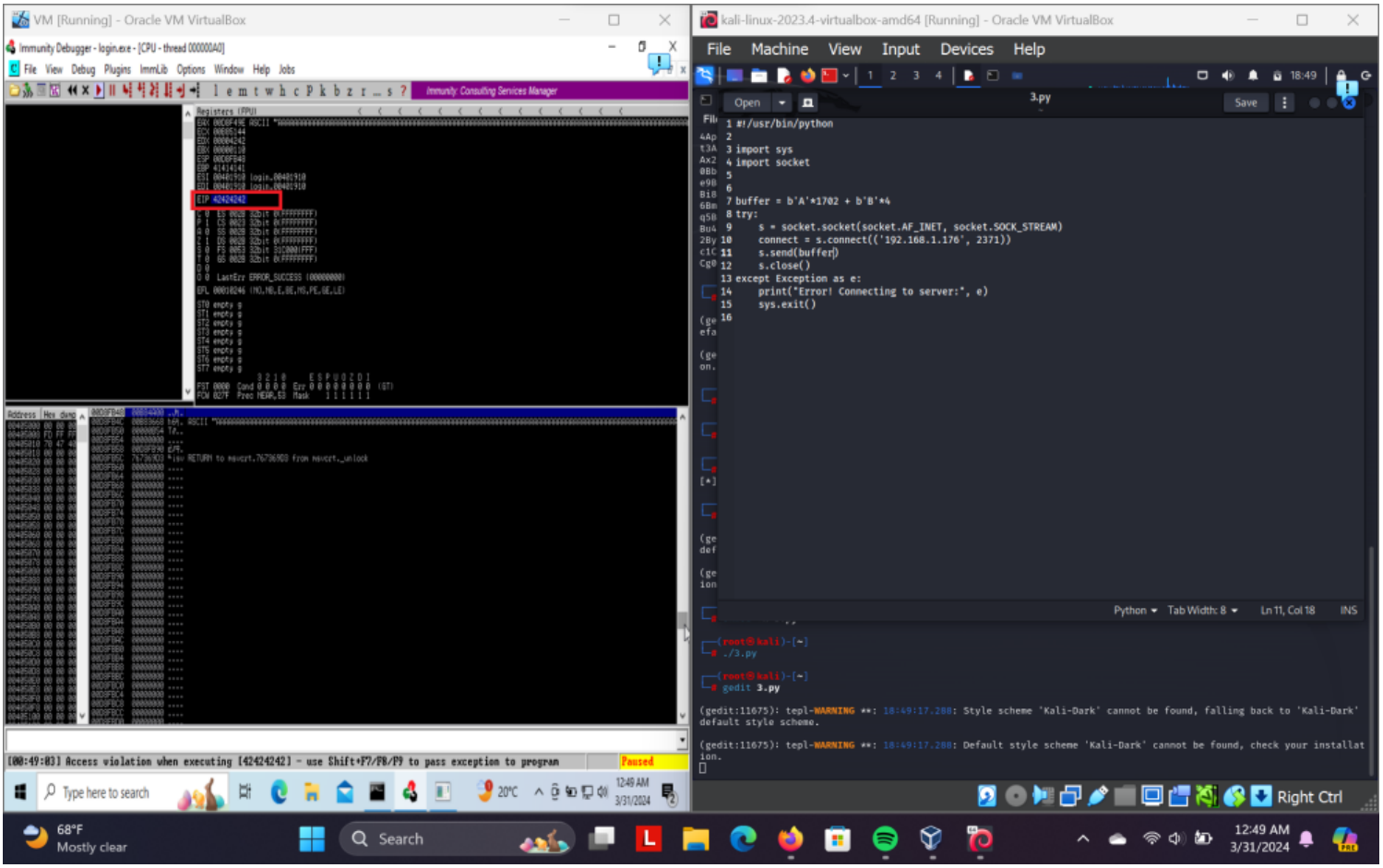


After generating the pattern, I successfully identified the EIP (Extended Instruction Pointer) by analyzing the crash. With this crucial information in hand, my next step is to craft a payload to overwrite the EIP, thereby gaining control over the program's execution flow.



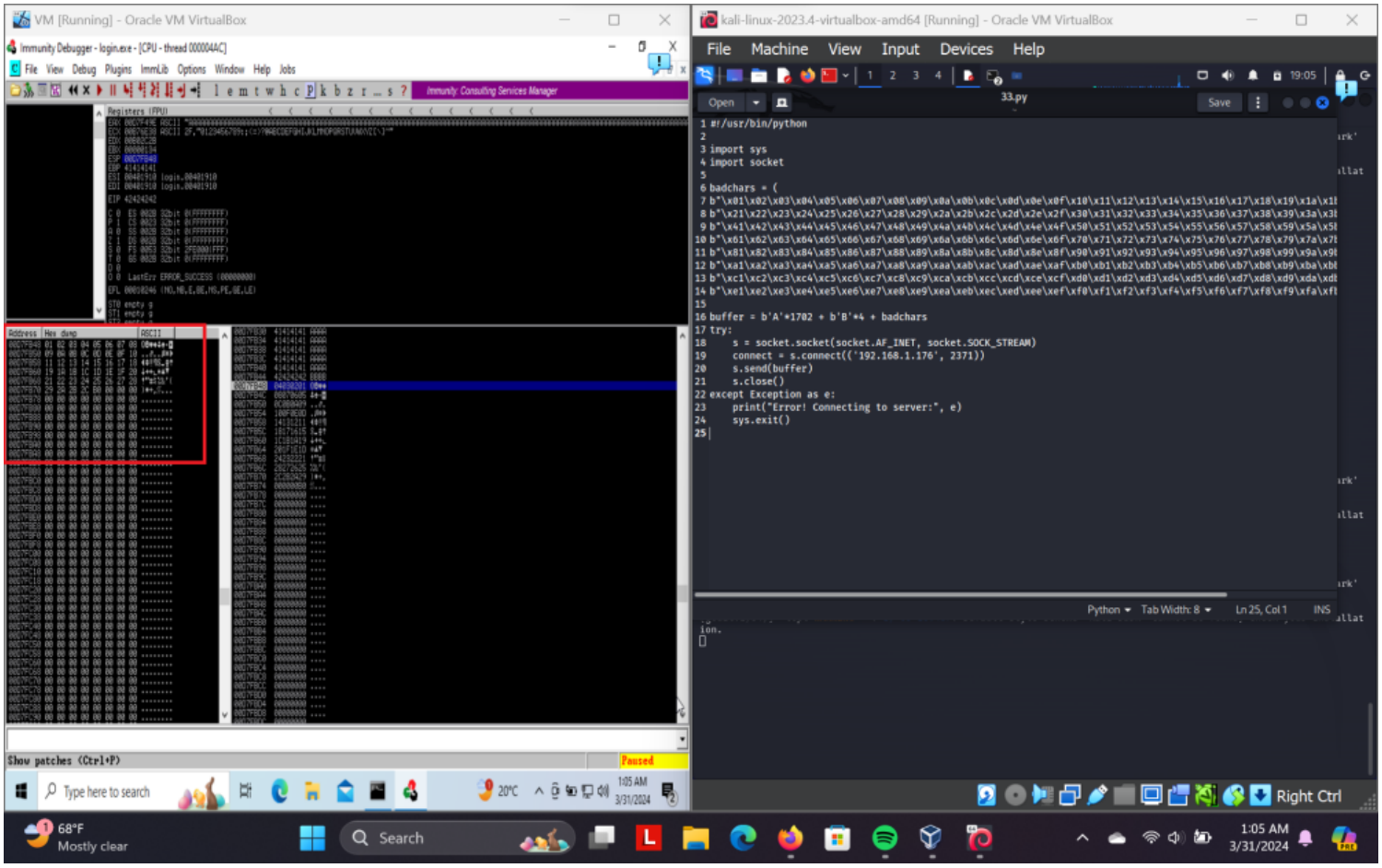
After identifying the EIP of the program, I utilized msf-pattern_create to generate a pattern of characters. Following this, I employed msf-pattern to determine the offset of the specified sequence within the generated pattern.

Overwriting the EIP



After determining the exact offset, I attempted to gain control of the EIP by crafting a buffer with the calculated offset, followed by replacing the EIP value with the specified four bytes ('BBBB' = 42424242). This was achieved with the following code: buffer = 'A' * offset + 'B' * 4.

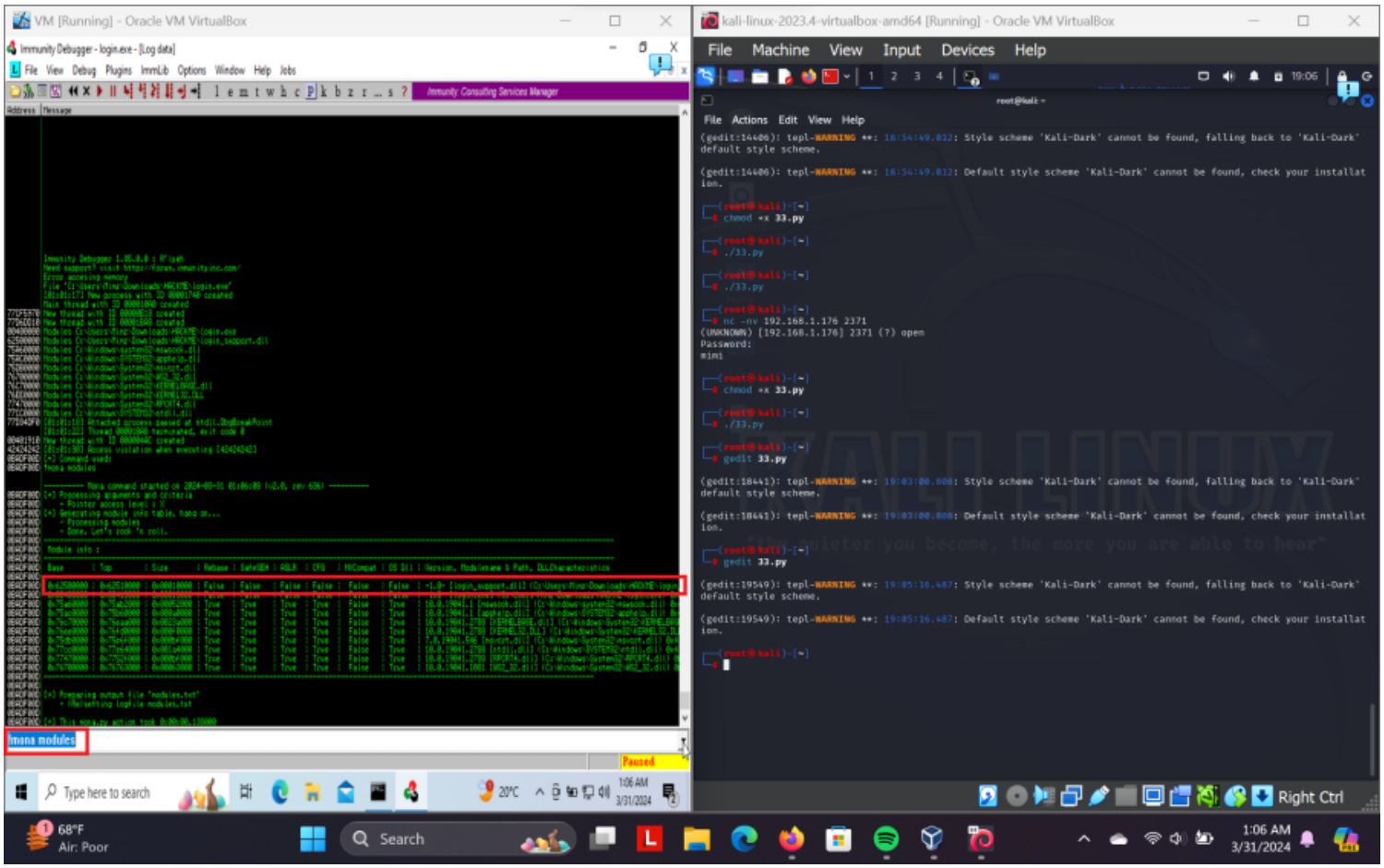
Finding Bad Chars



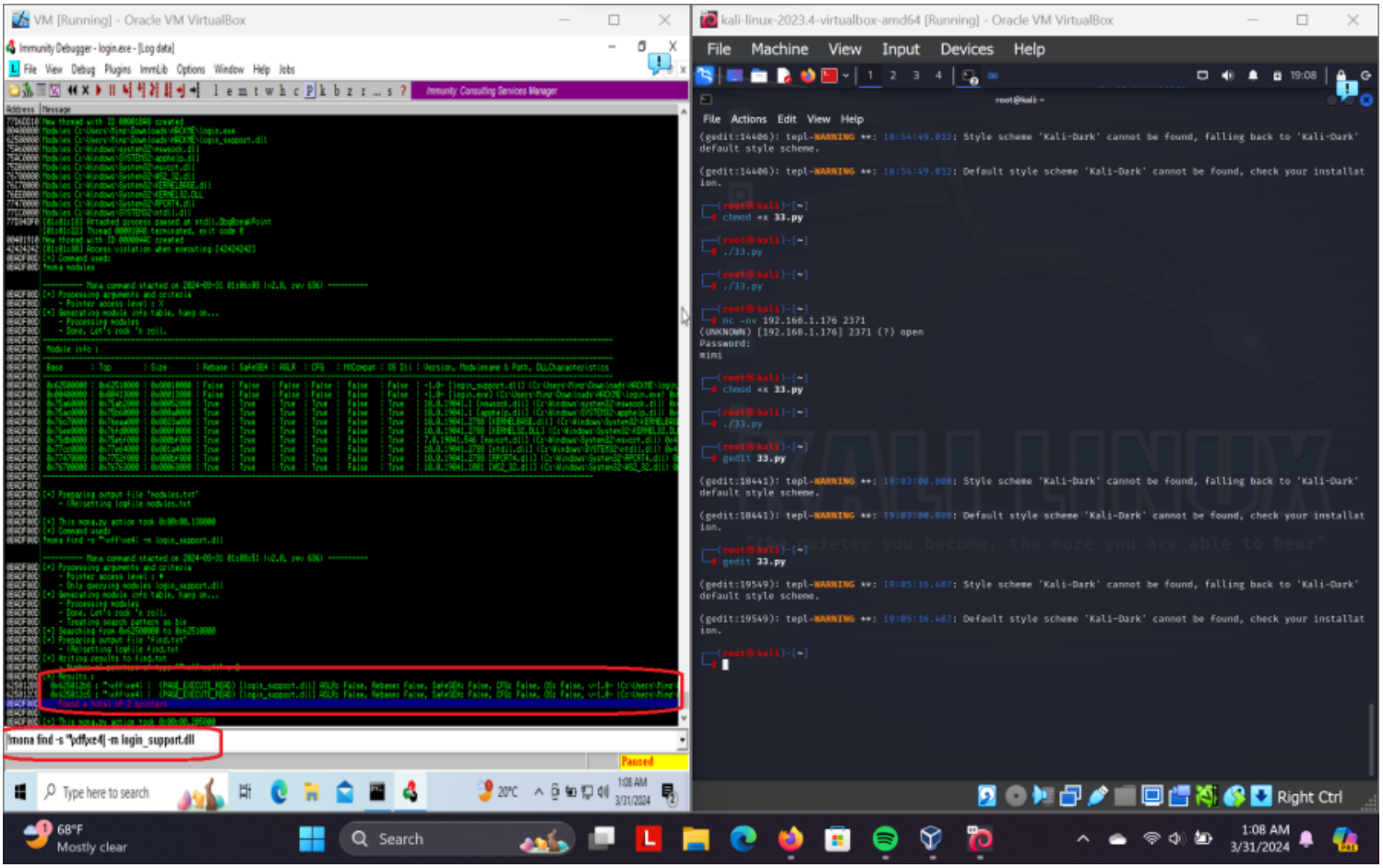
I proceeded to test for bad characters and discovered that the problematic ones were identified as:

0x00,
0x0d,
0x0e,
0x46,
0x47

Mona Module

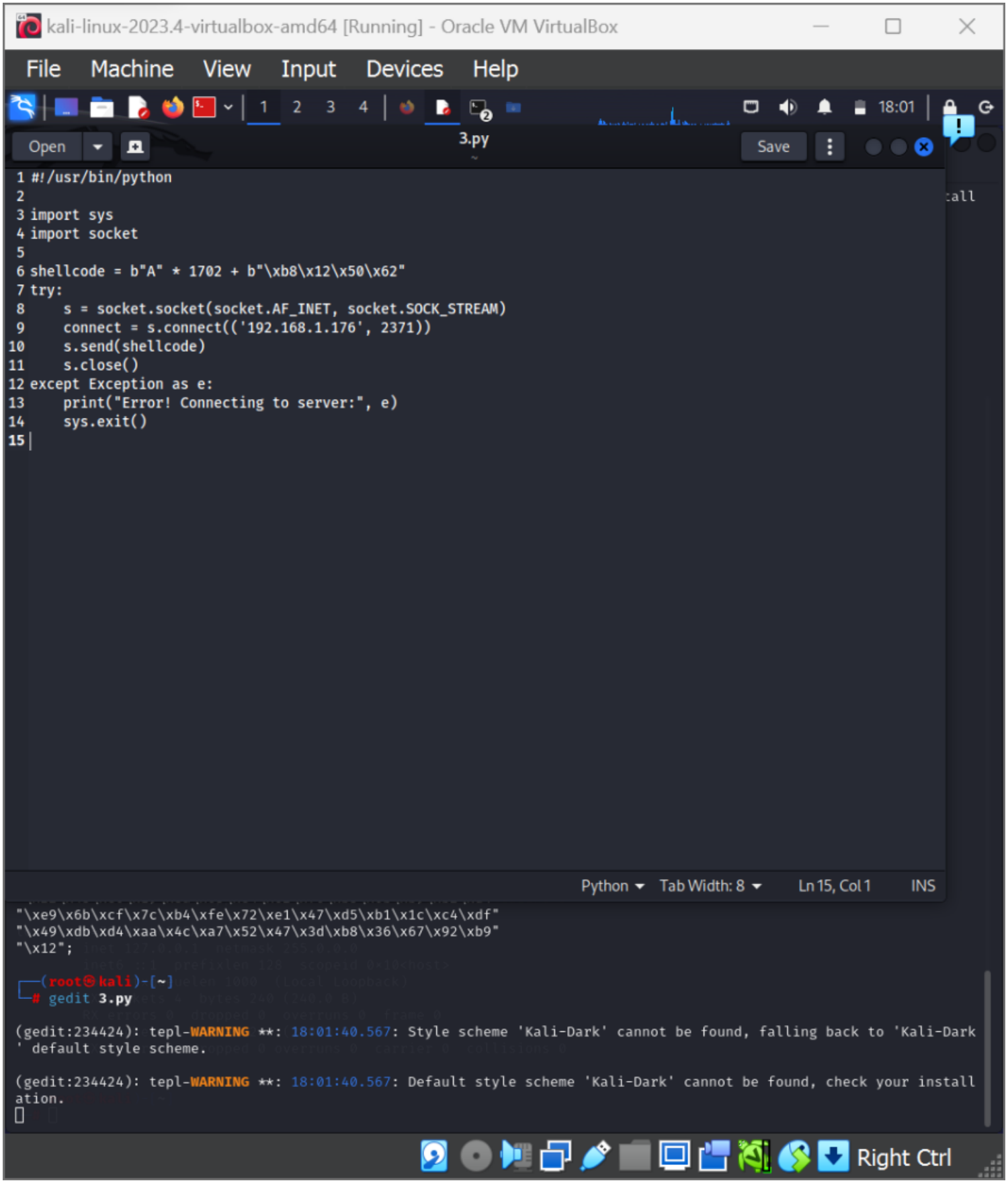


Here I can see the first file login_support.dll is least secure and can be used for exploitation.

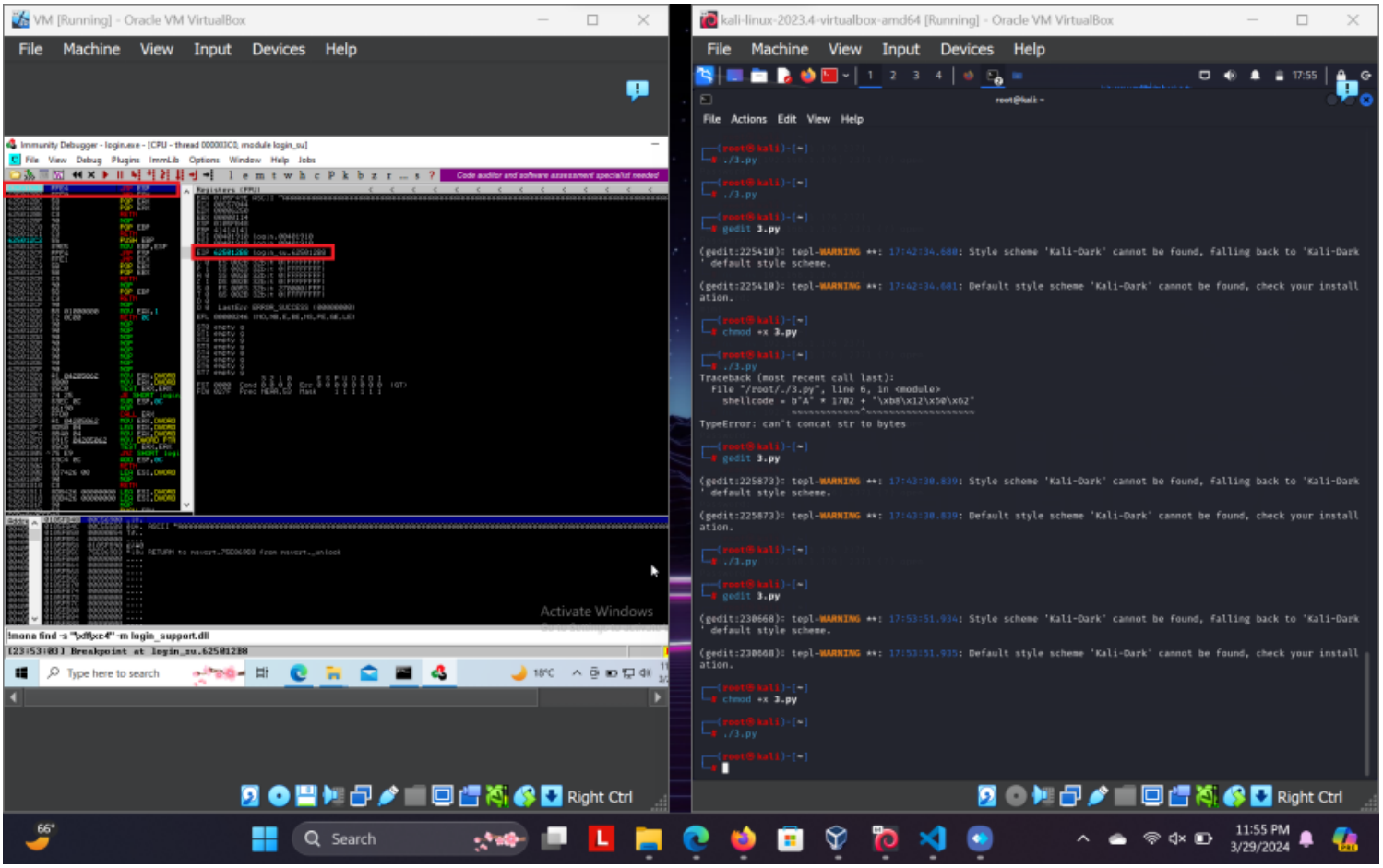


I utilized mona modules to identify suitable modules for exploitation. During the analysis, I discovered two pointers, 0x625012b and 0x625012c.

Notably, the pointer located at 0x625012b points to a JMP ESP instruction. This finding suggests a potential vulnerability within the program, as it could be exploited using techniques such as buffer overflow to redirect program execution flow

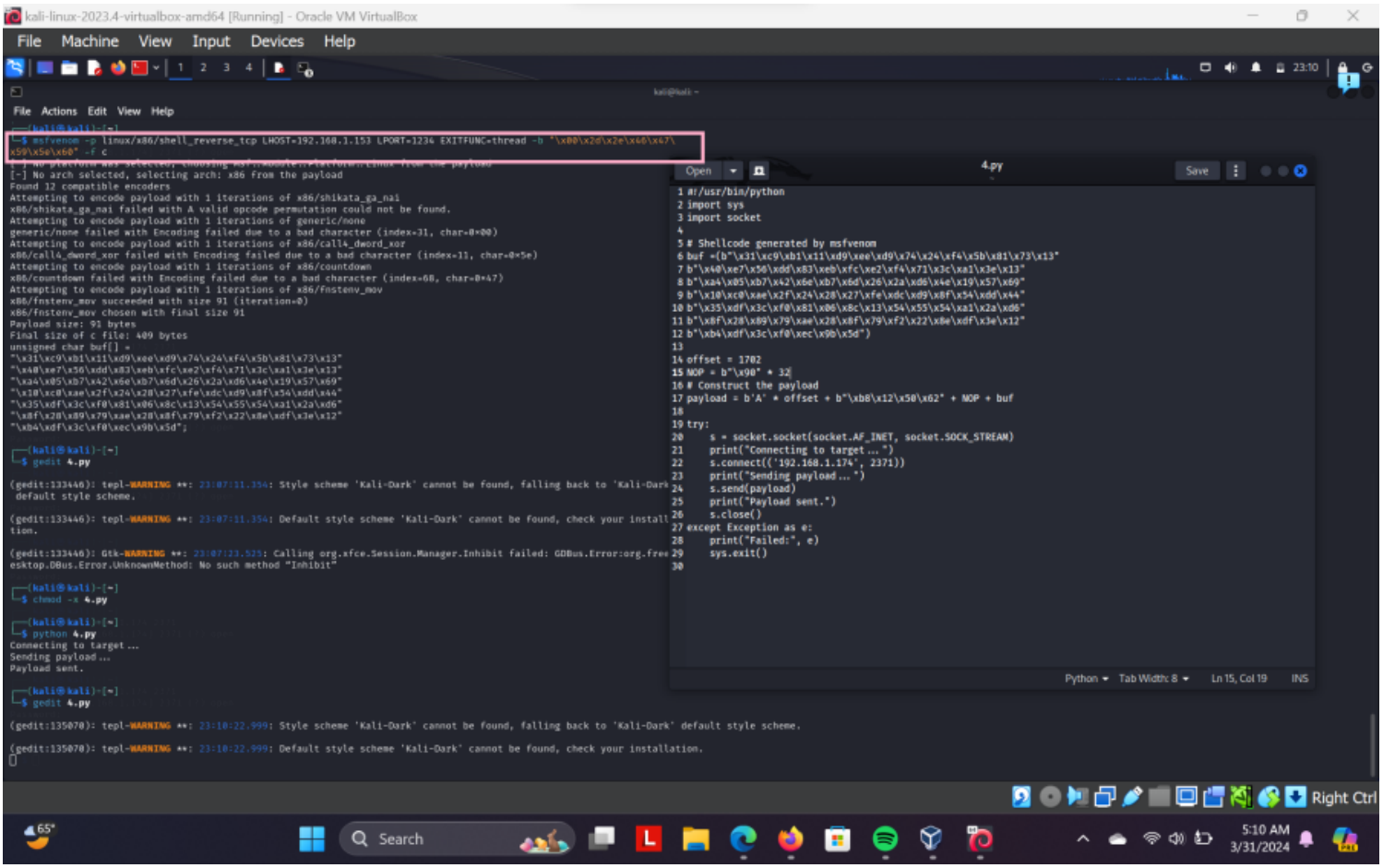


I set a breakpoint at address 0x625012b and executed a Python script to manipulate the EIP (Extended Instruction Pointer) using this address.

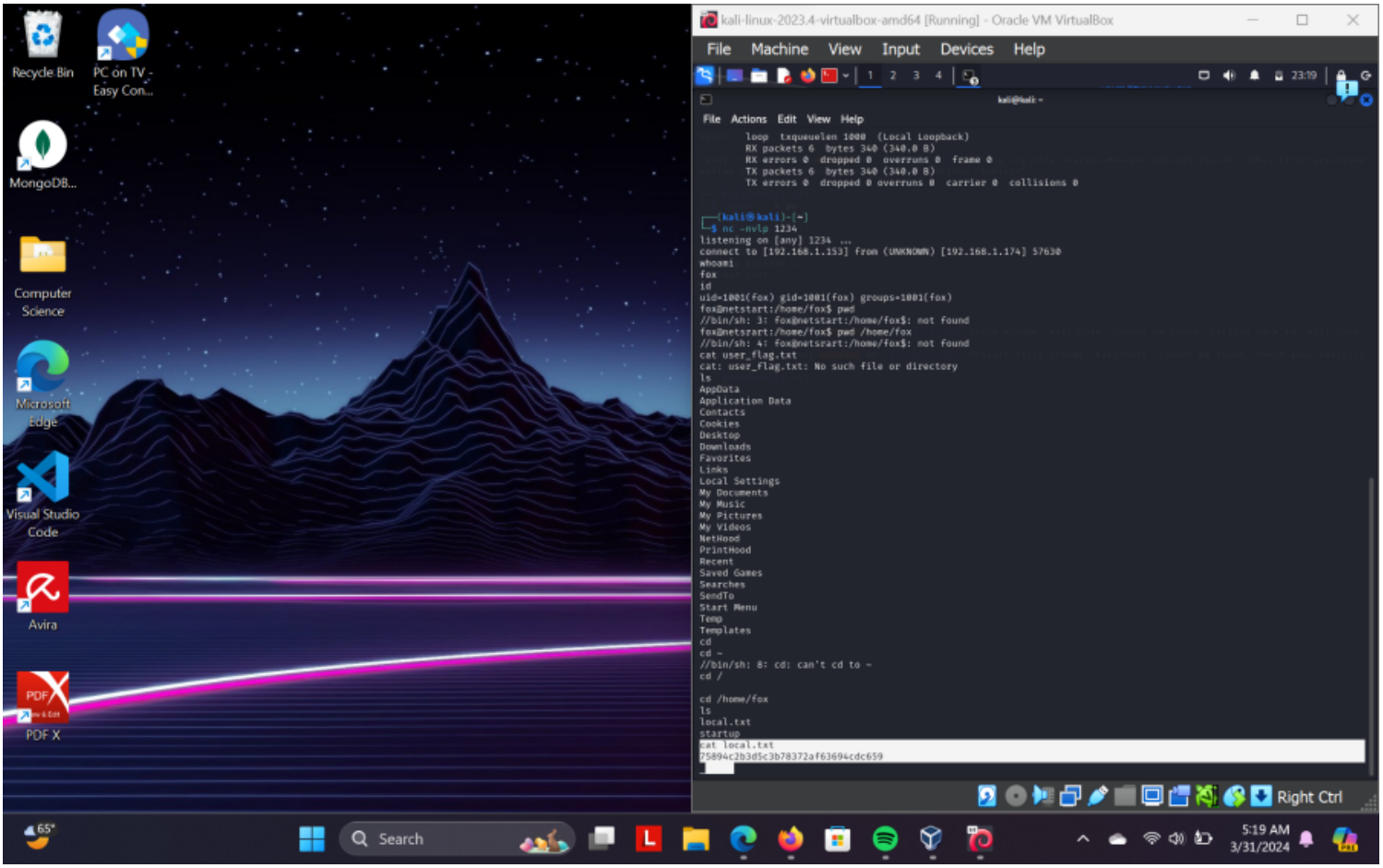


The execution was successful, and the program redirected to the specified address (0x625012b), triggering the breakpoint at the login_su function.

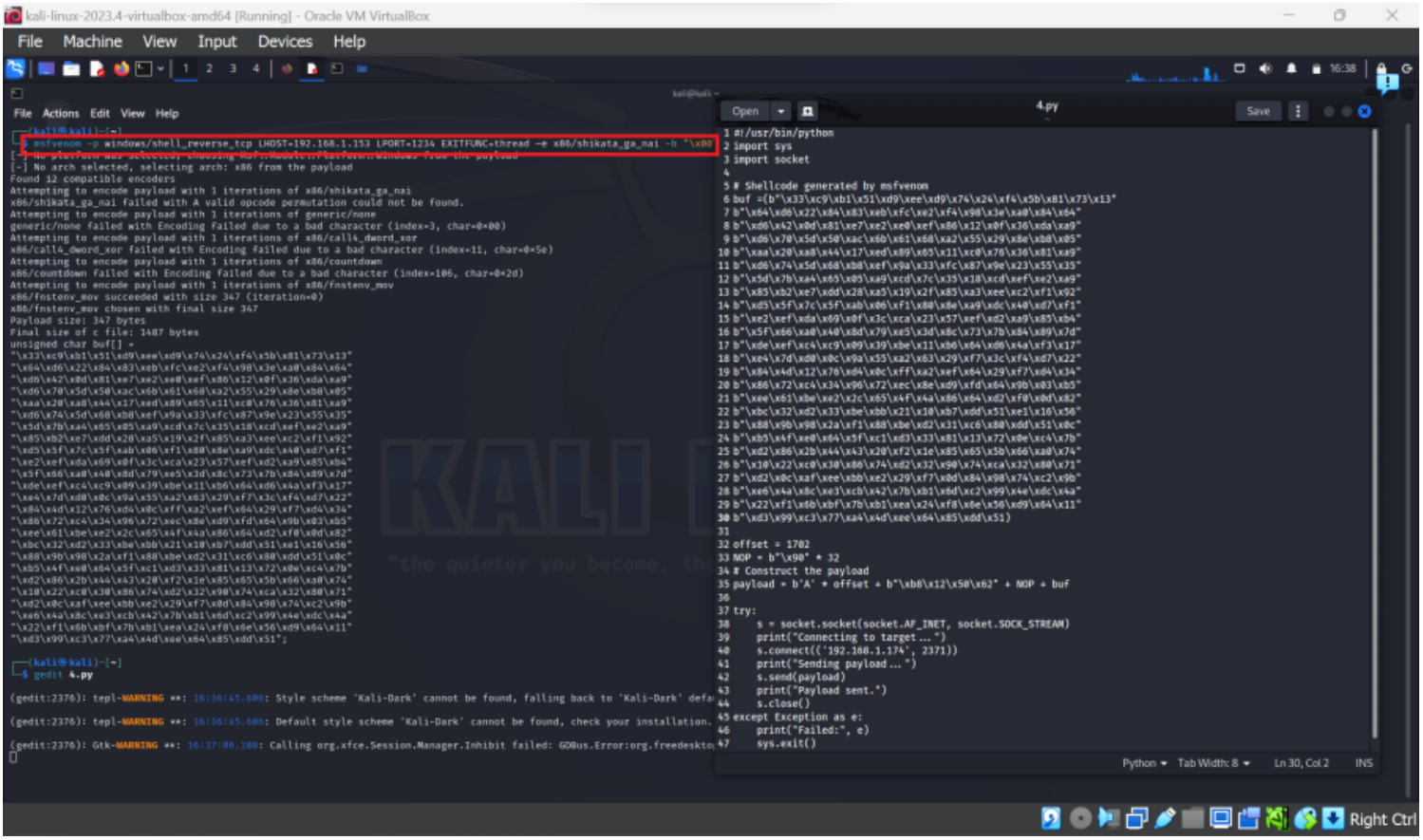
ShellCode



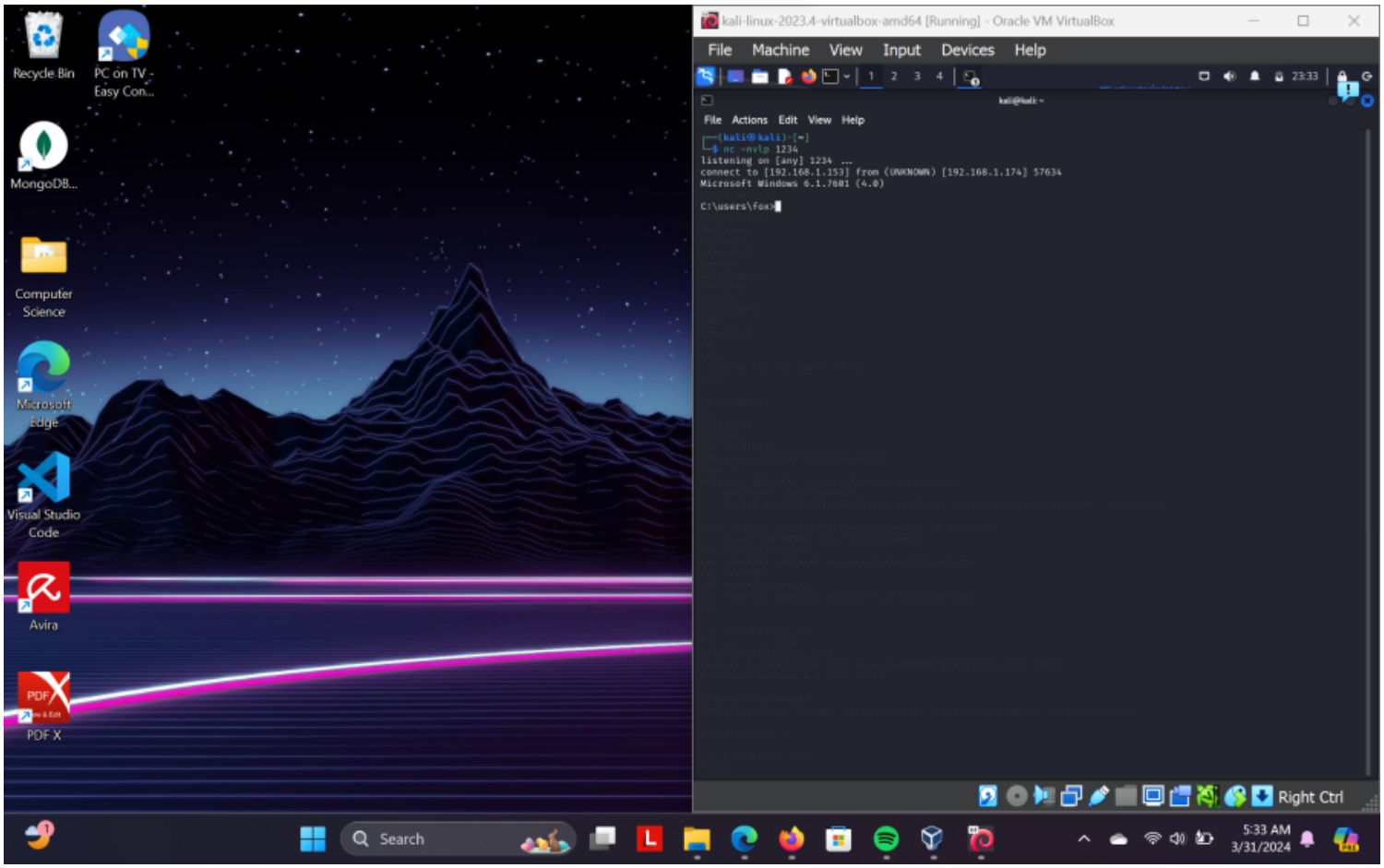
I utilized msfvenom to generate a reverse shell payload for Linux. I specified the LPORT and LHOST parameters to define the listening port and the attacker's (me) IP address, respectively. Additionally, I included the option to avoid certain bad characters, specifically '\x00\x0d\x0e\x46\x47\x59\x5e\x60'



Entering the program was successful. (Linux)



I performed the same process using msfvenom to generate a reverse shell payload for Windows as well.



I have successfully gained access to both a Windows and a Linux machine using my payloads