Structured Exception Handling (SEH) Overflow

About:

This write-up provides an in-depth exploration of Structured Exception Handling (SEH) overflow exploitation, a technique used to hijack the execution flow of a program by manipulating exception handlers. The document covers the basics of SEH, the process of identifying vulnerable applications, and a step-by-step guide to crafting an exploit to gain control over the program's execution. This write-up aims to help security researchers and pentesters understand how SEH overflow vulnerabilities can be discovered and leveraged, along with mitigation strategies to prevent such attacks.

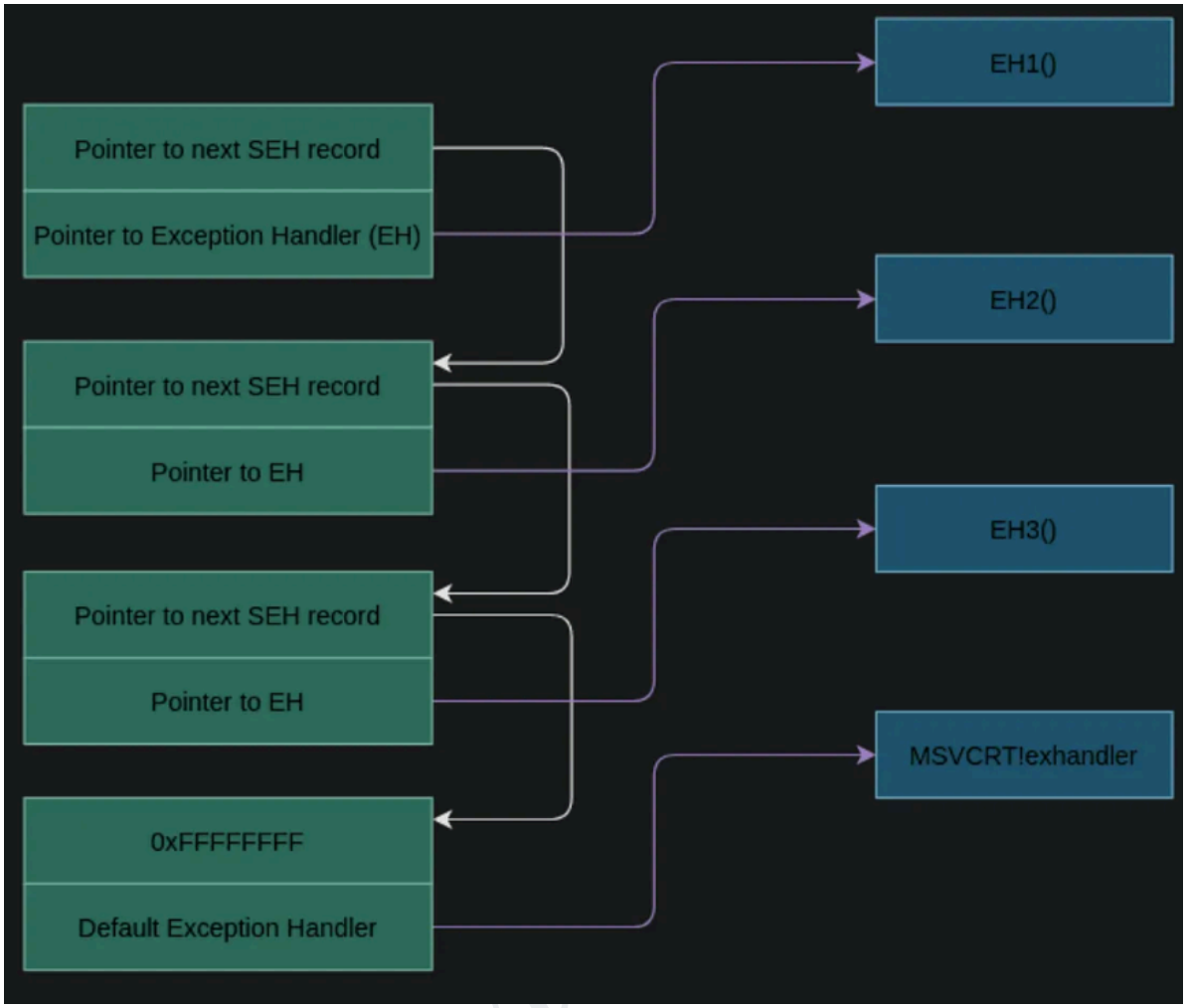
Objectives:

- Recreate the Blog & Exploit Development: Create a detailed reproduction of the original blog post available at the given link. Develop a working exploit.
- Proof of Concept Video:
  Develop a 1-2 minute video demonstrating the working exploit of SEH overflow.
- Report:
  Write a detailed report elaborating on the blog content and your findings from the video demonstration.
  The report should include background information, step-by-step analysis, and a conclusion that encapsulates the learning outcomes of the assignment.
  Format your report properly with appropriate headings, subheadings, and references.

Structured Exception Handler:

A Structured Exception Handler is a pattern of code developed to try and execute a particular line of code properly. If the code possesses some error or if it would throw an exception then the exception handler would handle the code and prevent the executable/app/code from being terminated abruptly showing the exception case. Exception handlers are stored in the form of a linked list and the default handler is Windows Exception Handler.

In an SEH chain, one exception record has 2 pointers of 4 bits each. One pointer points to the current exception handler and the other pointer points to the next exception record.



Source:https://miro.medium.com/v2/resize:fit:1400/format:webp/1*OGcJj4cWBOjJ3IQW3BO7sA.png

When the exception occurs, it is executed through the first record. If the exception handler of that record is able to resolve the issue then the code executes. Else it points to the next record of the handler and checks if the next record can solve the problem. This chain of forwarding the issue continues in the handler until a solution is found. If no solution is found then the program crashes.

SEH Vulnerability:

The SEH vulnerability is similar to the one where we have to overwrite the EIP register and gain shellcode access by pointing the EIP to ESP in the VulnServer vulnerability.

Here we have to create an error condition so that the SEH is initiated. In the handler we have to overwrite the pointer to the next exception record as well as the exception handler so that we can point to a specific area in the memory location which would be containing our exploit.

Steps for exploitation:

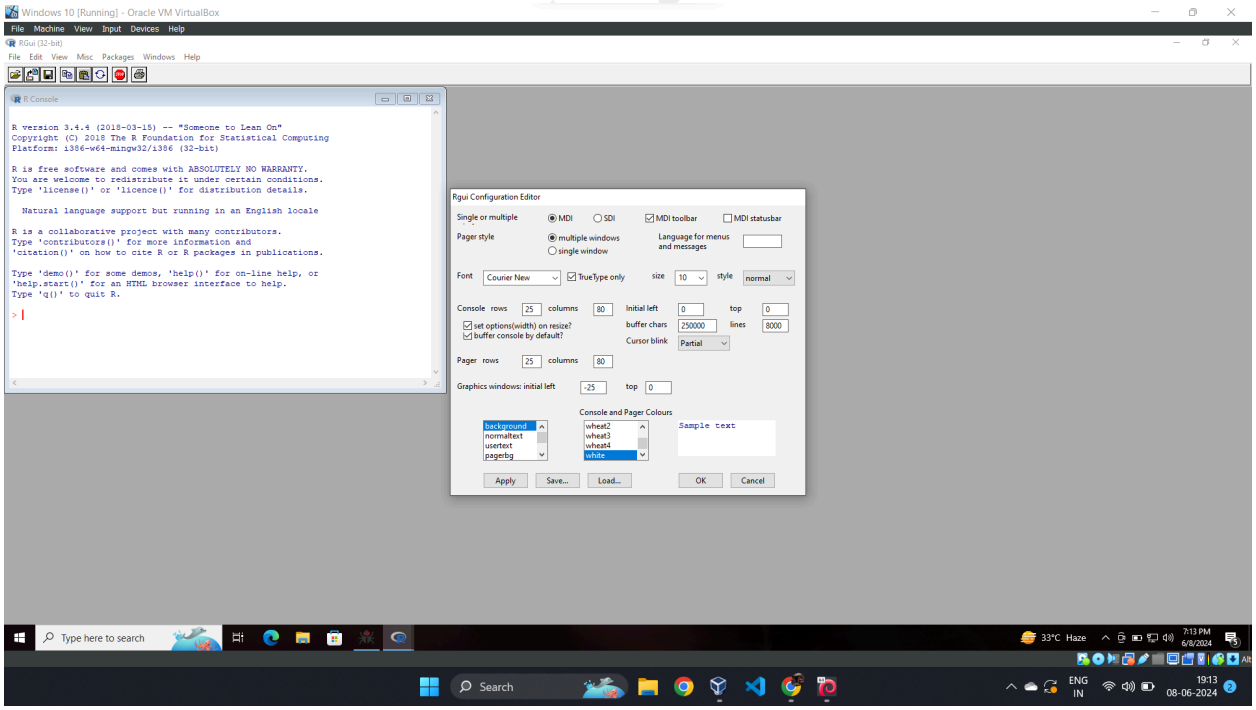
1. Identify the vulnerability: send input into the vulnerable placeholder to try crashing the running application.
2. SEH offset: find the offset, after how many input characters was the SEH record overwritten.
3. POP POP RET: find a set of pop pop return instructions to return EIP from the handler to the address.
4. Final: find a jump instruction to jump from the exception handler into the memory address and execute the malicious payload.

Setup:

Download X32dbg from the official website. Additionally, a copy of the ERC plugin for X32dbg will be required. Instructions for installing the plugin can be found on the Coalfire GitHub page.

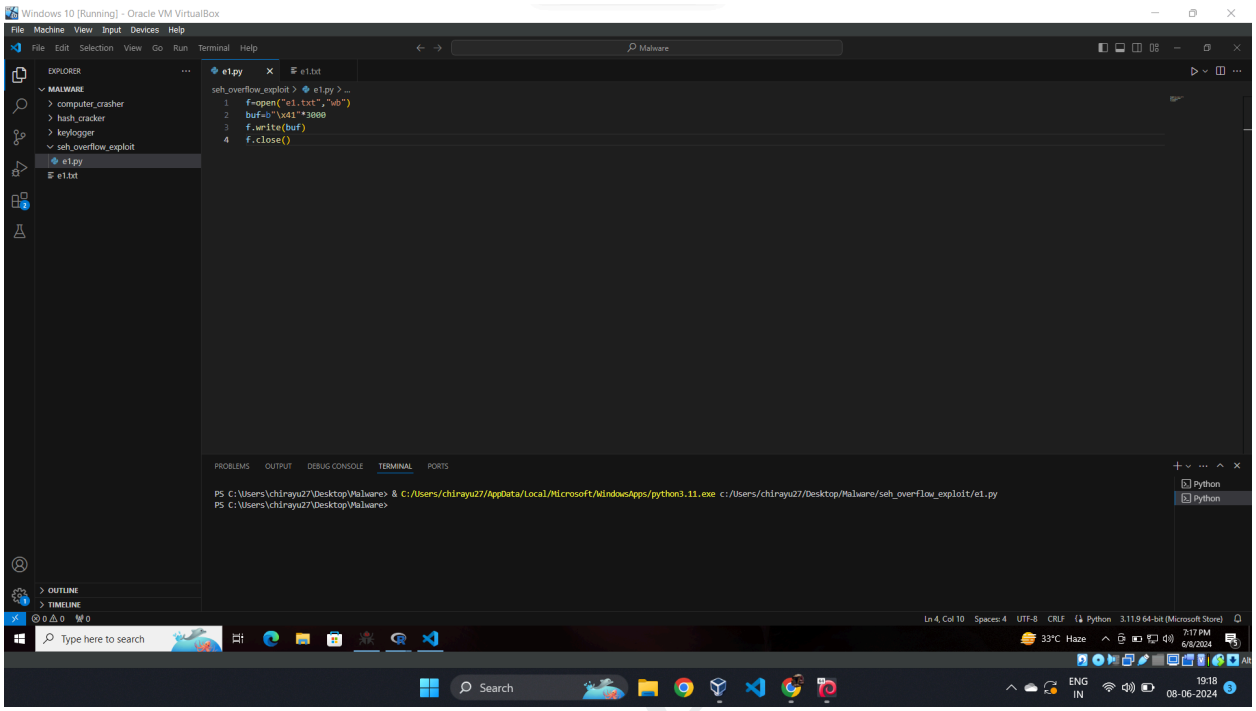
Download a copy of the vulnerable application (R.3.4.4). To confirm everything is working, X32dbg is started, and File -> Open is selected. The location where R.3.4.4 was installed is navigated to, and the executable is selected.

By clicking through the many breakpoints, the R.3.4.4 GUI interface appears.

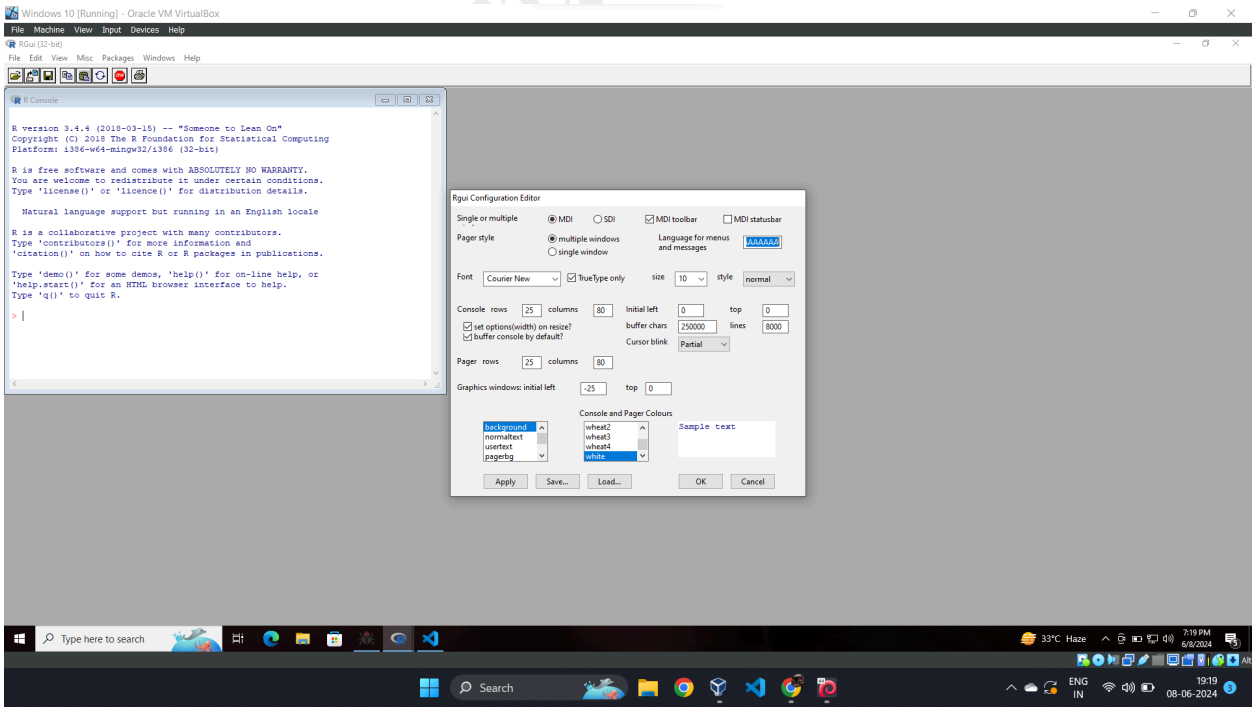


Procedure:

Confirm if a vulnerability really exists. Open the Rgui in x32dbg and navigate to the Rgui Configuration Editor through the Edit option.

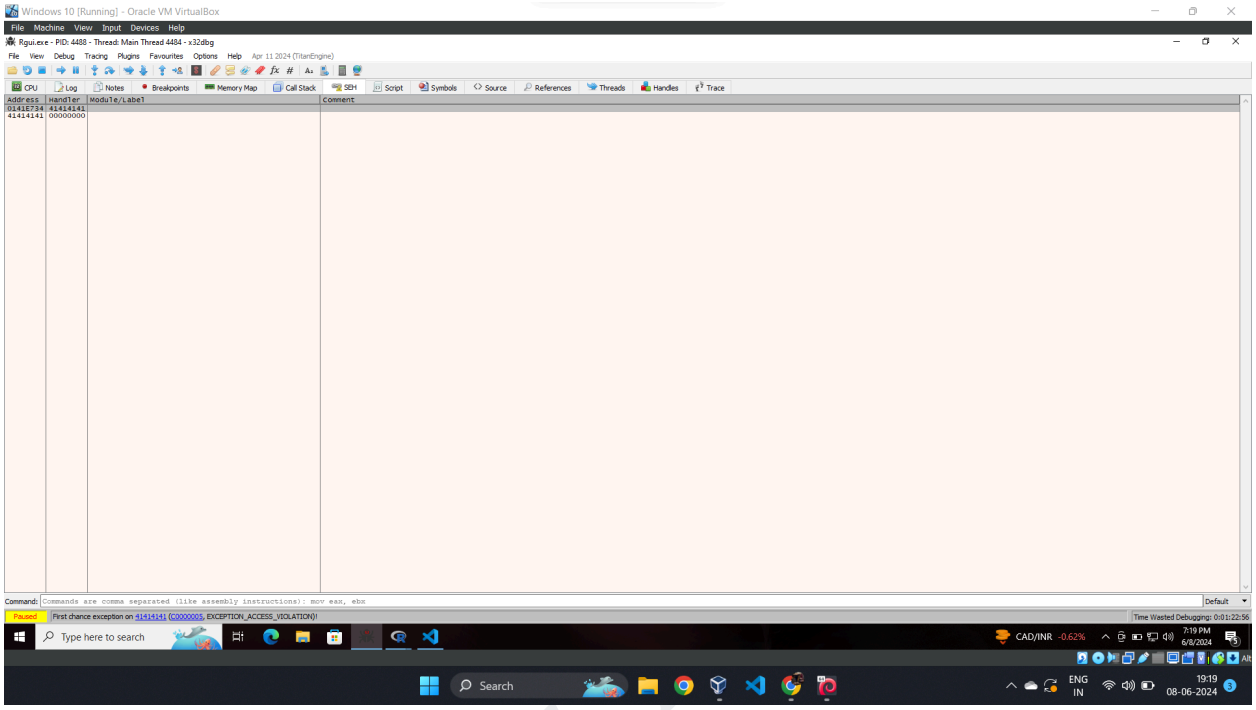


Develop a python code to generate a string of 3000 ‘A’s and generate a text file.



Copy the content of the text file and paste it into the empty space provided in ‘Language for menus and messages’ option.

Click OK and the application crashes.



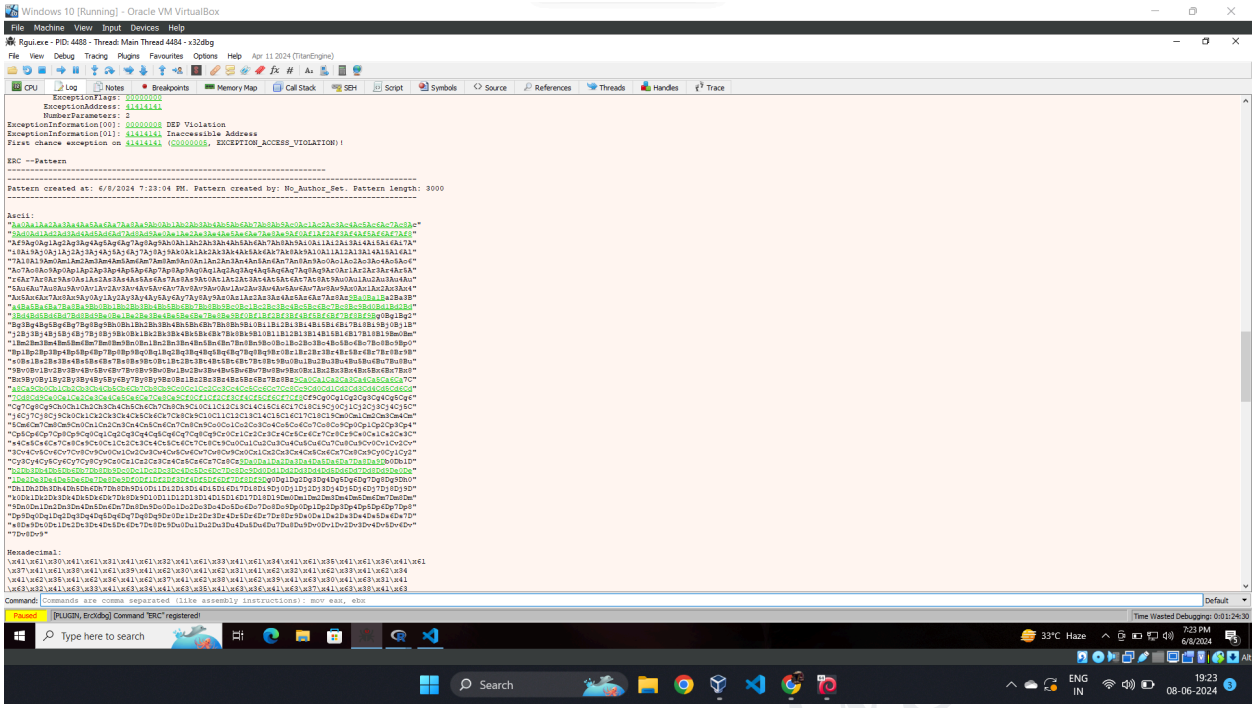
In the x32dbg SEH section it can be seen that the exception handler is overwritten with ‘41’ which is the hexadecimal code for ‘A’.

Thus the vulnerability is confirmed successfully.

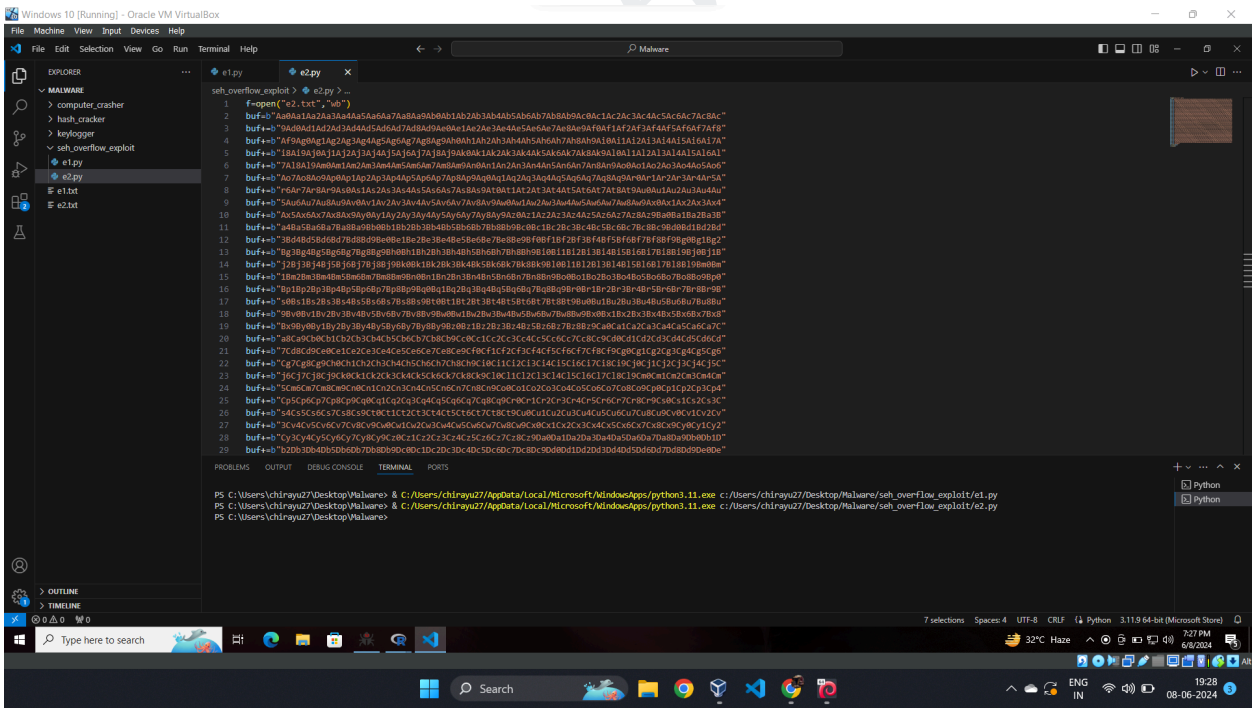
Now we want to know after how many characters were the SEH overwritten. To do this, first generate a pattern of characters by using the following command:

ERC --pattern c 3000

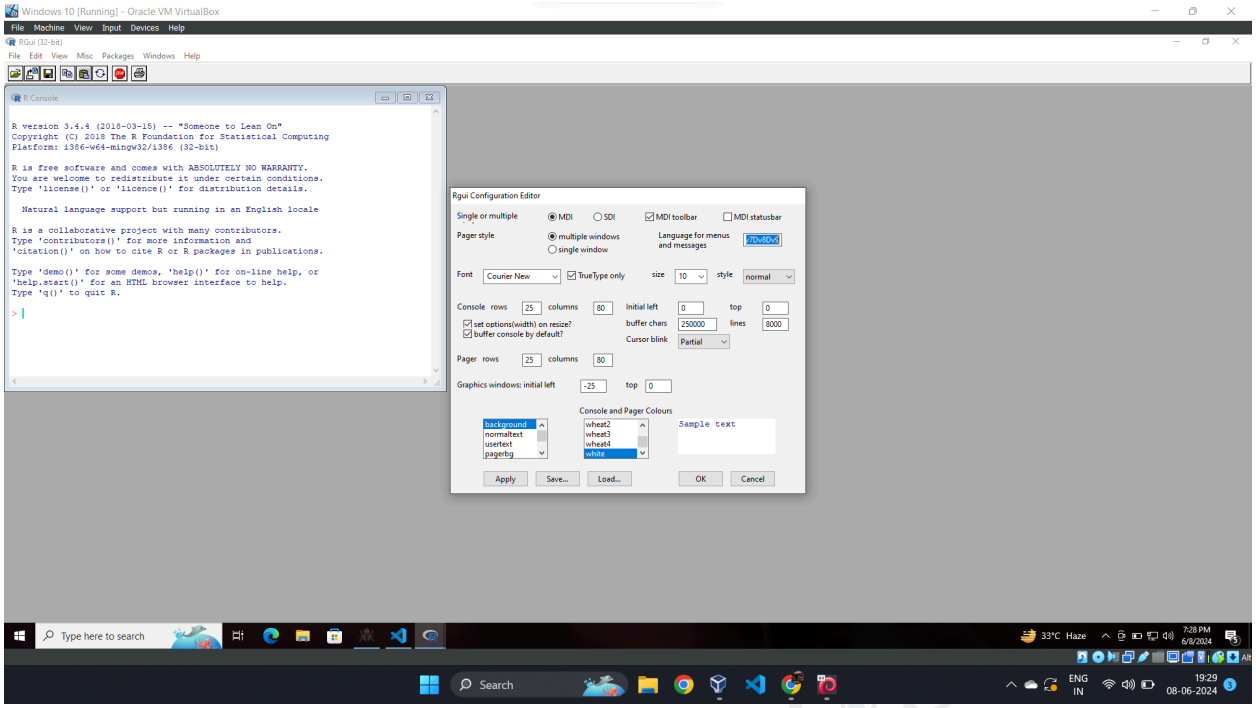
This command will generate a pattern of characters as follows. It can be traced in the Log section of the x32dbg.



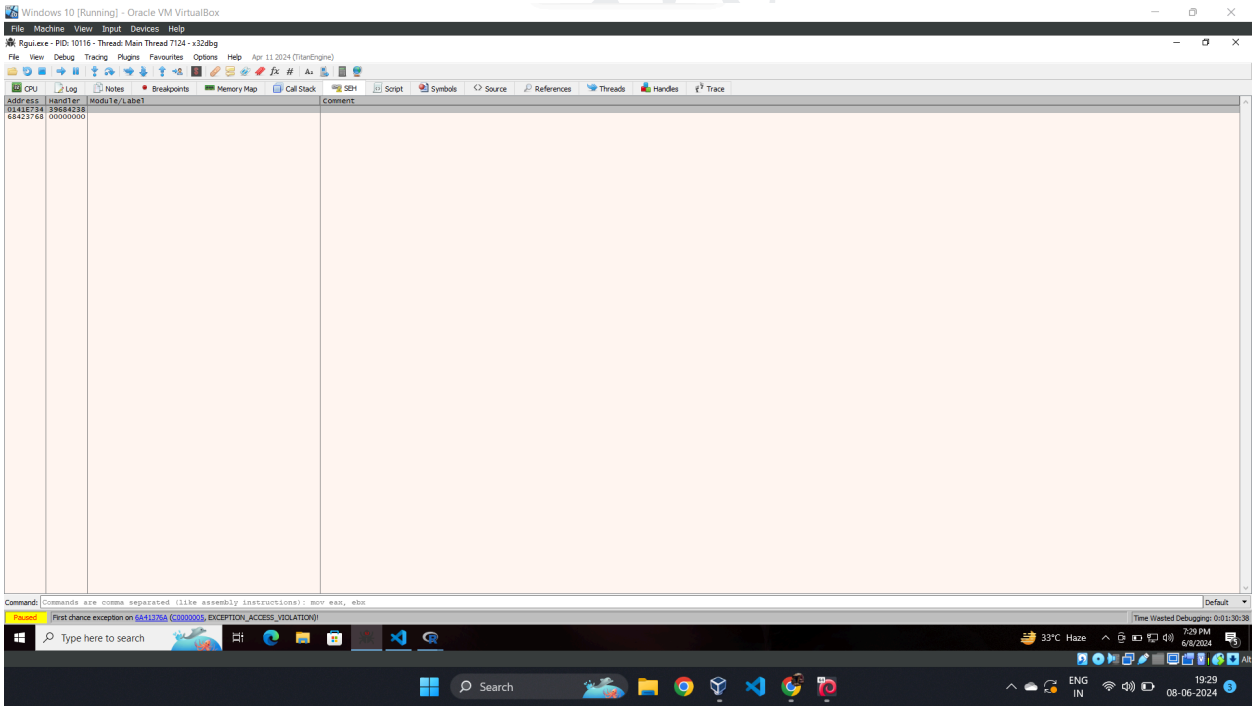
Copy the generated pattern and paste it in a python code to form a text file out of it.



Copy the pattern from the text file and input it in the vulnerable placeholder.



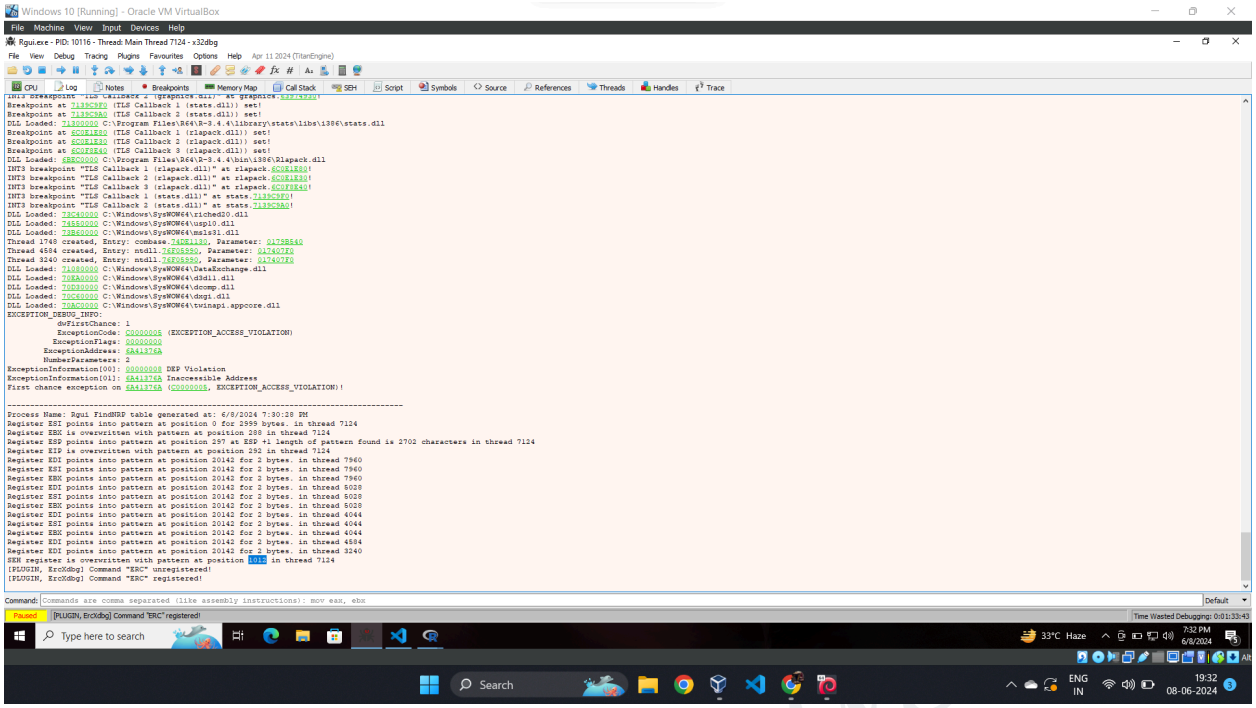
The application crashes again as expected.



The SEH is overwritten with a random set of hex values from the input payload.

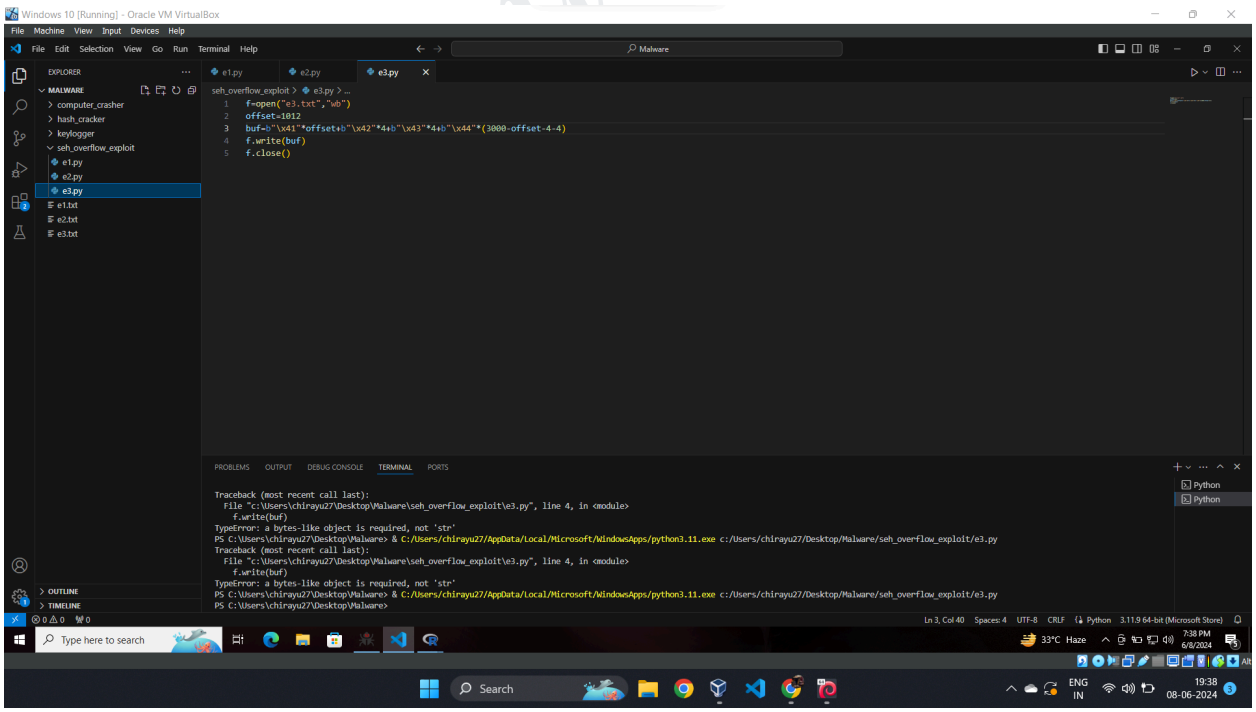
To find the offset for overwriting SEH, execute the following command:

ERC --FindNRP

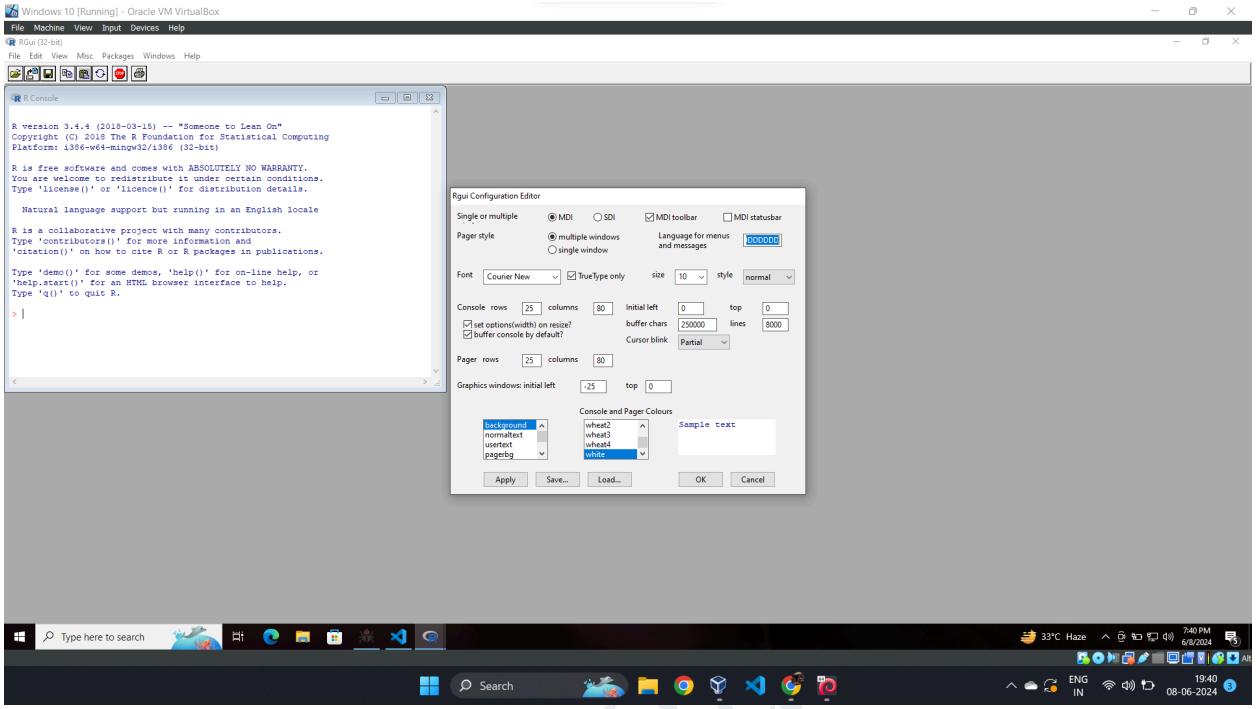


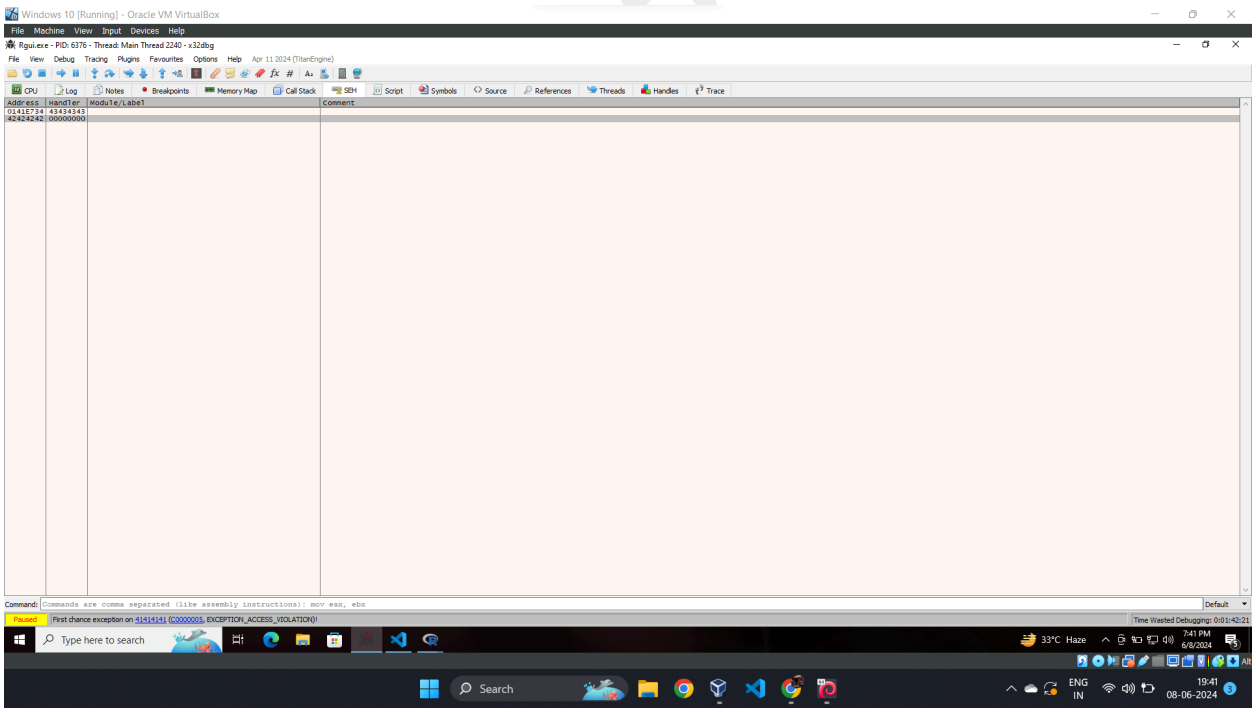
We can see that the SEH is overwritten with pattern at position 1012.

Now we have to confirm the offset value by creating another pattern of characters. Generate a python code with 1012 ‘A’s, 4’B’s, 4’C’s and rest ‘D’s.



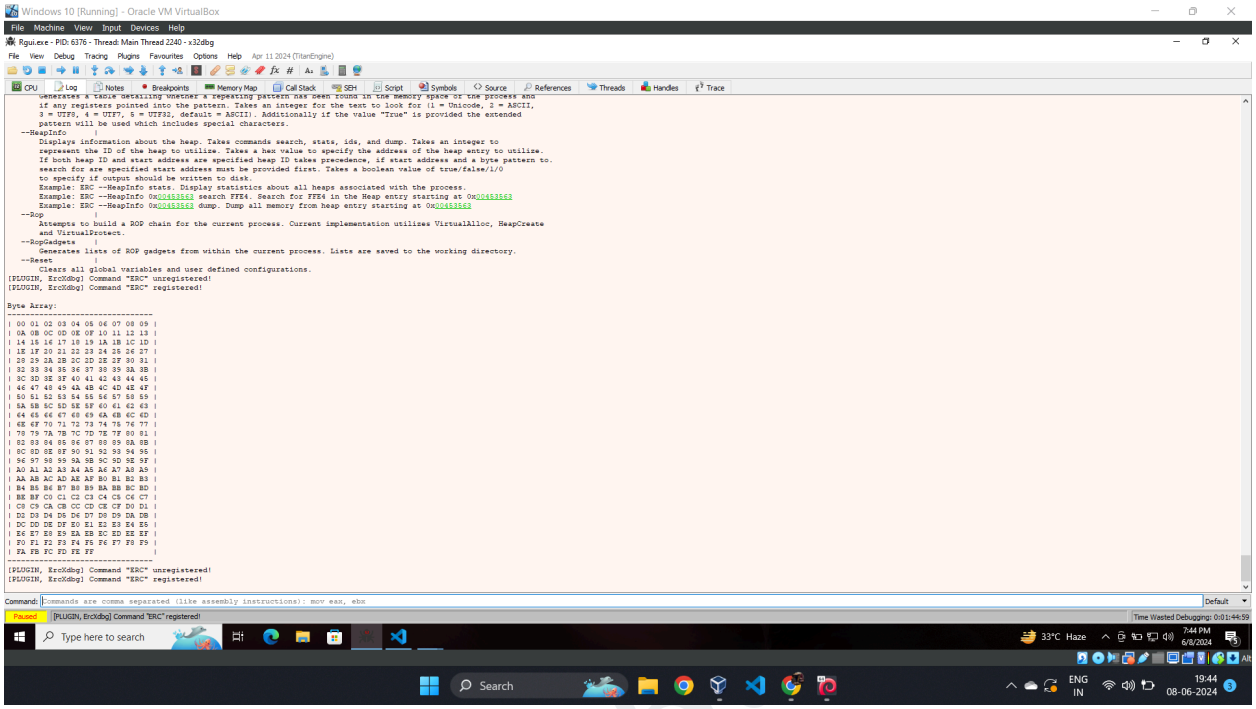
Copy the generated pattern from the text file and repeat the previous procedure.





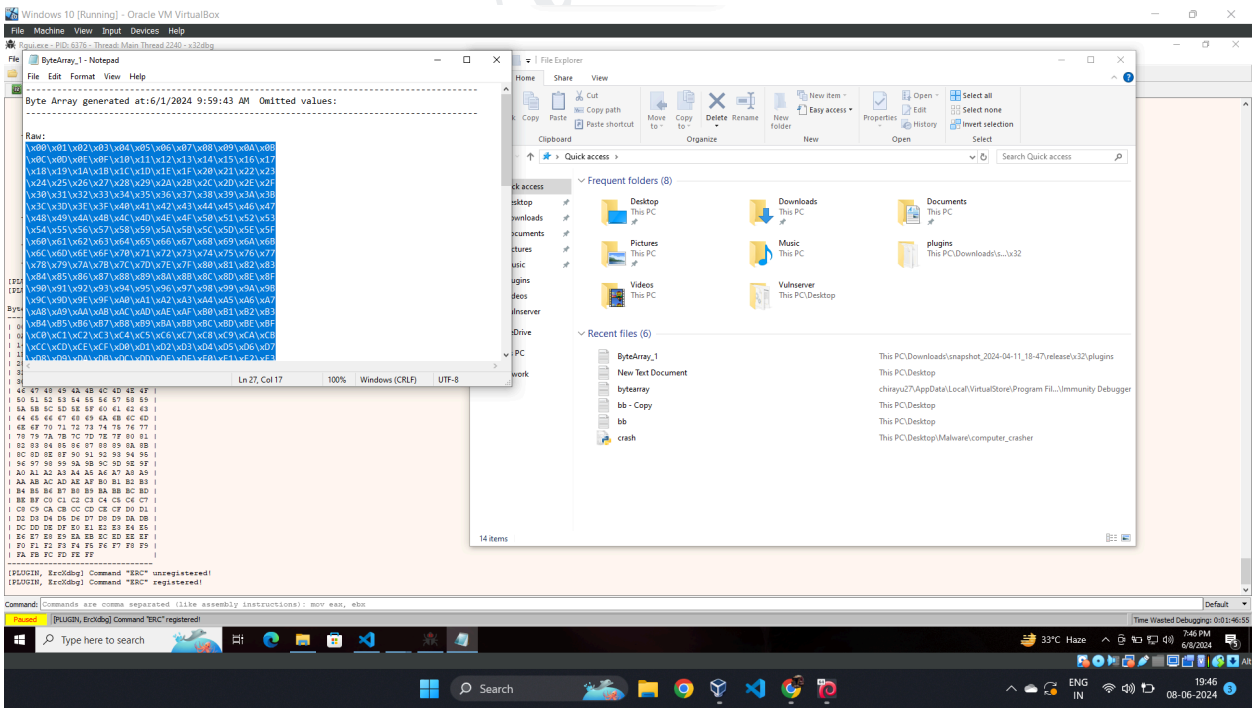
Thus the handler is overwritten with ‘C’s and the next address is overwritten with ‘B’s. Now we have complete control over or SEH records.

Now, we have to find the possible bad characters to ensure smooth execution of the malicious payload we enter.

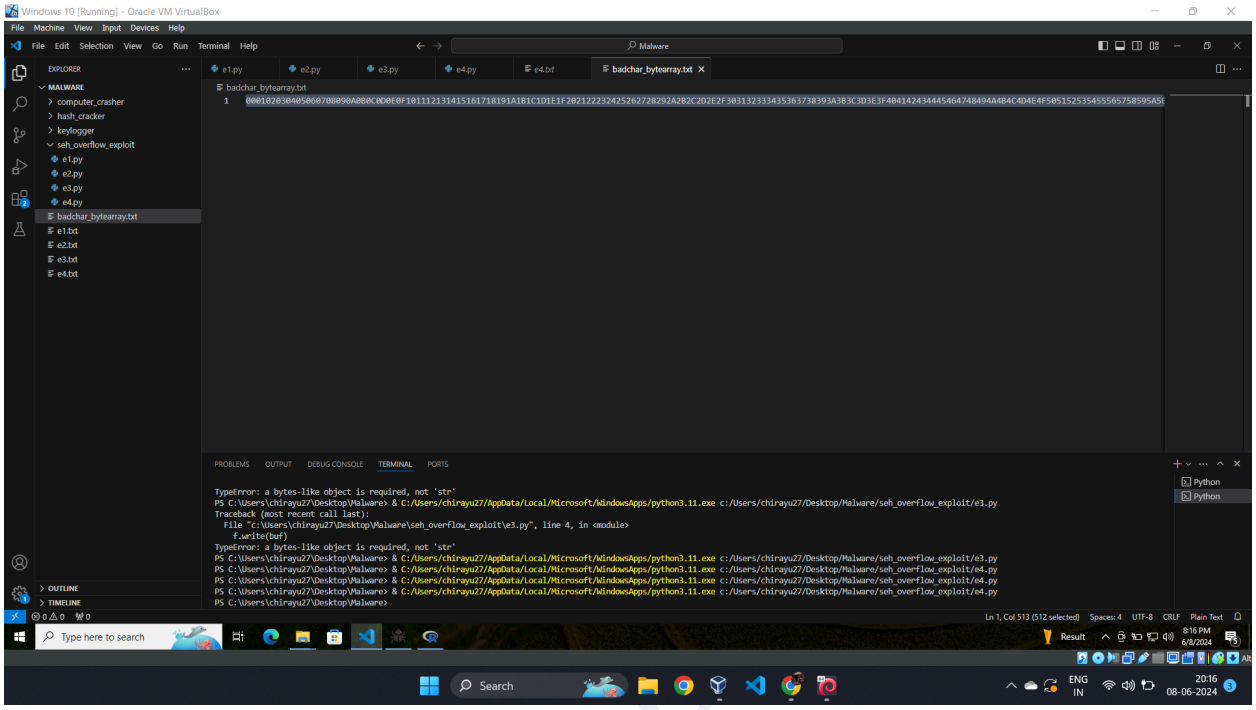


To find the bad characters first generate a bytearray by executing the following command:

ERC --bytearray



A text file and a binary file named *ByteArray_1* will be generated. Navigate to the location of the file and copy the bytearray.

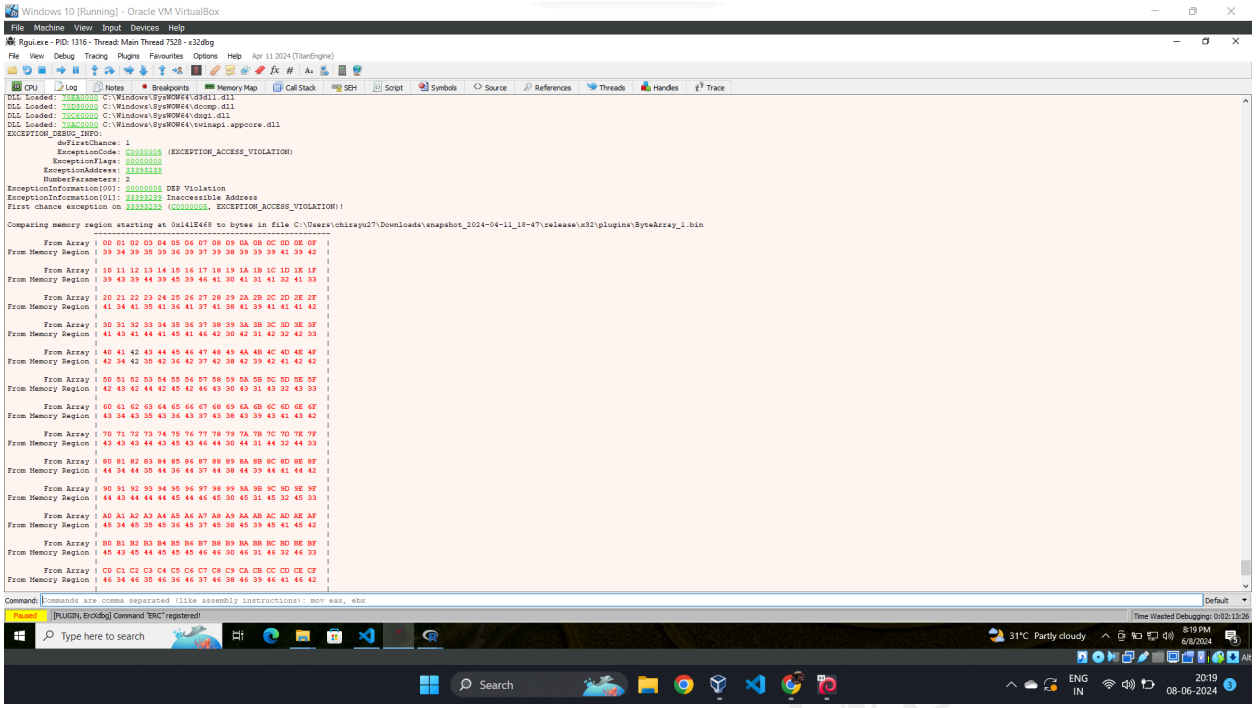


Enter it into the python code to generate a string of characters. Copy the contents of the generated text file and crash the application.

Compare the input characters with the ByteArray file to find out all the bad characters. To do so, use the following command:

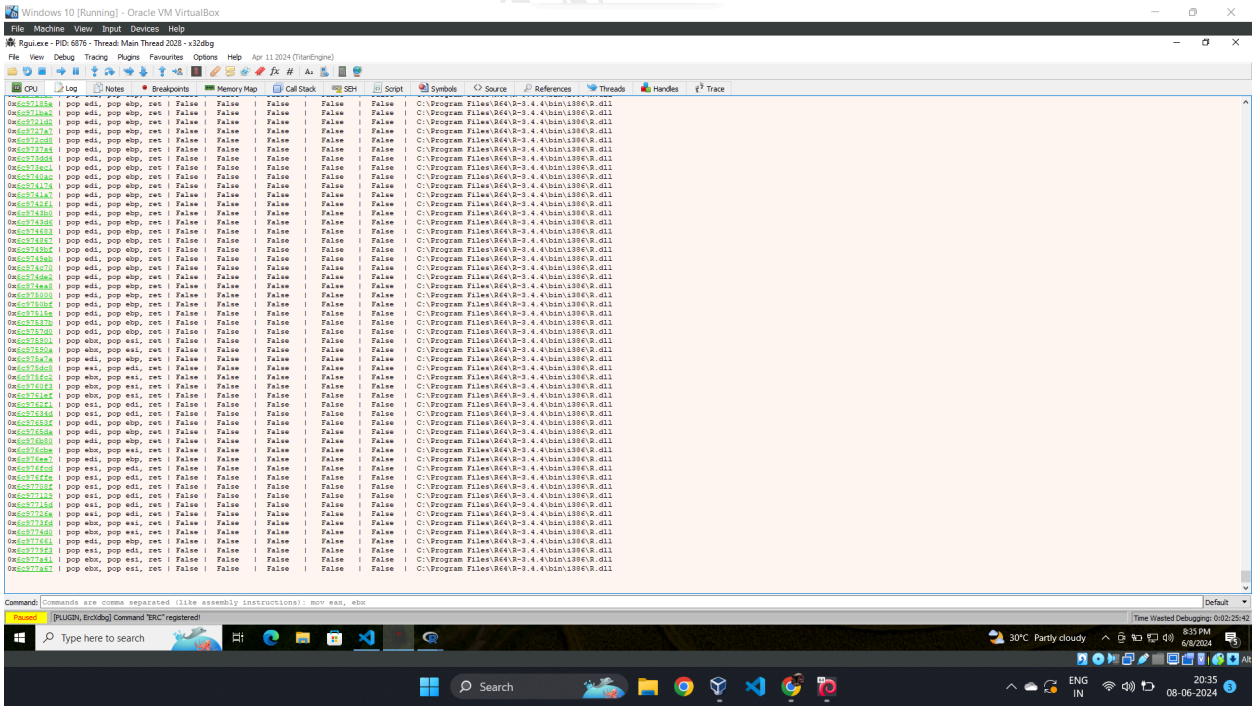
ERC --compare <address of ESP> <path to ByteArray_1.bin file>

All the characters that are not loaded into the memory properly are compared and displayed in the following output. As we can see, 00 is interpreted as 39 in the memory system and hence is a bad character. Remove 00 from the bytearray and continue the process all over again to find the next bad character after comparison. The bad characters for this application are 00, 0A and 0D.

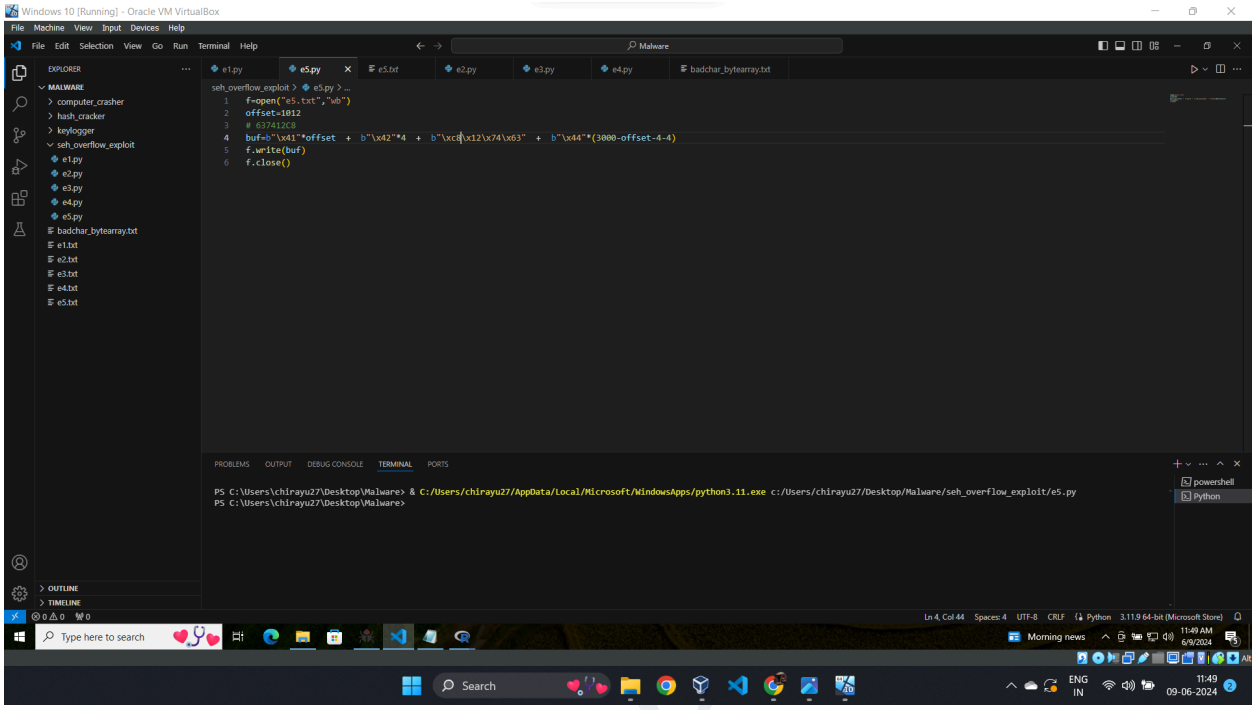


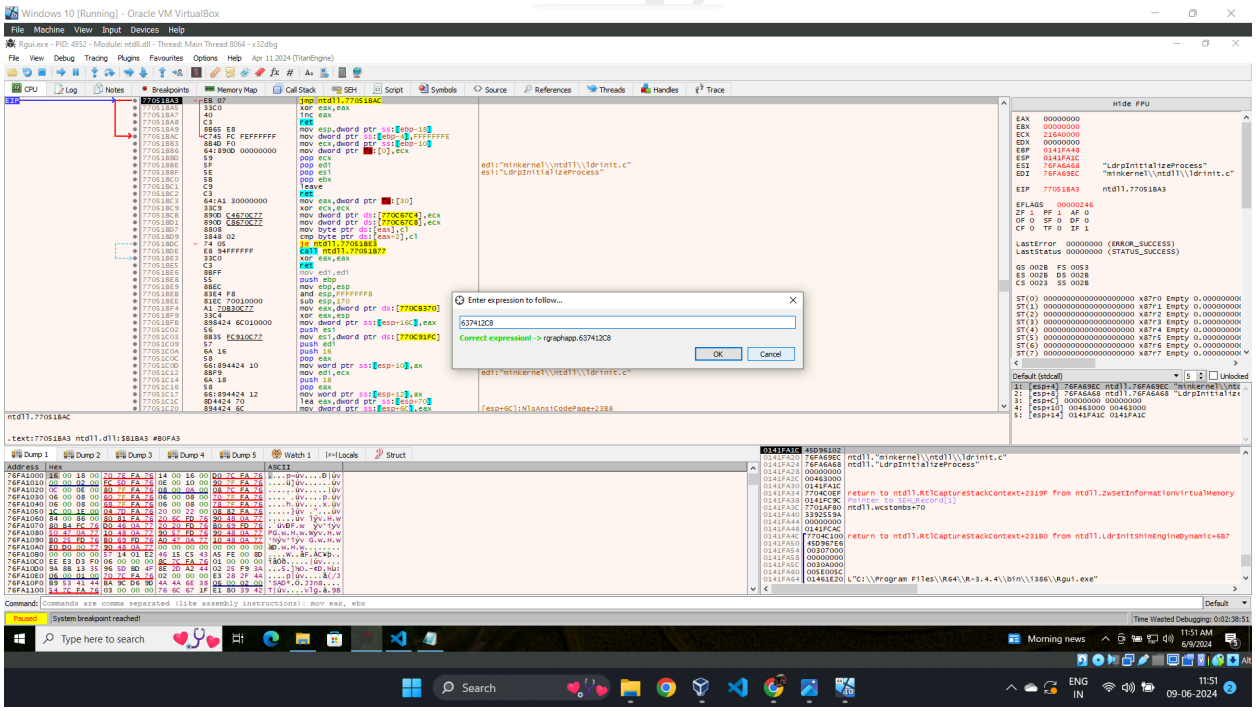
The next step in this exploit is to make the exception handler point to the next address. This can be done by inserting the address of a POP POP RET instruction into the handler. To find a POP POP RET instruction address execute the following command:

ERC --SEH



As we can see, the log section gives a list of such instruction addresses. Select an instruction address that is associated with the dll of the application and has all the security features disabled, i.e. False. Here, we have selected the address *637412C8*.

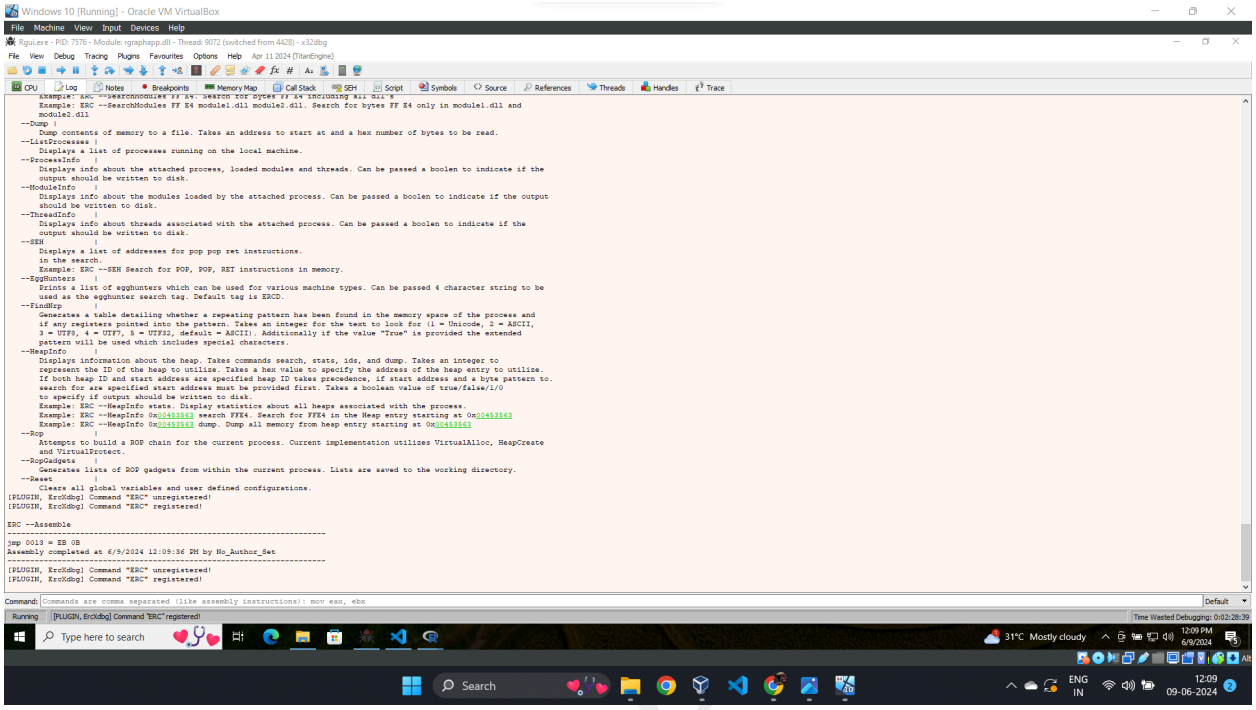




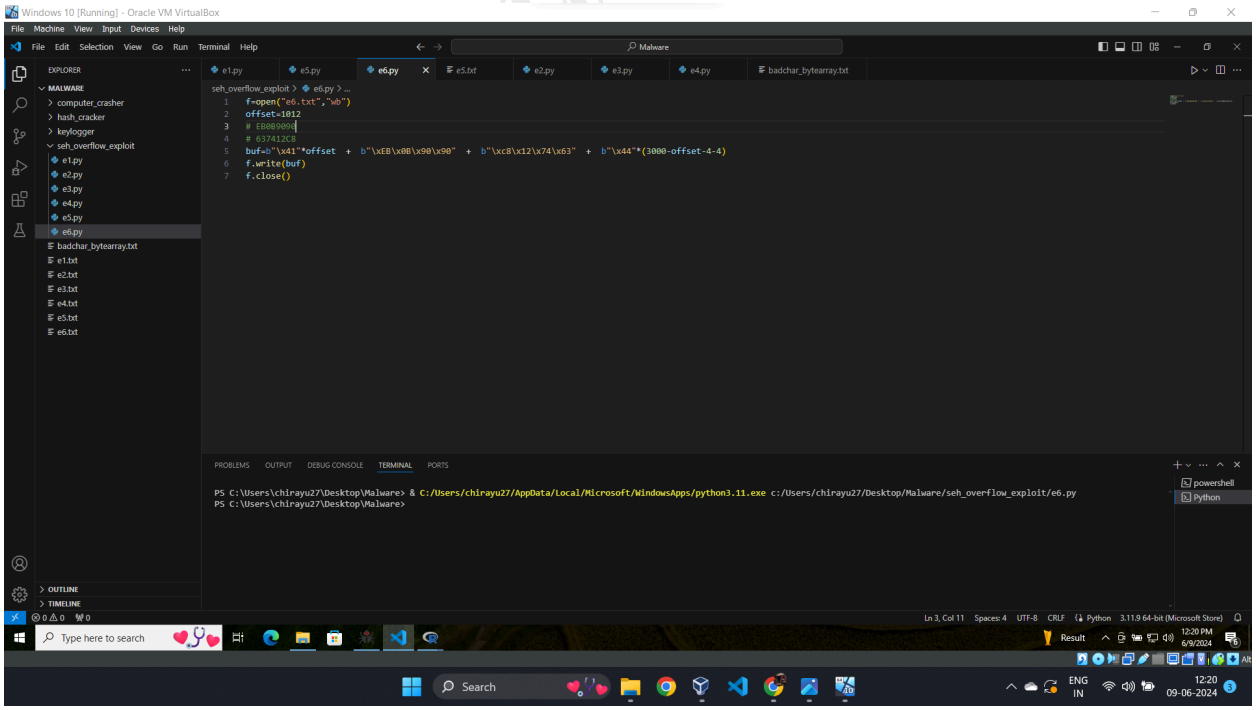
In the python code, replace the ‘C’s with the POP POP RET address. Copy the text generated and create a crash after inserting a breakpoint at the above address.

As our handler now points to the set of ‘B’s, we have to replace the ‘B’s with a short jump, so that the EIP lands into our memory location and then can execute our malicious payload. To get the jump address, execute the following command:

ERC --Assemble jmp 0013

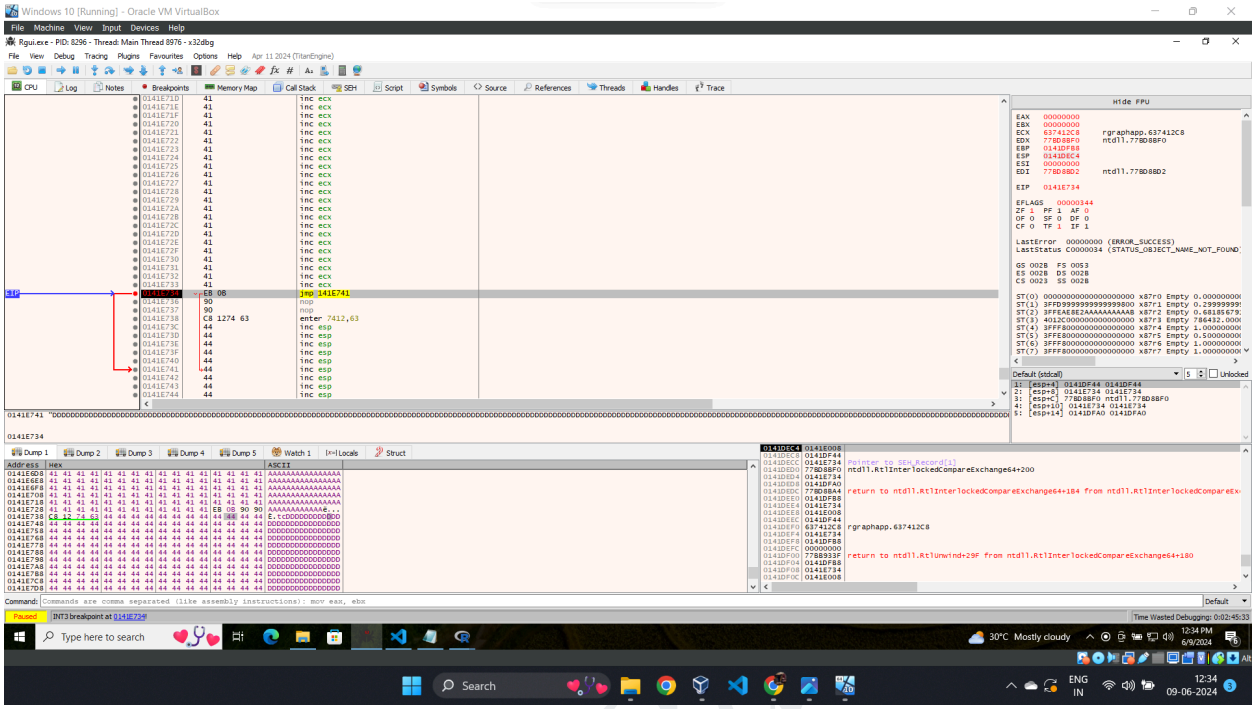


As we can see, the ERC has provided us with the jump address EB0B. Now, in the python code, replace the ‘B’s with the jump address ‘\xEB\x0B\x90\x90’.



Note that to complete the set of 4 bits, we have to add two NOPs.

Run the code and copy the text contents to crash the application.



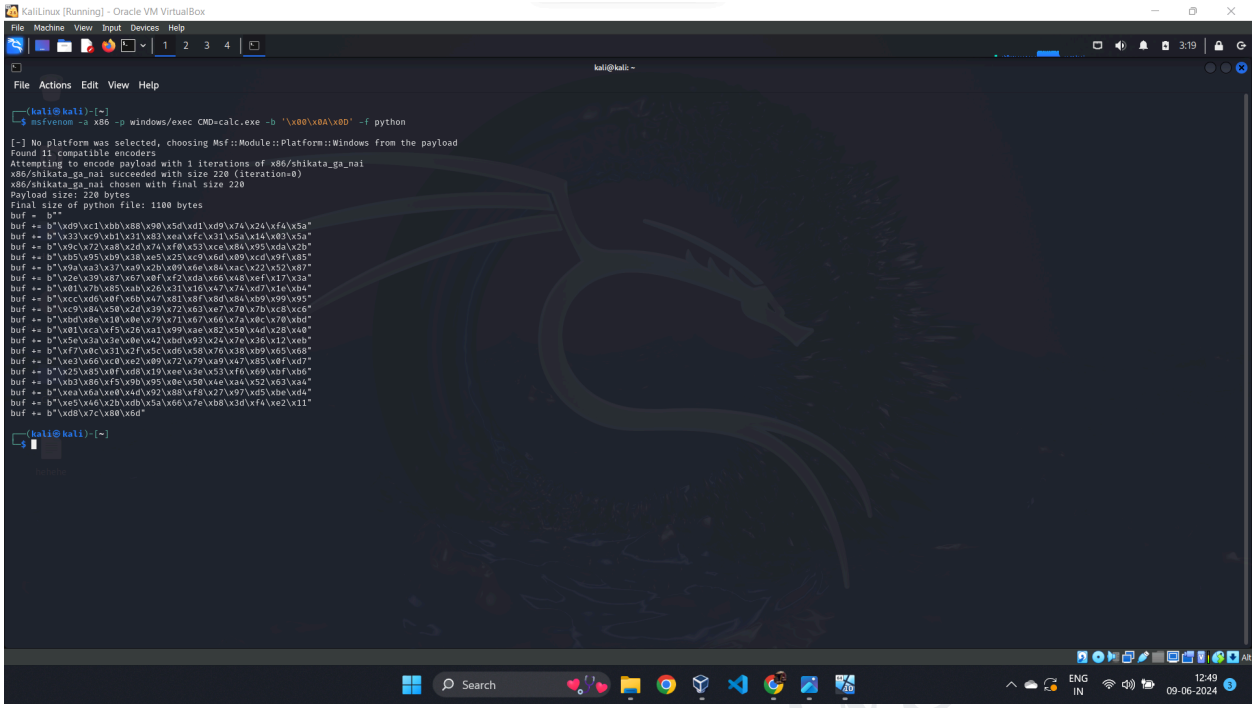
As we can see, the jmp instruction jumps the next few instructions and lands into the memory buffer directly where our ‘D’s are located with hex \x44.

Our exploit is at the last stage. Now we have to replace our ‘D’s with the payload we want to execute.

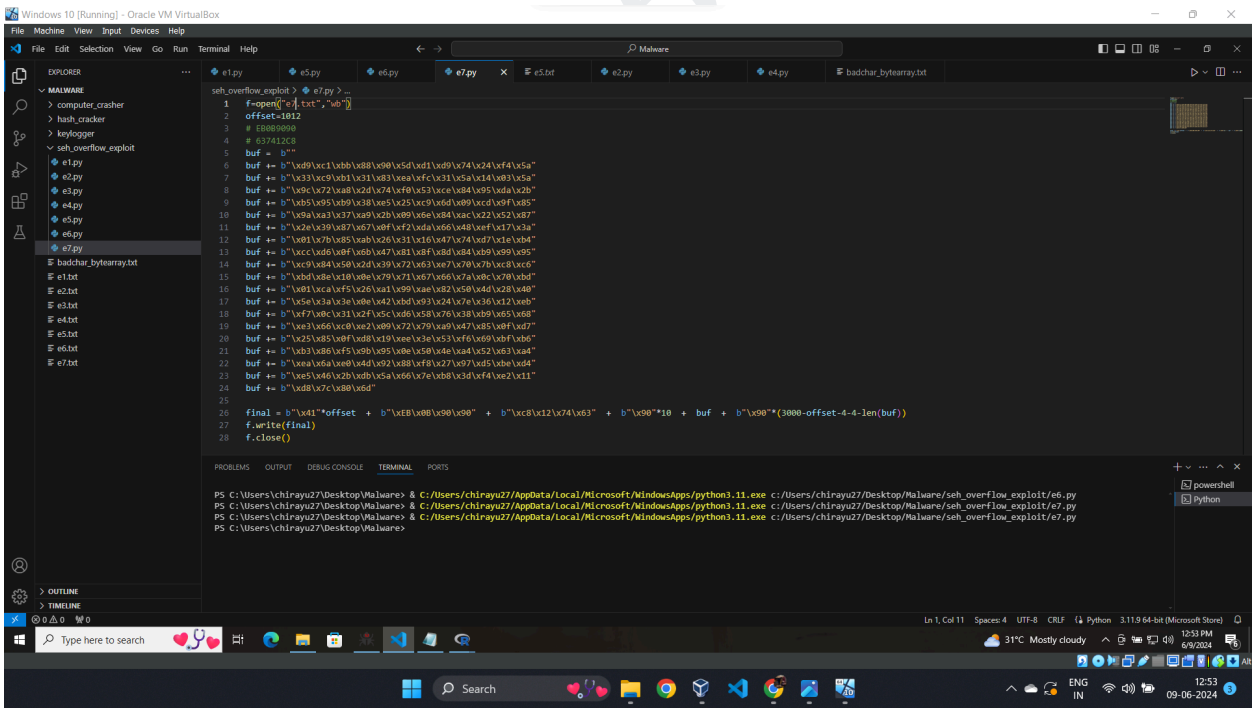
To generate payload, execute the following command in the terminal of your Kali or Parrot VM:

msfvemon -a x86 -p windows/exec CMD=calc.exe -b ‘\x00\x0A\x0D’ -f python

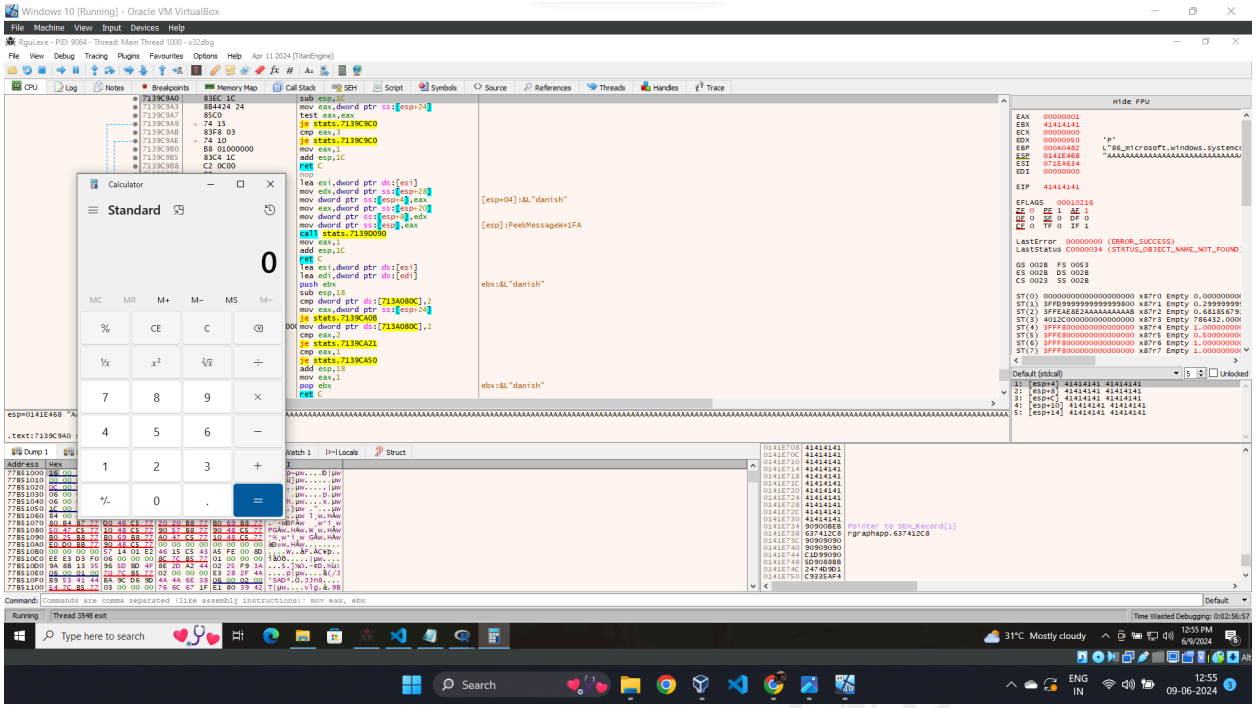
This command gives us a payload to start the Calculator application on triggering of the exploit. Copy the payload and paste it in the python exploit replacing the ‘D’s. as the jump instruction points to the memory location after skipping some instructions, do not forget to add a NOP sled before the payload.



Our final exploit looks as follows:



Execute the code and follow the same procedure to crash the application. Once the application crashes, navigate through the breakpoint and run the paused application. The Calculator application GUI appears on the screen.



Thus we have successfully exploited an SEH vulnerability in the R application.

Impact of SEH Vulnerability:

1. Data security reduced
2. Manipulation of file system
3. Denial of service
4. Financial loss/ransom
5. System damage

Potential Risks of SEH Vulnerability:

1. Attackers can execute arbitrary code.
2. Administrative privileges can be gained
3. Unauthorized data access
4. Can enter into organizational network

Mitigation Strategies:

1. Regular updates and software patches
2. Use SafeSEH
3. Application testing and monitoring
4. User education

Conclusion: The pointers pointing to the SEH stack can be overwritten to direct the flow of the code in a desired direction. SEH overflow vulnerability is restricted to 32bit applications. If exploited properly, this vulnerability is very dangerous for computer systems and data.