# INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 57- KERNEL

Contents

[INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 57- KERNEL 1](#_Toc40963026)

[HACKSYS – STACK OVERFLOW. 1](#_Toc40963027)

## HACKSYS – STACK OVERFLOW.

Bueno trataremos de explotar el stack overflow desde un script de Python, normalmente la explotación de kernel es local, o sea que uno ya exploto algún programa que no tiene privilegios de sistema y quiere escalar privilegios y ser SYSTEM lo cual nos quita las restricciones que tiene una explotación de un proceso que solo tiene limitados privilegios de usuario normal de la maquina.

Así que salvo muy raras excepciones los exploits de kernel son escalaciones de privilegios o priviledge escalation o como se diga jeje.

Por eso muchas veces vamos a ver el código de los mismos en un ejecutable compilado, o el código fuente del mismo, porque se supone que podemos bajarnos un archivo y ejecutarlo con permiso de usuario normal, ese ejecutable atacara en este caso nuestro driver y lo explotara consiguiendo la escalacion.

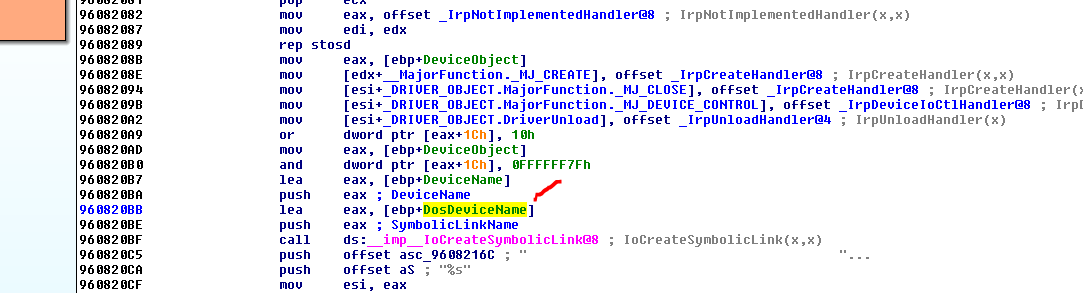
De cualquier manera tanto el código en C como en Python, están basados en los llamados a las mismas apis de Windows, CreateFile, DeviceIoControl, etc, asi que lo que se hace en uno es fácilmente portable al otro.

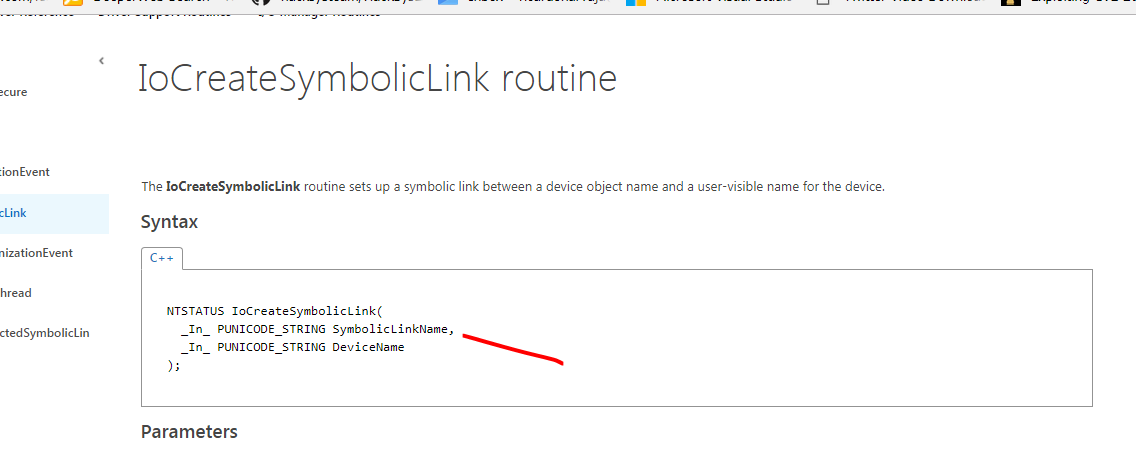
Recordemos nuestro modelo en Python del ejemplo anterior.



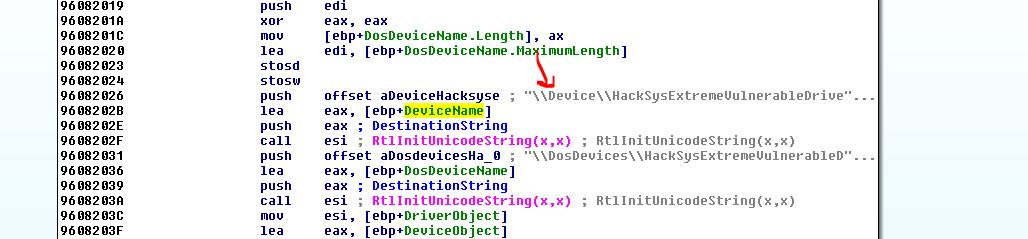
El primer paso sera cambiarle el nombre al driver para cuando haga CreateFile nos devuelva un handle correcto al mismo.

Recordemos que el nombre se genera en

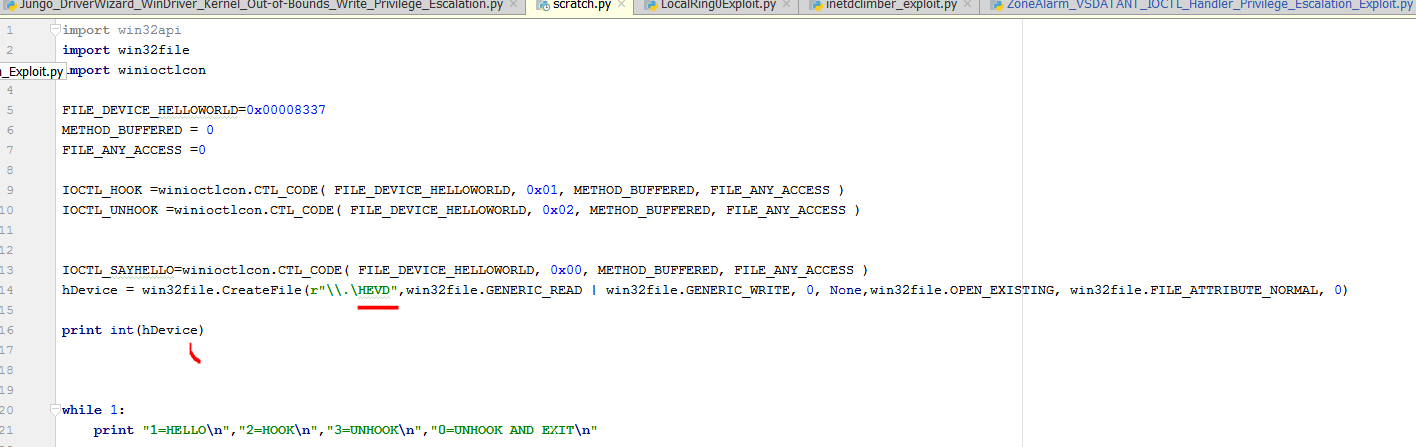




Alli devuelve el SymbolicName que viene de la string que esta en DeviceName.

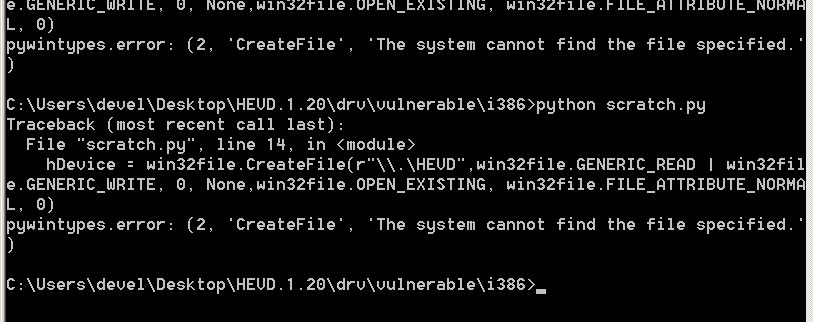


Sin mucho problema seguramente si reemplazo el nombre del viejo driver por el del nuevo funcionara, sin amargarme mucho probemos.

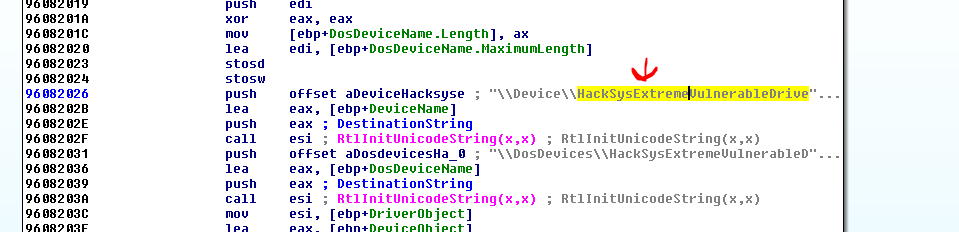


Podemos copiarlo a la maquina target y ver si me da error o un handle valido, sino seguiré mirando.

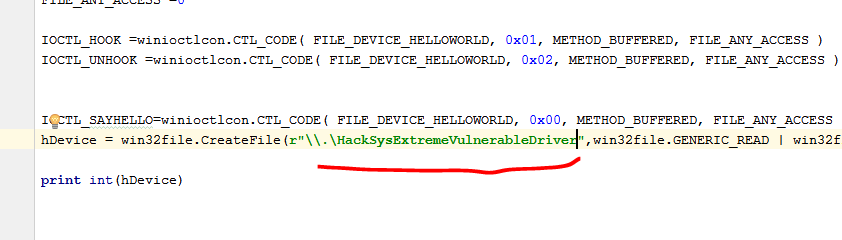
Esta el driver corriendo y arranco el script



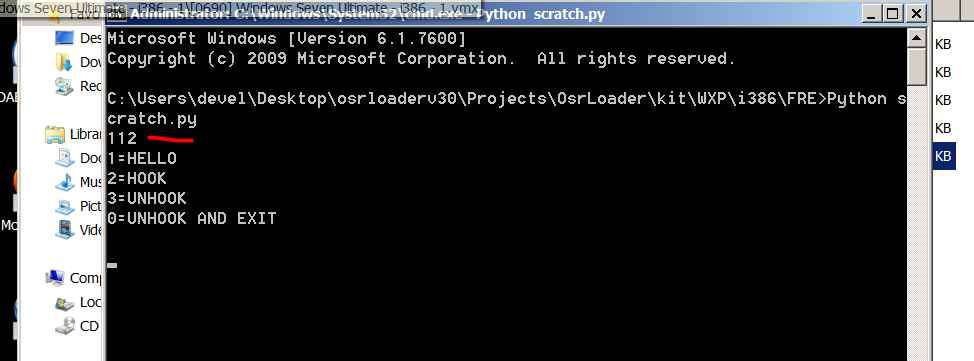
Como el SymbolicLink sale de acá tiene sentido que se use ese nombre



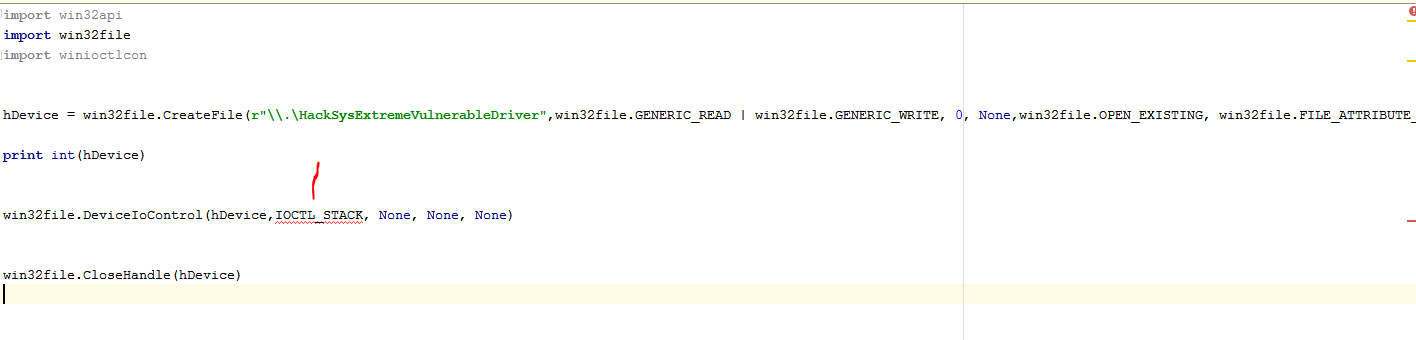
Probemoslo.



Vemos que ese funciono

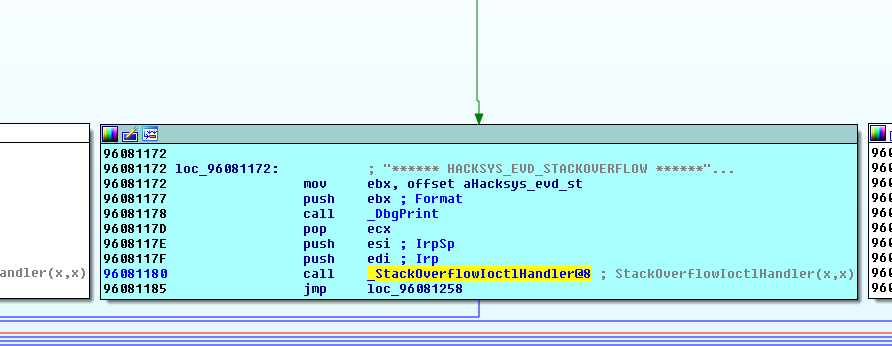


Me devolvió un valor positivo que es el handle al driver el resto no me interesa asi que lo cierro.

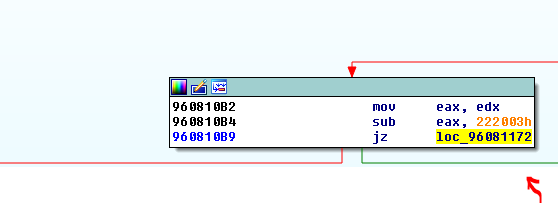


Por ahora le quitamos el while y todo el resto que no nos interesa y lo siguiente es ver cual era el IOCTL que nos lleva al bloque del stack overflow, para enviárselo.

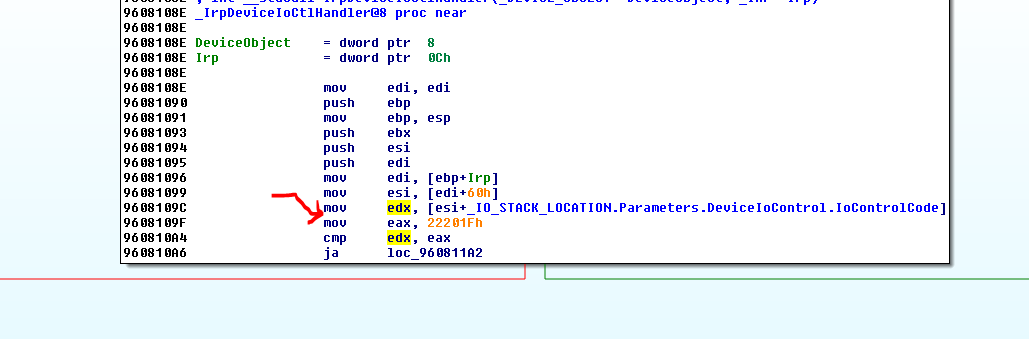
La cuestión es llegar alli



Vemos que viene de acá

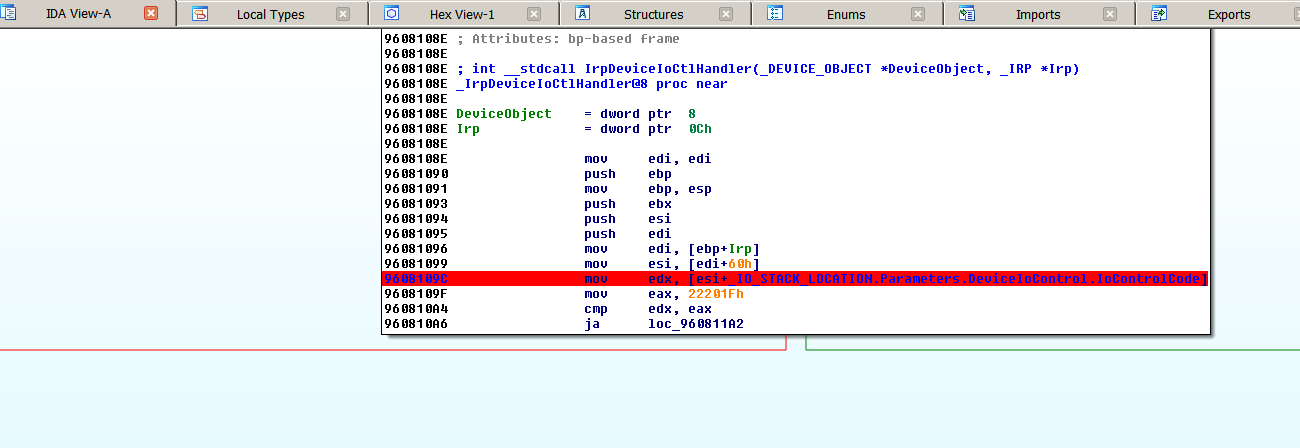


EDX tiene el IOCTL que salia de aca

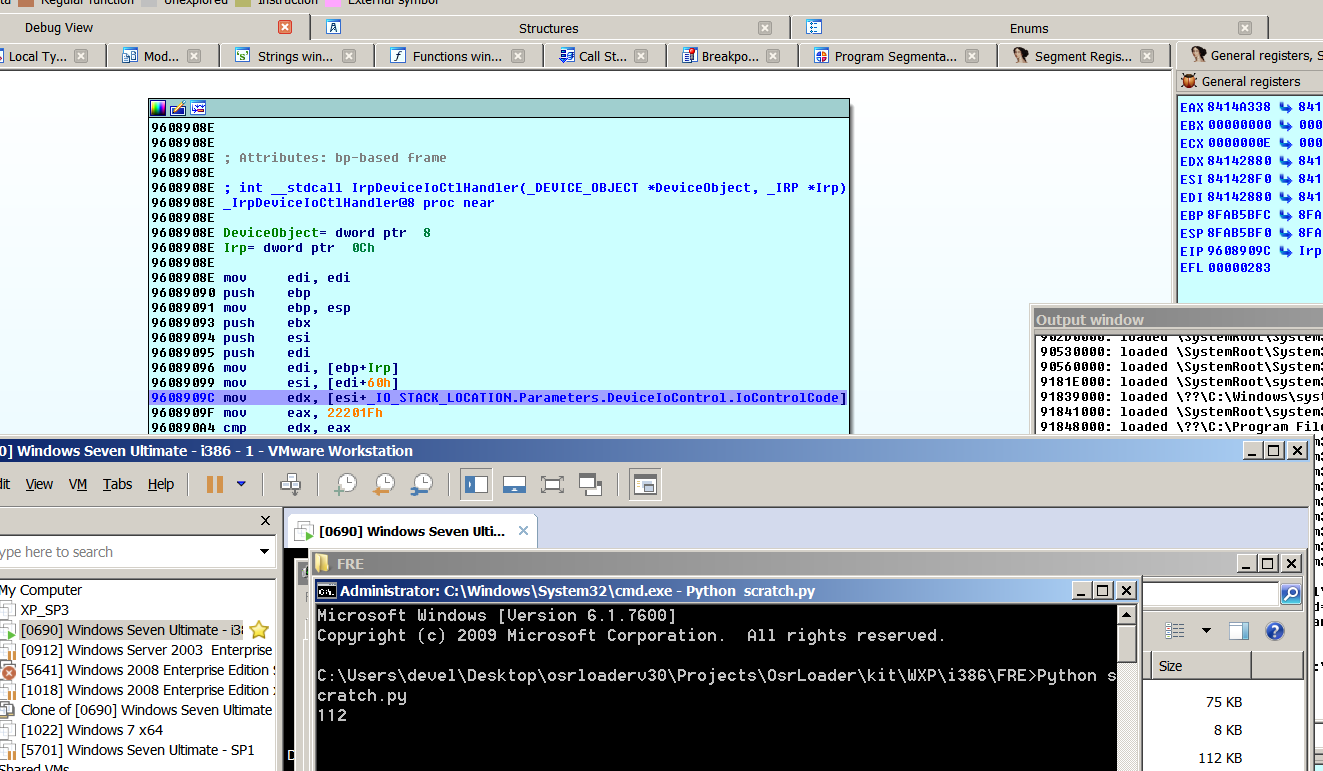


Lo resta con 0x222003 y si da cero va a la parte que necesitamos, probemos pongamos este IOCTL.

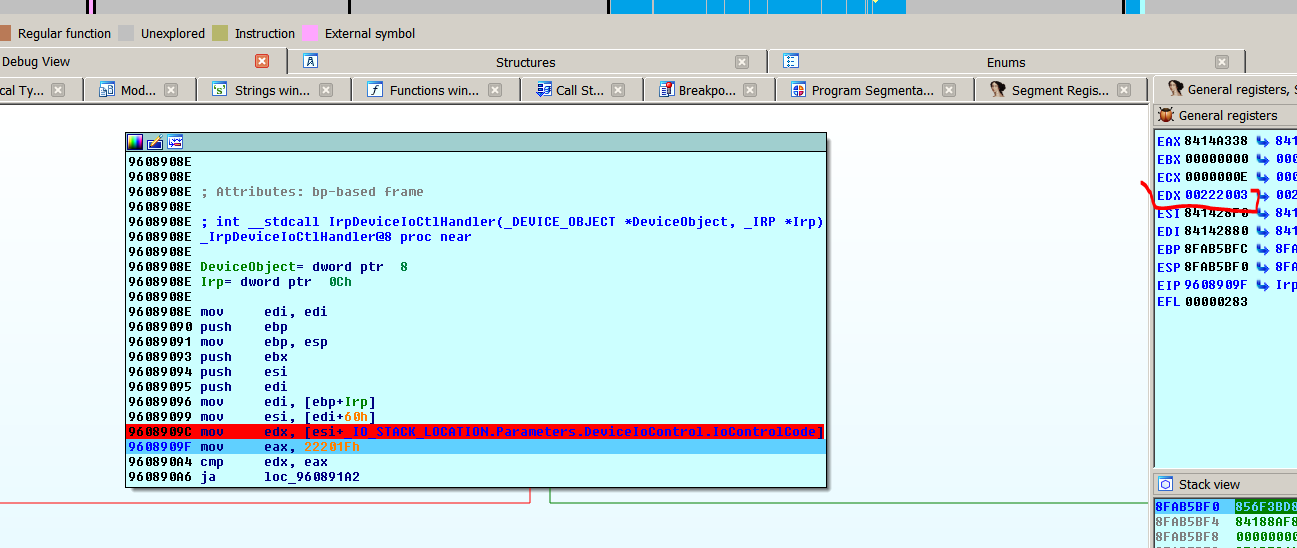
Veamos si llega al bloque que queremos atacheamos el IDA y pongamos un breakpoint.

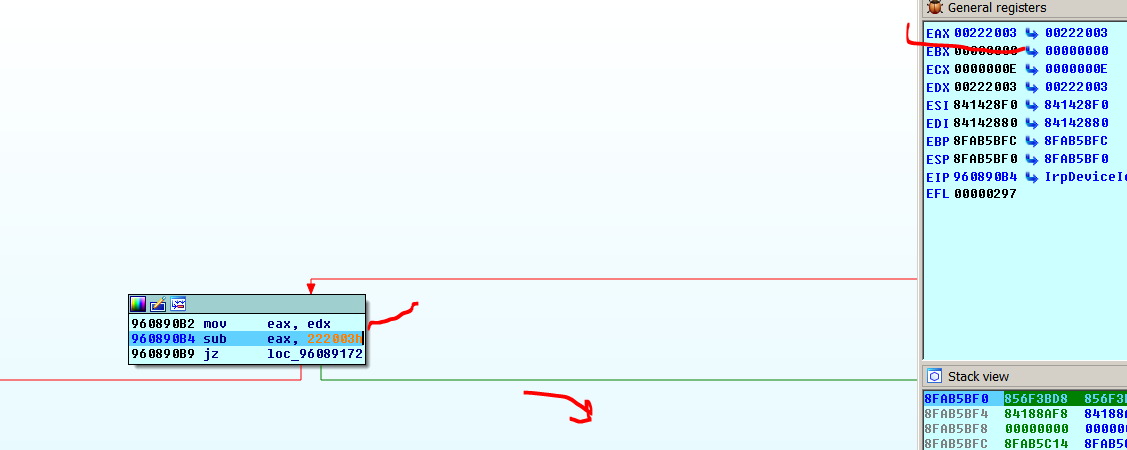


Apenas lo ejecuto en la maquina target para en el breakpoint

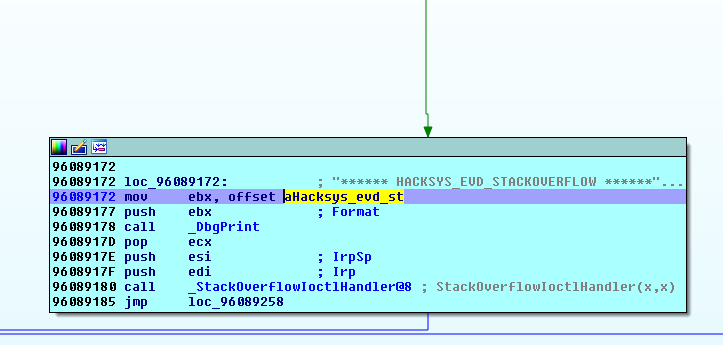


Vemos que en EDX lee el IOCTL que le pase.



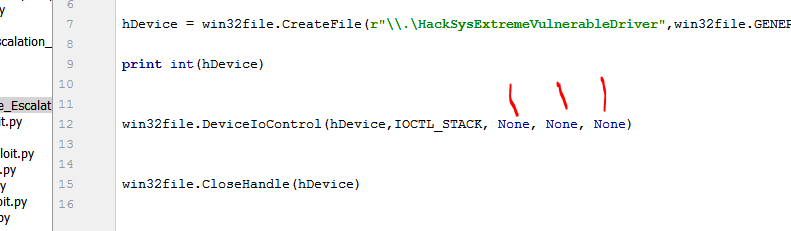


Lo mueve a EAX le resta 0x222003 y como da cero va al bloque vulnerable.

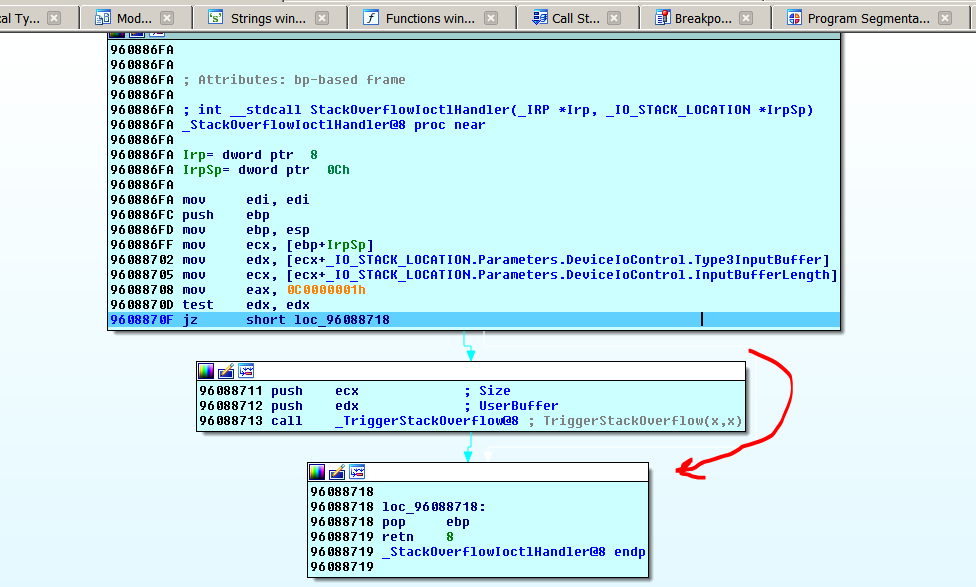




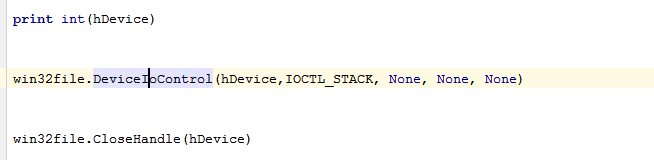
Lee el largo que es cero ya que no le pase argumentos aun salvo el IOCTL.

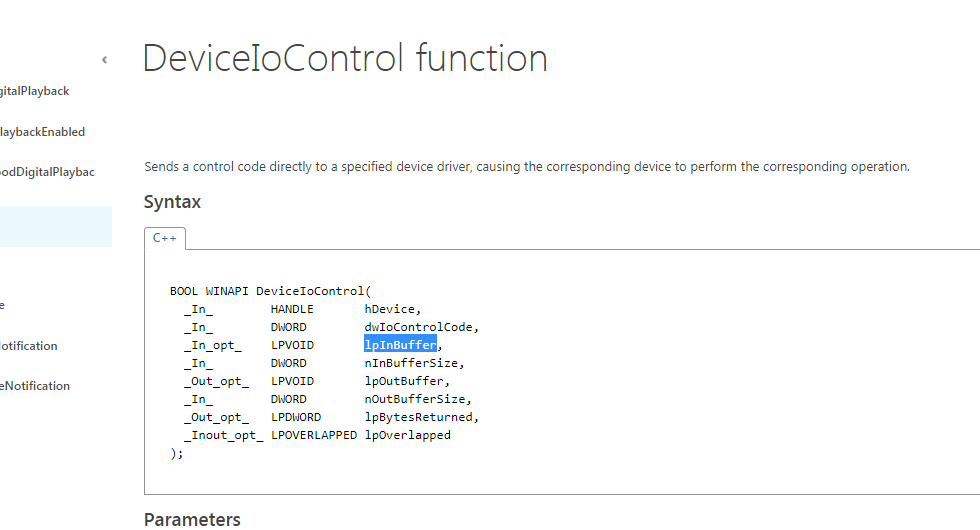


Como es cero saltea la función donde se triggerea el stack overflow.

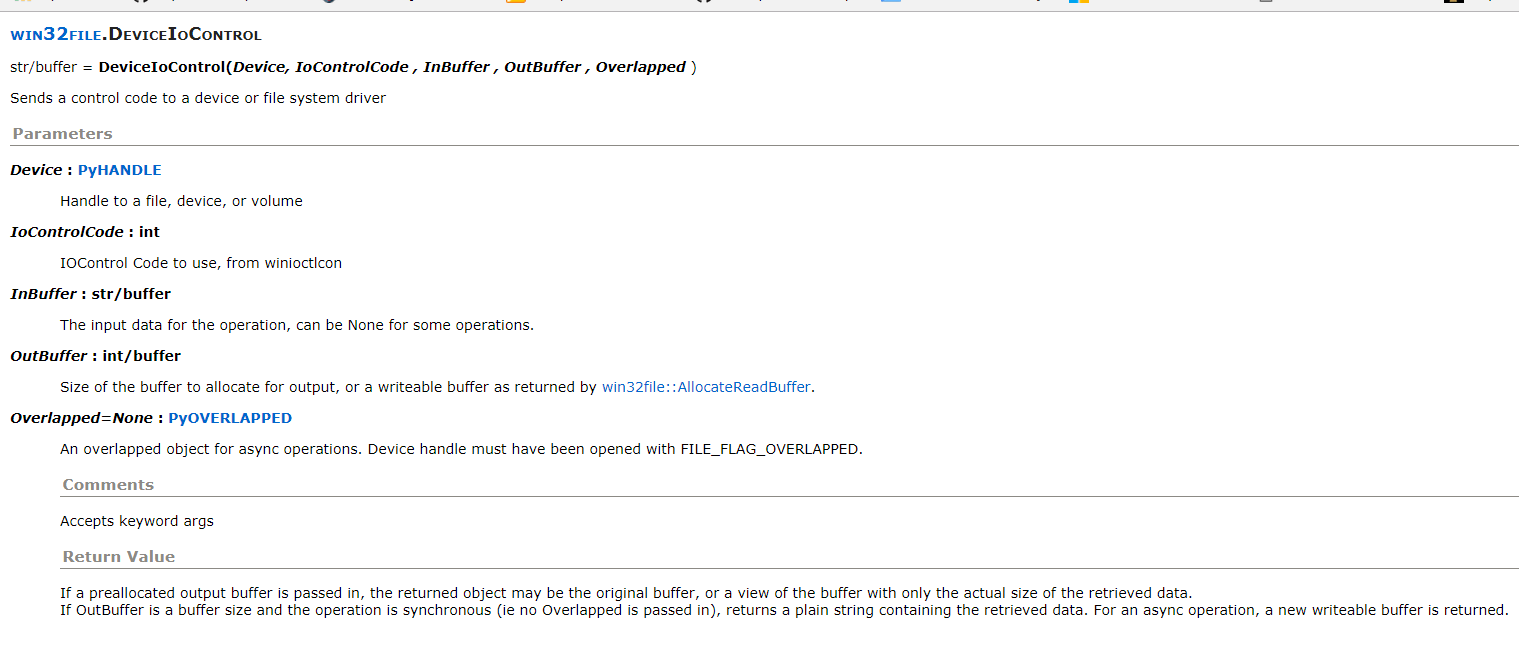


Como la api llamada desde win32file en Python tiene menos argumentos que la original que tiene mas.

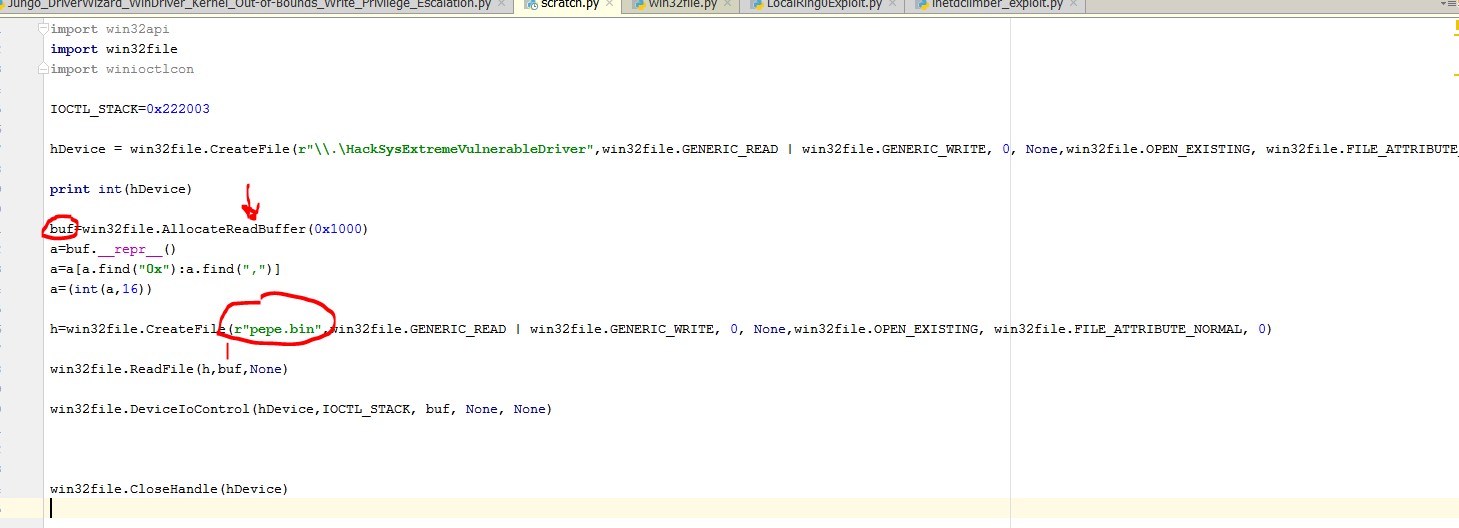




Acá esta la definición de la de win32file de Python



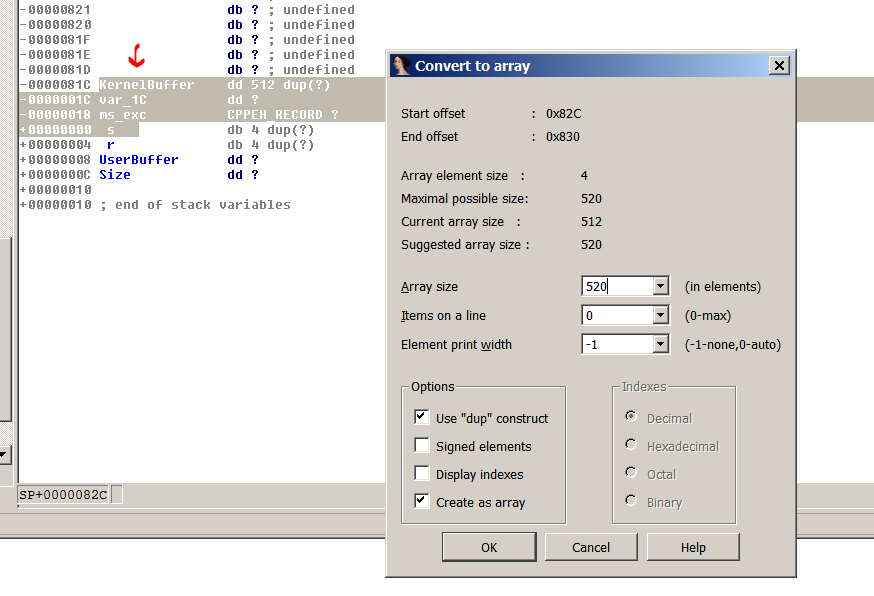
Veremos si con esto nos alcanza necesitamos hacer un buffer para pasárselo a la entrada, asi que como necesitamos un puntero al mismo, lo podemos hacer con AllocateReadBuffer, que nos alocara en el heap la cantidad que necesitamos para pasarle el buffer de entrada.(otra opción win32file no nos da)



Como win32file no tiene acceso a todas las apis no permite copiar directamente al buffer como memcpy o cosas asi de esta forma podemos solo copiarlo con ReadFile, asi que hacemos un archivo pepe.bin lleno de 0x1000 Aes, lo pasamos a CreateFile para que lo abra y nos devuelva el handle y eso lo pasamos a ReadFile con el argumento buf del buffer que allocamos y copiara el contenido del archivo alli.

Obviamente si lo hacemos en C, C++ o el lenguaje que sea podremos usar VirtualAlloc, y copiar fácilmente con memcpy o lo que queramos, pero aquí tenemos ciertas limitaciones y nos tenemos que adaptar.

Necesitamos saber ademas el largo justo de lo que debemos enviar, miremos el buffer de destino en el IDA.



Marcamos desde el inicio del buffer hasta justo antes del return address y nos de 520 decimal por 4 del element size.

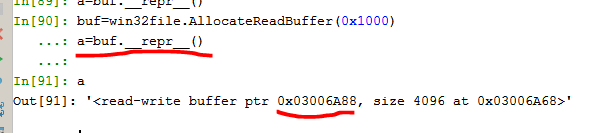
Asi que:

hex(520\*4)

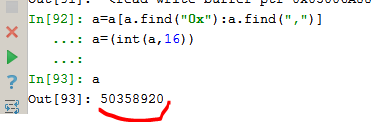
'0x820'

Asi que deberemos enviar 0x820 + ret

Ahora como sabemos en Python la direccion del buffer que creamos para pasarle, bueno un truco medio sucio es usar repr.



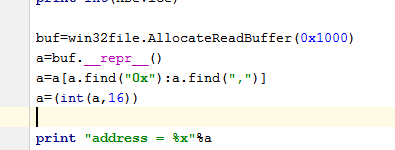
Vemos que me devuelve la direccion donde puedo escribir, en una string asi que busco 0x y busco la coma en dicha string y puedo stripear la direccion.



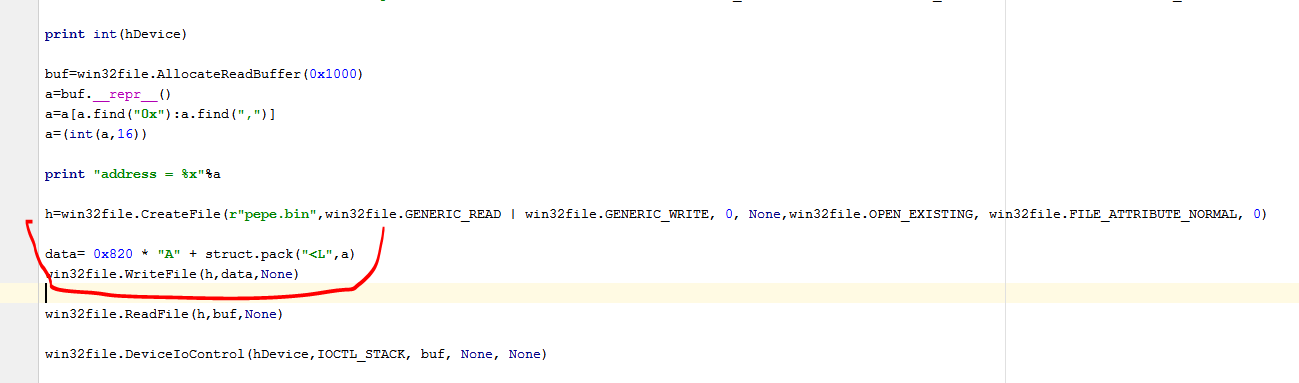


Alli esta la direccion la podemos pasar con struct.pack luego de los 0x820 Aes, lo molesto es que debemos escribirlo en el archivo que leerá, uf.

Por lo tanto una vez que tengo la direccion del buffer

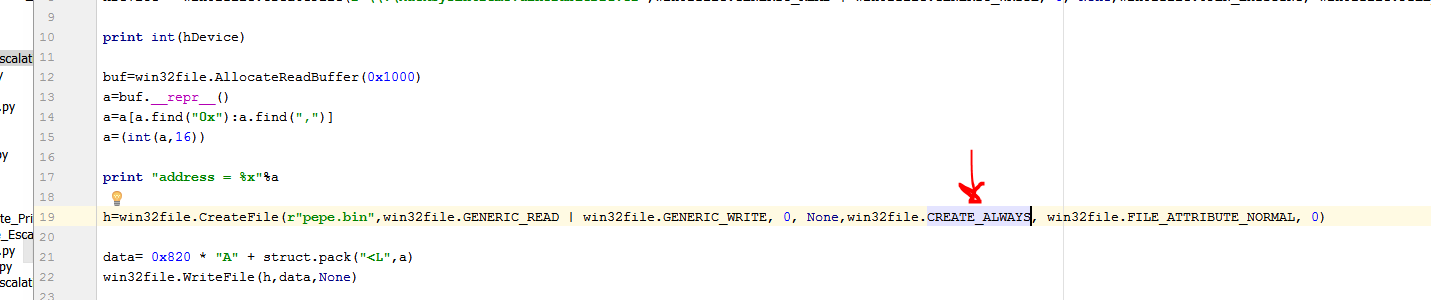


Escribo en el archivo la fruta que quiero enviar

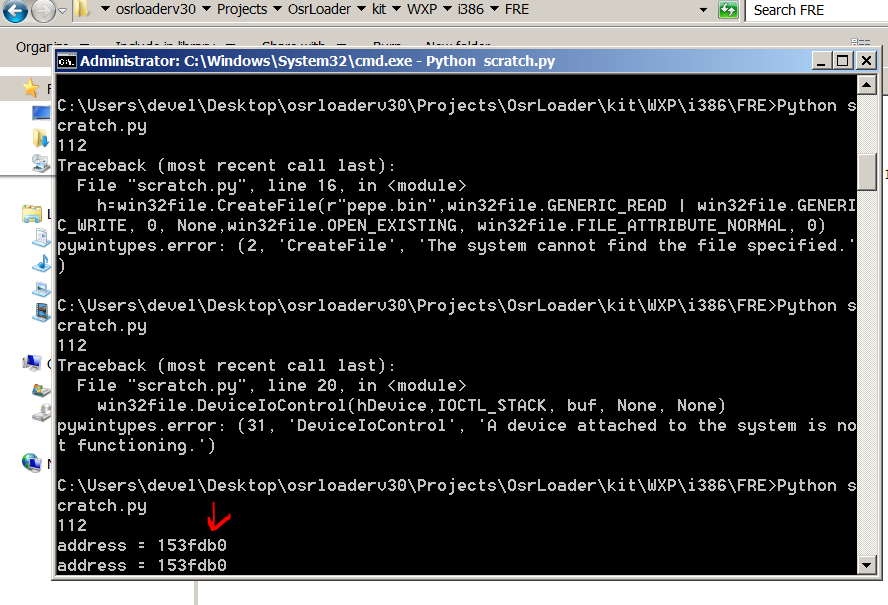


Antes de Con ReadFile copiarla al buffer.

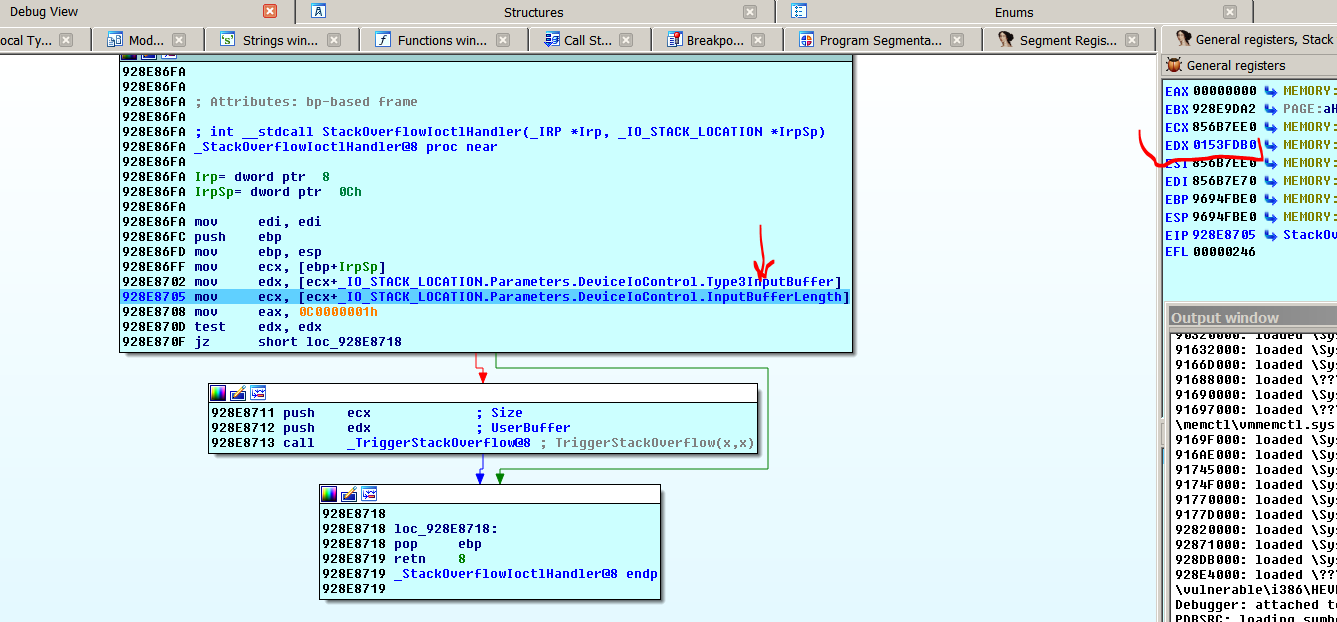
Veamos si sirve, en este caso no es necesario tener el archivo pues lo creara y lo llenara asi que borramos el anterior.



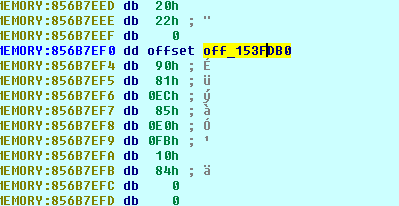
Para eso cambiamos el argumento a CREATE ALWAYS lo cual si no esta lo creara.

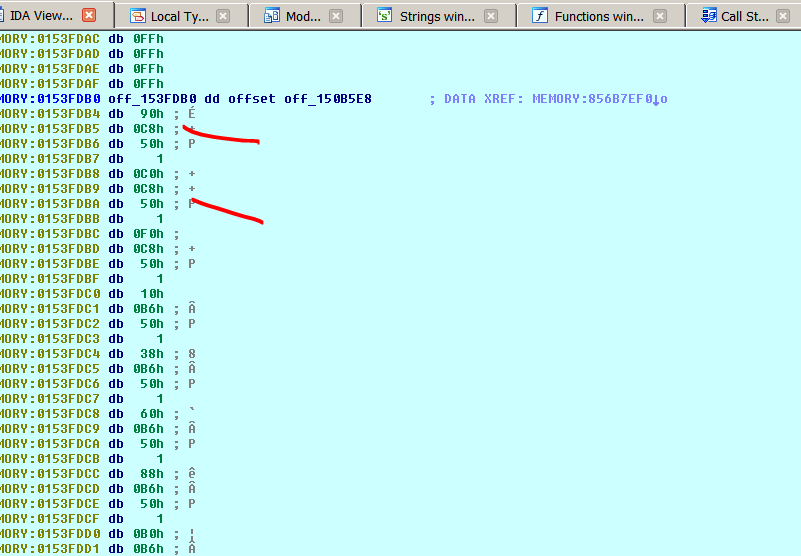


Bueno supuestamente ahí esta nuestro buffer con las Aes, el IDA paro, veamos que paso.

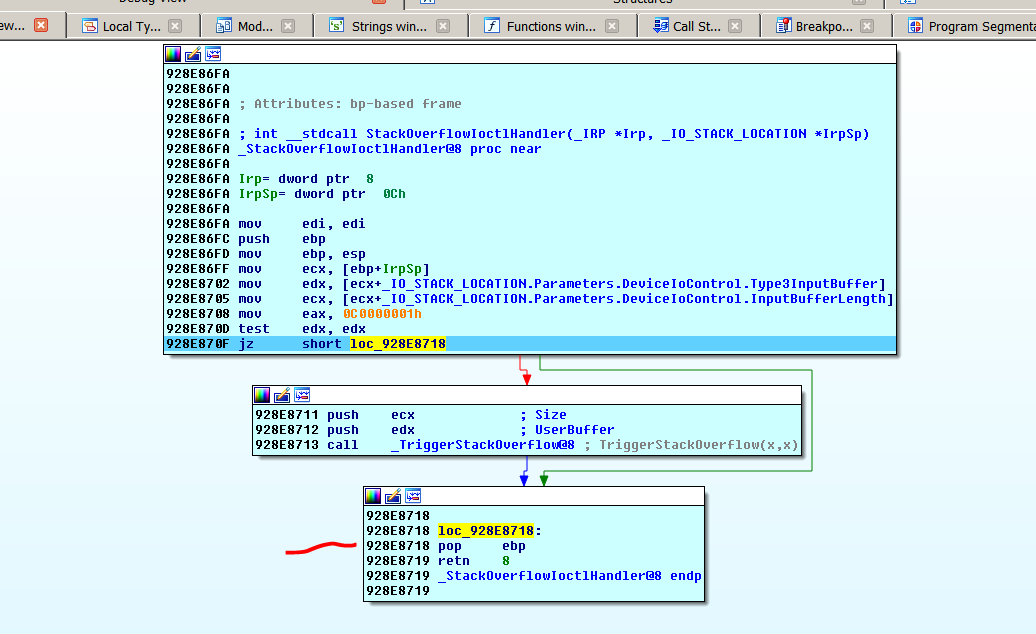


Vemos que el buffer de entrada que se pasa a EDX nos muestra el mismo valor, veamos si están las Aes.



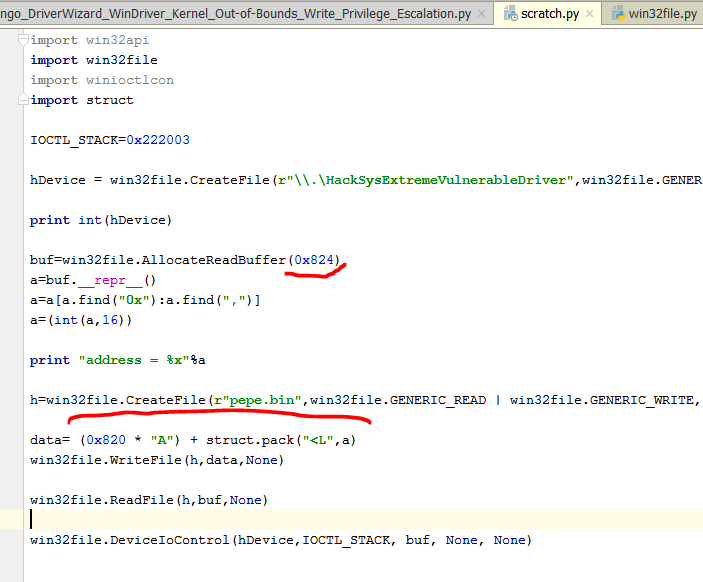


Uf no me puso las Aes me repito el puntero salteemos la explotación y arreglemos el script.



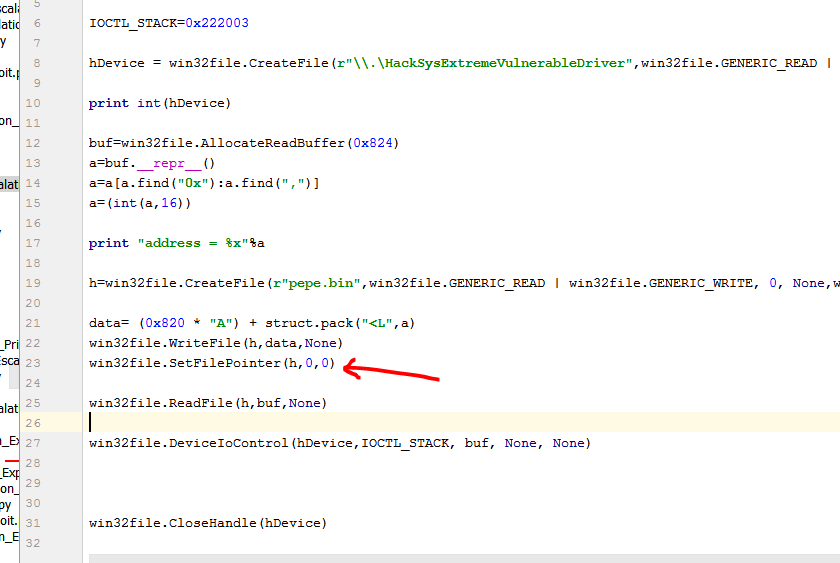
Cambio el EIP alli con click derecho-SET IP y doy RUN.

Creo que el problema es que debe coincidir el size del buffer con el size del archivo, justito jeje.

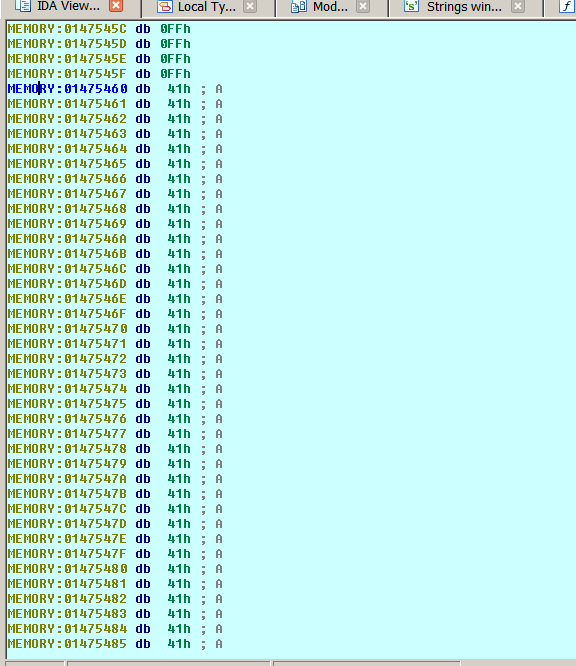


Veamos asi, no va

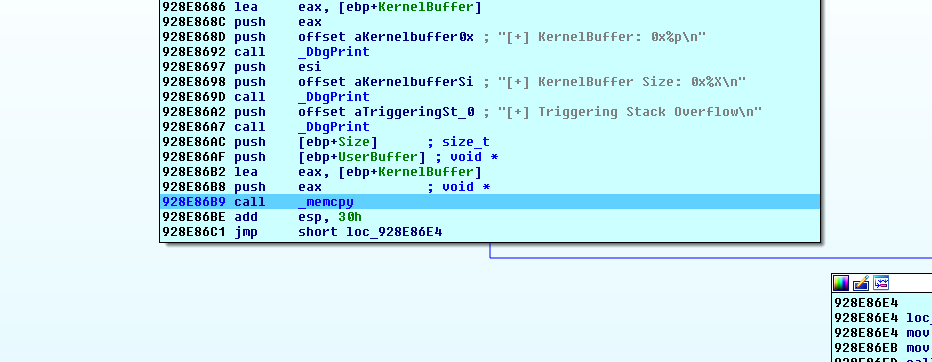
Ah ya caí



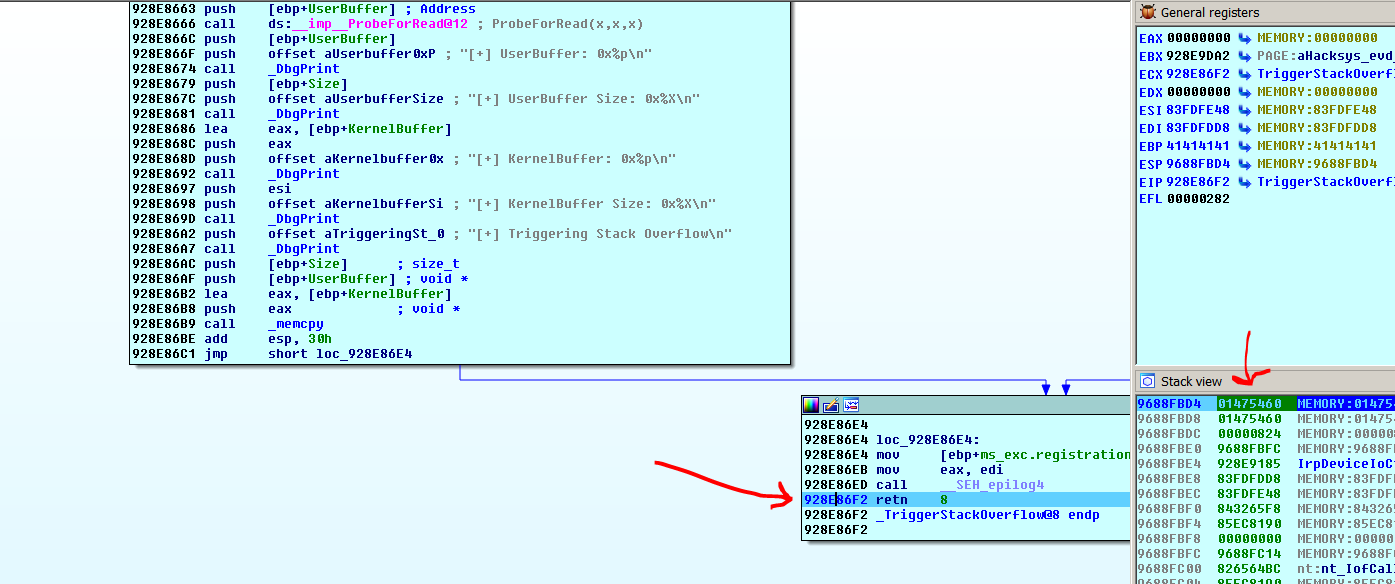
Le faltaba setear al inicio del archivo el file pointer, sino leerá desde el final donde lo dejo WriteFile, por eso no leía las Aes, ahora si.



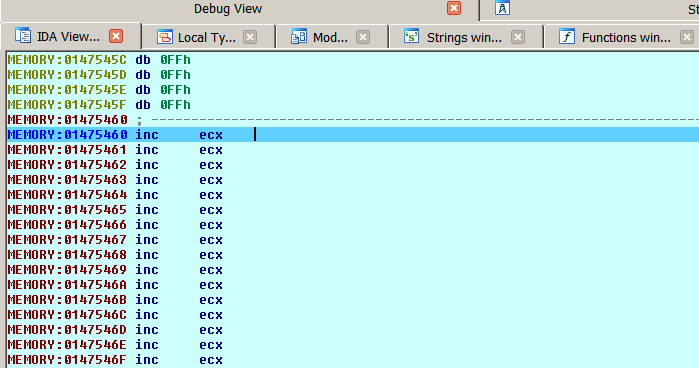
Traceo hasta el memcpy



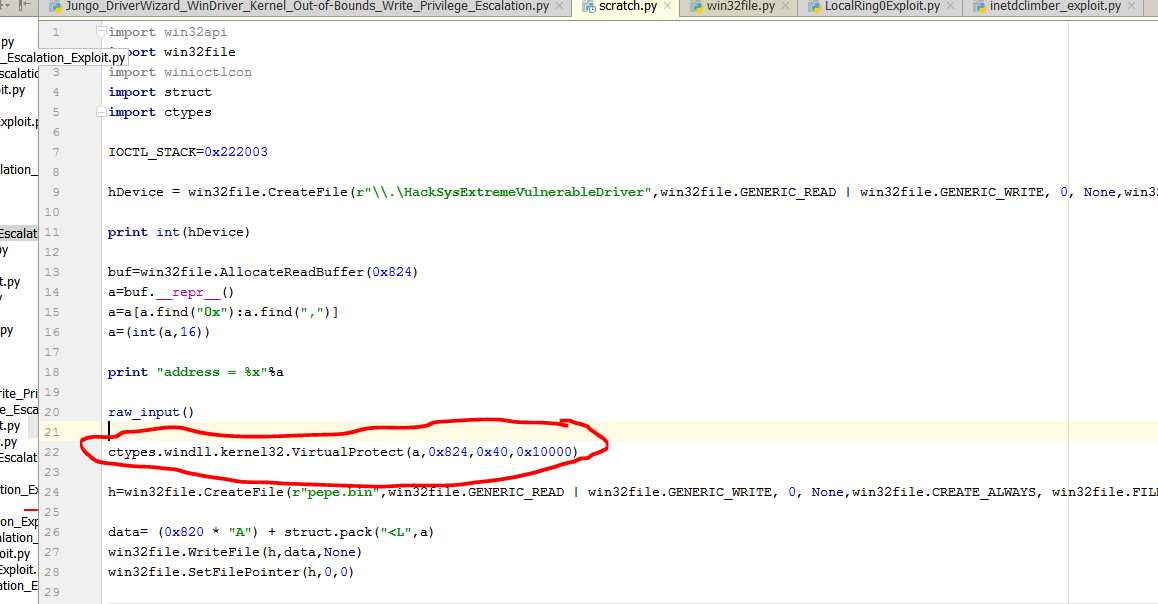
Llego hasta el ret.



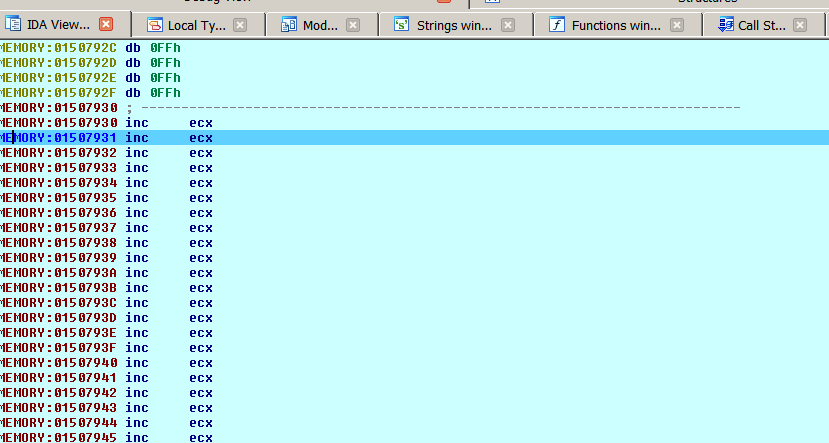
Sigo con F7.



Veo que llego al buffer el problema es que como no lo alloque con VirtualAlloc y lo alloque con la api esa de win32file AllocateReadBuffer no le da permiso de ejecución, asi que tendré que buscar la forma de allocar código ejecutable de alguna otra forma.

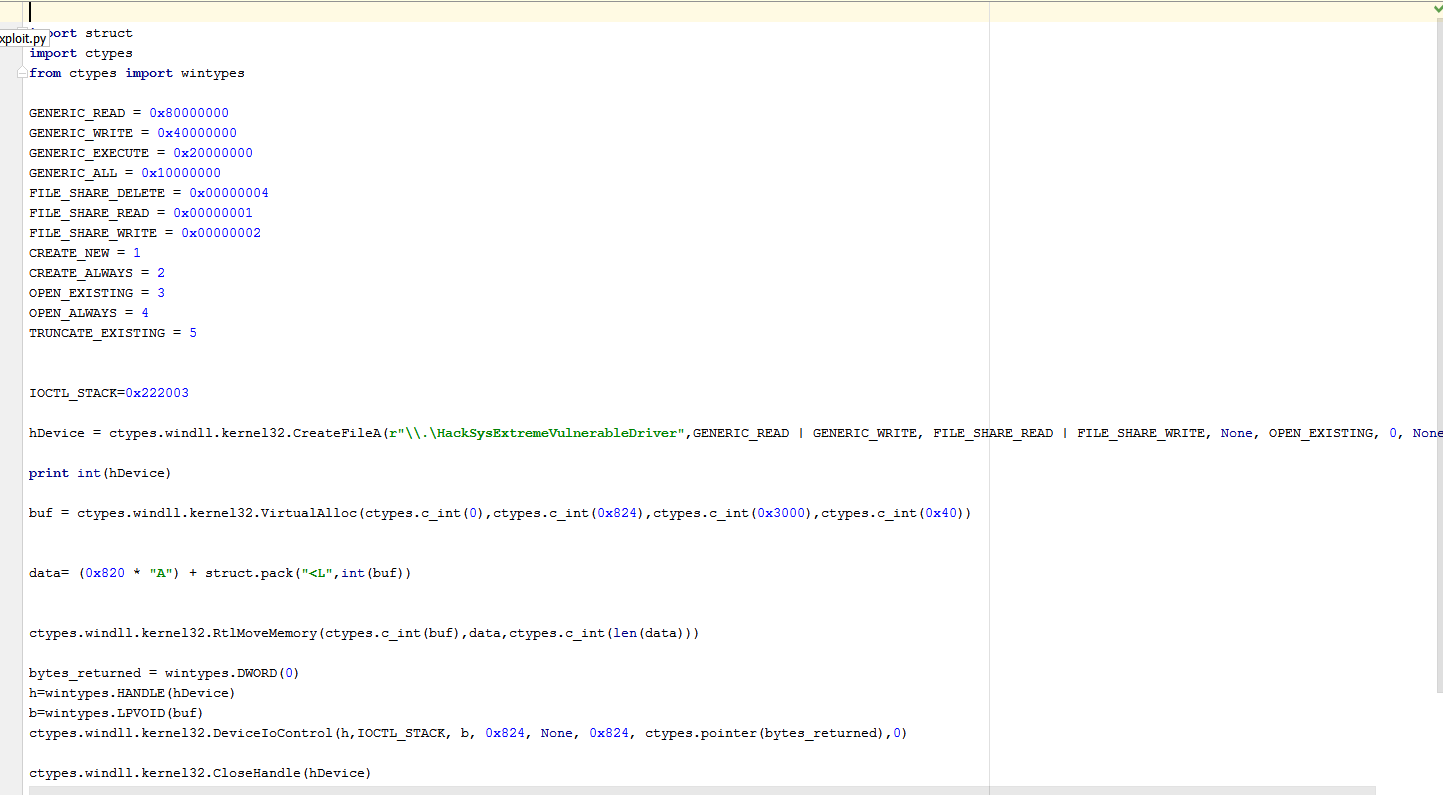


Le agregue el import ctypes y este tiene VirtualProtect asi que le di permiso de ejecución y ahora no hay problema.

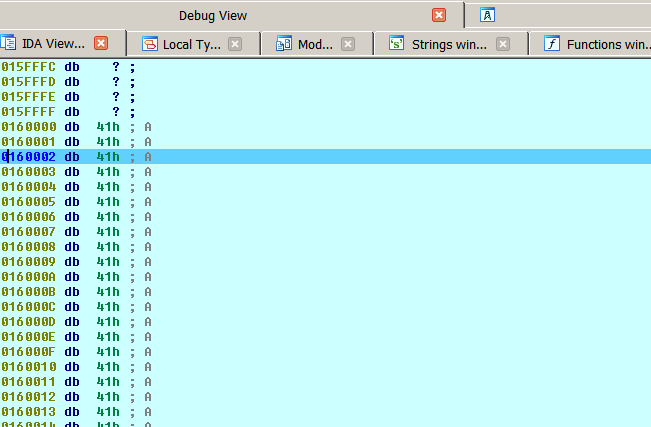


Salta y ejecuta sin problema, realmente veo que ctypes esta mas avanzado que win32api, por lo cual podría hacerse todo llamando directamente a VirtualAlloc de ctypes sin tanta vuelta.

La versión con ctypes es



Usa directamente CreateFile, VirtualAlloc, RtlMoveNemory y DeviceIoControl, solo hay que tener cuidado con alguno de los tipos, pero funciona bien.



Alli esta ejecutando, la cuestión ahora es que ejecutamos código, nos quedaría hacer el shellcode porque asi tendremos solo una bonita pantalla azul.

El shellcode lo analizaremos y armaremos en la parte siguiente.

Hasta la parte 48

Ricardo Narvaja