# INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 19.

Contents

[INTRODUCCIÓN AL REVERSING CON IDA PRO DESDE CERO PARTE 19. 1](#_Toc40949989)

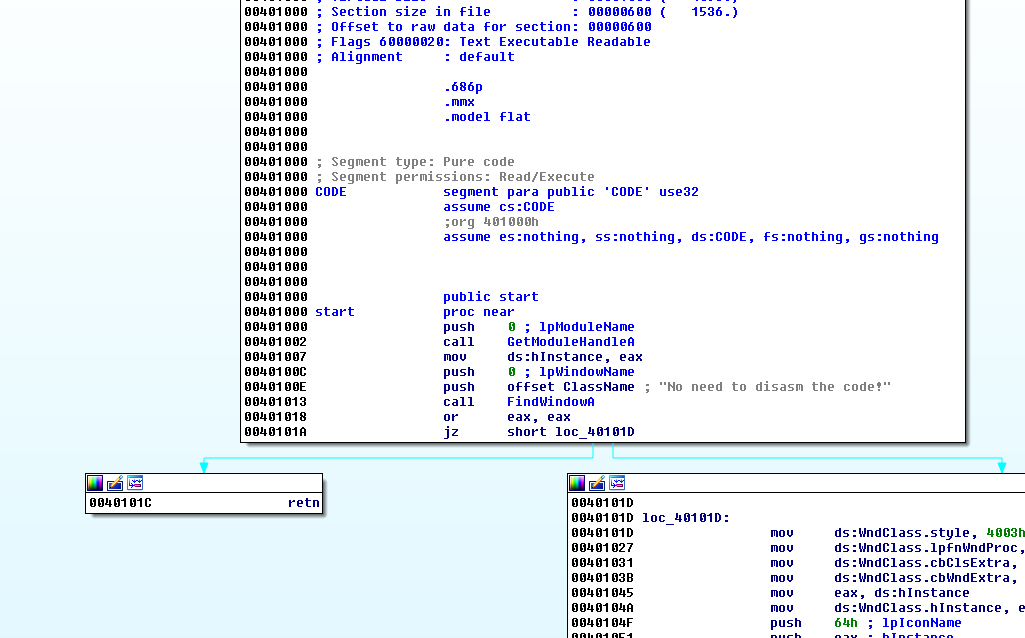
[REVERSEANDO EL CRACKME DE CRUEHEAD. 1](#_Toc40949990)

[KEYGEN CRACKME DE CRUEHEAD. 15](#_Toc40949991)

## 

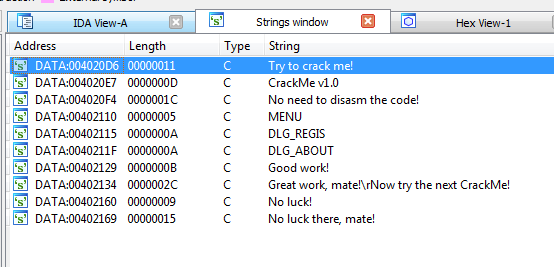
## REVERSEANDO EL CRACKME DE CRUEHEAD.

En este capítulo ya estamos en condiciones de reversear el crackme de Cruehead original y eso haremos, así que lo abrimos en el LOADER sin tildar MANUAL LOAD total es solo reversing el original no está empacado, así que no es necesario cargarlo manualmente.

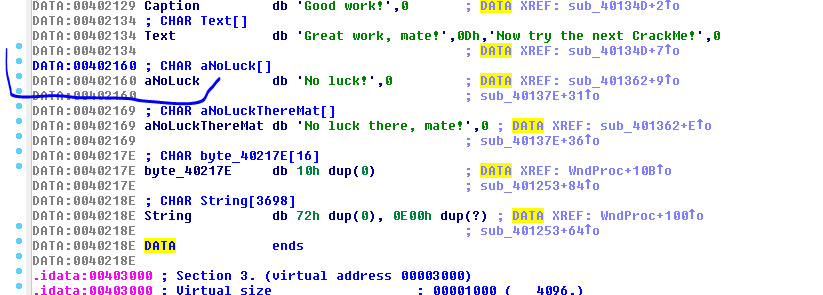


Allí se detuvo, en este caso al no ser una aplicación de consola, no es solo analizar la función main, sabemos que en aplicaciones con ventanas existe el LOOP de mensajes que van procesando lo que el usuario va interactuando con la ventana, los click que realiza etc. y según cada acción del usuario, está programado para ejecutar diferentes funciones con su código.

Sabemos que lo primero siempre es intentar por el lado de las strings, si esto falla se deben mirar las apis o funciones que utiliza el programa, en este caso las strings están bien visibles así que encararemos por este camino.

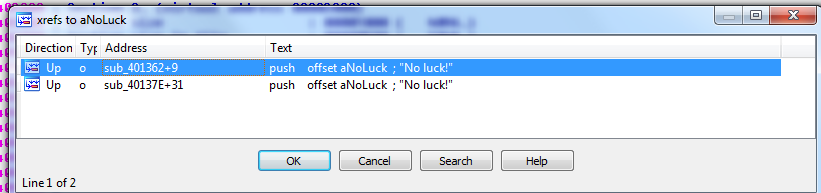


Siguiendo la string NO LUCK habíamos llegado a la zona donde tomaba la decisión, ya que haciendo doble click en dicha string vamos a

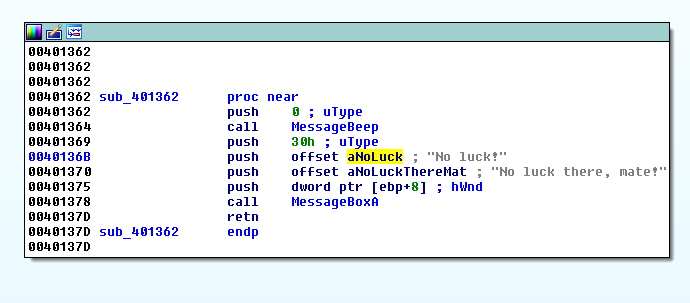


Viendo la referencia con X o CTRL +X

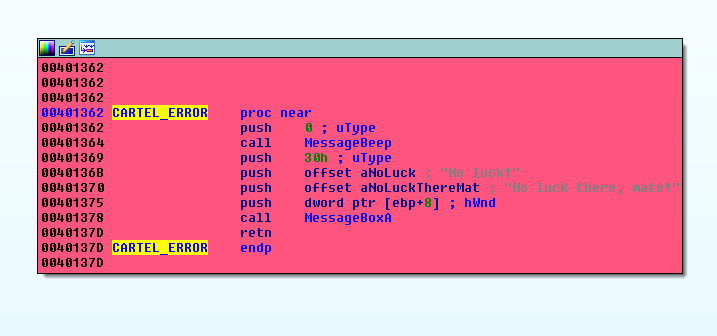
Vemos que hay dos referencias a dicha string.



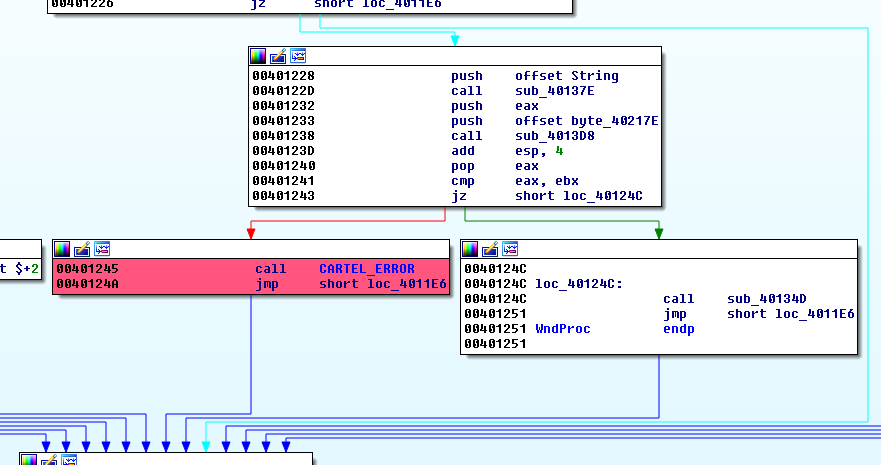
Veamos la primera.



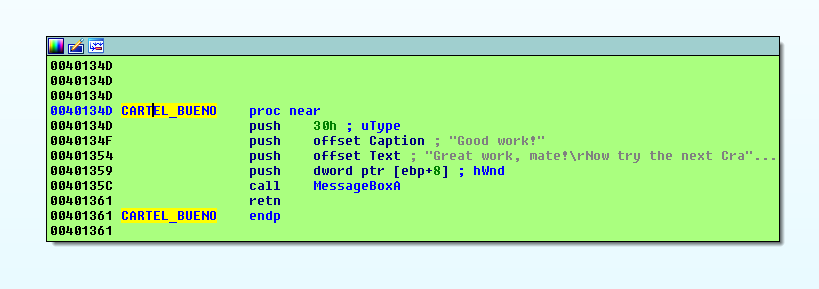
Pinto de rojo el bloque ya que es uno de error o chico malo.



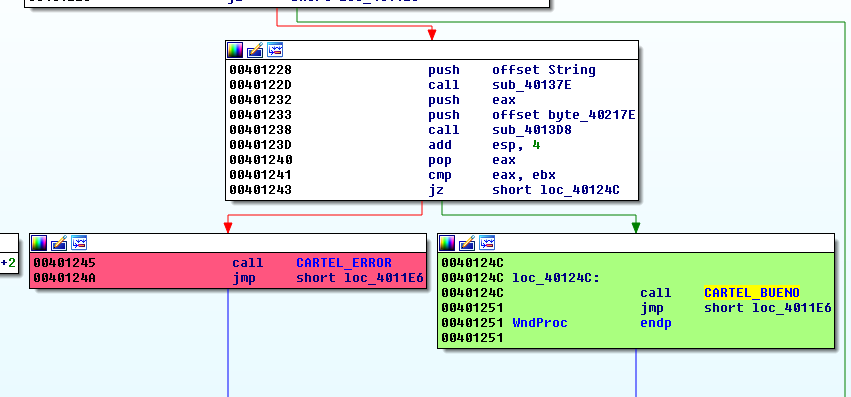
Veamos de donde viene.



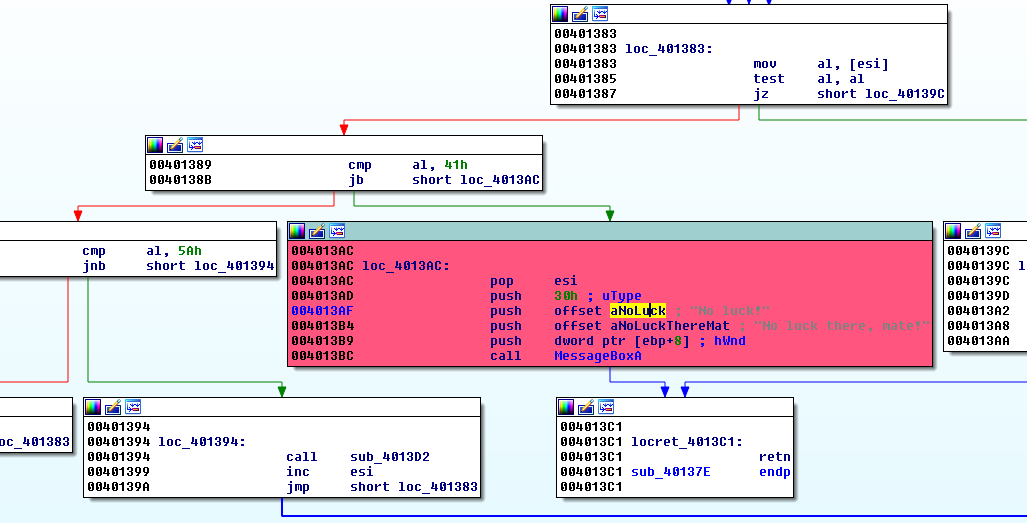
Así que el otro camino debe ser el chico bueno, miremos dentro de la función 0x40134d.



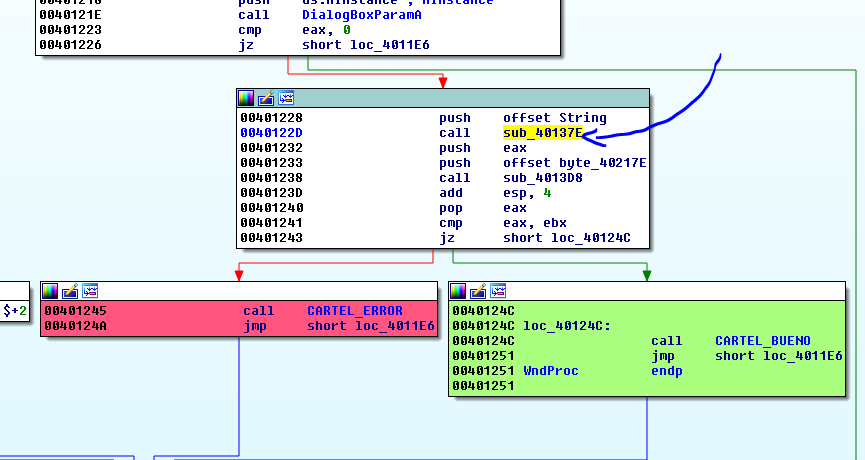
Así que en la referencia.

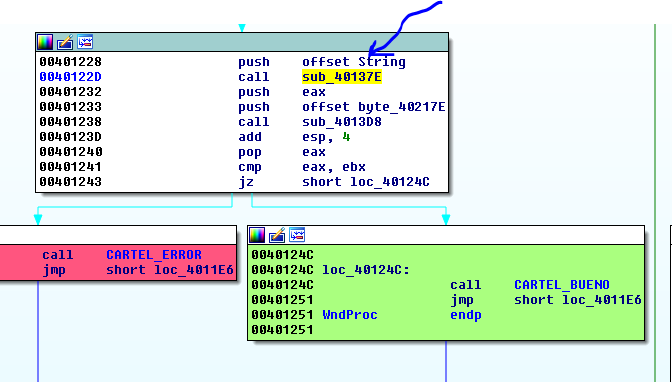


La otra referencia a la string de NO LUCK venia de acá.

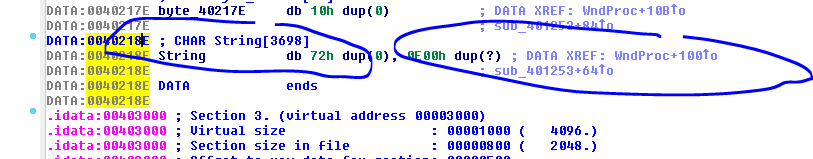


Y la referencia de este otro cartel de error viene de acá.



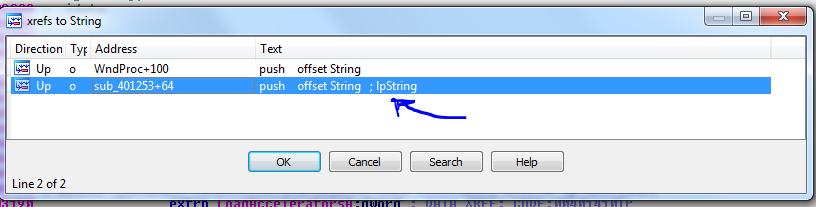


Vemos que el argumento que le pasa a esa función es la dirección (OFFSET) de una variable global que se llama STRING, vemos si hacemos click en ella donde está ubicada.

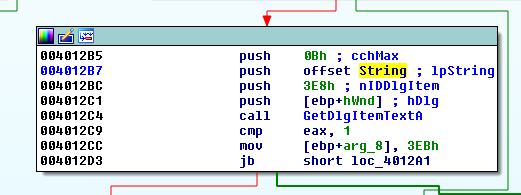


Es un buffer de 3698 bytes de largo en la dirección 0x40218e en la sección DATA.

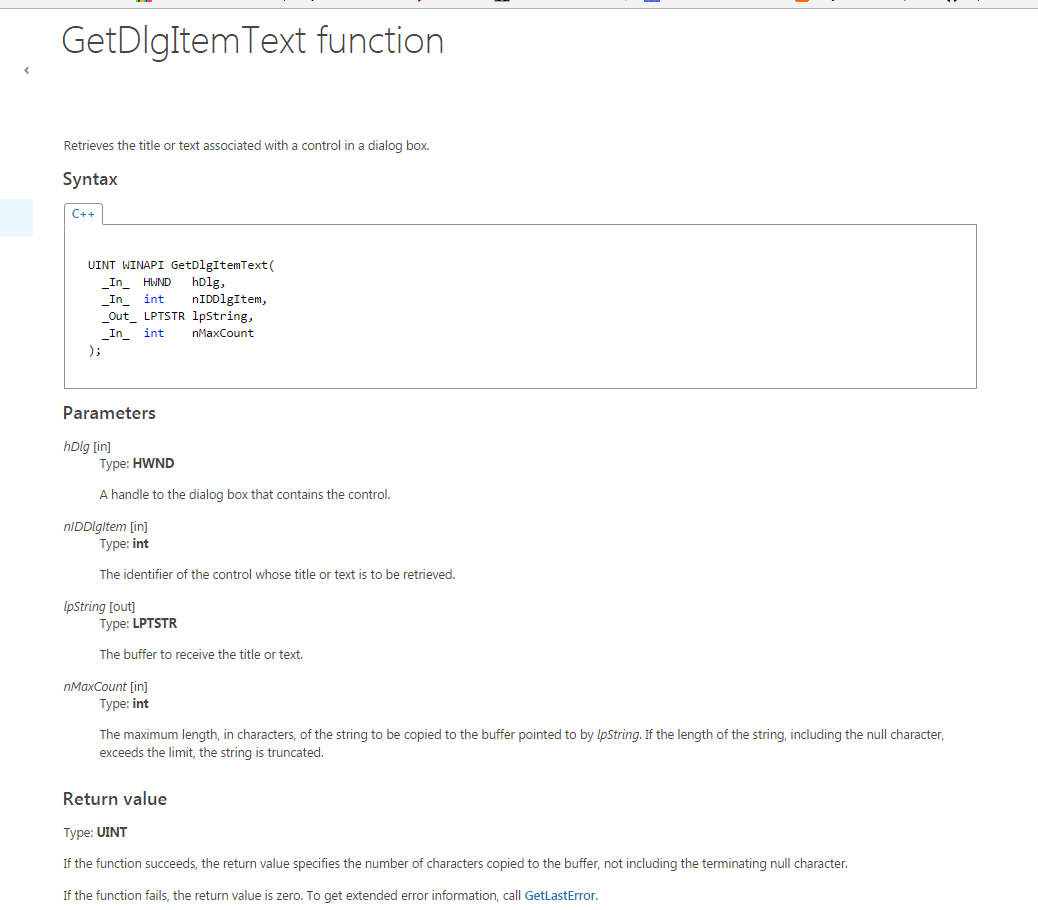
Vemos que tiene un par de referencias para ver que string guardara allí.



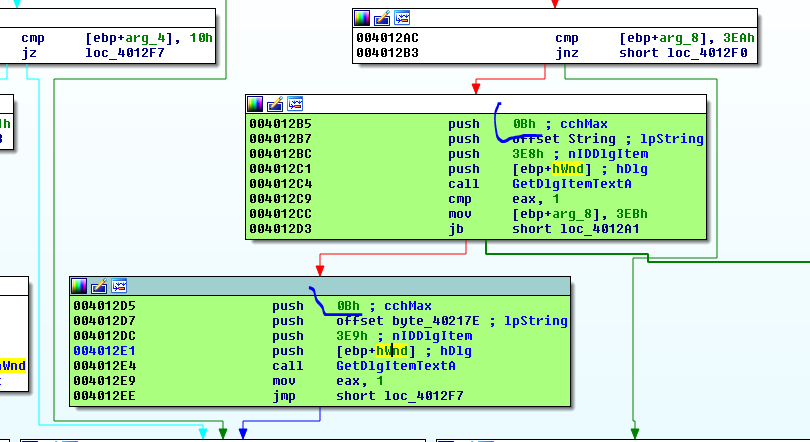
Vemos que hay una referencia que viene de aquí.



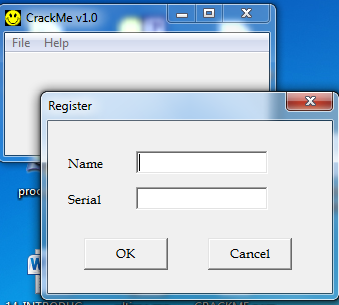
La api GetDLGItemTextA se utiliza para introducir algún dato, veamos.



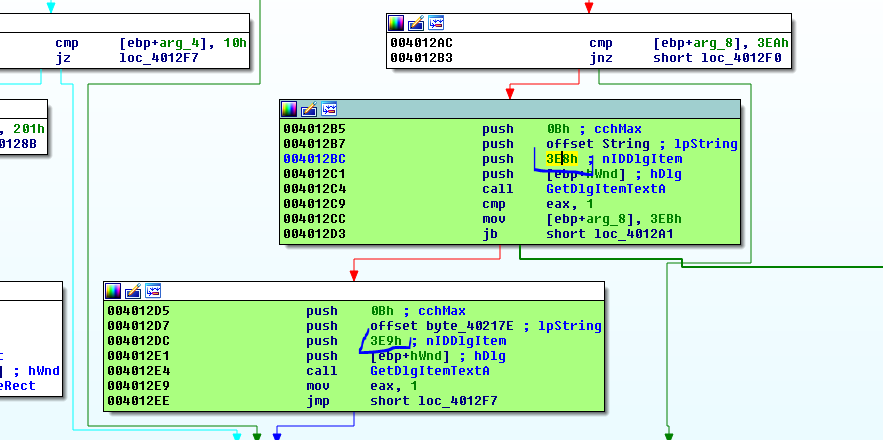
Bueno ahí va a entrar algún texto con el teclado.



Vemos que hay dos entradas continuadas con el mismo handle hWnd, por lo tanto supongo que deben ser las entradas para user y password que el crackme tiene cuando entras a REGISTER.

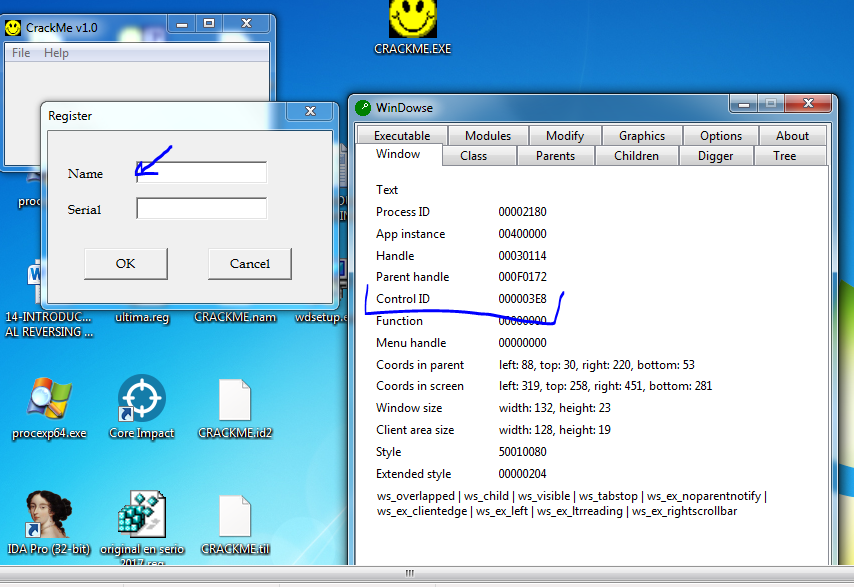


También vemos que ambos tienen el número de control nIDDlgItem de cada uno, 0x3e8 y 0x3e9.



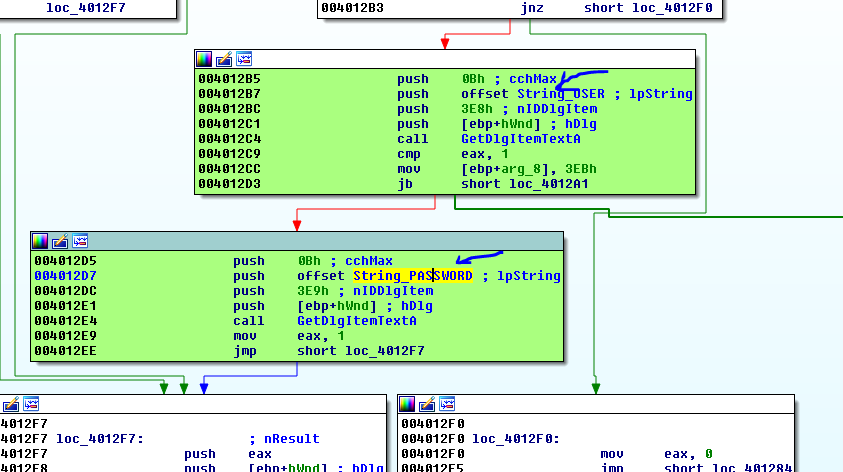
Utilizando el programa GREATIS WINDOWSE que nos da información acerca de las ventanas donde pasamos el mouse.

<http://www.greatis.com/wdsetup.exe>



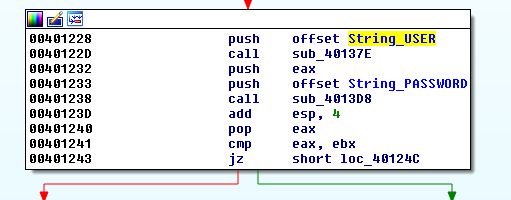
Verifico que el control id de la caja de texto de arriba es 0x3e8 y la de abajo 0x3e9.

También puedo renombrar los buffers donde recibe las strings, el primero como corresponde al user a String\_USER y el segundo como String\_PASSWORD, ambos solo aceptan 0x0b caracteres de largo máximo, a pesar de ser los buffers mas grandes.

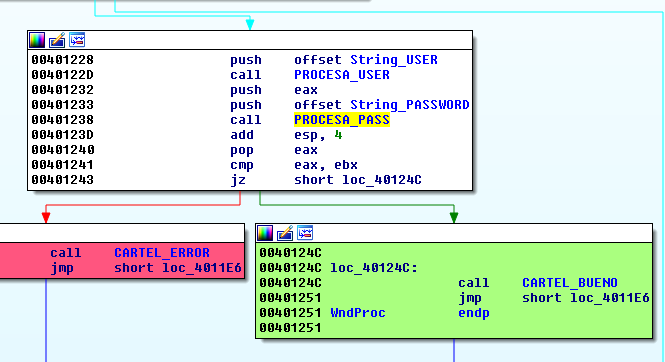


Bueno ya sabemos dónde se guarda el user y password que se tipea, veamos que hace con las misma.

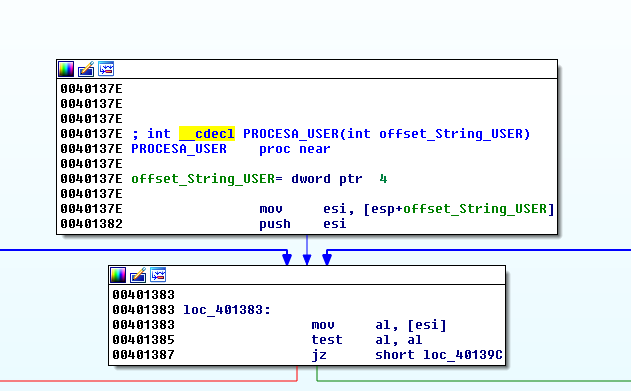
La String\_USER ya habíamos visto que la procesaba aquí dentro.



Analicemos que hace con la misma, pero antes podemos cambiar los nombres de las funciones ya que aparentemente la primera procesa la String\_USER y la segunda procesa la String\_PASSWORD

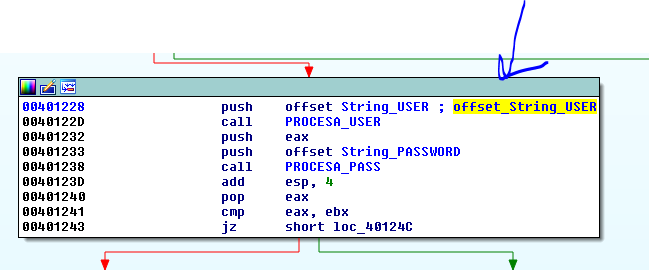


Ahora si analicemos PROCESA\_USER.

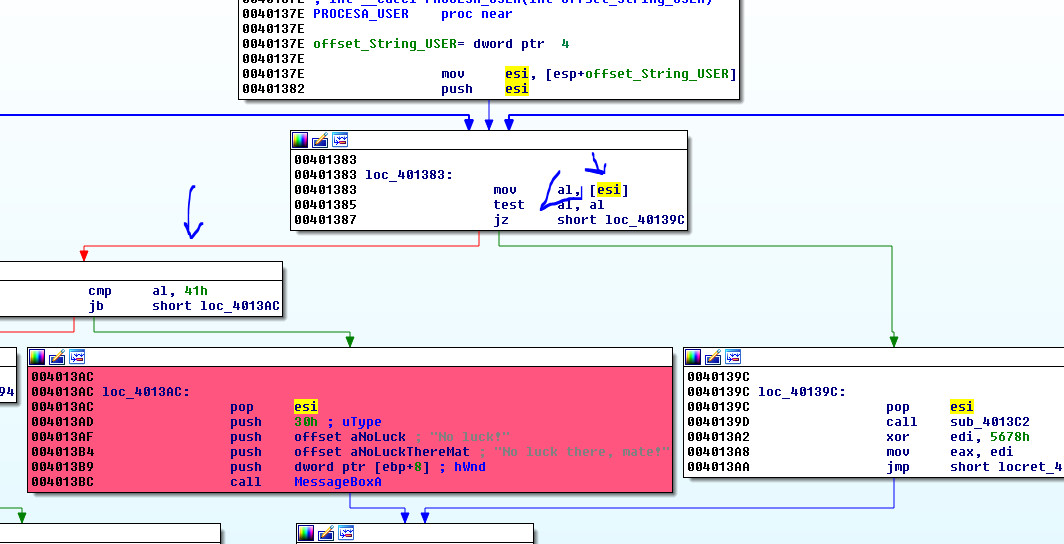


Renombrando la variable uso para respetar el mismo nombre que uso IDA, aunque podría haber puesto p\_String\_USER ya que es también es el puntero a la misma.

Con SET TYPE propago la función y me fijo si coincide el argumento en la referencia.

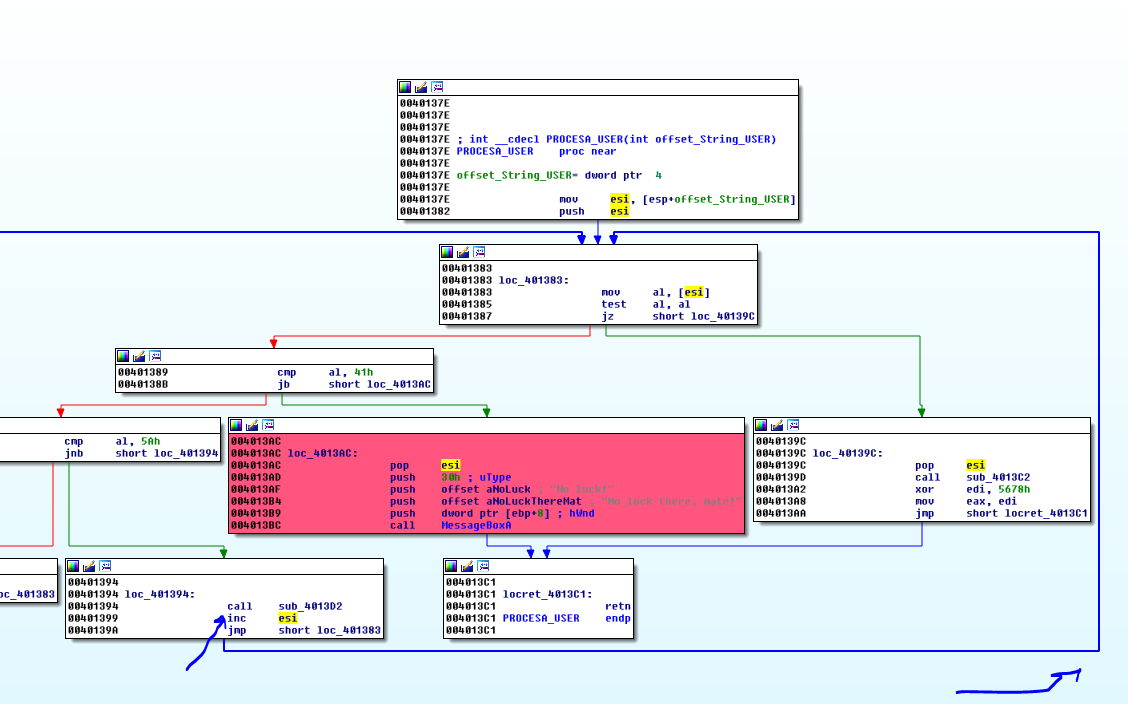


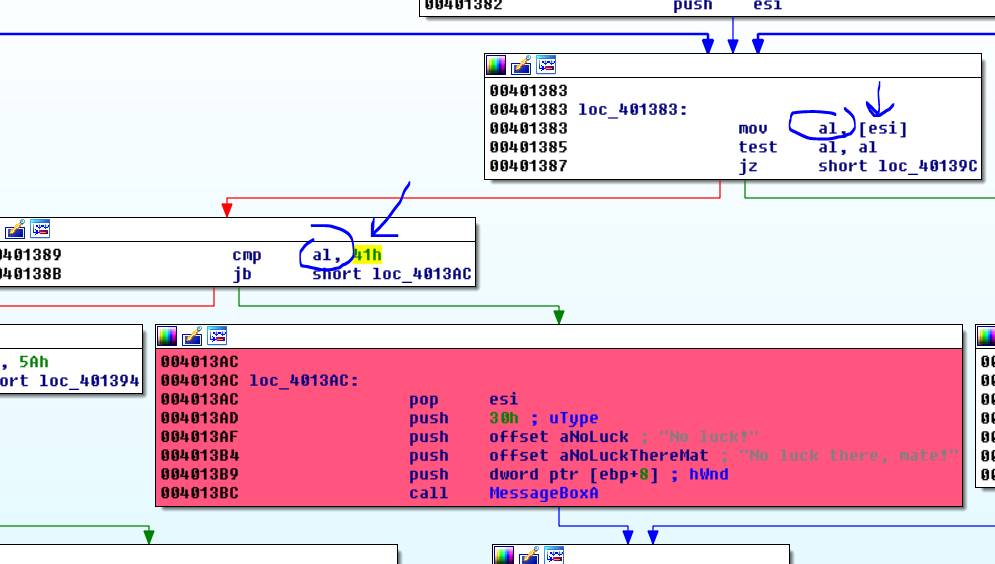
Vemos que la aclaración que agrego coincide con el nombre del argumento.



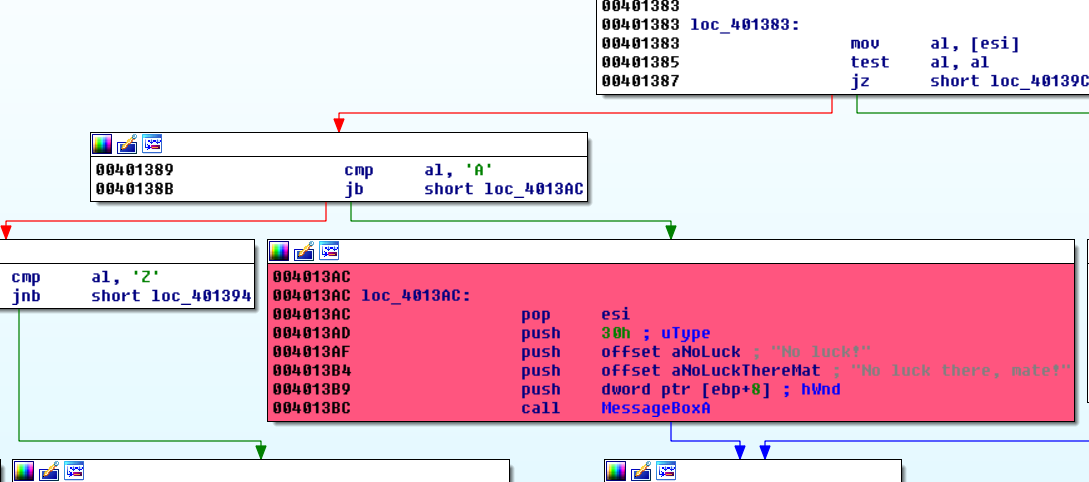
Vemos que hay un LOOP que ira leyendo los BYTES de la String\_USER, mientras que no sea cero el byte o sea hasta que no termine la string se repetirá el LOOP e ira por el camino de la flecha ROJA.

Allí se ve el LOOP completo va incrementando ESI para ir leyendo byte a byte cada carácter de la String\_USER, cada uno que toma los compara con 0x41.

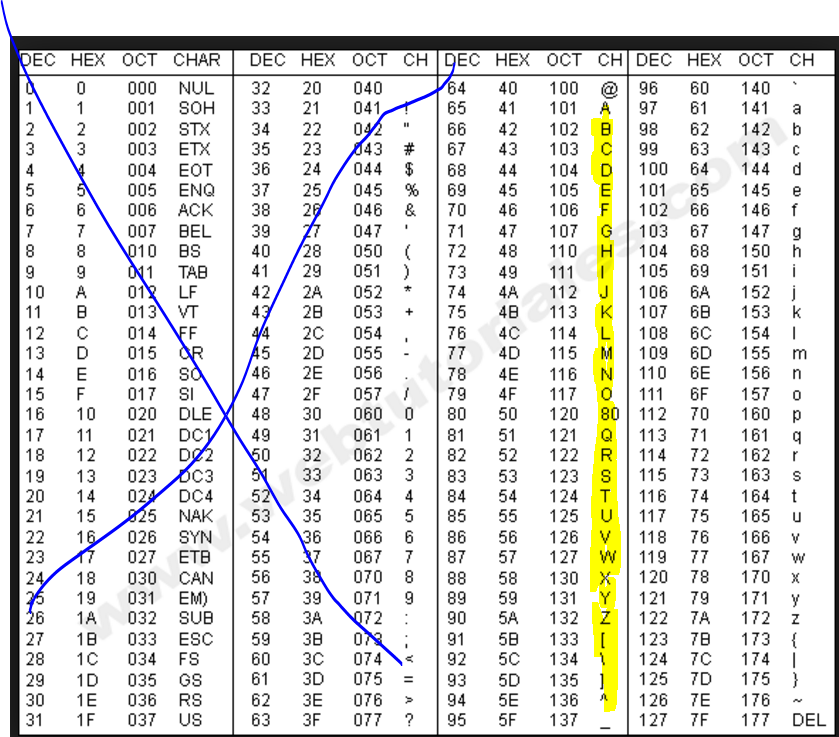




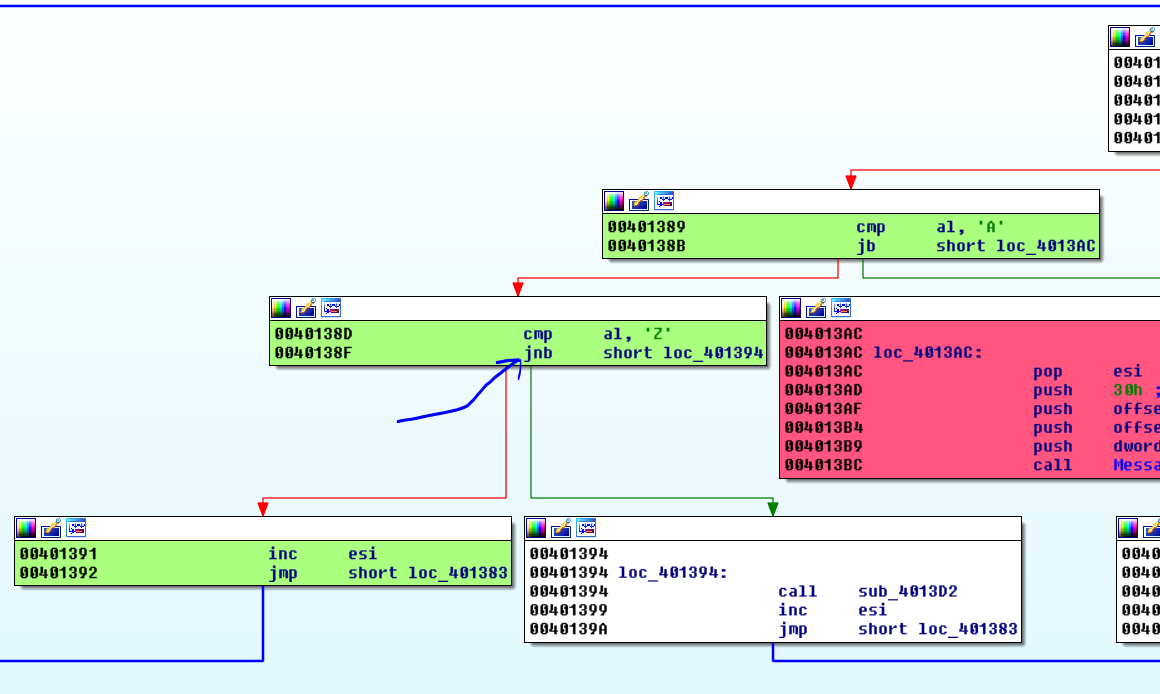
Podemos hacer click derecho en el 0x41 y cambiar por la A que es el carácter ASCII de ese valor.



La cuestión es que si es más bajo que el carácter A te tira a NO LUCK, así que si vemos la tabla ASCII, no acepta números en USER solo letras ya que deben ser mayores o iguales que A.

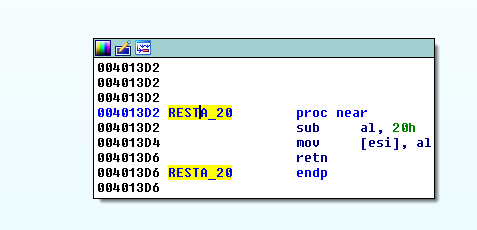


Así que chequea que todos los caracteres de la String\_USER sean mayores que 0x41 o sea A por lo menos.

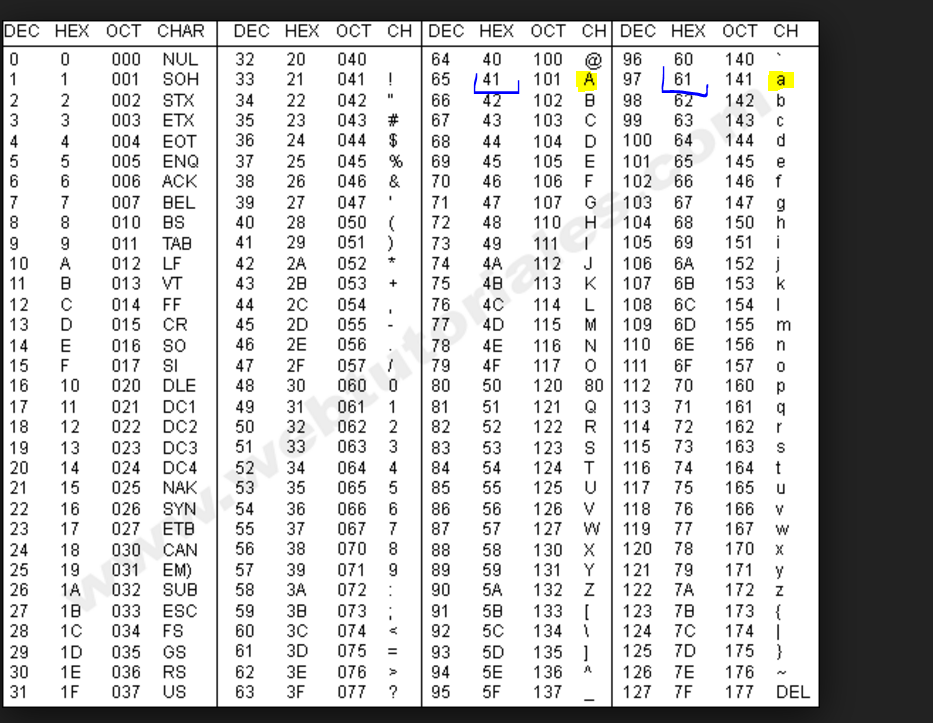


También chequea JNB si no es más bajo que Z lo manda a ese BLOQUE en 0x401394, sino lo deja como esta y sigue con el siguiente carácter por el camino de bloques verdes.

Así que acepta las letras mayúsculas salvo la Z los caracteres mayores o iguales a Z van a ese bloque veamos que hace en el mismo.

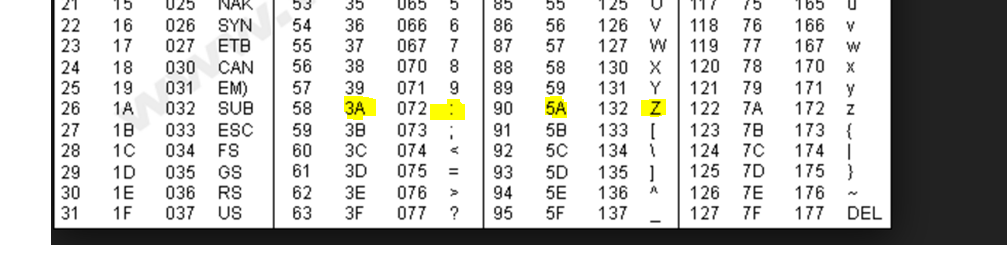


Le llame resta\_20 porque eso es lo que hace, si es más grande que Z le resta 20 y lo guarda.



O sea que si pasas 0x61 que es la “a” minúscula al restarle 0x20 quedara valiendo 0x41 que es la “A “ mayúscula, hace lo mismo con todos los caracteres más grandes o igual que Z.

Si es la Z menos 0x20 nos da 0x3a que es el símbolo de dos puntos.



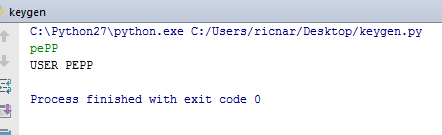
## KEYGEN CRACKME DE CRUEHEAD.

La cuestión es que podemos ir armando un script de Python para ir armando el keygen.

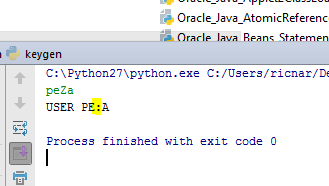
user=raw\_input()  
largo=len(user)  
**if** (largo> 0xb):  
 exit()  
  
userMAY=**""  
  
for** i **in** range(largo):  
 **if** (ord(user[i])<0x41):  
 **print "CARACTER INVALIDO"** exit()  
 **if** (ord(user[i]) >= 0x5a):  
 userMAY+= chr(ord(user[i])-0x20)  
 **else**:  
 userMAY+= chr(ord(user[i]))  
  
  
**print "USER"**,userMAY

Vemos que el script hace lo mismo que el programa, va tomando uno a uno los caracteres de la string user y compara con 0x41 si es menor te dice que es un carácter invalido y te tira a EXIT sino ve si es mayor o igual que 0x5a y si es así le resta 0x20 y lo va agregando a la string userMAY.

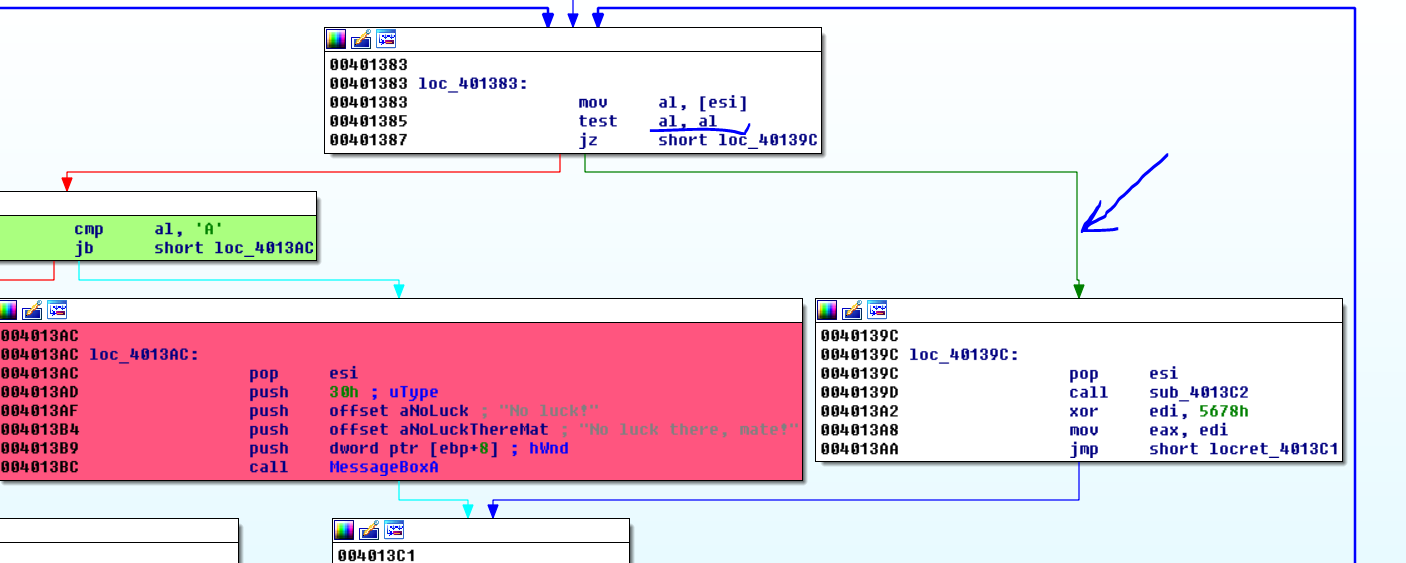
Vemos que si ingreso pePP me lo transforma en PEPP.



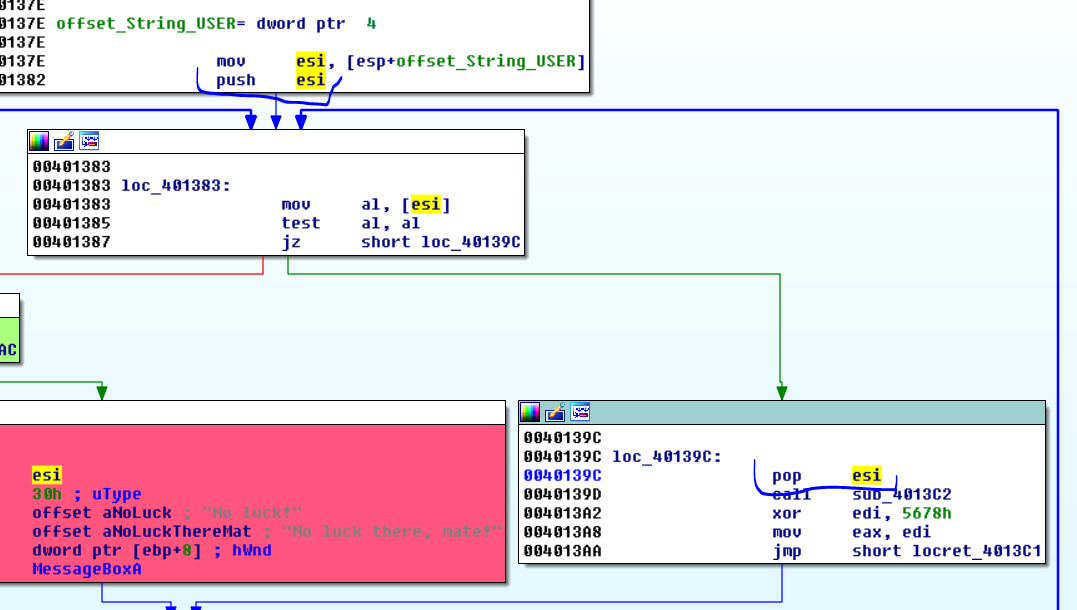
Y si ingreso la Z me la transforma en dos puntos como vimos.



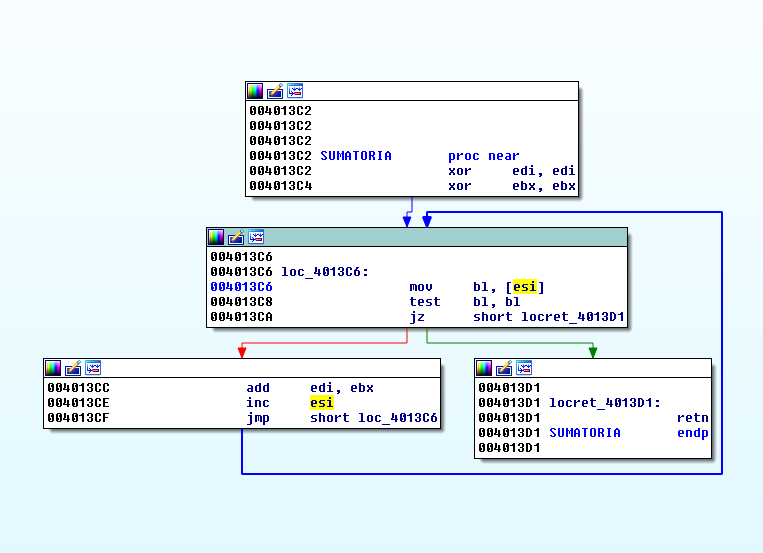
Así que hasta ahí vamos igual que el programa, cuando sale del LOOP veamos que hace sigue por aquí.



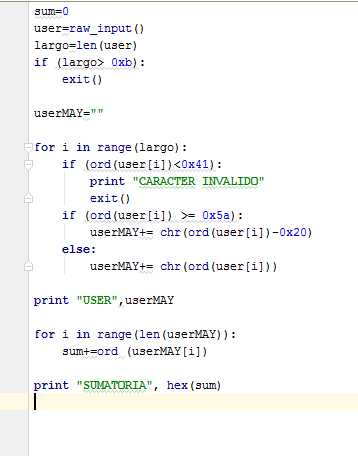
Cuando halle un carácter que sea cero terminara el LOOP e ira al bloque en 0x40139c.

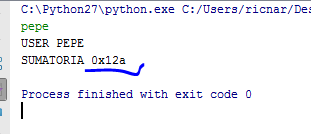


Vemos que antes de ir incrementando ESI, lo había PUSHEADO al stack para guardar el valor original que apunta al inicio de la string, y allí con el POP ESI lo recupera antes de entrar al call 0x4013c2.



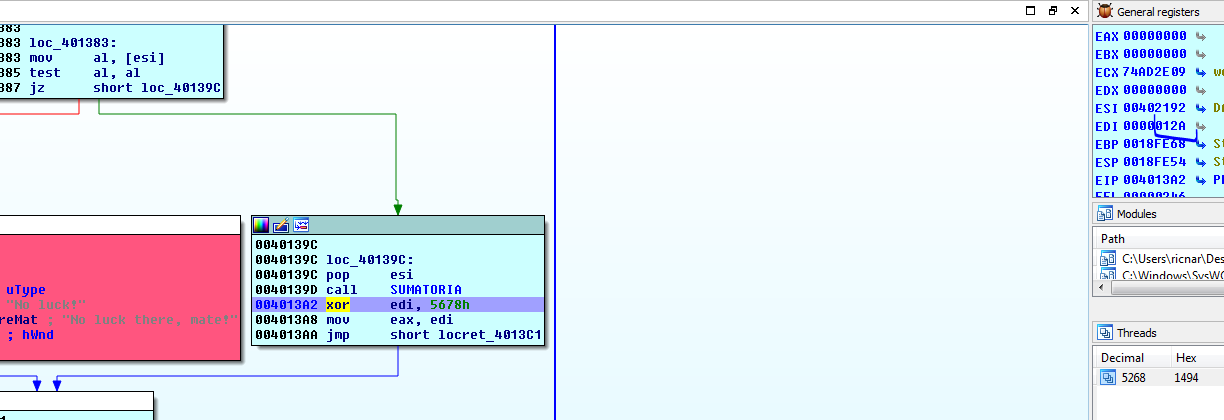
Vemos que es un LOOP que suma todos los bytes por eso lo llame SUMATORIA, así que podemos agregar eso en nuestro script.





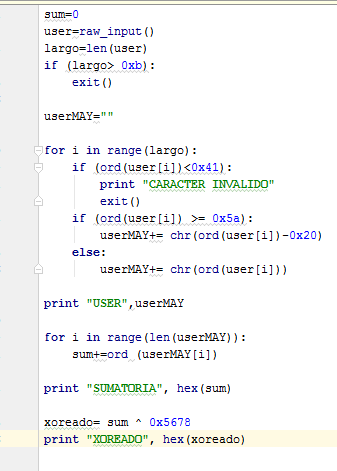
Sumo todos los bytes e imprimo la sumatoria.

Para comprobar si voy bien pongo un breakpoint al salir de la sumatoria y pongo pepe en el campo user y password 989898.

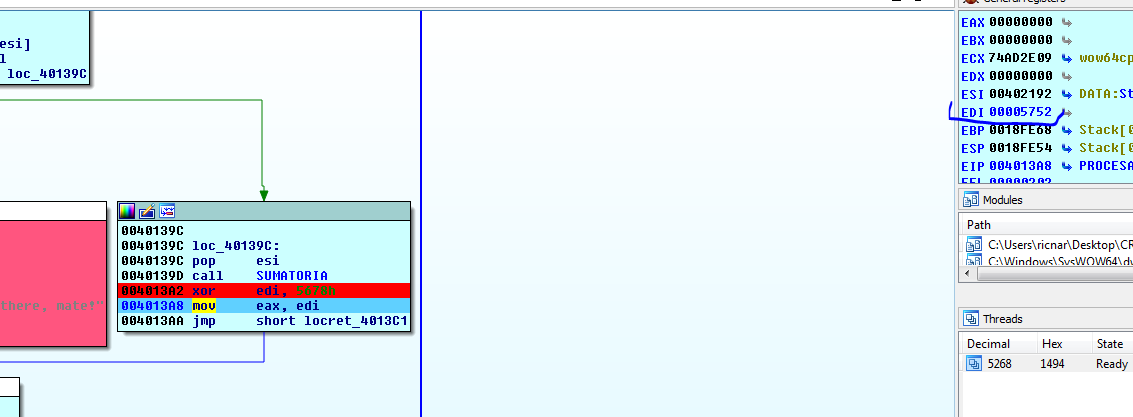


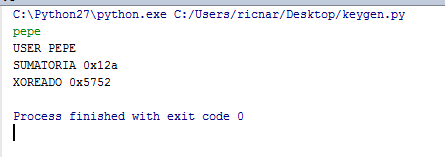
Veo que la sumatoria da 0x12a así que vamos bien.

Justo en esa línea XOREA con 0x5678 así que lo agrego al script.

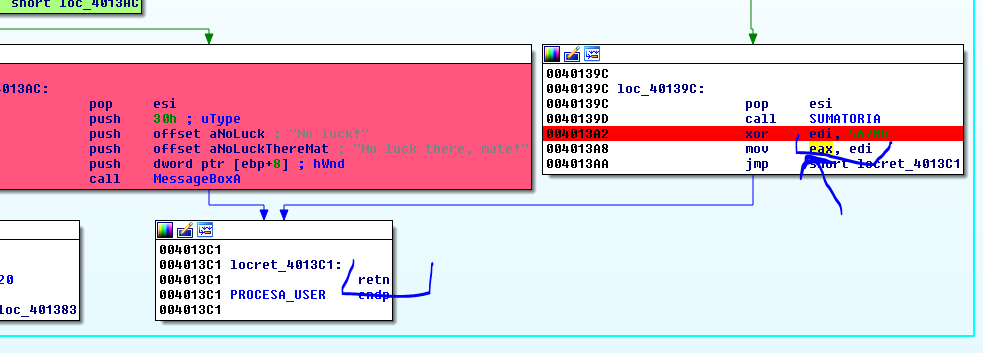


Si ejecuto esa línea XOREA y da lo mismo que el script.

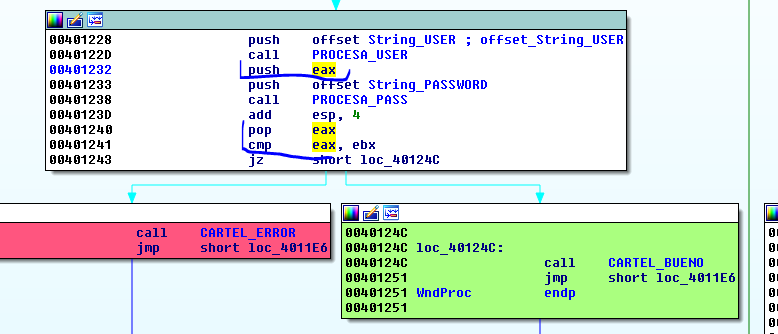




Luego mueve el resultado a EAX y sale

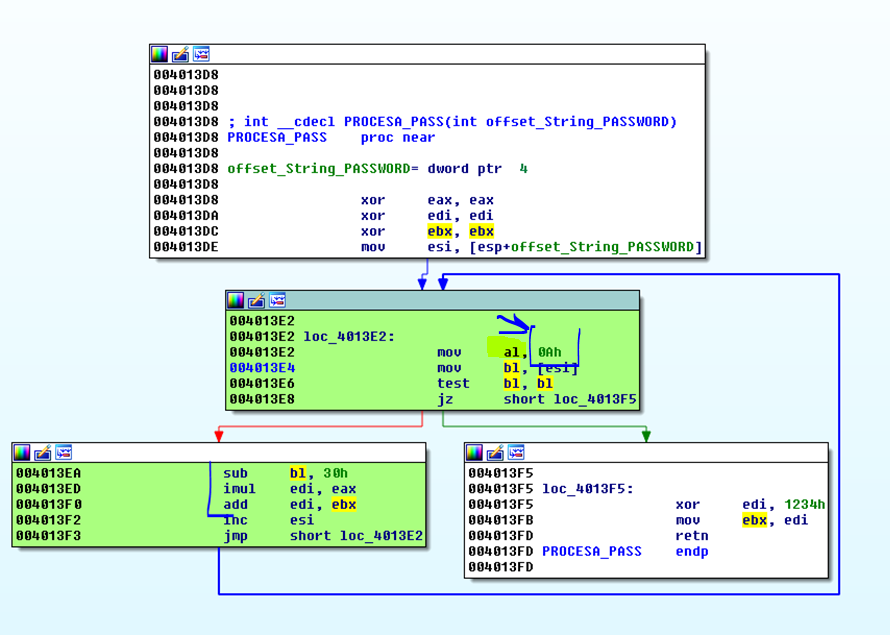


Luego PUSHEA EAX lo recuperara con POP EAX antes de la comparación final, o sea que en el CMP EAX, EBX, el primer miembro será este valor que sale de PROCESA\_USER.



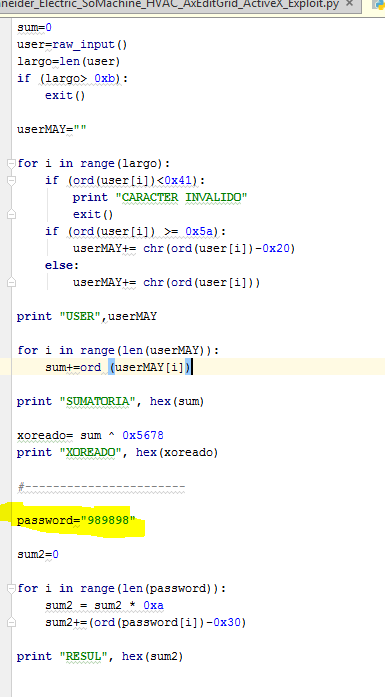
Veamos ahora que hace con el password en PROCESA\_PASS.

Allí vemos



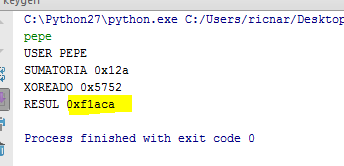
Va leyendo cada byte lo mueve a BL y a cada uno le resta 0x30 que queda en EBX, luego multiplica EDI que tiene la sumatoria por 0x0a y le suma EBX.

Hare otra parte del script con esto.



No ingreso el password por teclado solo estoy probando ya que en el keygen solo se ingresa el usuario, pero vero que si mi password es por ejemplo 989898.

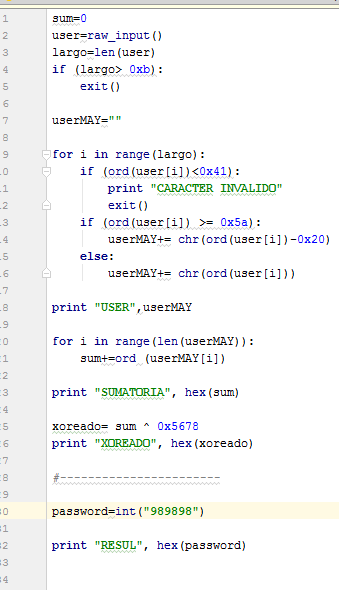
A sum2 que es la sumatoria que va guardando lo multiplica por 0xa y luego le suma el byte menos 0x30 como hace el programa.



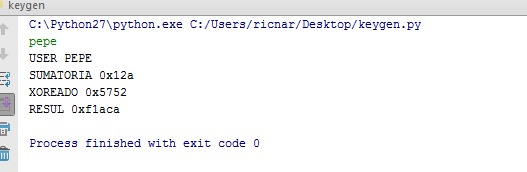
Al ejecutarlo veo que el password 989898 me dio como resultado de todo eso el valor f1aca que es el valor hexadecimal de la string 989898.



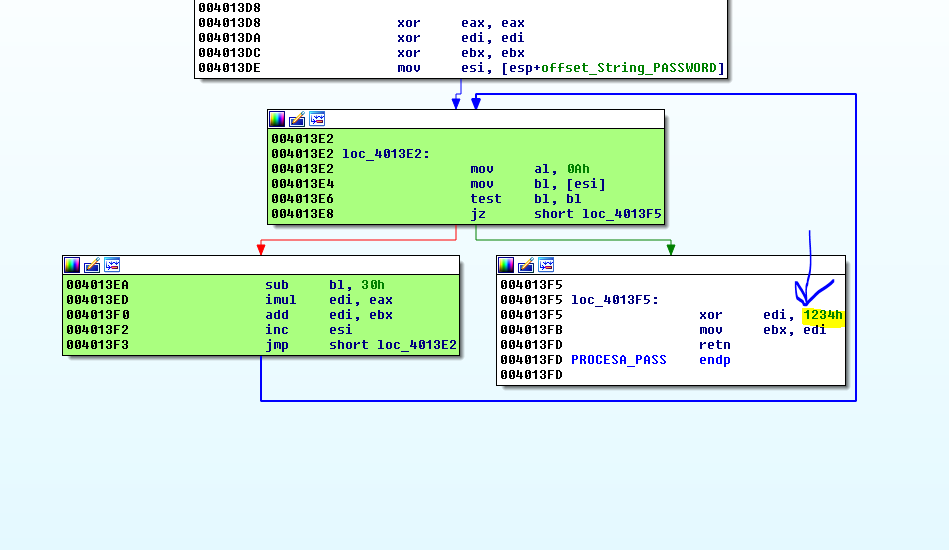
Así que todo eso en el script se puede resumir a pasar a hexadecimal la string con la función hex().



Me da exactamente lo mismo.



Al terminar XOREA ese resultado con 0x1234 y sale a compararlo con el valor que recupera en EAX que devolvió PROCESA\_USER.



Así que la formula genérica seria.

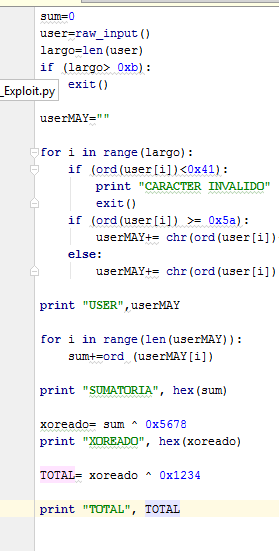
hex(password)^0x1234 = XOREADO

Donde XOREADO es el resultado que me devolvía PROCESA\_USER.

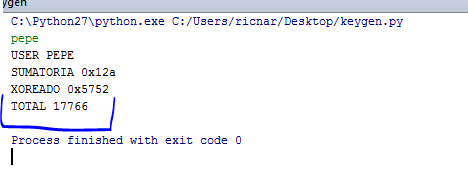
Si despejo

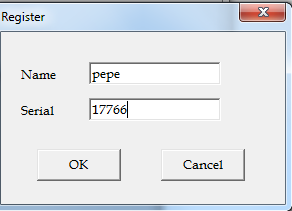
hex(password) = XOREADO ^ 0x1234

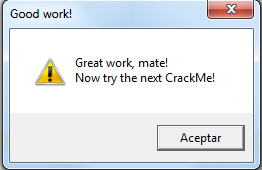
O sea que si al resultado que tenía lo xoreo por 0x1234 ya casi lo tengo veamos.



Si corremos eso con la string pepe





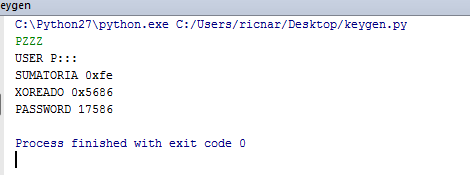


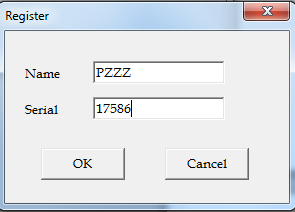
Así que ya tenemos el keygen y como antes no es necesario pasar el resultado a decimal porque ya Python nos hace la conversión.

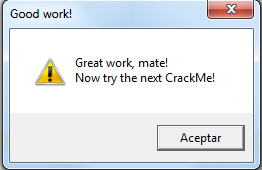
Aquí lo copio al keygen.

sum=0  
user=raw\_input()  
largo=len(user)  
**if** (largo> 0xb):  
 exit()  
  
userMAY=**""  
  
for** i **in** range(largo):  
 **if** (ord(user[i])<0x41):  
 **print "CARACTER INVALIDO"** exit()  
 **if** (ord(user[i]) >= 0x5a):  
 userMAY+= chr(ord(user[i])-0x20)  
 **else**:  
 userMAY+= chr(ord(user[i]))  
  
**print "USER"**,userMAY  
  
**for** i **in** range(len(userMAY)):  
 sum+=ord (userMAY[i])  
  
**print "SUMATORIA"**, hex(sum)  
  
xoreado= sum ^ 0x5678  
**print "XOREADO"**, hex(xoreado)  
  
TOTAL= xoreado ^ 0x1234  
  
**print "PASSWORD"**, TOTAL

Incluso en casos medios raros con la Z







Así que reverseamos e hicimos un keygen del crackme de Cruehead, nos vemos en la parte 20.

Hasta la próxima.

Ricardo Narvaja