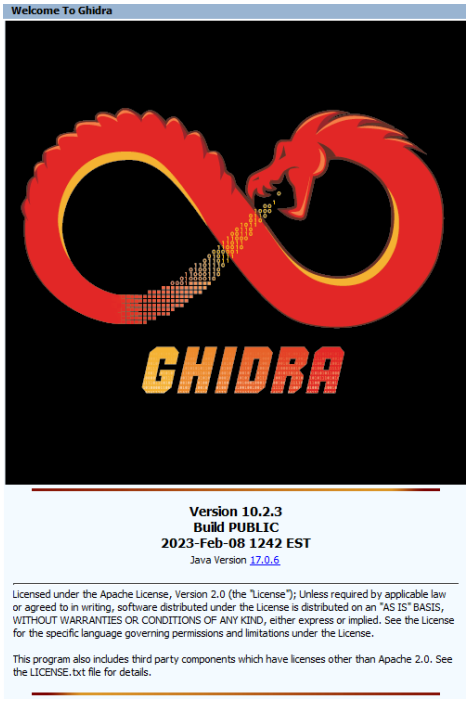
Ingenieria inversa básica con Ghidra

Publicado el El Informatico - 1 de marzo de 2023 -



Aunque no es del todo perfecto, Ghidra es un programa bastante potente para realizar ingenieria inversa sobre archivos binarios ejecutables, y lo mejor de todo es que es bastante sencillo de usar, en general. Más incluso que otros programas similares como Radare o IDA. En éste tutorial propongo un ejemplo muy sencillo de cómo usar Ghidra para analizar un ejecutable PE de Windows y cómo modificarlo desde el propio programa.

Para éste ejemplo, he creado un programa muy sencillo que solicita una clave , y usa un algoritmo muy simplón similar a algunos algoritmos que se usaban antíguamente para verificar las claves de CD.

> Puedes descargar el ejecutable que estoy usando en el ejemplo aquí.

Al ejecutar el programa, nos solicita una clave. Obviamente como creador del programa, conozco la(s) claves válidas y lo que hace el programa. Pero supongamos

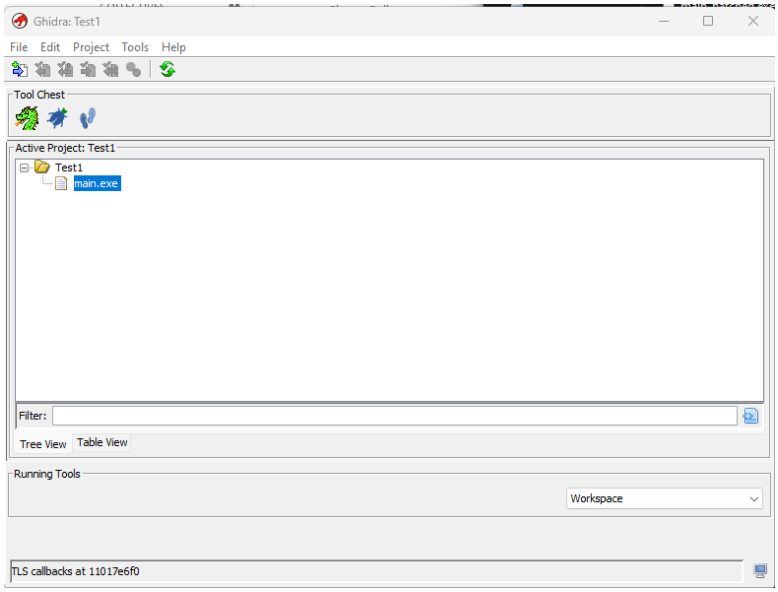
que no tenemos ni idea de qué es lo que hace. Al introducir claves al azar, el programa nos dice que son incorrectas.

```
@DESKTOP-IKUIJA1 MINGW64 ~/src/test
$ ./main
Introduzca la clave: 12345
Clave incorrecta!
@DESKTOP-IKUIJA1 MINGW64 ~/src/test
$ ./main
Introduzca la clave: hola1234
Clave incorrecta!
@DESKTOP-IKUIJA1 MINGW64 ~/src/test
$
```

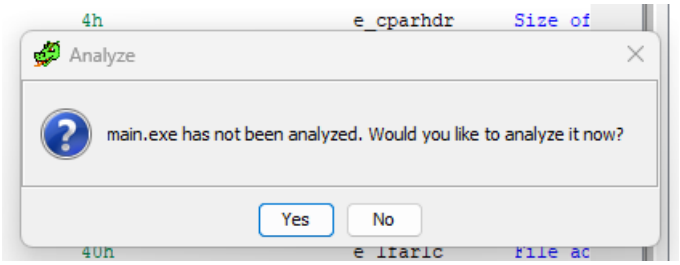
Supongamos que queremos conseguir acceder al programa, o al menos una clave válida para el mísmo. La única forma de conseguirlo es, obviamente, analizando el código del programa (si bien éste programa usa un algoritmo tan sencillo, que es muy probable que cualquiera pueda averiguarlo simplemente mediante ensayo y error, pero es un ejemplo).

Creando el proyecto...

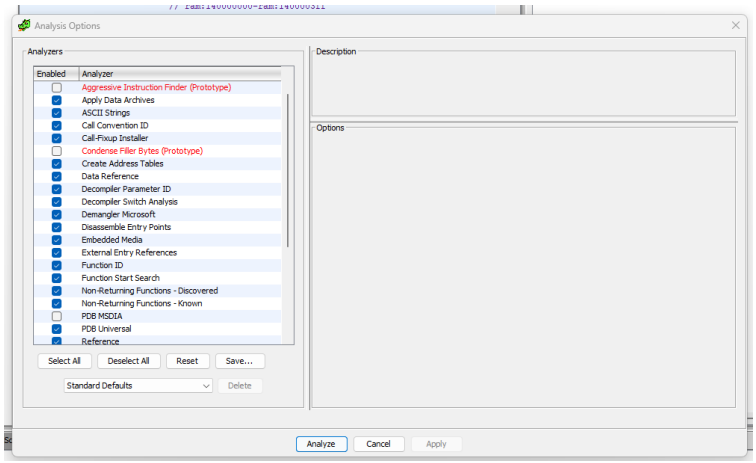
Con Ghidra, es muy sencillo. Podemos empezar por abrir Ghidra y crear un nuevo proyecto pulsando New > Project, o las teclas Ctrl + N. Con el proyecto ya creado, arrastramos el ejecutable, que en éste caso es un PE de Windows, a la carpeta del proyecto. El tipo de ejecutable será PE de Windows.



Si hacemos doble click sobre el ejecutable en el proyecto de Ghidra, nos abrirá la ventana con el desensamblado del código. La primera vez que lo hagamos, nos preguntará si queremos analizar el código.



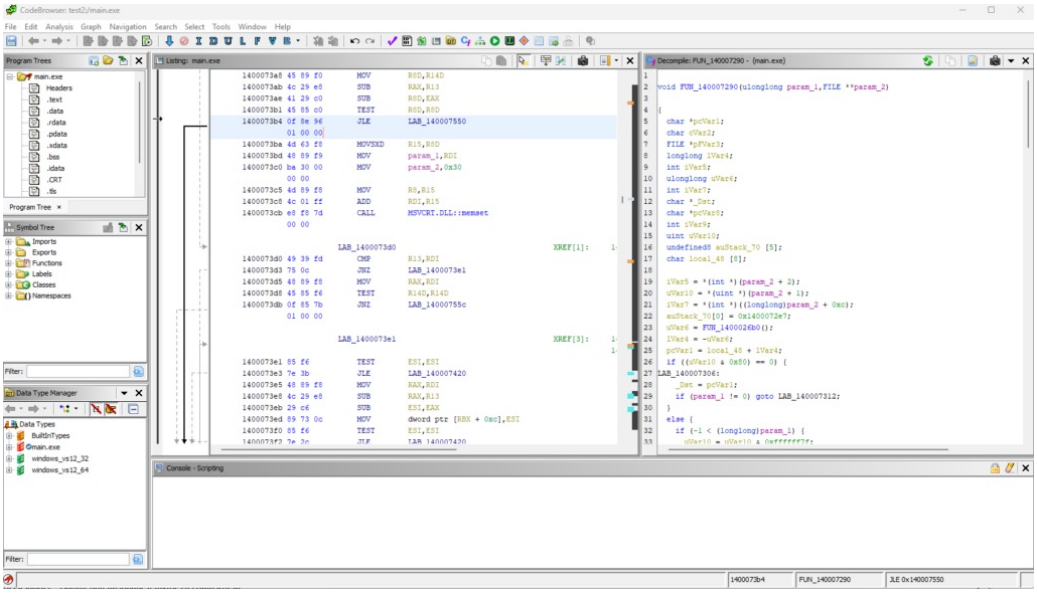
Es importante decirle que «*sí*» (Yes) siempre que sea posible, ya que al analizarlo Ghidra podrá reconocer todas las rutinas importadas de otras librerias, cadenas de texto, e incluso generará un pseudo código en C con la representación del código desensamblado que nos puede ser de mucha ayuda, asumiendo que sepas usarlo correctamente.



En la ventana de opciones de análisis, dejamos todo por defecto y pulsamos sobre «Analyze». El análisis puede tardar unos minutos, y el progreso se muestra en la esquina inferior derecha de la ventana.

La interfaz principal de Ghidra

Este es un buen momento para familiarizarse con la interfaz de Ghidra:



La interfaz principal se divide en las siguientes ventanas:

- **Program Trees**: Muestra una lista de las diferentes secciones del ejecutable. Haciendo doble click sobre cada una de ellas, podemos ir al inicio de dicha sección en la ventana del desensamblado.
- **Symbol Tree**: Aquí se muestra una lista categórica de las rutinas en el programa, y las rutinas usadas por el programa, así como clases y espacios de nombres para otro tipo de ejecutables. Esta vista es de las más importantes, ya que aquí puedes ver a qué funciones llama el programa, y cuales exporta para

ser usadas de forma externa.

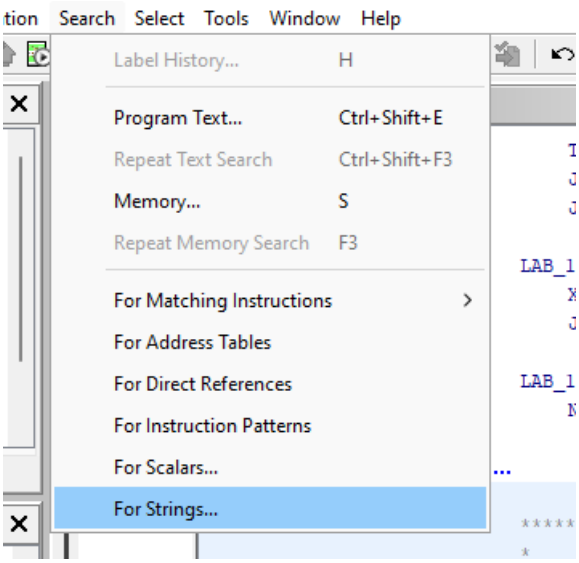
- **Data Type Manager**: Nos muestra una lista de tipos de datos y cabeceras usadas por el programa. Puede ser útil en casos muy específicos.
- **Listing**: Esta es la vista del código desensamblado del programa.
- **Decompile**: Muestra un equivalente en C al código desensamblado del programa. Es importante entender que el código mostrado es una interpretación literal del código desensamblado. Por lo general, es conveniente tener en cuenta las dos vistas de código y no centrarse sólo en una, ya que hay código que podría no ser visible en la descompilación.
- **Console**: Muestra la salida de los scripts usados en Ghidra.

Una vez finalice el análisis del ejecutable, podemos empezar a estudiar el programa. En éste caso, el objetivo principal es encontrar la rutina donde se encuentra la comprobación del código. Y necesitamos alguna pista que nos lleve a esa rutina, ya que ahora mismo no tenemos idea de donde puede estar.

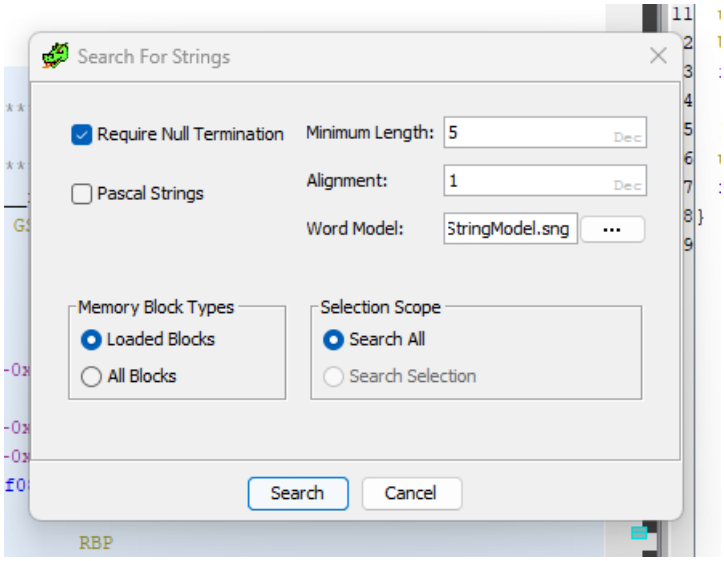
Si observamos el programa otra vez, podemos comprobar que el programa nos muestra dos textos. El primero dice «Introduzca la clave:». Y el segundo dice «Clave incorrecta!». Esto es perfecto, ya que podemos buscar esos textos en el programa, y ver en qué partes del mismo se referencian dichos textos.

Buscando un texto

Vamos al menú «Search» y pulsamos sobre «For String»:

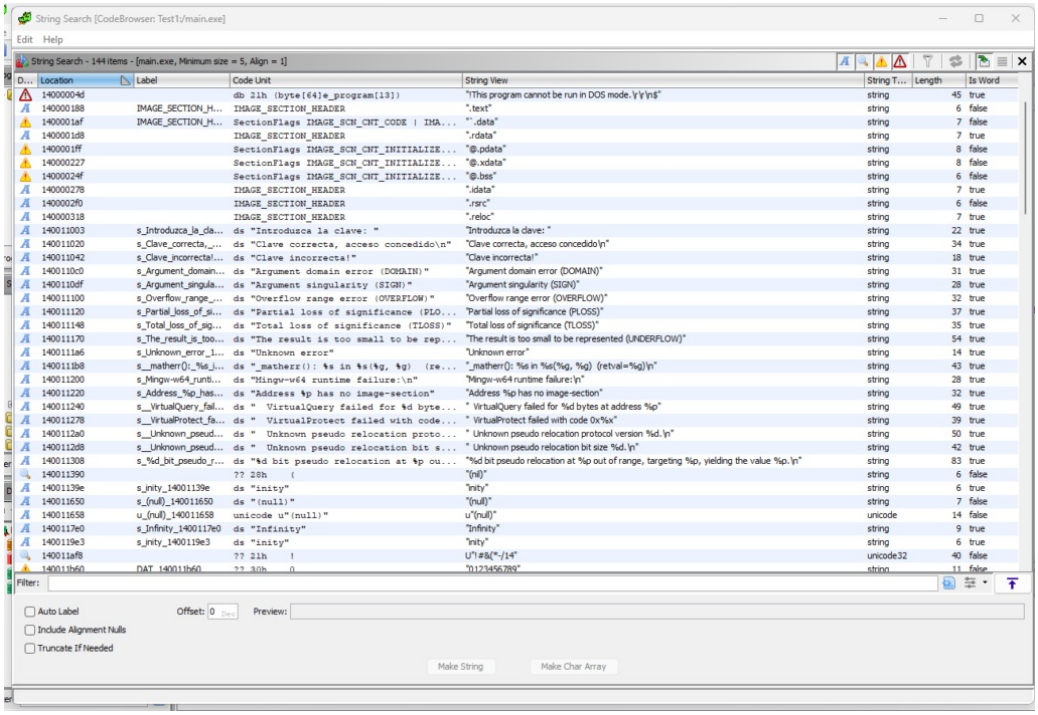


La siguiente ventana nos pide los parámetros de la búsqueda de texto.



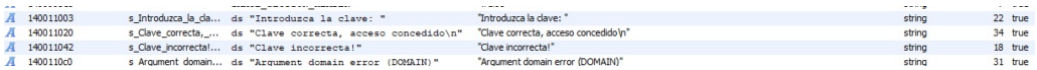
Es importante mantener la opción «Require Null Termination» activa, ya que los textos en C acaban con un caracter nulo ('\0') para indicar el final del mísmo. En caso contrario, Ghidra podría mostrar mucha «basura». Podemos modificar la longitúd minima para que se identifique como cadena de texto. En éste caso, lo dejamos todo por defecto y pulsamos sobre «Search».

Ghidra abrirá otra ventana donde tendremos una lista de todas las cadenas de texto que ha encontrado.

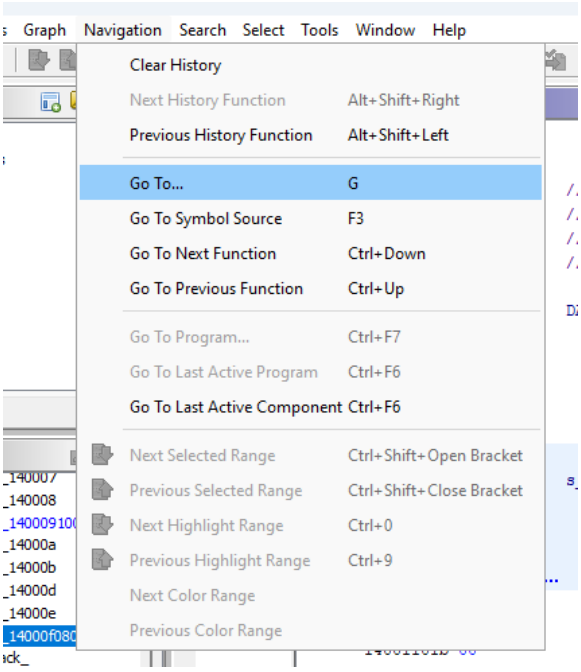


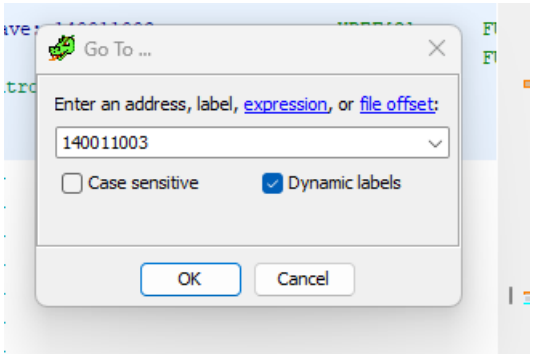
Podemos usar el campo «Filter» para buscar un texto específico, aunque en éste

caso no es necesario. Lo que estamos buscando está a la vista:



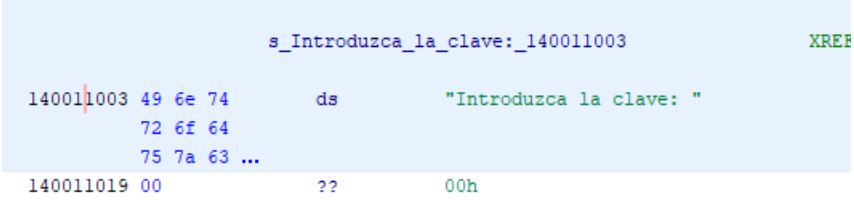
Ahora tenemos que buscar en qué partes del programa se referencian éstos textos. Podemos hacer doble click sobre cualquiera de los textos y cambiar a la ventana principal de Ghidra, o hacer click derecho sobre la dirección del texto (en la segunda columna) y decirle a Ghidra que nos muestre esa dirección en el desensamblado desde el menú Navigation > go To.





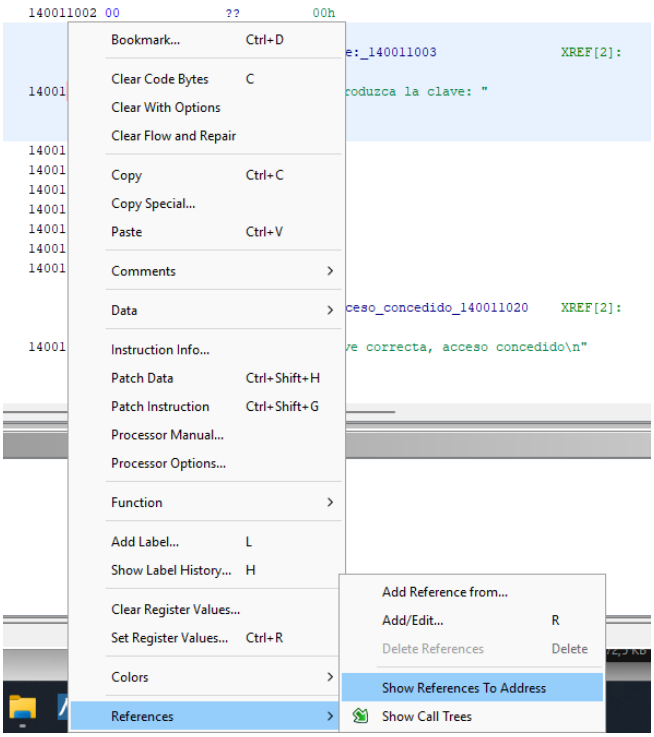
Referencias a direcciones

En cualquiera de los dos casos, en la ventana del desensamblado se mostrará la parte del programa en la que se encuentra el texto.

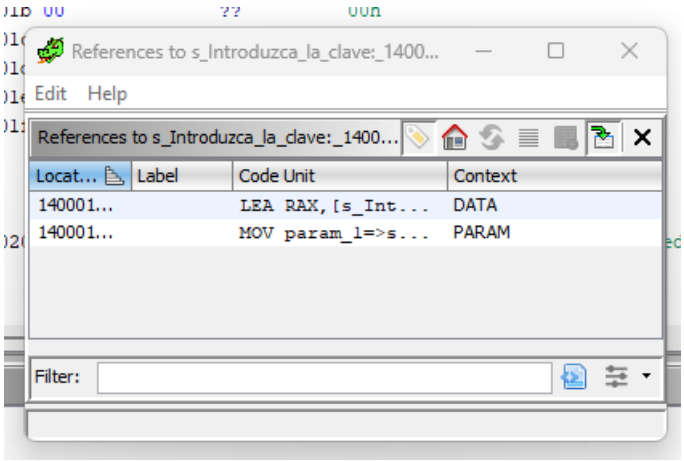


Para ver qué partes del código hacen referencia a éste texto, hacemos click dderecho sobre la dirección y seleccionamos «References > Show references to

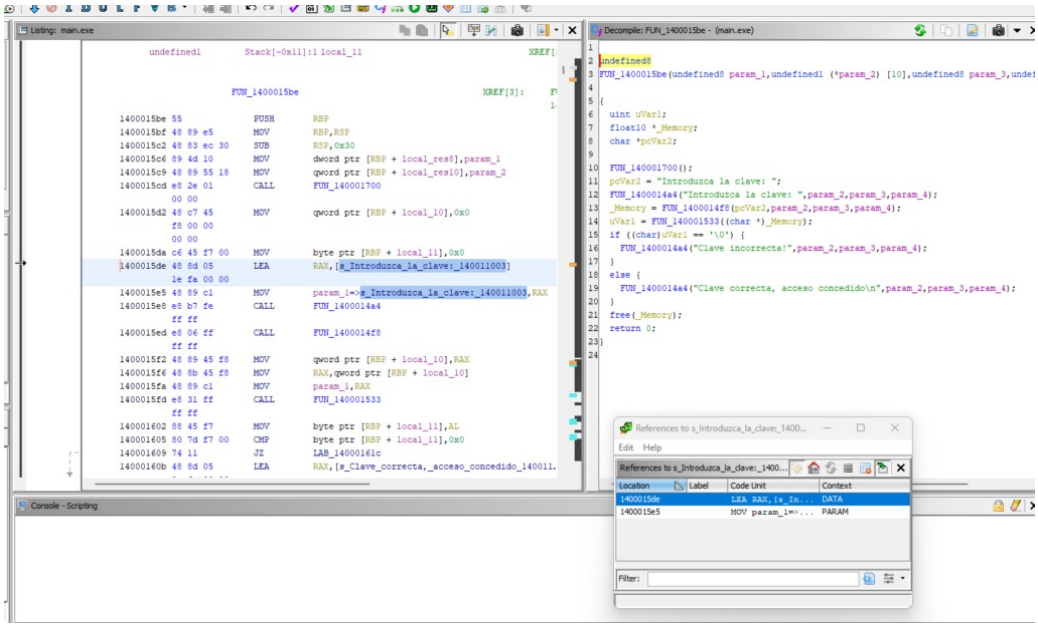
address».



Ghidra abrirá otra ventana donde podemos ver todas las referencias a ésta dirección.



Obviamente nos interesa la referencia en el contexto PARAM, que es donde se está usando como parámetro para llamar a otra función, si bien ambas referencias nos llevan al mismo punto del código. Podemos hacer doble click sobre la última referencia para que Ghidra nos lleve a esa parte del programa.

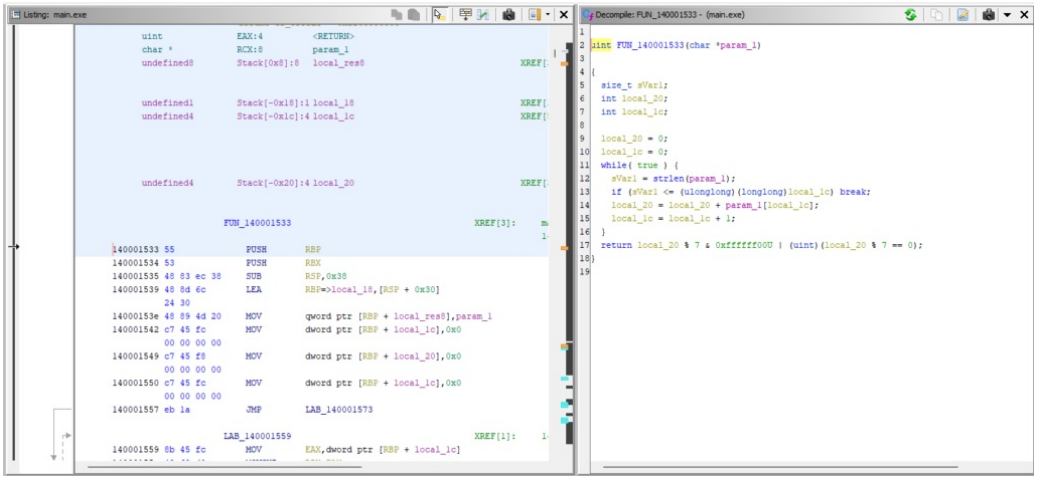


Análisis del código

Aquí es donde empieza lo divertido y donde podemos empezar a analizar la función en concreto. De entrada, observando el código en C, podemos comprobar que el programa verifica el valor retornado por la función FUN_140001533. Si es 0 literal, la clave es incorrecta. Si es cualquier otra cosa, la clave es verdadera. Esto se puede traducir como:

```
if(uVar1){
// Código aceptado
}else{
// Código no válido
}
```

Ya podemos empezar a estudiar cómo saltarnos esa verificación. Así que empezaremos por analizar la función FUN_140001533. Si hacemos doble click a cualquiera de las llamadas a esa función, Ghidra nos mandará al código de la función.



Observamos el siguiente código:

```
uint FUN_140001533(char *param_1)

{
  size_t sVar1;
  int local_20;
  int local_1c;

  local_20 = 0;
  local_1c = 0;
  while( true ) {
    sVar1 = strlen(param_1);
    if (sVar1 <= (ulonglong)(longlong)local_1c) break;
    local_20 = local_20 + param_1[local_1c];
    local_1c = local_1c + 1;
  }
  return local_20 % 7 & 0xffffff00U | (uint)(local_20 % 7 == 0);
}
```

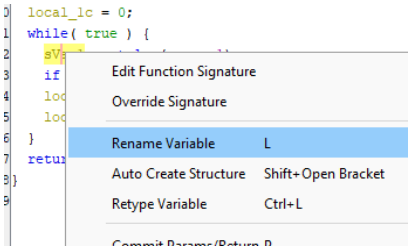
Aunque parezca un código algo complejo, no lo es tanto teniendo siempre en cuenta que es una traducción literal del código desensamblado. Podemos apreciar tres variables y un parámetro, y sabemos (o deberiamos de saber si nos metemos en ésto) que cada una de esas variables se almacenan en la pila de la función, mientras que el parámetro puede estar en la pila o en uno de los registros del procesador, en función de la arquitectura (x86 o x86_64) y de la convención usada. En cualquier caso, ésto es algo que puedes apreciar si analizas el desensamblado antes de la llamada a la función.

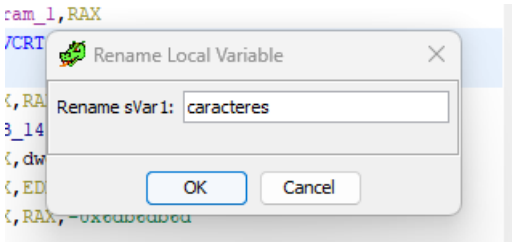
local_20 y local_1c se inicializan a 0, por tanto (y porque nos lo dice Ghidra) sabemos que son al menos íntegros. Despues se ejecuta un búcle infinito. El código sugiere que el búcle se ejecuta hasta que sVar1 sea menor o igual que local_1c. Podemos apreciar también que local_1c incrementa en cada ciclo. sVar1 recoge el valor de retorno de strlen(param_1). Podemos deducir que:

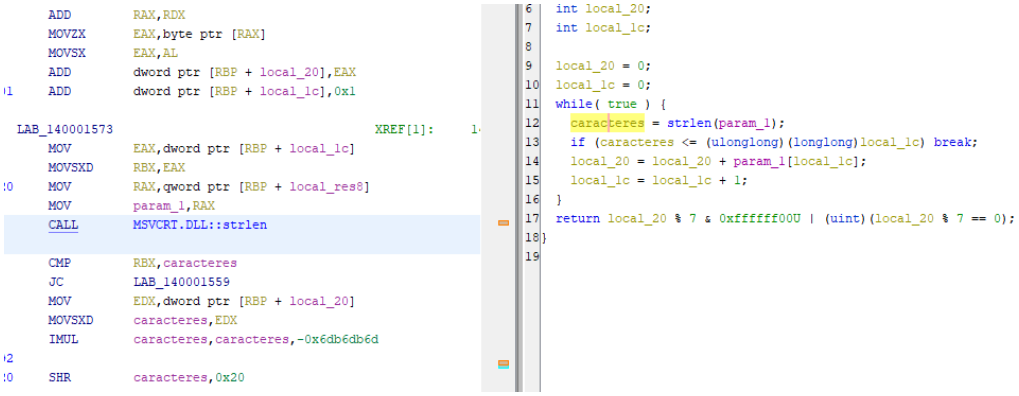
- param_1 es un string (una cadena de caracteres, o texto) con la contraseña.
- sVar1 contiene la longitud en caracteres del texto.
- local_1c es un contador de ciclos. Por cada ciclo ejecutado, se incrementa en 1.
- Cuando local_1c es mayor o igual al número de caracteres, el programa sale del búcle. Por lo tanto, está iterando a traves de los caracteres del texto.

Renombrando variables y funciones

Podemos dar nombre a éstas variables para hacer el código más fácil de comprender. Podemos hacer click derecho sobre cualquiera de las referencias a cada variable y seleccionar «Rename Variable» para renombrarla.







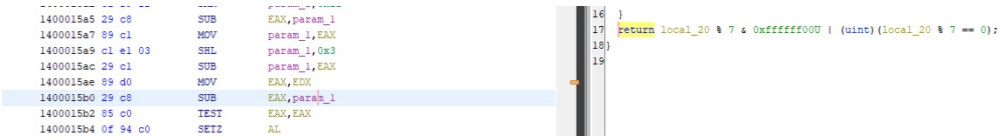
La vista del código se actualizará para reflejar el nuevo nombre de la variable. Podemos hacer lo mismo con el resto.

Por último, sólo queda ver qué es lo que hace el búcle con los caracteres. Si observamos la última linea de código que nos queda en él, podemos observar que suma el valor del caracter (el código ASCII del caracter) a la variable local_20. Por tanto, la variable local_20 contiene la suma de valores de los caracteres del texto. El búcle símplemente hace una suma de todos los caracteres del texto de la contraseña.

¿Y qué hace el programa con esa suma? En el valor de retorno se aprecia como aplica el módulo de 7 a la suma. Es decir, comprueba si la suma es divisible entre 7. Si lo es (local_20 % 7 == 0), el valor retornado es mayor que 0. En caso contrario, el valor

retornado es 0.

¡Ojo al código descompilado! En ese código se muestra que el valor retornado puede ser local_20%7==0 o local_20 % 7 & 0xffffff00U. En éste caso el módulo de local_20 y 7, y una máscara 0xffffff00U. Esa operación siempre va a dar resultado 0, siempre y cuando el resultado del módulo quepa en 16 bits. El valor de retorno depende sólo de la última parte, local_20 % 7 == 0.



En principio, ésta descompilación es muy confusa. No tiene sentido aplicar una operación cuyo resultado es siempre 0. Pero cobra sentido cuando entiendes que es una interpretación literal de lo que hace el código descompilado. La instrucción TEST aplica una operación AND entre dos operandos, en éste caso entre EAX y si mismo. Si EAX es 0, entonces ZF (zero-flag) se establece a 1. En caso contrario, es 0.

Por eso es SIEMPRE importante prestar atención a lo que hace el código desensamblado y no centrarse sólo en la descompilación, ya que todo el algoritmo en el que se calcula el módulo de local_20 y 7 se omite y se simplifica en el código descompilado de una manera que puede resultar confusa.

Podemos concluir entonces, que ésta función realiza la tarea de verificar la contraseña introducida. Tras cambiar el nombre de la función y cada variable, el resultado es el siguiente:

Decompile: verificacion - (main.exe)

```
1
2  uint verificacion(char *texto)
3
4  {
5    size_t caracteres;
6    int suma_caracteres;
7    int contador;
8
9    suma_caracteres = 0;
10   contador = 0;
11   while( true ) {
12     caracteres = strlen(texto);
13     if (caracteres <= (ulonglong)(longlong)contador) break;
14     suma_caracteres = suma_caracteres + texto[contador];
15     contador = contador + 1;
16   }
17   return suma_caracteres % 7 & 0xffffff00U | (uint)(suma_caracteres % 7 == 0);
18 }
```

Cálculo de las claves

Sabiendo ésto, ya podemos empezar a buscar qué contraseñas son válidas en el programa. Si la suma de los caracteres ASCII es divisible entre 7, entonces la contraseña será válida.

000	NUL	033	!	066	B	099	c	132	ä	165	Ñ	198	ã	231	þ
001	Start Of Header	034	"	067	C	100	d	133	à	166	ª	199	Ã	232	Þ
002	Start Of Text	035	#	068	D	101	e	134	å	167	º	200	╚	233	Ú
003	End Of Text	036	$	069	E	102	f	135	ç	168	¿	201	╔	234	Û
004	End Of Transmission	037	%	070	F	103	g	136	ê	169	®	202	╩	235	Ù
005	Enquiry	038	&	071	G	104	h	137	ë	170	¬	203	╦	236	ý
006	Acknowledge	039		072	H	105	i	138	è	171	½	204	╠	237	Ý
007	Bell	040	(	073	I	106	j	139	ï	172	¼	205	═	238	¯
008	Backspace	041	)	074	J	107	k	140	î	173	¡	206	╬	239	´
009	Horizontal Tab	042	*	075	K	108	l	141	ì	174	«	207	¤	240	-
010	Line Feed	043	+	076	L	109	m	142	Ä	175	»	208	ð	241	±
011	Vertical Tab	044	,	077	M	110	n	143	Å	176	░	209	Ð	242	‗
012	Form Feed	045	-	078	N	111	o	144	É	177	▒	210	Ê	243	¾
013	Carriage Return	046	.	079	O	112	p	145	æ	178	▓	211	Ë	244	¶
014	Shift Out	047	/	080	P	113	q	146	Æ	179	│	212	È	245	§
015	Shift In	048	0	081	Q	114	r	147	ô	180	┤	213	ı	246	÷
016	Delete	049	1	082	R	115	s	148	ö	181	Á	214	Í	247	¸
017	-- frei --	050	2	083	S	116	t	149	ò	182	Â	215	Î	248	°
018	-- frei --	051	3	084	T	117	u	150	û	183	À	216	Ï	249	¨
019	-- frei --	052	4	085	U	118	v	151	ù	184	©	217	┘	250	·
020	-- frei --	053	5	086	V	119	w	152	ÿ	185	╣	218	┌	251	¹
021	Negative Acknowledge	054	6	087	W	120	x	153	Ö	186	║	219	█	252	³
022	Synchronous Idle	055	7	088	X	121	y	154	Ü	187	╗	220	▄	253	²
023	End Of Transmission Block	056	8	089	Y	122	z	155	ø	188	╝	221	¦	254	■
024	Cancel	057	9	090	Z	123	{	156	£	189	¢	222	Ì	255	
025	End Of Medium	058	:	091	[	124	|	157	Ø	190	¥	223	▀		
026	Substitude	059	;	092	\	125	}	158	×	191	┐	224	Ó		
027	Escape	060	<	093	]	126	~	159	ƒ	192	└	225	ß		
028	File Seperator	061	=	094	^	127	⌂	160	á	193	┴	226	Ô		
029	Group Seperator	062	>	095	_	128	Ç	161	í	194	┬	227	Ò		
030	Record Seperator	063	?	096	`	129	ü	162	ó	195	├	228	õ		
031	Unit Seperator	064	@	097	a	130	é	163	ú	196	─	229	Õ		
032		065	A	098	b	131	â	164	ñ	197	┼	230	µ		

Tabla de caracteres ASCII (con valores en base 10)

Por ejemplo, supongamos la palabra «test». La operación sería

$$\frac{116 + 101 + 115 + 116}{7} = 64$$

Como la suma de los valores es divisible entre 7, la contraseña es una contraseña válida. Y si la introducimos en el programa, veremos que es así:

```
@DESKTOP-IKUIJA1 MINGW64 ~/src/test
$ ./main
Introduzca la clave: test
Clave correcta, acceso concedido
```

Sin embargo, si introducimos por ejemplo «testing»:

$$\frac{116 + 101 + 115 + 116 + 105 + 110 + 103}{7} = \frac{766}{7} = 109.4285...$$

La suma no es divisible entre 7, y por tanto no es una contraseña válida:

```
@DESKTOP-IKUIJA1 MINGW64 ~/src/test
$ ./main
Introduzca la clave: testing
Clave incorrecta!
```

Aunque ya tenemos las claves, aún podemos hacer algo más con el programa. Dado que Ghidra nos permite modificar el programa, podríamos aprovechar para hacer que el programa retorne siempre 1, de modo que siempre se ejecutará el bloque de la contraseña válida en el bloque 'if' del programa.

Modificar el código ya ensamblado

Ghidra nos permite reensamblar cualquier instrucción del programa, pero hay una serie de limitaciones muy importantes a tener en cuenta. Lo primero y más importante es que, en principio, no podemos añadir instrucciones al programa. Es decir, estamos limitados al espacio que tenemos (en bytes). **Esto quiere decir que sólo podemos sobreescribir bytes en el espacio que tenemos**. Y cada instrucción tiene un tamaño en bytes, que son los códigos de operación (opcodes) de la instrucción.

Por otra parte, si el tamaño de la nueva instrucción que introduzcamos supera el de la original, **sobreescribiremos las siguientes instrucciones**. Como ya puedes imaginar, habrá instrucciones que podamos sobreescribir, e instrucciones que no. Por eso hay que planificar muy cuidadosamente donde y qué vamos a escribir.

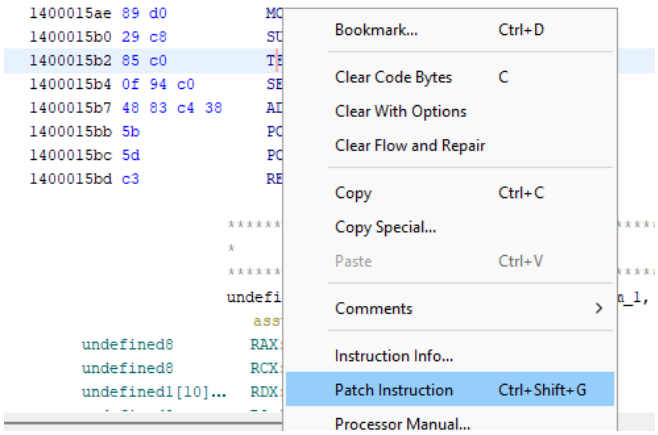
Si analizamos el final del código donde se realiza el módulo:

```
1400015b0 29 c8          SUB     EAX,param_1
1400015b2 85 c0          TEST    EAX,EAX
1400015b4 0f 94 c0       SETZ    AL
1400015b7 48 83 c4 38    ADD     RSP,0x38
1400015bb 5b             POP     RBX
1400015bc 5d             POP     RBP
1400015bd c3             RET
```

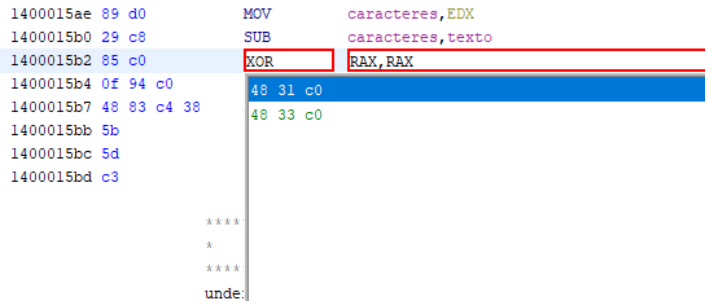
Las tres primeras instrucciones son donde se finaliza la comprobación. Mientras que las cuatro últimas instrucciones restablecen la pila y retornan al punto de la llamada. Esas cuatro últimas instrucciones no las podemos sobreescribir, pero sí las tres primeras, de modo que tenemos un total de 7 bytes en nuestra mano para hacer lo que queramos.

Por convención en éste caso, y dado que el programa es de 64 bits, el valor de retorno se almacena en RAX. El registro RAX es de 64 bytes. La parte menos significante de RAX es EAX, de 32 bits. Y a su vez EAX se divide en otros dos registros de 16 bits: AH (parte más significante de EAX) y AL (menos significante de EAX). Queremos que el valor de retorno sea 1, por lo que RAX tiene que ser 1 cuando se ejecute la instrucción RET.

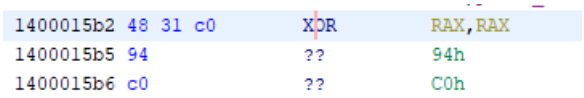
Tras ejecutar la instrucción TEST, no sabemos en qué estado estará RAX ya que depende de todo lo anterior. Así que podemos empezar por modificar esa instrucción. Tenemos 5 bytes. Podriamos entonces establecer RAX a 0, e incrementarlo en 1 en la siguiente instrucción. Si hacemos click derecho sobre TEST y seleccionamos «Patch Instruction»:



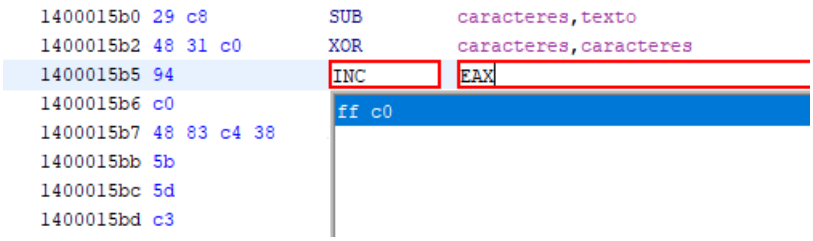
Podremos borrar el contenido de la instrucción e introducir la que queramos. En éste caso XOR RAX,RAX.



Ghidra nos muestra los tres opcodes que requiere la operación. Podemos elegir entre dos combinaciones posibles, en éste caso es irrelevante. Como la instrucción original TEST sólo usa dos bytes, la nueva instrucción tomará uno de los bytes de la siguiente instrucción, dejandonos sólo con dos libres.



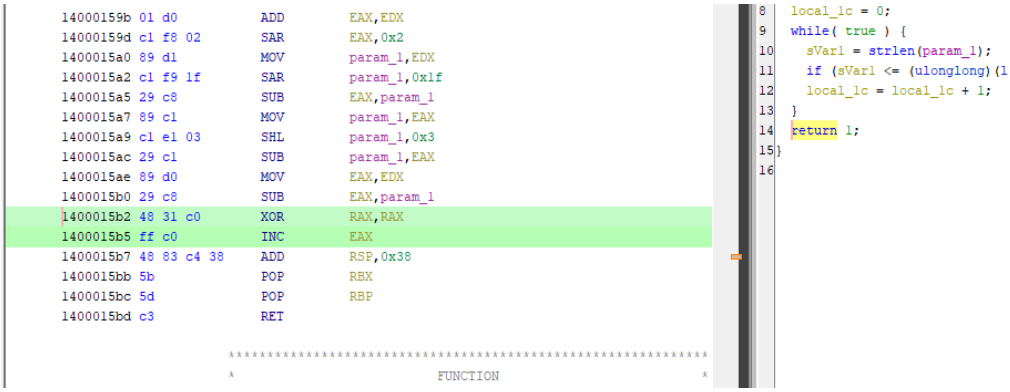
Para sobreescribir los dos siguientes, hacemos click derecho sobre el siguiente byte y seleccionamos de nuevo «Patch Instruction». En éste caso escribimos la instrucción INC EAX, que ocupa sólo dos bytes.



La operación XOR (OR exclusiva) entre un registro y sí mismo tiene de resultado 0, poniendo el registro RAX en éste caso a 0. INC EAX incrementa el valor de EAX en 1.

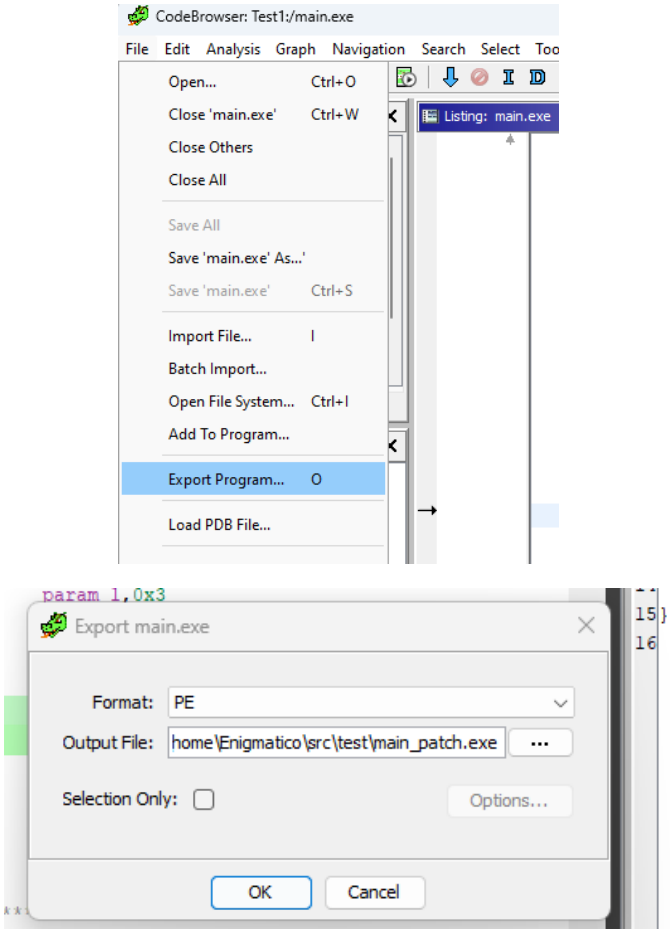
Pero EAX es una subdivisión de RAX, por tanto es lo mismo que poner RAX a 1, pero usando un byte menos que si usamos una instrucción de 64 bits (INC RAX). Igualmente podríamos incrementar AL y ocuparía dos bytes.

Podemos ver los cambios reflejados en el programa y en la descompilación, donde el valor de retorno ahora es 1.



Reensamblar el programa

Ya sólo queda reensamblar el ejecutable. Podemos hacerlo desde el menú File > Export Program.



Lo exportamos como archivo PE donde queramos tenerlo y pulsamos sobre OK. Esto exportará el archivo .exe con el código modificado. **Acuerdate de desmarcar la opción «Selection only» si la tienes marcada**, y pulsa Ok.

Ahora puedes probar de nuevo el programa y comprobar que el acceso te será

concedido sin importar la contraseña que introduzcas:

Comparación con el código original

El código original es el siguiente:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char* introducirClave()
{
    char* buf = (char*)calloc(1024,1);
    scanf("%s", buf);
    return buf;
}

char verificacion(char* passwd)
{
    int i = 0, vc = 0;
    for(i = 0; i < strlen(passwd); i++)
    {
        vc += (int)passwd[i];
    }
    return vc%7==0;
}

int main(int argc, char** argv)
{
    char* passwd = 0L;
    char vc = 0;
    printf("Introduzca la clave: ");
    passwd = introducirClave();

    vc = verificacion(passwd);

    if(vc)
    {
        printf("Clave correcta, acceso concedido\n");
    }else{
        printf("Clave incorrecta!");
    }

    free(passwd);
    return 0;
}
```

Hay varias cosas importantes que mencionar. Primero, **que hay una vulnerabilidad muy importante en éste código** (te dejo que la descubras tú si quieres, es bastante obvia si sabes C). La segunda, ¿Recuerdas éste código desensamblado?

```
while( true ) {
    sVar1 = strlen(param_1);
    if (sVar1 <= (ulonglong)(longlong)local_1c) break;
    local_20 = local_20 + param_1[local_1c];
    local_1c = local_1c + 1;
  }
```

Esta parte se corresponde con

```
for(i = 0; i < strlen(passwd); i++)
    {
        vc += (int)passwd[i];
    }
```

En éste caso, el compilador no ha optimizado esta parte del código. Por tanto, por cada ciclo ejecuta la función strlen, incluso cuando en éste caso el número de caracteres es constante y no cambia.

Por eso es **importante tener en cuenta éstos detalles a la hora de escribir código en C**. Si el resultado de una instrucción va a ser constante a lo largo del búcle, dejala fuera del búcle y usa sólo la variable con el valor de retorno de la función.

El resto del código se corresponde con lo que se ha mostrado en el desensamblado del ejecutable. El programa solicita una clave, comprueba que la suma del valor de los caracteres sea divisible entre 7, y si lo es da por buena la clave.

ANTERIOR

Acerca de la nueva ley transgénero (Y sobre la disfória de género)

SIGUIENTE

Estoy hasta las narices de la web moderna

Entradas Recientes

- ¿Necesito saber matemáticas para aprender informática?
- ¿Es el fín de los discos duros tradicionales?

Categorías

Actualidad | Android | Básicos | Ciberseguridad | Clima | Criptografía | Electronica | Emulación / Virtualización | FOSS | Hacking | Hardware | Informática | Internet | Juegos | Opinion | Otros | Personal | Privacidad | Programación | Tecnología | Time Machine | Tutoriales

RSS

Subscribirse al feed RSS