RandomWeird-Single [x64DBG][MVC++][SERIAL][LUISFECAB]

Software	**RandomWeird-Single by nextco**
Protección	**Serial.**
HERRAMIENTAS	Windows 7 Home Premium SP1 x32 Bits (SO donde trabajamos) X64DBG (Feb 14 2018) Keygener Assistant v2.1.0 Microsoft Visual Studio 2017 DESCARGAR HERRAMIENTAS DESCARGAR TUTO+ARCHIVOS
SOLUCIÓN	SERIAL. (FUERZA BRUTA-KEYGEN)
AUTOR	**LUISFECAB**
RELEASE	Mayo 8 2019 [TUTORIAL 014]

INTRODUCCIÓN

Tenía pendiente hacer este tutorial que sin las explicaciones de **nextco** creo que no hubiera podido completar el reto o me hubiera tomado años en resolverlo porque fijo lo hubiera abandonado y echado en el olvido.

Este reto es la versión sencilla que **nextco** me compartió de forma exclusiva para que pudiera entender mejor cómo resolver el **Crackme**, por eso lo llamé <**RandomWeird-Single**> porque la versión original aunque es muy similar no funciona con el serial que hallamos aquí y que no he resuelto aún; prefiero mejor escribir este tutorial y disfrutar de esta pequeña victoria que me ha enseñado mucho.

Ya lo he dicho en anteriores tutoriales, que por falta de conocimiento es que termino siendo derrotado por estos retos, pero que cuando supe por dónde era la cosa pude resolverlo. Aquí aprendí algo llamado **Generador lineal congruencial** (**GLC**) y no es más que la aplicación de la teoría de **Números Pseudoaleatorios**.

Para terminar esta pequeña introducción, quiero como siempre saludar a la lista de **CracksLatinoS** y por supuesto a mis amigos de **PeruCrackerS**.

ANALISIS INICAL

Aquí no hay mucho que analizarle al <**RandomWeird-Single**>. El **Crackme** está hecho en **MVC++** y nos pide que ingresemos un valor.

Lo que haremos en esta parte es profundizar en la teoría de **Números Pseudoaleatorios** enfocado en programación, claro resumido con mis palabras.

Teoría:

Si nos referimos a Números Aleatorios en programación caemos en un término mal utilizado porque en realidad los números no se generan aleatoriamente, si no, que se originan a partir de algoritmos que dan valores predeterminados, conociendo ciertos valores iniciales, y es por eso que nos referimos a Números Pseudoaleatorios.

Estos algoritmos se clasifican en algoritmos no congruenciales y congruenciales. Los algoritmos no congruenciales que podemos nombrar son: cuadrados medios, productos medios y multiplicador constante. Entre los algoritmos congruenciales se encuentran los algoritmos lineales y no lineales, como algoritmos lineales podemos referirnos al algoritmo congruencial lineal, multiplicativo y aditivo; y para los algoritmos no lineales, como el algoritmo de Blum, Blum y Shub, y el congruencial cuadrático[1].

Los diferentes lenguajes de programación han optado por implementar el algoritmo congruencial lineal para generar sus valores aleatorios.

Este algoritmo congruencial fue propuesto por D. H. Lehmer en 1951. El algoritmo congruencial lineal genera una secuencia de números enteros por medio de la siguiente ecuación recursiva:

$$x_{i+1} = (ax_i + c) mod(m) \quad i = 0, 1, 2, 3, \dots, n$$

[1] Eduardo García Dunna, H. G. (2013). Simulación y análisis con ProModel (Segunda ed.). México: PEARSON. Página 24.

donde x_0 es la semilla, a es la constante multiplicativa, c es una constante aditiva, y m es el módulo. $x_0 > 0$, $a > 0$,$c > 0$ y $m > 0$ deben ser números enteros[2].

Una explicación más adecuada con un énfasis en programación lo puedes leer en Wikipedia buscando por **Generador lineal congruencial.**

En la siguiente imagen podemos ver los diferentes valores iniciales del algoritmo para diferentes lenguajes y entre esos el que nos interesa el de **MVC++.**

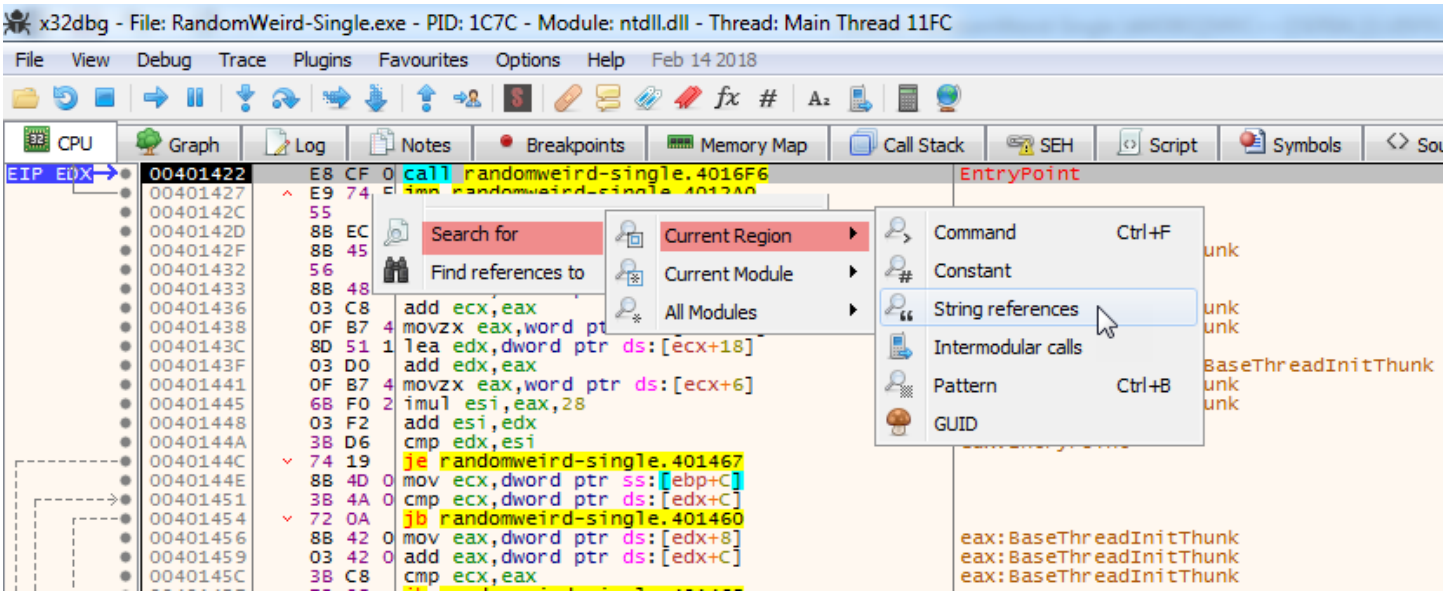
Fuente	*m*	(multiplicador) *a*	(incremento) *c*	Bits de salida en *rand()* o *Random(L)*
Numerical Recipes	2^{32}	1664525	1013904223	
Borland C/C++	2^{32}	22695477	1	bits 30..16 en *rand()*, 30..0 en *lrand()*
glibc (usado por GCC)[7]	2^{31} - 1	1103515245	12345	bits 30..0
ANSI C: Watcom, Digital Mars, CodeWarrior, IBM VisualAge C/C++[8] C99, C11: Suggestion in the ISO/IEC 9899[9]	2^{31}	1103515245	12345	bits 30..16
Borland Delphi, Virtual Pascal	2^{32}	134775813	1	bits 63..32 de *(semilla * L)*
Turbo Pascal	2^{32}	134775813 ($0x8088405_{16}$)	1	
Microsoft Visual/Quick C/C++	2^{32}	214013 ($343FD_{16}$)	2531011 ($269EC3_{16}$)	bits 30..16
Microsoft Visual Basic (6 and earlier)[10]	2^{24}	1140671485 ($43FD43FD_{16}$)	12820163 ($C39EC3_{16}$)	
RtlUniform from Native API[11]	$2^{31} - 1$	2147483629 ($7FFFFFED_{16}$)	2147483587 ($7FFFFFC3_{16}$)	
Apple CarbonLib, C++11's `minstd_rand0`[12]	$2^{31} - 1$	16807	0	ver MINSTD
C++11's `minstd_rand`[12]	$2^{31} - 1$	48271	0	ver MINSTD
MMIX by Donald Knuth	2^{64}	6364136223846793005	1442695040888963407	
Newlib, Musl	2^{64}	6364136223846793005	1	bits 63...32
VMS's **MTH$RANDOM**,[13] old versions of glibc	2^{32}	69069 ($10DCD_{16}$)	1	
.Random en Java, POSIX [lr]rand48, glibc [lr]rand48[_r]	$2^{48} - 1$	25214903917 ($5DEECE66D_{16}$)	11	bits 47...16
`random0`[14][15][16][17][18] Si X_n es par, entonces X_{n+1} será impar, y vice versa—el bit más bajo oscila a cada paso.	$134456 = 2^3 7^5$	8121	28411	$\frac{X_n}{134456}$
POSIX[19] [jm]rand48, glibc [mj]rand48[_r]	2^{48}	25214903917 ($5DEECE66D_{16}$)	11	bits 47...15
POSIX [de]rand48, glibc [de]rand48[_r]	2^{48}	25214903917 ($5DEECE66D_{16}$)	11	bits 47...0
cc65[20]	2^{23}	65793 (10101_{16})	4282663 (415927_{16})	bits 22...8
cc65	2^{32}	16843009 (1010101_{16})	826366247 (31415927_{16})	bits 31...16
Anteriormente de uso común: RANDU[6]	2^{31}	65539	0	

Ya con esta teoría podemos empezar nuestro ataque y ver cómo se implementa en **MVC++** la generación de números aleatorios (Pseudoaleatorios) con su función **rand().**

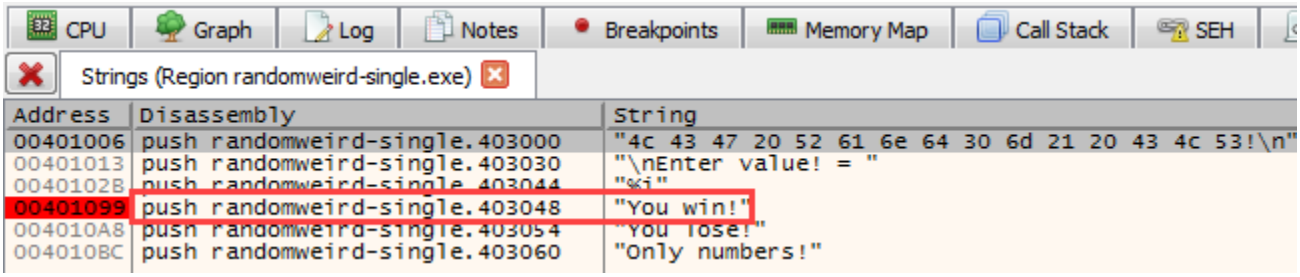
[2] Eduardo García Dunna, H. G. (2013). Simulación y análisis con ProModel (Segunda ed.). México: PEARSON. Página 27.

AL ATAQUE

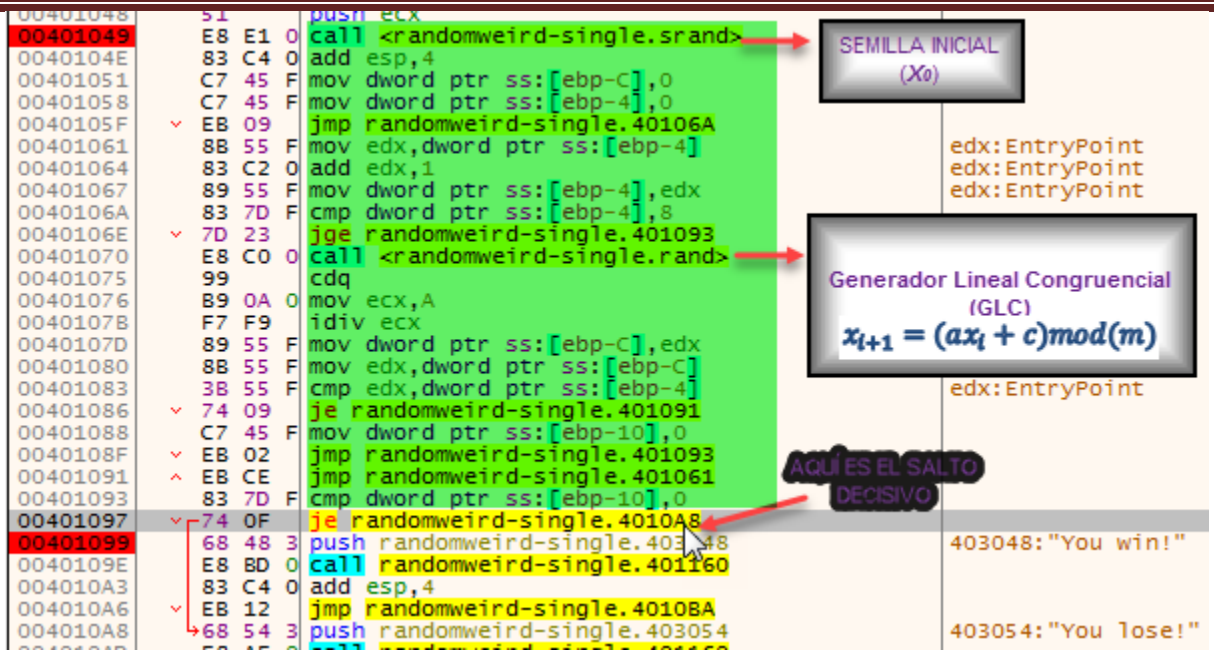
Carguemos el <**RandomWeird-Single**> en el **x64DBG** y buscamos en <**String references**> para ver si desde ahí podemos saltar a nuestra <**ZONA CALIENTE**>.



Perfecto, podemos ver que hay strings que nos cargarán buenas o malas noticias según le peguemos al serial correcto.



Me voy por el <**CHICO BUENO**> que dice "**You win!**" y desde ahí revisaré subiendo para ver desde dónde empieza a realizar cálculos.

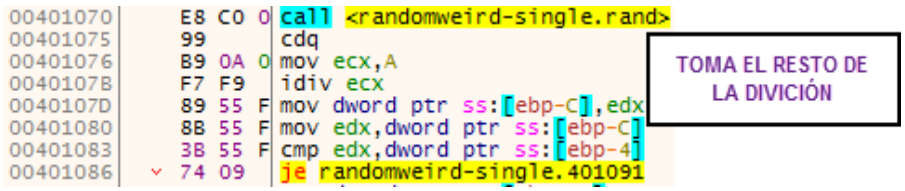


Tenemos el salto decisivo en la dirección **00401097** y lo **RESALTADO EN VERDE** es donde haré los cálculos con nuestro valor ingresado que viene siendo la **Semilla inicial** (X_0) y que en el programa es almacenada por la función **srand()** que se ejecuta en la dirección **00401049**. Luego se origina nuestro número aleatorio con la función **rand()** en la dirección **00401070**, ahí es que entra en acción el **Generador Lineal Congruencial**. Revisando lo que sucede en esa **ZONA**, pasa lo siguiente: nuestro valor ingresado (**Semilla inicial**) debe generarnos números terminados en 0,1,2,3,…7.

Primer número ---- XXXXXXXX0

Segundo número ---- XXXXXXXX1

Tercer número ---- XXXXXXXX2

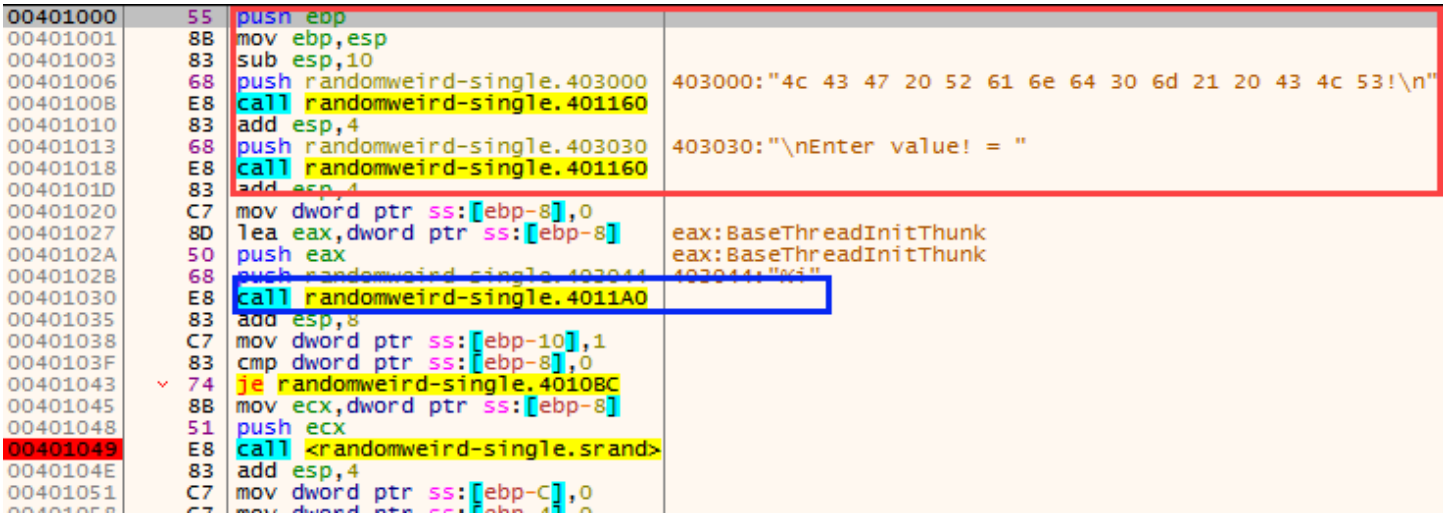


Después hace una división en **0040107B IDIV ECX**. Podemos ver en la instrucción anterior **00401076 MOV ECX,A** que vamos a dividir por **0xA** que es **10**, y con eso el resto de la división será el número en que termina el valor generado, y aquí esta lo más importante porque va comparando nuestros restos con un contador seteado a 0. Nuestro primer resto debe ser **0**, y para que eso suceda nuestro número aleatorio debe terminar en **0**, y como el contador es **0**, se cumple la comprobación, entonces el contador aumenta en **1** y volvemos a generar otro número aleatorio que debe terminar en **1** para poder obtener el resto igual a **1**, y así de esa forma cumplir la condición y no saltar al <**CHICO MALO**>. Lo anterior se repite hasta que el contador llegue a **8**. Todo eso ocurre en un **LOOP**. Ahora puede sonar un poco enredado pero cuando ingresemos un valor y lo probemos entenderemos mejor.

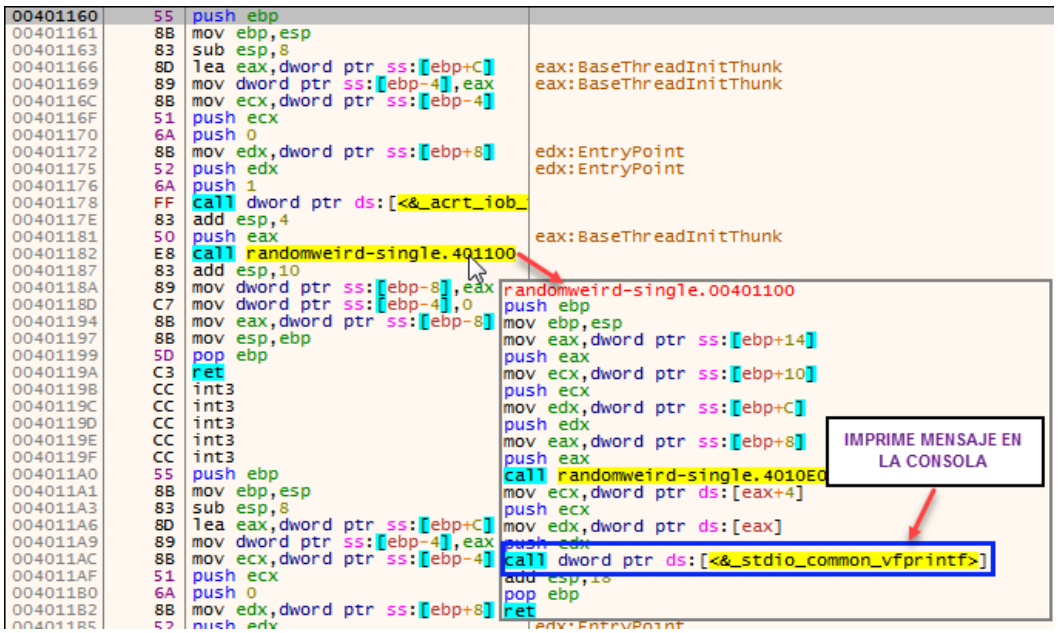
Lo que acabamos de explicar en el párrafo anterior es lo que debemos programar en nuestro **KeyGen** para hallar la **Semilla inicial**, que viene siendo el **Valor correcto** que nos pide el <**RandomWeird-Single**> que ingresemos.

Ya voy conociendo cómo se comporta estas aplicaciones de consola hechas de forma sencilla, a las que no le han puesto nada malicioso para confundirnos. Podemos ir

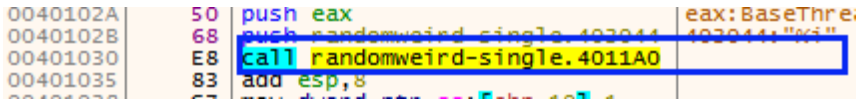
donde se carga el valor ingresado a puro ojo, ya de memoria. Si subimos un poco más, al inicio de ese procedimiento.



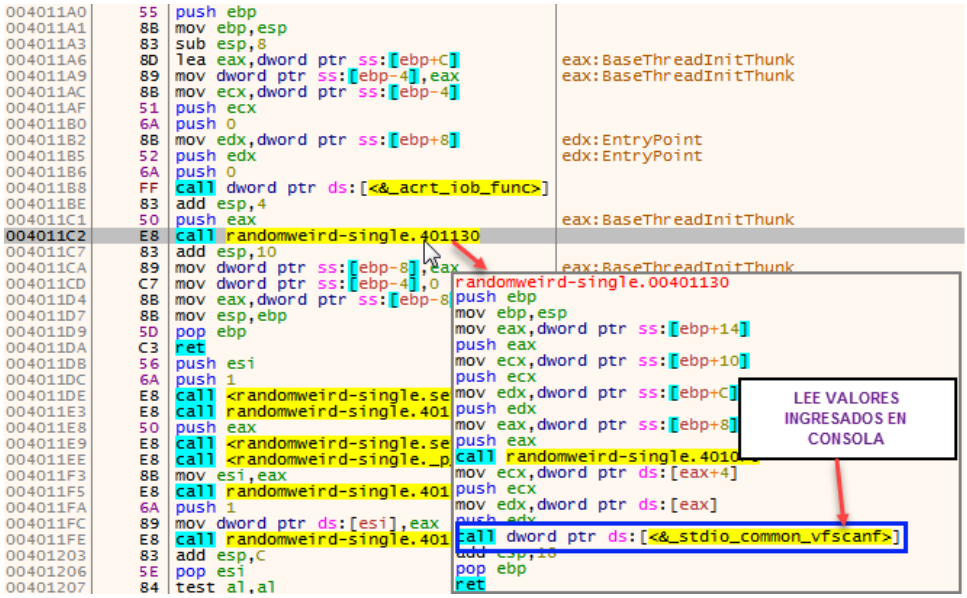
Podemos ver que va a mostrar o cargar texto en la consola, y sabemos que **MVC++** tiene la función **printf()** o similares. Miremos las direcciones **0040100B** y **00401018**, esas dos van al mismo `call randomweird-single.401160`, y desde ese se inicia el procedimiento para mostrar el texto en consola. Si entramos y lo revisamos.



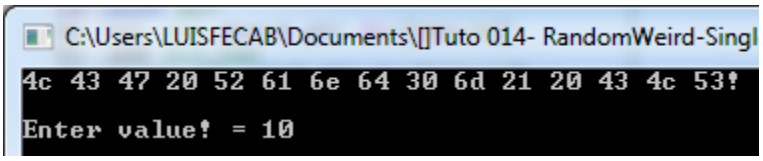
Ahí se ve claramente que todo eso es para imprimir el mensaje en la consola. Esto mismo ocurre cuando nosotros ingresamos el valor que nos piden el **Crackme**, solo que ahora usará una función para capturar el dato ingresado y será la función **scanf()** o similar. Volvamos al inicio en donde resaltamos el `call randomweird-single.4011A0` en el **RECUADRO AZUL** de la dirección **00401030**.



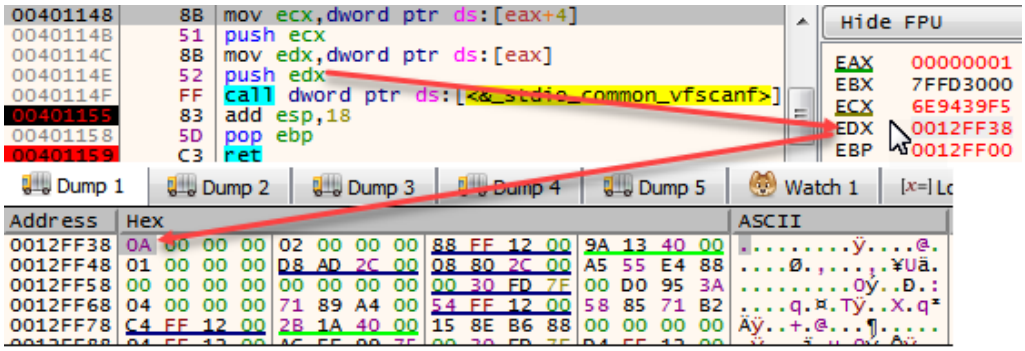
Entrando a ese **CALL** que es prácticamente el mismo procedimiento, solo que utilizará la función **scanf()**.



Pongamos un <**BREAKPOINT**> después de **0040114F** call dword ptr ds:[<&_stdio_common_vfscanf>]. Para detenernos ahí y ver en dónde guardo nuestro valor ingresado.

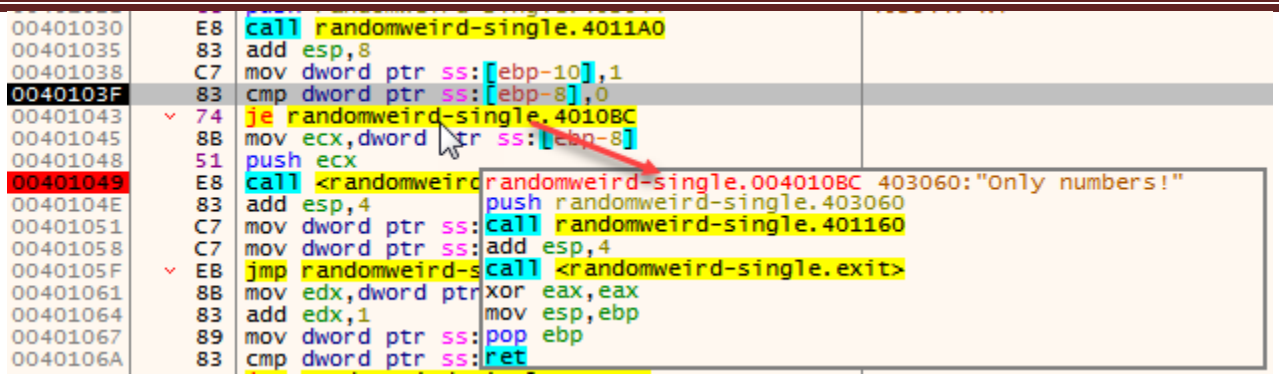


Ingreso **10**=0xA, y veamos donde guardó nuestro número.



Después de salir del **0040114F** call dword ptr ds:[<&_stdio_common_vfscanf>], si vemos en los registros **EDX** **0012FF38**, guarda nuestro **Valor ingresado**. No sé, me parece que toda esta explicación como que sale sobrando cuando recuerdo el nivel que tienen en Reversing mis amigos de **CracksLatinoS** y **PeruCrackerS**, pero uno no sabe, puede que este tuto caiga en manos de personas que necesitan y le sirve esta explicación como por ejemplo mi hermanita…jejeje.

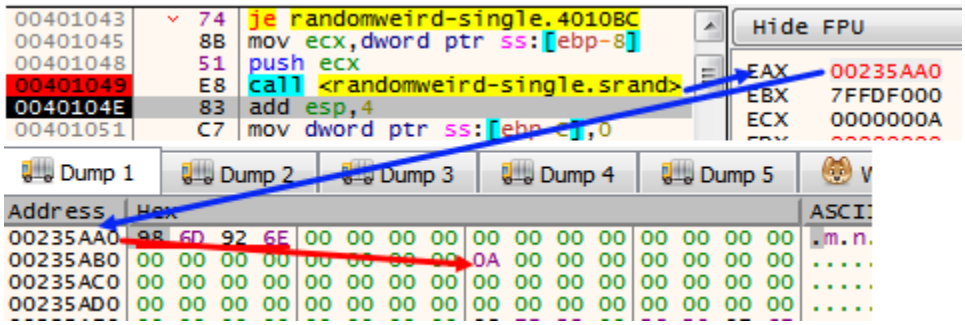
Continuemos traceando con <**F8**> hasta salir de todos esos procedimientos y llegar a la <**PARTE CALIENTE**> y detenemos el traceo en **0040103E**.



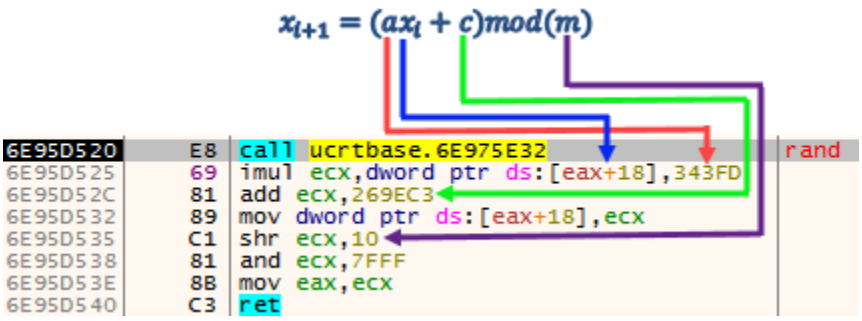
Si ingresamos un valor que no sea un número nos manda para afuera, así que solo números, no olvidarlo. Revisemos las siguientes instrucciones.

```
00401045 mov ecx,dword ptr ss:[ebp-8]    Mueve a ECX el valor ingresado.
00401048 push ecx                        Mueve al STACK el valor ingresado.
00401049 call <randomweird-single.srand> Guarda valor ingresado como nuestra Semilla inicial.
```

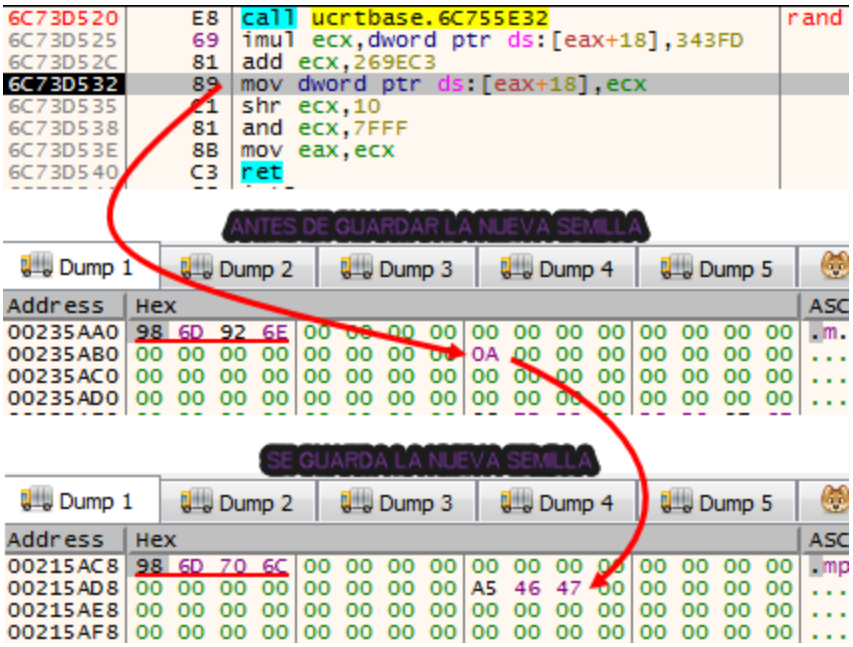
El **00401049** call <randomweird-single.srand> no hace otra cosa que guardar nuestro valor ingresado (**Semilla inicial**) en memoria para ser utilizada cuando se ejecute la función **rand()**. Recordar toda la teoría antes planteada. Pasemos con <**F8**> el call <randomweird-single.srand> y podemos observar que en **EAX** **00235AA0** queda una dirección que se comporta como un **Offset** o **Puntero** desde el cual permite llegar a nuestra **Semilla inicial**.



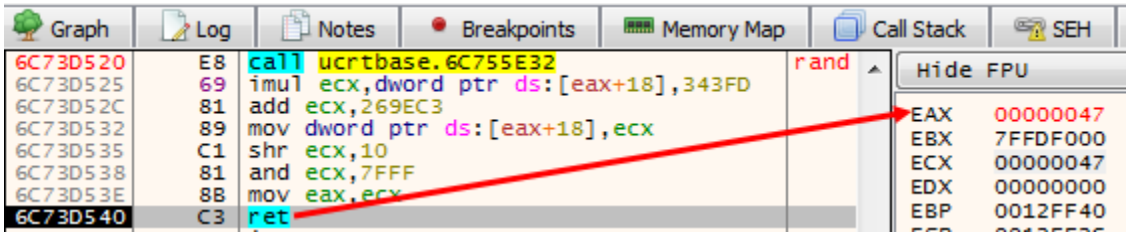
Nuestra **Semilla inicial** está en **[00235AA0+0x18]** o **[EAX+0x18]**. Recordemos esto porque así es como la función **rand()** coge nuestra **Semilla inicial**. Lleguemos a **00101070** call <randomweird-single.rand> y entremos con <**F7**> y sigamos hasta donde se ejecuta instrucción por instrucción función **rand()**.



Algo relevante, es que lo primero que se hace en un `call ucrtbase.6C755E32` y es solo para obtener nuestro **Offset** en **EAX** para cargar nuestra **Semilla inicial**. Sigamos traceando con <**F8**> para pasar ese **CALL** y lleguemos hasta `mov dword ptr ds:[eax+18],ecx`.



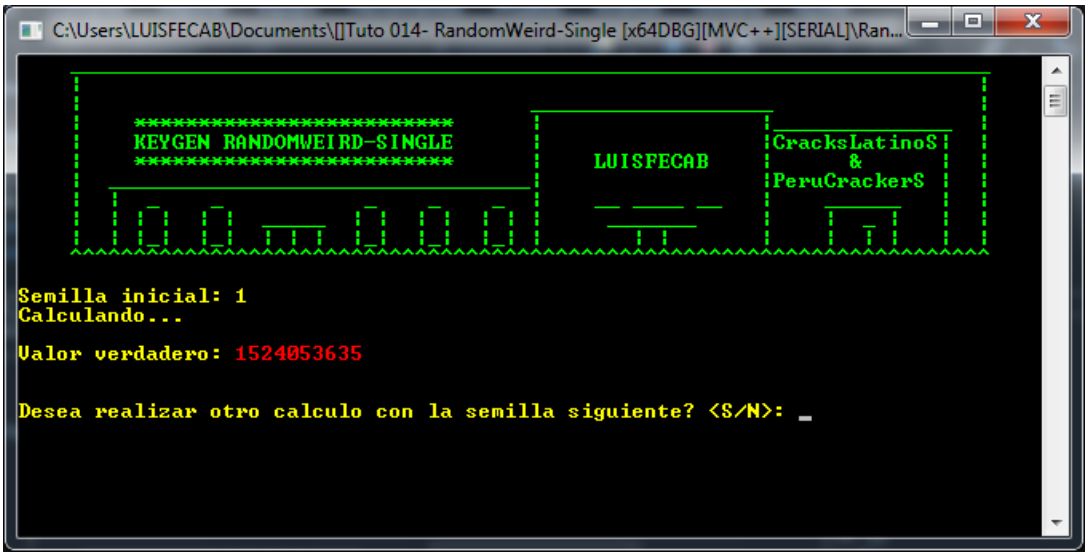
Podemos ver que se origina una **Nueva semilla** y con esa se hallará un nuevo número aleatorio cuando se llame de nuevo la función **rand()**, y así sucesivamente. Pues esa es la **Semilla** la que debemos hallar y que sabemos nos debe dar números que nos finalicen desde **0** hasta **7**. Según lo que pude leer y pude entender es que no hay un formula fija para hallar una **Semilla inicial** pero podemos programar un procedimiento que nos halle una mediante fuerza bruta, conociendo las condiciones de inicio. Traciemos hasta el **RET**.



Listo, ya tenemos nuestro primer número aleatorio en **EAX** **00000047**, **0x47=71**. Como vemos el número termina en **1** y ya con eso sabemos que no sirve porque **71mod10=1** y nosotros sabemos que el primer resto debe ser **0**.

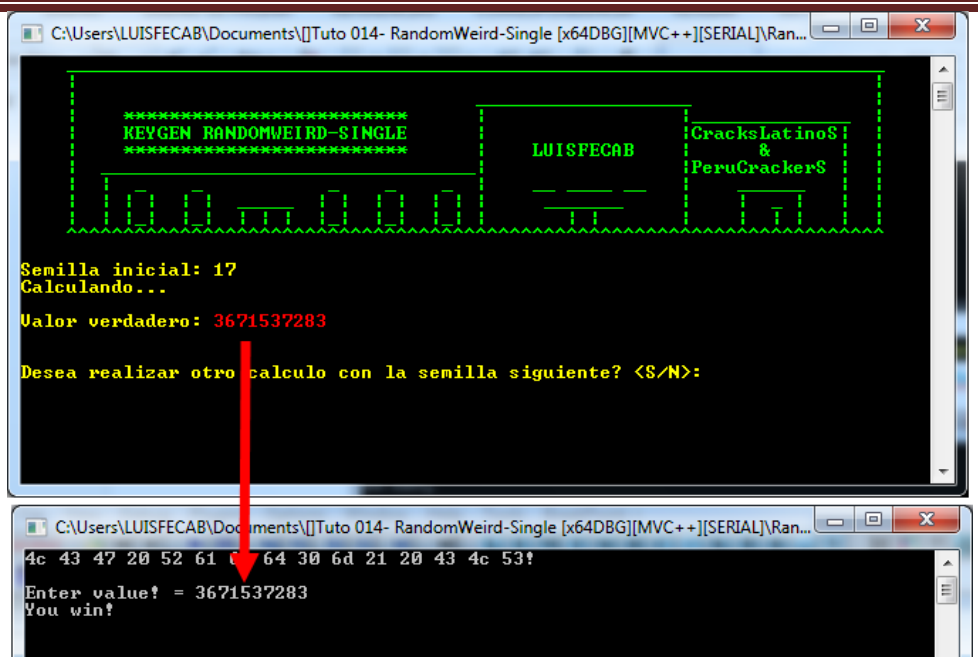
Supongo que deben haber diferentes formas de programar para hallar la **Semilla Inicial**. Yo opté por iniciar al cálculo siempre con **Semilla inicial=1** y si la **Semilla inicial** no me sirve, entonces se tome como **Nueva Semilla** la que se origina en la función **rand()**. Esto tiene una gran desventaja, y es que mi **KeyGen** es poco flexible ya que siempre tendré el mismo resultado al ejecutarlo y si sigo calculando nuevos valores, serán en el mismo sentido porque no he permitido ingresar una **Semilla inicial** si no que la va aumentando en **1**.

Ya hemos analizado todo y sabemos las condiciones iniciales de cómo se generan los números aleatorios en **MVC++** y también sabemos la condición del **Crackme** de **nextco** para poderlo vencer. Queda solo hacer mi **KeyGen** y que me halle un valor correcto.



Como ven mi **Semilla inicial: 1**, y por eso siempre tendré **Valor Verdadero: 1524053635**, pero lo importante era hallar un valor que venciera el **Crackme** <RandomWeird-Single>.

Perfecto, ya pudimos hallar un valor correcto. Si quisiera calcular un **Valor verdadero** para **Semilla inicial: 17** debo realizar más cálculos hasta llegar a **17**. No es cómodo pero sirve para su cometido. Otra cosa es que se nos repetirán Valores verdaderos, no es que hallemos un **Valor verdadero** por cada **Semilla inicial**; recordemos que es aquel número que nos permite tener números finalizados desde **0** a **7** y como es por **FUERZA BRUTA** pues se repetirán.


C:\Users\LUISFECAB\Documents\[]Tuto 014- RandomWeird-Single [x64DBG][MVC++][SERIAL]\Ran...
KEYGEN RANDOMWEIRD-SINGLE
LUISFECAB
CracksLatinoS
&
PeruCrackerS
Semilla inicial: 17
Calculando...
Valor verdadero: 3671537283
Desea realizar otro calculo con la semilla siguiente? <S/N>:

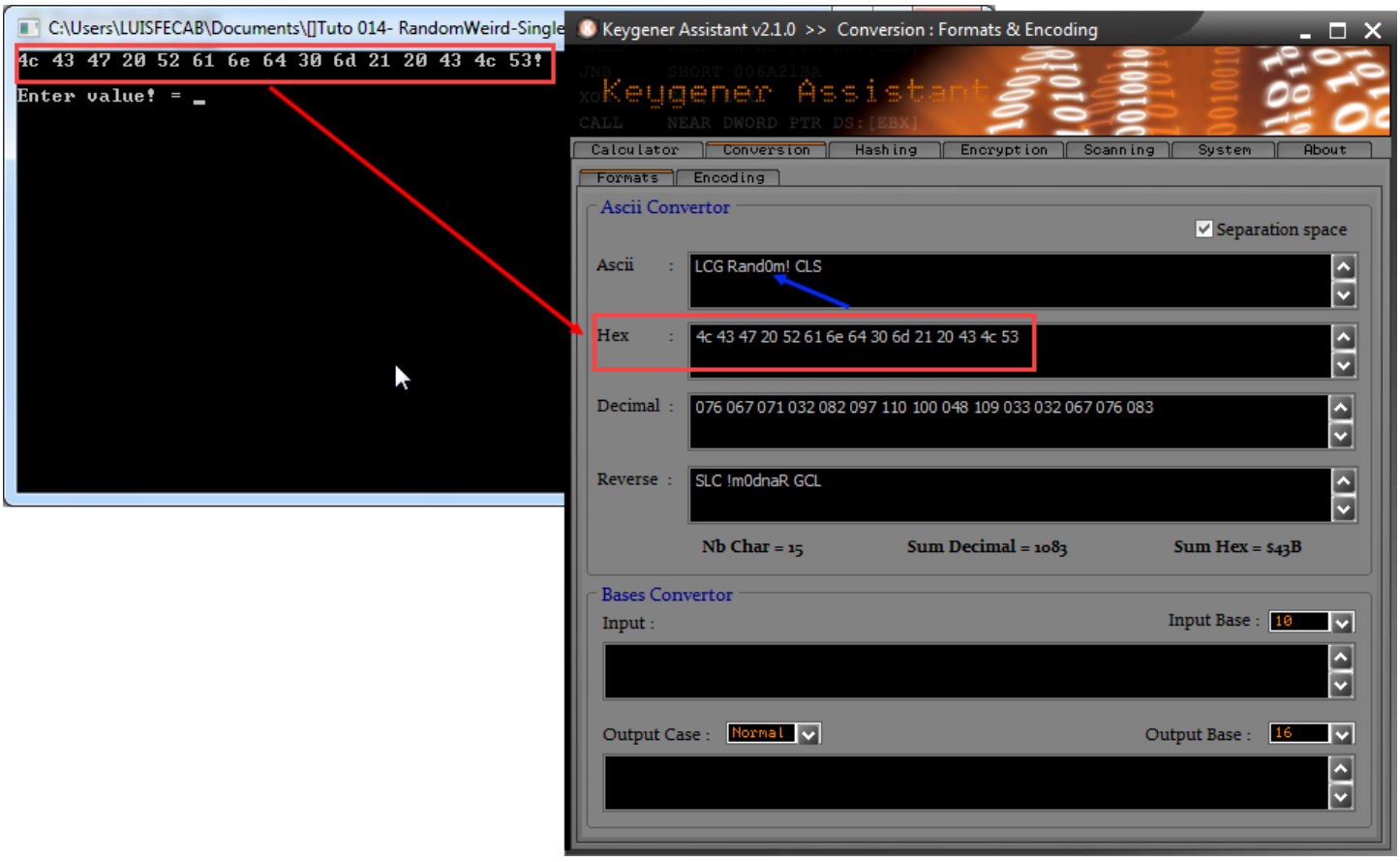
C:\Users\LUISFECAB\Documents\[]Tuto 014- RandomWeird-Single [x64DBG][MVC++][SERIAL]\Ran...
4c 43 47 20 52 61
64 30 6d 21 20 43 4c 53!
Enter value! = 3671537283
You win!

PARA TERMINAR

Ya vamos terminando, creo que podemos vencer el **Crackme** original <**RandomWeird**> con todo lo aprendido en este tutorial pero que lo enfrentaré después, porque con este he tenido suficiente, créanme cuando le digo eso. Cuando hagamos ese **Crackme** también trataremos de mejorar el **KeyGen** para que podamos ingresar una **Semilla inicial**, o un rango, o talvez otras opciones; luego veremos eso.

Este **Crackme** tiene lo suyo, recuerdo que **nextco** me dijo – ***Has este reto, te divertirás haciéndolo*** –. Yo venía muy vanidoso y creído porque había hecho otros y pensé que iba ser pan cómodo y resultó que el pan era yo, qué aterrizada me pegó este **Crackme** y me hizo recordar que en esto del **Reversing** uno debe tener humildad como muchas veces lo ha dicho el **Maestro Ricardo**.

Pistas que tiene el **crackme** que descubrí desde un inicio pero en mi ignorancia no supe aprovechar. Recordemos que se cargan unos valores hexadecimales que fijo traen un mensaje; para saber qué decía hice uso de la tool <**Keygener Assistant v2.1.0**>.



"**LCG Rand0m! CLS**", más claro no canta un gallo, **Lineal Congruential Generator (LCG)**.

Si hay algo incorrecto que he escrito o alguna observación para mejorar lo expuesto aquí, son bienvenidas porque entre más aprendemos, mucho mejor para mí y me ayudará

a hacer mejores tutoriales que contengan mejor información que en algún momento servirá a otros.

Amigos, esto es todo, "***hasta aquí me trajo el rio***". Saludos a todos los que en cierta forma me conocen y para los que me leen y no me han tratado también un saludo afectuoso.

Y qué dijeron que no me despido de mis grupos, no señor, no puedo irme sin despedirme de mi **PeruCrackerS** y **CracksLatinos.**

@LUISFECAB