

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)



Software	Realist KeyGenMe 1 by Bakasura
Protección	Serial. (Diferentes Licencias)
Herramientas	Windows 7 Home Premium SP1 x32 Bits (S.O donde trabajamos.) OllyDBG v1.10 RDG Packer Detector v0.7.6.2017 ExeinfoPe v0.0.5.0 2018.03.31 Microsoft Visual Studio 2017 DESCARGAR HERRAMIENTAS DESCARGAR TUTO+ARCHIVOS
SOLUCIÓN	KEYGEN
AUTOR	LUISFECAB
RELEASE	Octubre 19 2018 [TUTORIAL 011]

INTRODUCCIÓN

Empieza una nueva semana y con ella empiezo a escribir un nuevo tuto, el del reto <Realist KeyGenMe 1 by Bakasura>. Este es el último del par de retos que tenía pendientes y que me tomó un buen tiempo en terminar estos retos, y ojo, no por falta de este si no por **FALTA DE NIVEL**, pero que con paciencia y mucha práctica logramos adquirir nuevo conocimiento para lograrlos; ya nos estamos volviendo unos viejos con experiencia.

Este <Realist KeyGenMe 1 by Bakasura> me recordó lo importante de formular una solución acertada, para uno no perderse por el camino, y también reconocer la solución cuando uno la tiene al frente. Más adelante les contaré lo que me pasó y que son el motivo de estas palabras.

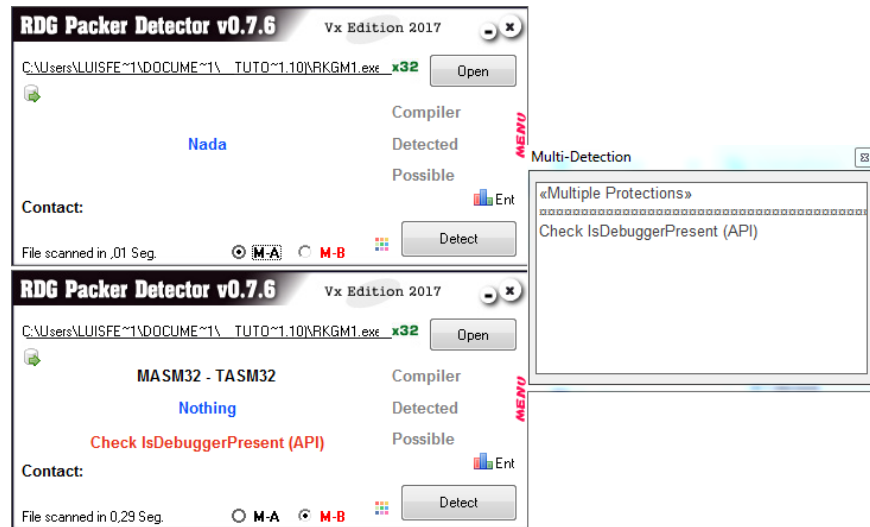
Recuerdo haber leído que si un reto se te hace difícil, mayor será tu satisfacción cuando lo resuelves. Algo parecido a eso sentí cuando por fin me di cuenta por dónde era la solución.

Bueno, trataremos de plantear de la mejor forma la experiencia de hacer este <Realist KeyGenMe 1 by Bakasura> y lo más importante escribir lo aprendido aquí.

Me despido de esta **INTRODUCCIÓN**, saludando a toda la lista de **CracksLatinoS** y **PeruCrackerS**. Saludos especiales para al **Ricardo Narvaja** y para **Bakasura** por compartir su reto.

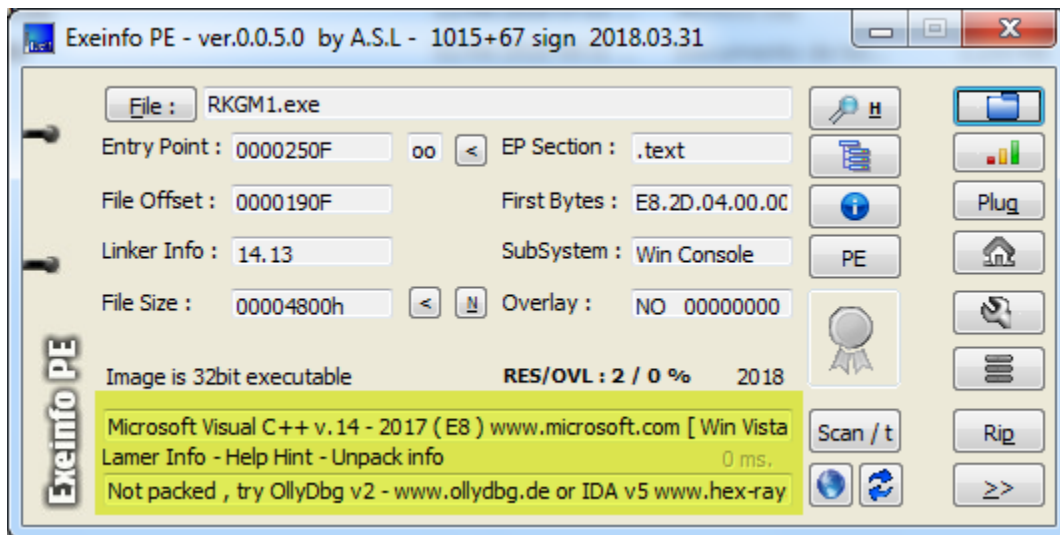
ANALISIS INICAL

Aquí la rutina de siempre. Revisemos con el <RDG Packer Detector v0.7.6.2017> qué tenemos.



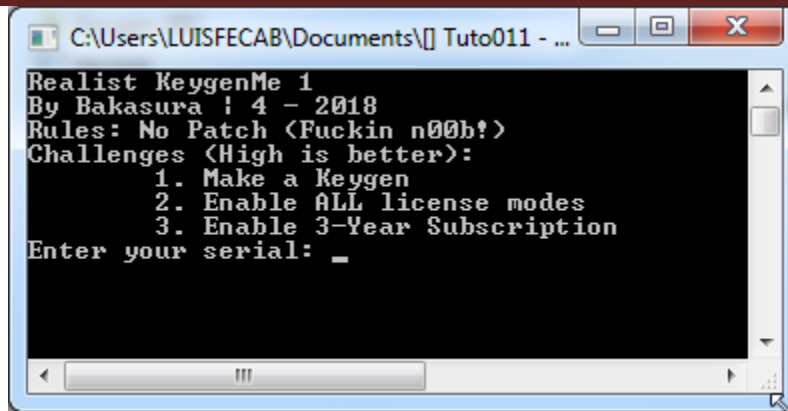
El <RDG Packer Detector v0.7.6.2017> está muy loco, **M-A** no muestra nada y **M-B** dice que es un **MASM32 - TASM32**. Este reto está hecho en MVC++, no sé de dónde sale con eso. Ahora lo de la **API_IsDebuggerPresent** si está presente; yo inhabilité mis **Plugins** para ver si me cerraba la aplicación y no me la cerraba, muy de buenas yo, me evité que lidiar con ella, aunque ya sabemos cómo evitarla yo quería pasarla a mano en este tuto para practicar pero no se pudo.

Para no dejar un manto de duda, utilicemos otro detector para corroborar lo dicho.



En esta ocasión voy a utilizar el <ExeinfoPe v0.0.5.0 2018.03.31> que lo han actualizado. Como vemos es un **Microsoft Visual C++ v14 - 2017** y no esta empackado. Ahora sí, ejecutemos el <Realist KeyGenMe 1 by Bakasura> para ver con qué nos sale.

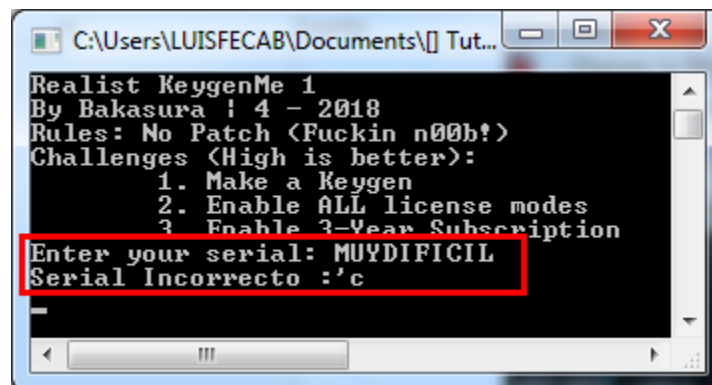
Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)



```
Realist KeygenMe 1
By Bakasura ! 4 - 2018
Rules: No Patch <Fuckin n00b!>
Challenges <High is better>:
1. Make a Keygen
2. Enable ALL license modes
3. Enable 3-Year Subscription
Enter your serial: _
```

Pues **Bakasura** nos pone como regla no **Parchear** y de paso nos trata con fuertes palabras si lo hacemos. Yo creo que cuando tenemos un reto **KeyGenMe**, entonces está más que claro que debemos hacer un **KeyGen** pero si queremos poner en práctica nuestros primeros conocimientos en hacer un **Patch** o **Crack**, pues yo creo que no es malo hacerlo si nos sirve para aprender, eso sí, no pretendas en pasar el reto de esa forma porque el reto no pide eso, una cosa es utilizarlo para practicar nuestros **Patches** y otra muy distintita vencer al reto como lo pide.

Tenemos 3 retos a vencer. Hacer un **KeyGen**, activar todas las licencias y activar la suscripción por tres años. Nos pide ingresar nuestro **Serial**, así que metamos uno y el mío es el de siempre "**MUYDIFICIL**".

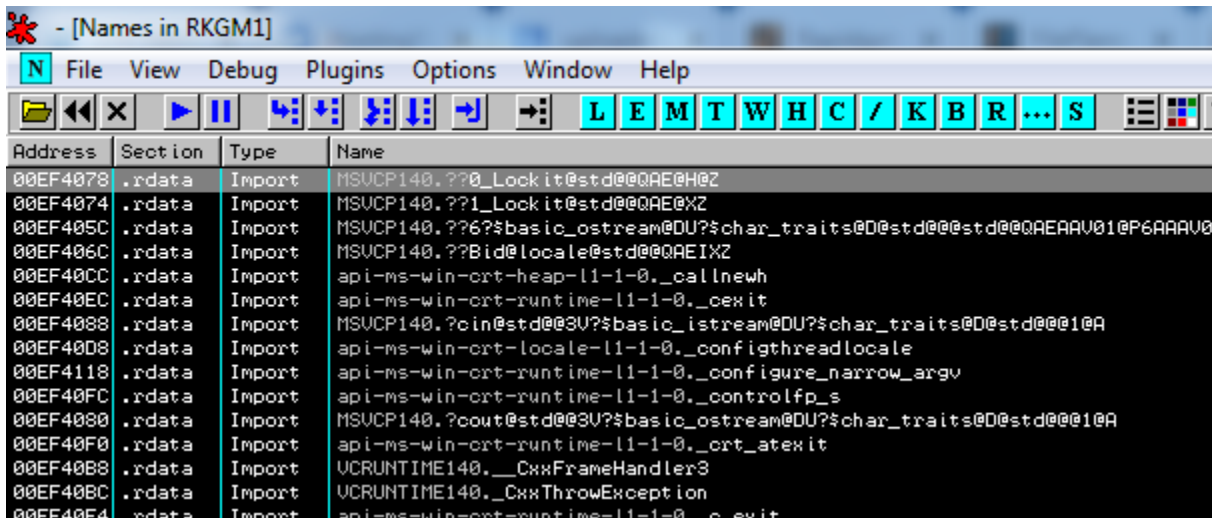


```
Realist KeygenMe 1
By Bakasura ! 4 - 2018
Rules: No Patch <Fuckin n00b!>
Challenges <High is better>:
1. Make a Keygen
2. Enable ALL license modes
3. Enable 3-Year Subscription
Enter your serial: MUYDIFICIL
Serial Incorrecto :\'c
```

Ahí lo metí y presioné <ENTER> y como resultado tenemos más abajo el mensaje "**Serial Incorrecto**". Bueno, no dice mucho pero ya lo venceremos. Listo, a por ello que estoy que me escribo.

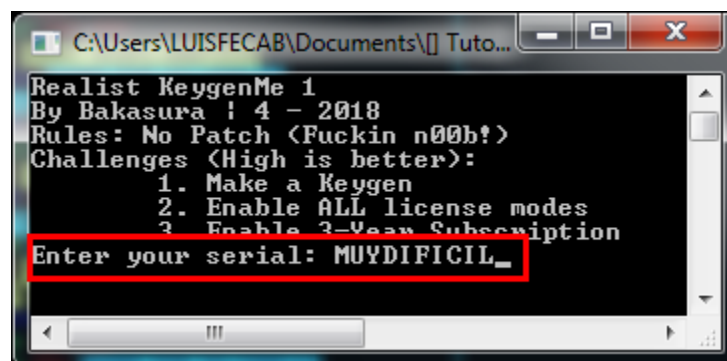
AL ATAQUE

Vamos a utilizar a mi querido y bien ponderado <OlllyDBG v1.10>. Lo dije en el tuto anterior que voy a omitir las obviedades, creo que ya estamos en capacidad de no ser tan detallados, esto lo digo para los que me han acompañado desde mi primer tutorial. Carguemos nuestro <Realist KeyGenMe 1 by Bakasura> y vallamos a ver las **API's** que usa, <CTRL+N>.



Espero no mentir o caer en una imprecisión, arriba en la imagen podemos ver una parte de las **API's** que usa esta aplicación y si observamos bien no se parecen en nada a las que normalmente usamos y visualizamos en esa ventana - **N** - del **Olly**, pero primero y lo que según entiendo, esto es una forma de reconocer un **MVC++** es el uso de esas **DLL** como las **MSVCP**, **VCRUNTIME** o el resto que vemos ahí, además esas **API's** como que no están descifradas; y esa misma consulta la hicieron en la lista y **Ricardo** la resolvió. El maestro dijo esto: "*No podrás solucionarlo con Olly porque no trabaja con símbolos sino que trae harcodeados ficciones de ciertas DLLs si no está entre las que trae no se puede agregar.*" Habla de *ficciones*, no se a lo mejor se refiere a funciones. Por eso es que **Olly** nos muestra esas **API's** de esa forma. Aclaro de nuevo, que es lo que pienso yo y puede que esté más perdido que el hijo de Limberg.

Después de algo de teoría que puede ser muy buena o hasta peligrosa porque son mis suposiciones, pongamos a correr la aplicación con **<F9>** y le metemos nuestro **Serial "MUYDIFICIL"**.



Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Una vez metido nuestro serial, lo probamos. Ya sabemos que no sirve, eso no importa es solo que debemos empezar por el principio; ahí tenemos el **"Serial Incorrecto"** así que reiniciemos todo y buscamos en las **Strings** ese mensaje porque esas **API's** no me ayudan mucho.

```
00041792 MOV EDX,RKGM1.00044340 ASCII "Serial Correcto! \o/"
00041821 PUSH RKGM1.00044358 ASCII " plus a 3-Year Subscription"
00041873 PUSH RKGM1.00044374 ASCII "GLOBAL CORPORATE"
0004188A PUSH RKGM1.00044388 ASCII "UNLIMITED SITE"
000418A1 PUSH RKGM1.00044398 ASCII "25"
000418B5 PUSH RKGM1.0004439C ASCII "20"
000418C9 PUSH RKGM1.000443A0 ASCII "15"
000418DD PUSH RKGM1.000443A4 ASCII "10"
00041933 MOV ESI,RKGM1.000443A8 ASCII "Congratulations! The purchase key is valid. This software is now authorized"
000419AF MOV EDX,RKGM1.0004420C ASCII "Serial Incorrecto : 'c'"
00041A35 MOV EDX,RKGM1.00044444 ASCII "Realist Keygenme 1"
00041A5C MOV EDX,RKGM1.0004442C ASCII "By Bakasura : 4 - 2018"
00041A7F MOV EDX,RKGM1.00044458 ASCII "Rules: No Patch (Fuckin n00b!)"
00041AAF MOV EDX,RKGM1.000444CC ASCII "Challenges (High is better):"
00041AC5 MOV EDX,RKGM1.000444B8 ASCII 09,"1. Make a"
00041ADD MOV EDX,RKGM1.00044498 ASCII 09,"2. Enable"
00041AF5 MOV EDX,RKGM1.00044478 ASCII 09,"3. Enable"
00041B12 MOV EDX,RKGM1.000444EC ASCII "Enter your serial: "
```

Tenemos unas muy buenas **Strings**, ya es cuestión de nosotros si decidimos poner **<BREAKPOINTS>** a todas las que nos parezcan interesantes y luego debuggear y si paramos en unos de esos **<BREAKPOINTS>** revisar si vale la pena a no. Por mi parte yo me voy por la **String** **"Serial Incorrecto"** y desde ahí miro por dónde me voy.

```
00041998 . 33CD XOR ECX,EBP
0004199A . E8 F9080000 CALL RKGM1.00042298
0004199F . 8BE5 MOV ESP,EBP
000419A1 . 5D POP EBP
000419A2 . C3 RETN
000419A3 > 58 C01F0400 PUSH RKGM1.00041FC0
000419A8 . 51 PUSH ECX
000419A9 . 8B0D 80400400 MOV ECX,DWORD PTR DS:[<MSUCP140. ?count@std@03V?%basic_ostream@DU?%char_traits@0@std@0010A
000419AF . BA 0C420400 MOV EDX,RKGM1.0004420C ASCII "Serial Incorrecto : 'c'"
000419B4 . E8 B7030000 CALL RKGM1.00041D78
000419B9 . 83C4 04 ADD ESP,4
000419BC . 8BC8 MOV ECX,EAX
000419BE . 55 LEAVE
00041FC0=RKGM1.00041FC0
Jumps from 00041423, 0004142C, 00041440, 00041454, 00041468, 0004147C, 00041490, 000414A4, 000414B8, 000414CC, 000414E0, 0004
```

En el recuadro **ROJO**, tenemos **000419AF** que carga nuestro **"Serial Incorrecto"**. Podemos ver más arriba en **000419A3** desde dónde se viene a parar al **<CHICO MALO>** y que las aclaraciones del Olly en el recuadro **VERDE**, nos muestran que salta de muchísimos lugares, parece que comprueba un montón de veces nuestro **Serial**. Sigamos ese salto, subamos esa flecha roja hasta su inicio.

```
00051421 . 3B08 CMP EBX,EAX COMPARA CALCULO PRIMER SEGMENTO = QUINTO SEGMENTO.
00051423 . 0F85 7A050000 JNZ RKGM1.000519A3 Salto decisivo
00051429 . 83FB 01 CMP EBX,1
0005142C . 0F86 71050000 JBE RKGM1.000519A3
00051432 . BA 24420500 MOV EDX,RKGM1.00054224
00051437 . 8BCF MOV ECX,EDI
00051439 . E8 72FDFFFF CALL RKGM1.000511B8
0005143E . 84C0 TEST AL,AL
00051440 . 0F85 5D050000 JNZ RKGM1.000519A3
00051446 . BA 2C420500 MOV EDX,RKGM1.0005422C
0005144B . 8BCF MOV ECX,EDI
0005144D . E8 5EFDFFFF CALL RKGM1.000511B8
00051452 . 84C0 TEST AL,AL
```

En **00051423** es de donde viene ese salto, ahí le puse **Salto decisivo** y si vemos un pelin más arriba en **00051421** **CMP EBX,EAX**, he puesto **COMPARA CALCULO PRIMER SEGMENTO = QUINTO SEGMENTO**. Les muestro un adelanto como avance, una parte del **Serial** verdadero se genera a partir de un **SEGMENTO** del **Serial** y con eso calculamos otra parte que en este caso yo la llamo **QUINTO SEGMENTO**. Pues ahí está truco del **Serial**

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

que apenas averigüemos cuáles son las condiciones para que sea verdadero podemos hacer nuestro **KeyGen** para todas las **Licencias**. Pongamos un **<BREAKPOINT>** en nuestro **Salto decisivo** y lo probamos con nuestro **Serial**, **"MUyDIFICIL"**.



```
00051421 . 3B08 CMP EBX,EBX
00051423 . 0F85 7A050001 JNZ RKG1.000519A3
00051429 . 83FB 01 CMP EBX,1
0005142C . 0F86 71050001 JBE RKG1.000519A3
00051432 . BA 24420500 MOV EDX,RKG1.00054224
00051437 . 8BCF MOV ECX,EDI
00051439 . E8 72FDFFFF CALL RKG1.000511B0
0005143E . 84C0 TEST AL,AL
00051440 . 0F85 5D050001 JNZ RKG1.000519A3
00051446 . BA 2C420500 MOV EDX,RKG1.0005422C
0005144B . 8BCF MOV ECX,EDI
0005144D . E8 5EFDFFFF CALL RKG1.000511B0
00051452 . 84C0 TEST AL,AL
00051454 . 0F85 4B050001 JNZ RKG1.000519A3

0024F6E8 0038B948
0024F6EC 5C126314 uortbase.5C126314
0024F6F0 7FFD6000
0024F6F4 00053138 RKG1.00053138
0024F6F8 0024F9E0 ASCII "MUyDIFICIL"
0024F6FC 00000000
```

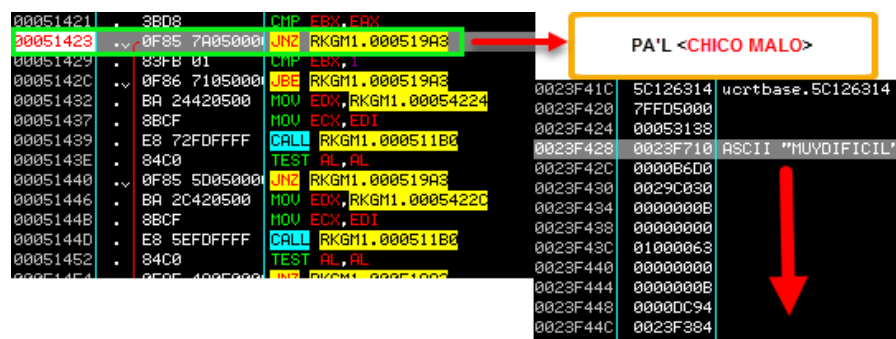
Vamos a tomar el salto que nos envía por mal camino hacia el **<CHICO MALO>**. No lo había comentado, pero voy a colocar cosas que hice que me ayudan a entender cuando estoy tratando de hacer un **KeyGen** o buscando cómo se originan y comprueban los **Seriales**. En ocasiones pruebo las aplicaciones con la pregunta, *¿Qué tal si...?*. Entonces si ustedes miran en el **STACK** o **PILA** en **0024F6F8** tenemos nuestro **Serial**, **"MUyDIFICIL"** pero *¿Qué tal si ingreso mi serial en minúsculas?*, normalmente he visto en aplicaciones comerciales que los **Seriales** los pasan a mayúsculas, así que reiniciemos todo y probamos nuestro **Serial**, **"muydifícil"**. Probemos hasta parar de nuevo en nuestro **Salto decisivo**.



```
00051421 . 3B08 CMP EBX,EBX
00051423 . 0F85 7A050001 JNZ RKG1.000519A3
00051429 . 83FB 01 CMP EBX,1
0005142C . 0F86 71050001 JBE RKG1.000519A3
00051432 . BA 24420500 MOV EDX,RKG1.00054224
00051437 . 8BCF MOV ECX,EDI
00051439 . E8 72FDFFFF CALL RKG1.000511B0
0005143E . 84C0 TEST AL,AL
00051440 . 0F85 5D050001 JNZ RKG1.000519A3
00051446 . BA 2C420500 MOV EDX,RKG1.0005422C
0005144B . 8BCF MOV ECX,EDI
0005144D . E8 5EFDFFFF CALL RKG1.000511B0
00051452 . 84C0 TEST AL,AL
00051454 . 0F85 4B050001 JNZ RKG1.000519A3

0022F668 0030B948
0022F66C 5C126314 uortbase.5C126314
0022F670 7FFD6000
0022F674 00053135
0022F678 0022F960
0022F67C 00000000
```

Paramos y como vemos no hay presencia de nuestro **Serial** **"muydifícil"**. Repito esto de nuevo, sé que se alarga el tutorial pero creo que esto que coloco es lo más importante en mis tutos porque estoy seguro que es la enseñanza principal. Mejor sigamos, a mí eso se me hizo muy raro, pensé, *-Ummmmmmmmmm! Sospechosa la cosa, qué pasó con mi serial-*. Entonces me dije, *-en algún lugar debe de trabajar con mi Serial-* y opté por buscarlo en el **STACK**, mi suposición es que el **STACK** debe mostrarme un **CALL** o varios que sean desde la aplicación o que retornaran a la misma donde estuviera el **Serial**. Para eso, reinicié todo de nuevo y metí mi **Serial** **"MUyDIFICIL"** que es el que no desaparece hasta parar de nuevo en nuestro **Salto decisivo**.



```
00051421 . 3B08 CMP EBX,EBX
00051423 . 0F85 7A050001 JNZ RKG1.000519A3
00051429 . 83FB 01 CMP EBX,1
0005142C . 0F86 71050001 JBE RKG1.000519A3
00051432 . BA 24420500 MOV EDX,RKG1.00054224
00051437 . 8BCF MOV ECX,EDI
00051439 . E8 72FDFFFF CALL RKG1.000511B0
0005143E . 84C0 TEST AL,AL
00051440 . 0F85 5D050001 JNZ RKG1.000519A3
00051446 . BA 2C420500 MOV EDX,RKG1.0005422C
0005144B . 8BCF MOV ECX,EDI
0005144D . E8 5EFDFFFF CALL RKG1.000511B0
00051452 . 84C0 TEST AL,AL
00051454 . 0F85 4B050001 JNZ RKG1.000519A3

0023F41C 5C126314 uortbase.5C126314
0023F420 7FFD5000
0023F424 00053138
0023F428 0023F710 ASCII "MUyDIFICIL"
0023F42C 0000B0D0
0023F430 0029C030
0023F434 0000000B
0023F438 00000000
0023F43C 01000063
0023F440 00000000
0023F444 0000000B
0023F448 0000DC94
0023F44C 0023F384
0023F450 73F50000
```


Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Ahí lo que hice fue bajar y bajar en el **STACK** a ver si mi suposición era cierta, después de todo nada se perdía por probar. Pues buenas noticias, encontré algo interesante.

```
0023F678  FFFFFFFF
0023F67C  0023F6B8
0023F680  011B1D32 RETURN to RKGM1.011B1D32 from MSUCP140.??1_Lockit@std@@QAE@XZ
0023F684  5F69FB8F
0023F688  0023F710 ASCII "MUYDIFICIL"
0023F68C  5C042C90 OFFSET MSUCP140.?cin@std@@@3U?$basic_istream@DU?$char_traits@D@std@@@10A
0023F690  011B1D4C RETURN to RKGM1.011B1D4C from RKGM1.011B2298
0023F694  5C042C90 OFFSET MSUCP140.?cin@std@@@3U?$basic_istream@DU?$char_traits@D@std@@@10A
0023F698  0023F71A
0023F69C  00000000
```

Aparece mi **Serial**, "MUYDIFICIL" en medio de dos **CALL's**. Observemos el **CALL** de abajo, **RETURN to RKGM1.011B1D4C from RKGM1.011B2298**, lo llama la aplicación y retorna en la aplicación y el otro **CALL** de arriba, **RKGM1.011B1D32 from MSUCP140.??1_Lockit@std@@QAE@XZ**, lo llama una **API** y retorna en la aplicación. Vallamos al retorno del primer **CALL** y desde ahí podemos ver si sucede algo con nuestro **Serial**.

011B1D42	. 8B4D	MOV ECX,DWORD PTR SS:[EBP-10]	
011B1D45	. 33CD	XOR ECX,EBP	
011B1D47	. E8 4C	CALL RKGM1.011B2298	Desde aqui busco donde se almacena el serial
011B1D4C	. 8BE5	MOV ESP,EBP	
011B1D4E	. 5D	POP EBP	
011B1D4F	. C3	RETN	
011B1D50	. 8D4D	LEA ECX,DWORD PTR SS:[EBP-24]	
011B1D53	. E8 38	CALL RKGM1.011B1090	
011B1D58	. 68 A8	PUSH RKGM1.011B4CA8	
011B1D5D	. 8BC1	MOV EAX,ECX	

0023F678	FFFFFFF
0023F67C	0023F6B8
0023F680	011B1D32 RETURN to RKGM1.011B1D32 from MSUCP140.??1_Lockit@std@@QAE@XZ
0023F684	5F69FB8F
0023F688	0023F710 ASCII "MUYDIFICIL"
0023F68C	5C042C90 OFFSET MSUCP140.?cin@std@@@3U?\$basic_istream@DU?\$char_traits@D@std@@@10A
0023F690	011B1D4C RETURN to RKGM1.011B1D4C from RKGM1.011B2298
0023F694	5C042C90 OFFSET MSUCP140.?cin@std@@@3U?\$basic_istream@DU?\$char_traits@D@std@@@10A
0023F698	0023F71A
0023F69C	00000000

[Go to expression](#) [Ctrl+G](#)
[Follow in Disassembler](#) [Enter](#)
[Follow in Dump](#)

Ahí vemos el retorno y el **CALL** que lo llamó, **011B1D47 CALL RKGM1.011B2298**, ponemos en ese **CALL** un <**BREAKPOINT**> para parar cuando hagamos todo de nuevo y ahí si traceamos con <**F7**> para observar qué sucede.

Address	Hex dump	Disassembly	Comment
011B1D42	. 8B4D	MOV ECX,DWORD PTR SS:[EBP-10]	
011B1D45	. 33CD	XOR ECX,EBP	
011B1D47	. E8 4C	CALL RKGM1.011B2298	Desde aqui busco donde se almacena el serial
011B1D4C	. 8BE5	MOV ESP,EBP	
011B1D4E	. 5D	POP EBP	
011B1D4F	. C3	RETN	
011B1D50	> 8D4D	LEA ECX,DWORD PTR SS:[EBP-24]	
011B1D53	. E8 38	CALL RKGM1.011B1090	

Ahí estamos detenidos en el **011B1D47 CALL RKGM1.011B2298**. Ahora a tracear con <**F7**> abriendo bien el ojo para cuando entre en escena nuestro **Serial**.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Address	Hex	dump	Disassembly	Comment
011B20C3	. 83CF		OR EDI,FFFFFFFF	
011B20C6	. 8945		MOV DWORD PTR SS:[EBP-18],EAX	
011B20C9	> 8B4C3		MOV ECX,DWORD PTR DS:[ECX+ESI]	
011B20CD	. FF15		CALL DWORD PTR DS:[&MSUCP140	MSUCP140.?sgeto@?%basio_str
011B20D3	> 83C7		ADD EDI,-1	Recupera nuestro serial.
011B20D6	. 8BD0		MOV EDX,EAX	Caracter del serial
011B20D8	. 8355		ADC DWORD PTR SS:[EBP-18],-1	
011B20DC	~ 78 5D		JS SHORT RKG1.011B213B	
011B20DE	~ 7F 04		JG SHORT RKG1.011B20E4	
011B20E0	. 85FF		TEST EDI,EDI	
011B20E2	~ 74 57		JE SHORT RKG1.011B213B	
011B20E4	> 83FA		CMP EDX,-1	Compara BYTE caracter
011B20E7	~ 75 05		JNZ SHORT RKG1.011B20EE	
011B20E9	. 8D5A		LEA EBX,DWORD PTR DS:[EAX+2]	
011B20EC	~ EB 4D		JMP SHORT RKG1.011B213B	
011B20EE	> 8B45		MOV EAX,DWORD PTR SS:[EBP-20]	
011B20F1	. 0FB6C		MOVZX ECX,DL	
011B20F4	. 8B40		MOV EAX,DWORD PTR DS:[EAX+C]	
011B20F7	. F6044		TEST BYTE PTR DS:[EAX+ECX*2],4	
011B20FB	~ 75 3E		JNZ SHORT RKG1.011B213B	
011B20FD	. 84D2		TEST DL,DL	
011B20FF	~ 74 3A		JE SHORT RKG1.011B213B	
011B2101	. 8B45		MOV EAX,DWORD PTR SS:[EBP-14]	Aqui guarda nuestro serial
011B2104	. 8810		MOV BYTE PTR DS:[EAX],DL	
011B2106	. 40		INC EAX	
011B2107	. 8945		MOV DWORD PTR SS:[EBP-14],EAX	
011B210A	. 8B06		MOV EAX,DWORD PTR DS:[ESI]	
011B210C	. 8B40		MOV EAX,DWORD PTR DS:[EAX+4]	
011B210F	. 8B4C3		MOV ECX,DWORD PTR DS:[ECX+ESI]	
011B2113	. FF15		CALL DWORD PTR DS:[&MSUCP140	MSUCP140.?snexto@?%basio_s
011B2119	. EB B8		JMP SHORT RKG1.011B20D3	

ESTA API RECUPERA CARACTER POR CARACTER DEL SERIAL PARA SER GUARDADO EN MEMORIA.

VA GUARDANDO NUESTRO SERIAL

Aquí ya entra en juego nuestro ojo de cracker y nuestra experiencia que vamos adquiriendo a medida que superamos retos que nos ayudan a reconocer las zonas de interés. Esa rutina va a guardar nuestro **Serial** en **Memoria**, cabe aclarar que la dirección en **Memoria** no será igual porque cada vez que reiniciamos el reto esta cambia, pero podemos colocar un <BREABKPOIT> en **011B2104 MOV BYTE PTR DS:[EAX],DL** que es el lugar donde se guarda nuestro **Serial**. Reiniciemos todo de nuevo pero colocando nuestro **Serial** en minúsculas, "muydifícil", porque lo que queremos es averiguar el motivo de su misteriosa desaparición. Lleguemos a este último <BREAKPOINT>, los otros <BREAKPOINTS> cuando nos detengan los quitamos y seguimos con <F9> hasta llegar a **011B2104**.

011B2101	. 8B45	MOV EAX,DWORD PTR SS:[EBP-14]	
011B2104	. 8810	MOV BYTE PTR DS:[EAX],DL	Aqui guarda nuestro serial
011B2106	. 40	INC EAX	
011B2107	. 8945	MOV DWORD PTR SS:[EBP-14],EAX	
011B210A	. 8B06	MOV EAX,DWORD PTR DS:[ESI]	

DL=6D ('m')

Stack DS:[0023FD20]=00

Address	Hex	dump	ASCII
0023FD20	00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0023FD30	00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0023FD40	00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0023FD50	00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Vemos en las aclaraciones que va a guardar nuestro primer carácter que es la "m" a partir de la dirección de **Memoria** en el **DUMP**, **0023FD20**. Ya con esto sabemos que en algún momento accederá a esa dirección de **Memoria** para trabajar con nuestro **Serial**. Pasemos ese ciclo con cuidado de no pasar de largo hasta pasar todo el **serial**.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

```

011B2101 . 8B45 MOV EAX,DWORD PTR SS:[EBP-14]
011B2104 . 8B10 MOV BYTE PTR DS:[EAX],DL      Aqui guarda nuestro serial
011B2106 . 40 INC EAX
011B2107 . 8945 MOV DWORD PTR SS:[EBP-14],EAX
011B210A . 8B06 MOV EAX,DWORD PTR DS:[ESI]
EAX=0023F849

```

Address	Hex dump	ASCII
0023F840	60 75 79 64 69 66 69 63 69 6C 00 00 00 00 00 00	muydifícil.....
0023F850	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0023F860	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Breakpoint

Search for

Go to

Memory, on access

Memory, on write

Hardware, on access

Hardware, on write

Hardware, on execution

Byte

Word

Dword

Colocaremos un <HARDWARE-BREAKPOINT ON ACCES-BYTE> para parar muy seguramente cuando valla a hacer una pilatuna por estar en minúsculas. Sigamos con <F9> para detenernos en nuestro <HBP>.

```

011B128F . 8A07 MOV AL,BYTE PTR DS:[EDI]
011B1291 . 89BD MOV DWORD PTR SS:[EBP-2E0],EDI  Guarda direccion del serial almacenado.
011B1297 . 84C0 TEST AL,AL
011B1299 < 74 42 JE SHORT RKGM1.011B12DC
011B129B . 0F DB 0F
011B129C . 1F DB 1F
011B129D . 44 DB 44  CHAR 'D'
011B129E . 00 DB 00
011B129F . 00 DB 00
011B12A0 > 3C 41 CMP AL,41  Comprueba que serial sean mayusculas o numeros
011B12A2 < 7C 34 JL SHORT RKGM1.011B12A8  de lo contrario va eliminando el serial por no cumplir
011B12A4 . 3C 5A CMP AL,5A  cumplir esa condicion
011B12A6 < 7E 26 JLE SHORT RKGM1.011B12AE
011B12A8 > 2C 30 SUB AL,30
011B12AA . 3C 29 CMP AL,9
011B12AC < 77 33 JA SHORT RKGM1.011B12B1
011B12AE > 46 INC ESI
011B12AF < EB 26 JMP SHORT RKGM1.011B1207
011B12B1 > 8D55 LEA EDX,DWORD PTR DS:[ESI+1]
011B12B4 . 8BCA MOV ECX,EDX
011B12B6 . 8D5B LEA EBX,DWORD PTR DS:[ECX+1]
011B12B9 . 0F DB 0F
011B12BA . 1F DB 1F
011B12BB . 80 DB 80
011B12BC . 00 DB 00
011B12BD . 00 DB 00
011B12BE . 00 DB 00
011B12BF . 00 DB 00
011B12C0 > 8A01 MOV AL,BYTE PTR DS:[ECX]
011B12C2 . 41 INC ECX
011B12C3 . 84C0 TEST AL,AL
011B12C5 < 75 F9 JNZ SHORT RKGM1.011B12C8
011B12C7 . 2BCB SUB ECX,EBX
011B12C9 . 8D41 LEA EAX,DWORD PTR DS:[ECX+1]
011B12CC . 50 PUSH EAX

```

Paramos en 011B1291, recordemos que los <HBP> se detienen una instrucción adelante así que en realidad la instrucción 011B128F MOV AL,BYTE PTR DS:[EDI] fue la que activó el <HBP> y es debido a que mueve a AL nuestro primer carácter "m" = 0x6D. Bueno, ya es cuestión de entender esa rutina que es un LOOP que revisa qué tenemos en nuestro Serial y que deben ser números o letras mayúsculas porque de lo contrario los elimina, y ese es el motivo de la desaparición misteriosa del Serial, "muydifícil". Bueno, podríamos seguir la ruta que sigue el serial pero ya estuvo bien por ahora de esta ruta, solo quería dejar algo de teoría como enseñanza.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Quitemos el <HBP> y el último <BP> para que no estorben. Reiniciemos todo, metemos nuestro serial "**MUYDIFICIL**" y paramos de nuevo en nuestro **Salto decisivo**.

Address	Hex dump	Disassembly	Comment	Registers (FPU)
011B1423	0F85	JNZ RKG1.011B19A3	Salto decisivo	EAX 00000000
011B1429	83FB	CMP EBX,1		ECX 0013FDCC
011B142C	0F86	JBE RKG1.011B19A3		EDX 0000000F
011B1432	BA 24	MOV EDX,RKG1.011B4224	ASCII "C58CA"	EBX 00001743
011B1437	8BCF	MOV ECX,EDI		ESP 0013FAF8
011B1439	E8 72	CALL RKG1.011B11B0		EBP 0013FDE8
011B143E	84C0	TEST AL,AL		ESI 00000005
011B1440	0F85	JNZ RKG1.011B19A3		EDI 0013FDF0 ASCII "MUYDIFICIL"
011B1446	BA 2C	MOV EDX,RKG1.011B422C	ASCII "EA84A"	EIP 011B1423 <RKG1.Salto decisivo>
011B144B	8BCF	MOV ECX,EDI		C 0 ES 0023 32bit 0(FFFFFFFF)
011B144D	E8 5B	CALL RKG1.011B11B0		P 0 CS 001B 32bit 0(FFFFFFFF)
011B1452	84C0	TEST AL,AL		A 0 SS 0023 32bit 0(FFFFFFFF)
011B1454	0F85	JNZ RKG1.011B19A3		Z 1 DS 0023 32bit 0(FFFFFFFF)
011B145A	BA 34	MOV EDX,RKG1.011B4234	ASCII "F16EA"	S 0 FS 003B 32bit 7FFDE000(4000)
011B145F	8BCF	MOV ECX,EDI		

Para no desesperarte mi querido lector, ya entramos de lleno a resolver el reto. Como ven en la imagen de arriba en **011B1423**, si tomamos el salto iremos al <**CHICO MALO**> y para evitar eso cambiamos **FLAG-Z = 1**; con eso seguimos el camino del <**CHICO BUENO**>. He resaltado con un recuadro **VERDE** para que notemos la existencia del **CALL RKG1.011B11B0** y que de ahí para abajo es llamado unas cuantas veces, ese lo que hace es buscar mediante una comparación si nuestro **Serial** contiene una de esas parte **ASCII**, y si las tiene lo toma como un **Serial** erróneo y nos manda al lugar odiado con **JNZ RKG1.011B19A3**. La tarea de ustedes es que revisen el **CALL RKG1.011B11B0** para que miren cómo es que hace esa comparación. Sigamos pasando esas validaciones hasta llegar a la dirección **012F1740**.

012F172A	8D4D DC	LEA ECX,DWORD PTR SS:[EBP-24]		
012F172D	E8 EEF9FFFF	CALL RKG1.012F1120	TRABAJA CON TERCER SEGMENTO. 5 CARACTERES.	
012F1732	8B8D 1CFDFFF	MOV ECX,DWORD PTR SS:[EBP-2E4]	MUEVE EL VALOR DEL PRIMER SEGMENTO XORREADO	
012F1738	81E1 FFFF0F0	AND ECX,0FFFFFF		
012F173E	3BC8	CMP ECX,EAX	TERCER SEGMENTO = PRIMER SEGMENTO XOR 53135	
012F1740	0F85 5D02000	JNZ RKG1.012F19A3	COMPRUEBA TERCER SEGMENTO DE NUESTRO SERIAL	
012F1746	8D4D D4	LEA ECX,DWORD PTR SS:[EBP-2C]		
012F1749	E8 D2F9FFFF	CALL RKG1.012F1120	TRABAJA CON CUARTO SEGMENTO. 5 CARACTERES.	
012F174E	8B8D 24FDFFF	MOV ECX,DWORD PTR SS:[EBP-2DC]		
012F1754	81E1 FFFF0F0	AND ECX,0FFFFFF		
012F175A	3BC8	CMP ECX,EAX	CUARTO SEGMENTO = RUTINA CALCULO PRIMERSEGMENTO.	
012F175C	0F85 4102000	JNZ RKG1.012F19A3	COMPRUEBA CUARTO SEGMENTO DE NUESTRO SERIAL	
012F1762	807D EC 41	CMP BYTE PTR SS:[EBP-14],41	DEBE SER UNA LETRA COMO PRIMER CARACTER EN PRIMER SEFMENTO.	
012F1766	0F8C 3702000	JL RKG1.012F19A3		
012F176C	807D F8 42	CMP BYTE PTR SS:[EBP-8],42	ULTIMO CARACTER DEL SEGMENTO2 DEBE SER "B"	
012F1770	0E85 2002000	JNZ RKG1.012F19A3		
012F1776	57	PUSH EDI		
012F1777	FF15 00402F0	CALL DWORD PTR DS:[&KERNEL32.lstrlenA]	[String lstrlenA	
012F177D	83F8 19	CMP EAX,19	LONGITUD DE NUESTRO SERIAL DEBE SER 0x19=25	
012F1780	0F8C 1D02000	JL RKG1.012F19A3		
012F1786	68 C01F2F01	PUSH RKG1.012F1FC0		
012F178B	51	PUSH ECX		
012F178C	8B8D 80402F0	MOV ECX,DWORD PTR DS:[&MSUCP140.?cout@	MSUCP140.?cout@std@03U?%basic_ostream@DU?%char_traits@0@std	
012F1792	BA 40432F01	MOV EDX,RKG1.012F4340	ASCII "Serial Correcto! \n/"	
012F1797	E8 D4050000	CALL RKG1.012F1D70		
012F179C	83C4 04	ADD ESP,4		

Esa imagen muestra casi todo resuelto y nos da respuestas de qué debe tener nuestro **Serial** para que sea válido. Todos esas condiciones para generar nuestro **Serial** válido las descubrí de a poquitos mientras traceaba y traceaba, y reiniciaba todo de nuevo una infinidad de veces. Aquí lo relevante es que lo primero que descubrí fue la longitud de nuestro **Serial** debe ser **0x19=25** caracteres. Venía acostumbrado que lo primero que hacían las aplicaciones era comparar la longitud del **Serial** ingresado pero aquí es lo último que hace. Bueno, ya conociendo nuestra longitud del **Serial**, reiniciamos todo con un nuevo **Serial**, "**11111-22222-33333-44444-55555**".

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OlllyDBG v1.10)

Habrán notado que lo separé por guiones (-) pero que no afectan la longitud de 25 porque si recuerdan la aplicación elimina todo lo que no sea números o letras mayúscula previamente. Reiniciemos todo con nuestro nuevo **Serial**.

The screenshot displays the OlllyDBG interface with the following components:

- Assembly View:** Shows instructions from address 012F13E2 to 012F1432. Key instructions include `CALL RKG1.012F1120` at 012F141C and `CMP EBX, EAX` at 012F1421.
- Registers (MMX):** Lists registers with values: EAX 00055555, ECX 000221F4, EDX 00055555, EBX 00055534, ESP 0024F660, EBP 0024F950, ESI 00000005, EDI 0024F958 (ASCII "1111122222333334444455555"), and EIP 012F1423.
- Memory Dump:** Shows a hex dump and ASCII representation. The ASCII string at address 0024F958 is "1111122222333334444455555".

Todo es cuestión de tener la sana costumbre de revisar siempre la información que tenemos, en los **REGISTROS** y el **STACK** nos muestra nuestro **Serial** y si vamos al **DUMP** vemos el **Serial** y un poco más arriba podemos ver el **Serial** pero partido de a 5 caracteres y ese fue el motivo de que mi nuevo **Serial** fuera "11111-22222-33333-44444-55555", así es como poco a poco fui entendiendo las maniobras para el **Serial**. Yo repetí esto muchísimas, estoy tratando de mostrarles lo que hice picando aquí y picando allá. Bueno, después me enfoqué en buscar cómo obtener una comparación `012F1421 CMP EBX, EAX` válida para no tomar el salto y eso ocurre cuando **EBX=EAX**.

This screenshot shows a different section of the assembly code and registers:

- Assembly View:** Instructions from address 012F140E to 012F1429. Key instructions include `CALL RKG1.012F1120` at 012F141C and `CMP EBX, EAX` at 012F1421.
- Registers (MMX):** Lists registers with values: EAX 00022222, ECX 0012FE24 (ASCII "55555"), EDX 00022222, EBX 00055534, ESP 0012FB50, EBP 0012FE40, ESI 00000005, and EDI 0012FE48 (ASCII "1111122222").

Lo que hice fue poner un `<BP>` en `012F141C CALL RKG1.012F1120` y ver qué hacía y qué retornaba. Les cuento que tracié un montón de veces ese `CALL` con mi serial "MUYDIFICIL" y no podía entender qué hacía, es que sigo siendo un novato a pesar que mi buen amigo DavicoRm me dijo una vez que ya no lo era. Lo vine a descifrar cuando cambié mi serial a "11111-22222-33333-44444-55555" pero primero echemos un vistazo en los registros, **EAX 00022222** que es la segunda parte de nuestro **Serial**, "22222", por eso hice mi **Serial** separado con guiones (-) para que fuera compuesto por 5 partes y así es más fácil explicar cómo tener nuestro **Serial** válido, y en **EBX 00055534**, este valor ni idea por ahora pero cuando hagamos el `012F141C CALL` lo sabremos. Notemos también que **ECX 0012FE24 ASCII "55555"** viene siendo la **PARTE 5** del **Serial**, "55555", yo en mis comentarios del Ollly los llamé **SEGMENTOS** que es la misma vaina; algo muy importante para resaltar es que **ECX** toma ese valor en `012F1413 LEA ECX, DWORD PTR SS:[EBP-1C]`. La explicación se me enreda un poco pero espero lo puedan entender y seguir, vamos a ver en el **DUMP** de dónde sacó la **PARTE 5** del **Serial**

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

para **ECX**; lo que haremos es posicionarnos sobre **012F1413** **LEA ECX, DWORD PTR SS:[EBP-1C]** y escogerlo como nuestro nuevo origen.

Address	Hex dump	Disassembly	Comment
012F140E	. 83FE 05	CMP ESI,5	
012F1411	.^ 7C F0	JL SHORT RKG1.012F1408	
012F1413	. 804D E4	LEA ECX,DWORD PTR SS:[EBP-1C]	
012F1416	. 81E3 FFF	AND EBX,0FFFFFFF	
012F141C	. E8 FFFCF	CALL RKG1.012F1120	New origin here Ctrl+Gray *
012F1421	. 3B08	CMP EBX,EAX	Go to
012F1423	. 0F85 7A0	JNZ RKG1.012F19A3	Follow in Dump
012F1429	. 83FB 01	CMP EBX,1	

Eso lo podemos hacer porque **EBP** no ha cambiado, sigue manteniendo el mismo valor. Con eso podemos saber la dirección en el **DUMP** y ver de dónde tomó esa parte del **Serial**.

012F1411	.^ 7C F0	JL SHORT RKG1.012F1408	
012F1413	. 804D E4	LEA ECX,DWORD PTR SS:[EBP-1C]	
012F1416	. 81E3 FFF	AND EBX,0FFFFFFF	
012F141C	. E8 FFFCF	CALL RKG1.012F1120	TRABAJA CON QUINTO SEGM
Stack address=0012FE24, (ASCII "55555")			
ECX=0012FE24, (ASCII "55555")			
Address	Hex dump	ASCII	
0012FE24	35 35 35 35 35 00 FF 7F 31 31 31 31 31 00 12 00	55555. 011111.0.	
0012FE34	32 32 32 32 32 00 2F 01 56 FE F4 09 BC FE 12 00	22222./0U=7P=0.	
0012FE44	32 1B 2F 01 31 31 31 31 31 32 32 32 32 32 33 33	2+0111112222233	
0012FE54	33 33 33 34 34 34 34 34 35 35 35 35 35 00 00 00	3334444455555...	
0012FE64	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00		

Si miramos en el **DUMP** vemos que lo tomó de las partes en las que separa el **Serial**, lo ven, ahí el motivo de mi **Serial**. Listo, volvamos donde estábamos, haciendo lo mismo, seleccionamos la dirección **012F141C** y la dejamos como nuestro nuevo origen y entramos a ese **CALL RKG1.012F1120**, ¿recuerdan?, es el **CALL** que no entendía.

Address	Hex dump	Disassembly	Comment
012F111F	. CC	INT3	
012F1120	. 56	PUSH ESI	
012F1121	. 8BF1	MOV ESI,ECX	
012F1123	. 85F6	TEST ESI,ESI	
012F1125	. 75 04	JNZ SHORT RKG1.012F1128	
012F1127	. 33C0	XOR EAX,EAX	
012F1129	. 5E	POP ESI	
012F112A	. C3	RET	
012F112B	. 8A06	MOV AL,BYTE PTR DS:[ESI]	
012F112D	. 84C0	TEST AL,AL	
012F112F	.^ 74 F6	JE SHORT RKG1.012F1127	
012F1131	. 33C9	XOR ECX,ECX	
012F1133	. 33D2	XOR EDX,EDX	
012F1135	.> 3C 31	CMP AL,31	SALTA SI AL<31. SALTARA UNICAMENTE CON UN 0.
012F1137	. 7C 04	JL SHORT RKG1.012F1130	
012F1139	. 3C 39	CMP AL,39	SALTA SI AL<=39. COMPROBAR SI ES NUMERO. SALTA SI ES NUMERO.
012F113B	. 7E 19	JLE SHORT RKG1.012F1156	
012F113D	.> 3C 61	CMP AL,61	
012F113F	. 7C 04	JL SHORT RKG1.012F1145	
012F1141	. 3C 66	CMP AL,66	
012F1143	. 7E 11	JLE SHORT RKG1.012F1156	
012F1145	.> 3C 41	CMP AL,41	
012F1147	. 7C 04	JL SHORT RKG1.012F1140	COMPROBAMOS SI TENEMOS LETRA ENTRE A - E
012F1149	. 3C 46	CMP AL,46	
012F114B	. 7E 09	JLE SHORT RKG1.012F1156	SALTA AL<=46. DESDE A-F
012F114D	.> 8A4431 0	MOV AL,BYTE PTR DS:[ECX+ESI+1]	
012F1151	. 41	INC ECX	
012F1152	. 84C0	TEST AL,AL	
012F1154	.^ 75 DF	JNZ SHORT RKG1.012F1135	
012F1156	.> 8A040E	MOV AL,BYTE PTR DS:[ESI+ECX]	
012F1159	. 03F1	ADD ESI,ECX	
012F115B	. 84C0	TEST AL,AL	
ESI=00000005			
Local calls from 012F132F, 012F13F6, 012F141C, 012F172D, 012F1749			

Pues ese **CALL** lo que hace es tomar nuestra parte del **Serial** que es un **ASCII** y lo convierte en un valor **HEXADECIMAL**, jaaaaaa eso era todo y cuando lo entendí se me

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

"prendió el bombillo" y ahí comprendí que mi **Serial** solo debe tener valores **HEXADECIMALES**, así que nuestro **Serial** "trucho" por lo menos cumple esa condición y que gracias a eso pude entender ese **CALL**, muy de buenas yo al haber escogido mi serial con puros números. Si miramos en las aclaraciones del Olly este **CALL** es llamado 5 veces y debe ser para cuando va a trabajar con las 5 partes de nuestro **Serial** para pasarlos a un valor **HEXA**. Ya con esto la tuve clara, entonces pasemos ese **CALL** hasta salir de este.

Address	Hex dump	Disassembly	Comment	Registers (MM)
012F1411	. ^ 7C F0	JL SHORT RKG1.012F1403		EAX 00055555
012F1413	. 8D4D E4	LEA ECX,DWORD PTR SS:[EBP-1C]		ECX 000221F4
012F1416	. 81E3 FFF	AND EBX,0FFFFFFF		EDX 00055555
012F141C	. E8 FFFCF	CALL RKG1.012F1120	TRABAJA CON QUINTO SEGMENTO. 5 CARACTERES.	EBX 00055534
012F1421	. 3BD8	CMP EBX,EAX	COMPARA CALCULO PRIMER SEGMENTO = QUINTO SEGMENTO.	ESP 0012FB50
012F1423	. 0F85 7A0	JNZ RKG1.012F19A3	Salto desicivo	EBP 0012FE40
012F1429	. 83FB 01	CMP EBX,1		ESI 00000005

Ahí está claro, el **CALL RKG1.012F1120** pasó a **EAX 00055555** que es nuestra **PARTE 5** del **Serial** como un valor **HEXA** y lo compara con **EBX 00055534**, como ven **0x55555<>0x55534**, no hay igualdad y tomamos el salto al **<CHICO MALO>**. Podemos concluir que la **PARTE 5** de nuestro **Serial** debe ser **"55534"** para que la igualdad se dé y así podamos seguir derecho y no saltar; ese es el valor que dijimos que no teníamos ni idea, ¡lo recuerdan, no!, nuestro **Serial** pasa a ser **"11111-22222-33333-44444-55534"**. Probemos nuestro nuevo **Serial** para ver si ahora si tenemos igualdad.

Address	Hex dump	Disassembly	Comment	Registers (MM)
012F1413	. 8D4D E4	LEA ECX,DWORD PTR SS:[EBP-1C]		EAX 00055534
012F1416	. 81E3 FFF	AND EBX,0FFFFFFF		ECX 000221F3
012F141C	. E8 FFFCF	CALL RKG1.012F1120	TRABAJA CON QUINTO SEGMENTO. 5 CARACTERES.	EDX 00055534
012F1421	. 3BD8	CMP EBX,EAX	COMPARA CALCULO PRIMER SEGMENTO = QUINTO SEGMENTO.	EBX 00055534
012F1423	. 0F85 7A0	JNZ RKG1.012F19A3	Salto desicivo	ESP 002CF770
012F1429	. 83FB 01	CMP EBX,1		EBP 002CFA60

Hicimos el **012F1421 CMP EBX,EAX** y como tenemos igualdad no saltamos y seguimos por el camino correcto. Ya tenemos un **Serial** que funciona hasta este salto; ya con todo lo descubierto hasta aquí me sirve para plantear cómo seguir. Resumamos lo descubierto, el **Serial** se divide en 5 partes, esas partes se convierten a valores **HEXADECIMALES** y que luego son comparadas, bueno aquí podemos decir que apenas descubrimos una comparación con un valor que seguramente ha sido calculado previamente a partir del **Serial**; y por qué digo que a partir del **Serial**, pues porque con lo poco que he aprendido al hacer estos **KeyGen** que no dependen del usuario ingresado si no que estos utilizan una parte del **Serial** ingresado para generar la otra parte y luego si hacer su validación, ejemplo claro de esto es lo que acabamos de hacer, que nuestra **PARTE 5** del **Serial** la pudimos hallar de un valor que calculó la aplicación para ser comparado y que se calcula a partir de nuestro **Serial**, ya lo verán más adelante. Entonces, qué camino opté por seguir; decidí volver al inicio y seguir el **Serial**. Recuerdan cuando hicimos el seguimiento del **Serial** para saber el motivo de por qué las minúsculas desaparecían y lo dejamos hasta ahí cuando lo descubrimos, pues lo correcto ahora es seguir ese seguimiento y ver en donde coge nuestro **Serial** y lo separa en 5 partes porque seguro, muy seguro, tomará nuestras partes del **Serial** para hacer cálculos previos. Seguiremos el **Serial** a partir de donde dejamos ese seguimiento, eso sí seguiremos es a nuestro nuevo **Serial**, **"11111-22222-33333-44444-55534"**. Reiniciamos todo hasta llegar donde se purga nuestro serial.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Address	Hex dump	Disassembly	Comment
00CA128F	. 8A07	MOV AL,BYTE PTR DS:[EDI]	
00CA1291	. 89BD 20F	MOV DWORD PTR SS:[EBP-2E0],ED	Guarda direccion del serial almacenado.
00CA1297	. 84C0	TEST AL,AL	
00CA1299	~ 74 42	JE SHORT RKG1.00CA12D0	
00CA129B	0F	DB 0F	
00CA129C	1F	DB 1F	
00CA129D	44	DB 44	CHAR 'D'
00CA129E	00	DB 00	
00CA129F	00	DB 00	
00CA12A0	> 3C 41	CMP AL,41	Comprueba que serial sean mayusculas o numeros
00CA12A2	~ 7C 04	JL SHORT RKG1.00CA12A8	de lo contrario va eliminando el serial por no cumplir
00CA12A4	. 3C 5A	CMP AL,5A	cumplir esa condicion
00CA12A6	~ 7E 06	JLE SHORT RKG1.00CA12AE	
00CA12A8	> 2C 30	SUB AL,30	
00CA12AA	. 3C 09	CMP AL,9	
00CA12AC	~ 77 03	JR SHORT RKG1.00CA12B1	
00CA12AE	> 46	INC ESI	
00CA12AF	~ EB 26	JMP SHORT RKG1.00CA12D7	
00CA12B1	> 8D56 01	LEA EDX,DWORD PTR DS:[ESI+1]	
00CA12B4	. 8BCA	MOV ECX,EDX	
00CA12B6	. 8D59 01	LEA EBX,DWORD PTR DS:[ECX+1]	
00CA12B9	0F	DB 0F	
00CA12BA	1F	DB 1F	
00CA12BB	80	DB 80	
00CA12BC	00	DB 00	
00CA12BD	00	DB 00	
00CA12BE	00	DB 00	
00CA12BF	00	DB 00	
00CA12C0	> 8A01	MOV AL,BYTE PTR DS:[ECX]	
00CA12C2	. 41	INC ECX	
00CA12C3	. 84C0	TEST AL,AL	
00CA12C5	^ 75 F9	JNZ SHORT RKG1.00CA12C0	
00CA12C7	. 2BCB	SUB ECX,EBX	
00CA12C9	. 8D41 01	LEA EAX,DWORD PTR DS:[ECX+1]	Va a copiar en memoria nuestro serial. Quita primer caracte
00CA12CC	. 50	PUSH EAX	
00CA12CD	. 52	PUSH EDX	
00CA12CE	. 56	PUSH ESI	
00CA12CF	. E8 191D 00	CALL <JMP.&VCRUNTIME140.memcpy>	
00CA12D4	. 83C4 0C	ADD ESP,0C	
00CA12D7	> 8A06	MOV AL,BYTE PTR DS:[ESI]	
00CA12D9	. 84C0	TEST AL,AL	
00CA12DB	^ 75 C3	JNZ SHORT RKG1.00CA12A0	
00CA12DD	> 8B35 444	MOV ESI,DWORD PTR DS:[444]	

Registers (3DNow!)	
EAX	00000031
ECX	0030FC80 ASCII "11111-22222-33333-44444-55534"
EDX	0000000A
EBX	7FFD6000
ESP	0030F988
ESI	0030FC80 ASCII "11111-22222-33333-44444-55534"
EDI	0030FC80 ASCII "11111-22222-33333-44444-55534"

Address	Hex dump	ASCII
0030FC80	31 31 31 31 31 2D 32 32 32 32 32 33 33 33 33	11111-22222-33333
0030FC90	33 2D 34 34 34 34 34 2D 35 35 35 33 34 00 00 00	3-44444-55534...
0030FCA0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Hardware breakpoint 2 at RKG1.00CA1291 - EIP points to next instruction

Resumiendo un poco, hemos llegado donde nuestro **Serial** es limpiado de todo lo que no sea números o letras mayúsculas, que para nuestro caso son los guiones (-). He colocado las capturas de los **REGISTROS** y el **DUMP** en el **DESENSAMBLADO** para ganar espacio. Podemos notar que nos hemos detenido gracias al **<HARDWARE-BREAKPOINT-ON ACCES>**. Bueno, limpiemos nuestro **Serial** de la basurita y miremos cómo nos queda para hacerle su seguimiento.

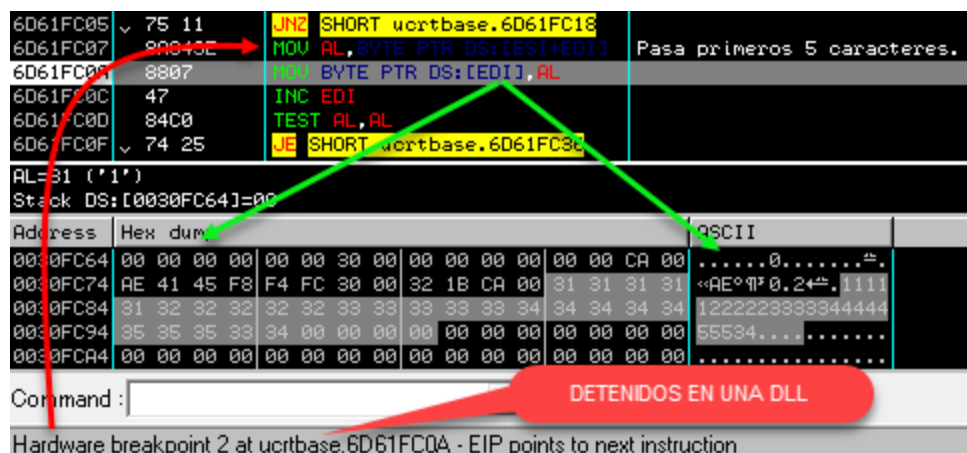
00CA12C7	. 2BCB	SUB ECX,EBX	
00CA12C9	. 8D41 01	LEA EAX,DWORD PTR DS:[ECX+1]	Va a copiar en memoria nuestro serial. Quita primer caracte
00CA12CC	. 50	PUSH EAX	
00CA12CD	. 52	PUSH EDX	
00CA12CE	. 56	PUSH ESI	
00CA12CF	. E8 191D 00	CALL <JMP.&VCRUNTIME140.memcpy>	
00CA12D4	. 83C4 0C	ADD ESP,0C	
00CA12D7	> 8A06	MOV AL,BYTE PTR DS:[ESI]	
00CA12D9	. 84C0	TEST AL,AL	
00CA12DB	^ 75 C3	JNZ SHORT RKG1.00CA12A0	
00CA12DD	> 8B35 444	MOV ESI,DWORD PTR DS:[444]	

Address	Hex dump	ASCII
0030FC80	31 31 31 31 31 32 32 32 32 32 32 33 33 33 34	1111122222333334
0030FC90	34 34 34 34 35 35 35 33 34 00 00 00 00 00 00 00	444455534.....

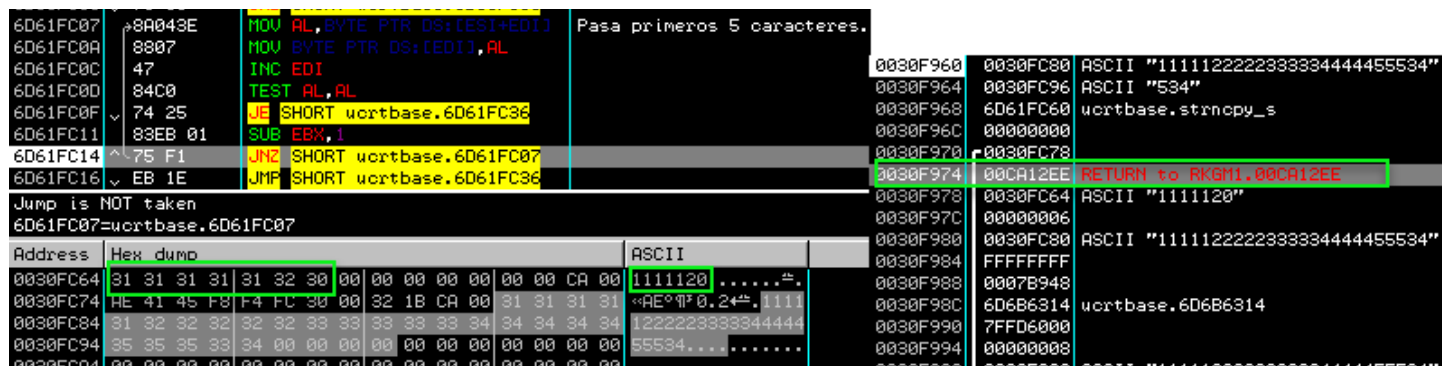
Hice esa rutina hasta llegar a **00CA12D9** y si miramos en el **DUMP** ahí mismo donde estaba el **Serial** con sus guiones (-), lo coloca nuevamente pero ya limpio. Cosas para resaltar y reforzar conocimientos, como pueden ver yo coloqué un **<BREAKPOINT>** en la dirección **00CA12C9** y dejé un comentario para saber que ahí guardamos nuestro **Serial** ya limpio, y para no repetir ese código paso a paso lo iba pasando con **<F9>** y así podía ver más rápido la limpieza, bueno y a qué viene esto, pues caigan en cuenta que el **<HARDWARE-BREAKPOINT-ON ACCES>** no se activó y es porque ese tipo de

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

<HARDWARE-BREAKPOINT> se activa solo cuando se lee lo que hay en memoria y aquí se escribió. Lo más seguro es que ya lo supieran pero lo quería dejar como teoría de refuerzo. Bueno, por fortuna el **Serial** está en nuestro <HARDWARE-BREAKPOINT-ON ACCES>, así que sigamos con <F9> que seguro parara cuando valla a leer el **Serial** para partirlo en 5 partes.



Ahí nos detuvimos, vuelvo y repito lo que ya sabemos pero me gusta reforzar, la flecha **ROJA** muestra la dirección que activó nuestro <HBP> y estamos detenidos una instrucción más abajo la cual guardará los 5 caracteres para tener una parte del **Serial** partido, para ser exactos lo ha mediante un **LOOP**. Si miramos bien esto lo hace una API porque estamos parador en una **DLL** del sistema. Hagamos ese **LOOP** para ver cómo los va moviendo.



Listo, hicimos el **LOOP** y podemos ver en el **DUMP** que movimos nuestros primeros caracteres que serían "11111" y un poco más pero que no es relevante, lo importante son los 5 primeros, Ahora nos queda regresar a nuestro **EXE** y ver de dónde es que venimos y para eso nos apoyamos en el **STACK** en donde podemos ver que retornamos a nuestra aplicación en **RETURN to RKG1.00CA12EE**, vayamos a esa dirección de retorno con la ayuda del **STACK**, eso no lo explico porque ya lo debemos saber.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Address	Hex dump	Disassembly	Comment
00CA12E3	. 8D45 EC	LEA EAX,DWORD PTR SS:[EBP-14]	
00CA12E6	. 6A FF	PUSH -1	VA A COPIAR PRIMER SEGMENTO SERIAL. 5 CARACTERES.
00CA12E8	. 57	PUSH EDI	
00CA12E9	. 6A 06	PUSH 6	
00CA12EB	. 50	PUSH EAX	
00CA12EC	. FFD6	CALL ESI	<api-ms-win-ort-string-l1-1-0.strncpy_s>
00CA12EE	. 6A FF	PUSH -1	
00CA12F0	. 8D47 05	LEA EAX,DWORD PTR DS:[EDI+5]	VA A COPIAR SEGUNDO SEGMENTO. 5 CARACTERES.
00CA12F3	. 50	PUSH EAX	
00CA12F4	. 8D45 F4	LEA EAX,DWORD PTR SS:[EBP-C]	
00CA12F7	. 6A 06	PUSH 6	
00CA12F9	. 50	PUSH EAX	
00CA12FA	. FFD6	CALL ESI	
00CA12FC	. 6A FF	PUSH -1	
00CA12FE	. 8D47 0A	LEA EAX,DWORD PTR DS:[EDI+A]	VA A COPIAR TERCER SEGMENTO. 5 CARACTERES.
00CA1301	. 50	PUSH EAX	
00CA1302	. 8D45 DC	LEA EAX,DWORD PTR SS:[EBP-24]	
00CA1305	. 6A 06	PUSH 6	
00CA1307	. 50	PUSH EAX	
00CA1308	. FFD6	CALL ESI	
00CA130A	. 6A FF	PUSH -1	
00CA130C	. 8D47 0F	LEA EAX,DWORD PTR DS:[EDI+F]	VA A COPIAR QUINTO CUARTO. 5 CARACTERES.
00CA130F	. 50	PUSH EAX	
00CA1310	. 8D45 D4	LEA EAX,DWORD PTR SS:[EBP-2C]	
00CA1313	. 6A 06	PUSH 6	
00CA1315	. 50	PUSH EAX	
00CA1316	. FFD6	CALL ESI	
00CA1318	. 83C4 40	ADD ESP,40	
00CA131B	. 8D47 14	LEA EAX,DWORD PTR DS:[EDI+14]	VA A COPIAR QUINTO SEGMENTO. 5 CARACTERES.
00CA131E	. 6A FF	PUSH -1	
00CA1320	. 50	PUSH EAX	
00CA1321	. 8D45 E4	LEA EAX,DWORD PTR SS:[EBP-1C]	
00CA1324	. 6A 06	PUSH 6	

En **00CA12EE** deberíamos retornar y ahí les muestro dónde se mueven las partes del **Serial** para dejarlo ya partido con sus **5** partes. Ustedes pueden analizar más a profundidad esa rutina. Una vez que separamos nuestro **Serial** debemos colocarle un **<HARDWARE-BREAKPOINT-ON ACCES>** a esas partes para saber qué caculos se hacen con ellas.

Como sabemos que el **<OllyDBG v1.10>** solo permite **4 <HARDWARE-BREAKPOINT>**, entonces seleccionamos las primeras **3** partes y le ponemos su **<HBP-ON ACCES>** y ejecutamos con **<F9>** para parar ya sean cuando acceda a leer una de esas **3** partes o el serial completo. Luego miro si estoy en el lugar correcto.

Address	Hex dump	Disassembly	Comment	Registers (32Now!)
0124112A	. C3	RETN		EAX 00000031
0124112B	> 8A06	MOV AL,BYTE PTR DS:[ESI]		ECX 001EF8E4 ASCII "11111"
0124112D	. 84C0	TEST AL,AL		EDX FFFFFFFF
0124112F	^ 74 F6	JE SHORT RKGM1.01241127		EBX 7FFDF000
01241131	. 39C9	XOR ECX,ECX		ESP 001EF600
01241133	. 39D2	XOR EDX,EDX		EBP 001EF8F8
01241135	> 3C 31	CMP AL,31		ESI 001EF8E4 ASCII "11111"
01241137	^ 7C 04	JLE SHORT RKGM1.0124113D	SALTA SI AL<31. SALTARA UNICAMENTE CON UN 0.	EDI 001EF900 ASCII "111112222333334444455534"
01241139	. 3C 39	CMP AL,39		
0124113B	> 7E 19	JLE SHORT RKGM1.01241156	SALTA SI AL<=39. COMPROBAR SI ES NUMERO. SALTAR	
0124113D	> 3C 61	CMP AL,61		
0124113F	^ 7C 04	JLE SHORT RKGM1.01241145		
01241141	. 3C 66	CMP AL,66		
01241143	^ 7E 11	JLE SHORT RKGM1.01241156		
01241145	> 3C 41	CMP AL,41		
01241147	> 7C 04	JLE SHORT RKGM1.0124114D	COMPROBAMOS SI TENEMOS LETRA ENTRE A - E	
01241149	. 3C 46	CMP AL,46		
0124114B	> 7E 09	JLE SHORT RKGM1.01241156	SALTA AL<=46. DESDE A-F	
0124114D	> 8A43 10	MOV AL,BYTE PTR DS:[ECX+ESI+10]		
01241151	. 41	INC ECX		
01241152	. 84C0	TEST AL,AL		
01241154	^ 75 DF	JNZ SHORT RKGM1.01241135		

RETORNO

Paramos cuando trabaja con nuestra **PARTE 1** del **Serial** y ya conocemos esa rutina que pasará nuestro **"11111"** a un valor **HEXA**, **0x11111**. Hagamos la rutina y lleguemos a la dirección del retorno **01241334**.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Address	Hex	dump	Disassembly	Comment	Registers (3DNow!)
01241318	. 83C4	4	ADD ESP,40		EAX 00011111
0124131B	. 8D47	1	LEA EAX,DWORD PTR DS:[EDI+14]	VA A COPIAR QUINTO SEGMENTO. 5 CARACTERES.	ECX FF FFFF0
0124131E	. 6A FF		PUSH -1		EDX 00011111
01241320	. 50		PUSH EAX		EBX 77 FDF000
01241321	. 8D45	E	LEA EAX,DWORD PTR SS:[EBP-1C]		ESP 01 EF608
01241324	. 6A 06		PUSH 6		EBP 001EF8F8
01241326	. 50		PUSH EAX		ESI 6D61FC60 uortbase.stncpy_s
01241327	. FFD6		CALL ESI		EDI 001EF900 ASCII "111112222233334444455534"
01241329	. 83C4	1	ADD ESP,10		
0124132C	. 8D4D	E	LEA ECX,DWORD PTR SS:[EBP-14]		
0124132F	. E8 EC		CALL RKGM1.01241120	TRABAJA CON PRIMER SEGMENTO. 5 CARACTERES.	
01241334	. 35 35	3	XOR EAX,53135		
01241339	. 33C9		XOR ECX,ECX		
0124133B	. 8985	1	MOV DWORD PTR SS:[EBP-2E4],EAX	GUARDA VALOR XOREADO.	
01241341	. 33FF		XOR EDI,EDI		

Salimos a una direcci3n que es contigua en donde se copia la **PARTE 5** del **Serial**, eso nos indica que podemos siempre tracear tranquilamente desde **0124132C** y ver qu3 sucede con el **Serial** completo o sus **5** partes. Aqu3 tenemos nuestro primer c3lculo que se hace con nuestra **PARTE 1** del **Serial** y es un **XOR** con **0x53135** y luego m3s abajo guarda ese valor **XOREADO** en **0124133B**.

Address	Hex	dump	Disassembly	Comment	Registers (3DNow!)
0124132F	. E8 EC		CALL RKGM1.01241120	TRABAJA CON PRIMER SEGMENTO. 5 CARACTERES.	EAX 00042024
01241334	. 35 35	3	XOR EAX,53135		ECX 00000000
01241339	. 33C9		XOR ECX,ECX		EDX 00011111
0124133B	. 8985	1	MOV DWORD PTR SS:[EBP-2E4],EAX	GUARDA VALOR XOREADO.	EBX 77 FDF000
01241341	. 33FF		XOR EDI,EDI		ESP 001EF608
01241343	. 898D	2	MOV DWORD PTR SS:[EBP-2DC],ECX		EBP 001EF8F8
					ESI 6D61FC60 uortbase.stncpy_s
					EDI 001EF900 ASCII "111112222233334444455534"

El valor a guardar ser3a **0x42024**, recordemos este valor porque m3s adelante es utilizado, record3moslo como **C3LCULO-1**.

Address	Hex	dump	Disassembly	Comment
0124134E	. 0000		ADD BYTE PTR DS:[EAX],AL	
01241350	> 8A543D		MOV DL,BYTE PTR SS:[EBP+EDI-14]	TRABAJA CON PRIMER SEGMENTO. 5 CARACTERES.
01241354	. 8D4A	9	LEA ECX,DWORD PTR DS:[EDX-61]	
01241357	. 80F9	0	CMP CL,5	
0124135A	. 77 08		JR SHORT RKGM1.01241364	SALTA SI AL>5. SIEMPRE SALTARA PORQUE NESECITA LETRA MINUSCULA PARA NO SALTAR.
0124135C	. 0BFEF2		MOVSX ESI,DL	
0124135F	. 83EE	5	SUB ESI,57	
01241362	. EB 22		JMP SHORT RKGM1.01241386	
01241364	> 8AC2		MOV AL,DL	
01241366	. 2C 41		SUB AL,41	
01241368	. 3C 05		CMP AL,5	
0124136A	. 77 08		JR SHORT RKGM1.01241374	SALTA SI AL>5. SI ES NUMERO. VALORES HEXA.
0124136C	. 0BFEF2		MOVSX ESI,DL	
0124136F	. 83EE	3	SUB ESI,37	
01241372	. EB 12		JMP SHORT RKGM1.01241386	PASA A LA SEGUNDA PARTE DEL SERIAL.
01241374	> 8AC2		MOV AL,DL	
01241376	. 2C 30		SUB AL,30	
01241378	. 3C 09		CMP AL,9	
0124137A	. 77 08		JR SHORT RKGM1.01241384	
0124137C	. 0BFEF2		MOVSX ESI,DL	
0124137F	. 83EE	3	SUB ESI,30	
01241382	. EB 02		JMP SHORT RKGM1.01241386	
01241384	. 33F6		XOR ESI,ESI	
01241386	> 8A4C3D		MOV CL,BYTE PTR SS:[EBP+EDI-C]	TRABAJA CON SEGUNDO SEGMENTO. 5 CARACTERES.
0124138A	. 8AC1		MOV AL,CL	
0124138C	. 2C 61		SUB AL,61	
0124138E	. 3C 05		CMP AL,5	
01241390	. 77 08		JR SHORT RKGM1.0124139E	SALTA SI AL>5. SIEMPRE SALTARA PORQUE NESECITA LETRA MINUSCULA PARA NO SALTAR.
01241392	. 0BFEF2		MOVSX EDX,CL	
01241395	. 83EA	5	SUB EDX,57	
01241398	. EB 22		JMP SHORT RKGM1.012413BC	
0124139A	> 8AC1		MOV AL,CL	
012413C9	. 0F1F		???	Unknown command
012413CB	. 8000	0	ADD BYTE PTR DS:[EAX],0	
012413CE	. 0000		ADD BYTE PTR DS:[EAX],AL	
012413D0	. 8BC1		MOV EAX,ECX	
012413D2	. C1E1	0	SAL ECX,4	
012413D5	. 2BC8		SUB ECX,EAX	
012413D7	. 83EB	0	SUB EBX,1	
012413DA	. 75 F4		JNZ SHORT RKGM1.012413D0	
012413DC	. 33D6		XOR EDX,ESI	
012413DE	. 47		INC EDI	
012413DF	. 0FAFD1		IMUL EDX,ECX	
012413E2	. 0195	2	ADD DWORD PTR SS:[EBP-2DC],EDX	GUARDA VALOR DEL CALCULO. PRIMERA Y SEGUNDA SECCION.
012413E8	. 83FF	0	CMP EDI,5	
012413EB	. 75 F2		JBE RKGM1.012413E0	SALTA SI EDI<5

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Si seguimos traceando hasta llegar a **012A1350** que es el inicio de un **LOOP** que realiza un nuevo cálculo con las partes **1** y **2** del **Serial** para hallar otro valor, el cual lo guarda cuando pasamos por la dirección **012A13E2** en donde va sumando siempre el valor de **EDX**. Este **LOOP** se repite **5** veces, hagámoslo todo y miremos cuál es el valor final de esa suma en **012A13E2 ADD DOWRD PTR SS:[EBP-2DC],EDX**.

```
012A13DC > 33D6 XOR EDI,ESI
012A13DE . 47 INC EDI
012A13DF . 0FAFD1 IMUL EDI,EAX
012A13E2 . 0195 24F0FF ADD DWORD PTR SS:[EBP-2DC],EDX GUARDA VALOR DEL CALCULO. PRIMERA Y SEGUNDA SECCION
012A13E8 . 83FF 05 CMP EDI,5
012A13EB . ^ 0F82 5FFFFF JB RKG1.012A1350 SALTA SI EDI<5

Jump is NOT taken
012A1350=RKG1.012A1350

Address Hex dump ASCII
0023F9FC CD 7B 02 00 30 C0 3A 00 0B 00 00 00 00 00 00 00 =(0.04:..0.....
0023FA0C 63 00 00 01 00 00 00 00 0B 00 00 00 94 DC 00 00 c..0....0...0...
```

Listo, ya repetí mi **LOOP** **5** veces y me encuentro detenido en **012A13EB**, en donde podemos ver que ya no tomamos el salto para repetir el **LOOP**. Ahora lo importante es saber el valor final de la suma en **012A13E2 ADD DOWRD PTR SS:[EBP-2DC],EDX**, que terminó siendo **0x27BCD**. A este nuevo valor lo llamaremos **CÁLCULO-2**. Ya saliendo de ese **LOOP**, pasamos a la instrucción siguiente.

```
012A13F1 . 804D F4 LEA ECX,DWORD PTR SS:[EBP-0]
012A13F4 . 33F6 XOR ESI,ESI
012A13F6 . E8 25F0FFFF CALL RKG1.012A1120 TRABAJA CON SEGUNDO SEGMENTO. 5 CARACTERES.
012A13FB . 8BD0 20F0FFFF MOV EDI,DWORD PTR SS:[EBP-2E0] TRABAJA CON PRIMER SEGMENTO. ORIGINA 5 SEGMENTO.
012A1401 . 33DB XOR EBX,EBX
012A1403 > 0FB0C37 MOVZX ECX,BYTE PTR DS:[EDI+ESI]
012A1407 . 46 INC ESI
012A1408 . 41 INC ECX
012A1409 . 0FAFC8 IMUL ECX,EAX
012A140C . 03D9 ADD EBX,ECX
012A140E . 83FE 05 CMP ESI,5
012A1411 . ^ 7C F0 JN SHORT RKG1.012A1403
012A1413 . 804D E4 LEA ECX,DWORD PTR SS:[EBP-1C]
012A1416 . 81E3 FFFF0F AND EBX,0FFFFFFF
012A141C . E8 FFFCFFFF CALL RKG1.012A1120 TRABAJA CON QUINTO SEGMENTO. 5 CARACTERES.
012A1421 . 3BD8 CMP EBX,EAX COMPARA CALCULO PRIMER SEGMENTO = QUINTO SEGMENTO.
012A1423 . ^ 0F85 7A050000 JNZ RKG1.012A19A3 Salto decisivo
```

Registers (FPU)	
EAX	0000002F
ECX	0023FCCC ASCII "22222"
EDX	00025143
EBX	00000000
ESP	0023F9E8
EBP	0023FCD8
ESI	00000000
EDI	00000005

Tenemos resaltado en dos recuadros para explicarlo mejor, notemos que hacemos dos veces el **CALL RKG1.012A1120**, vuelvo y lo repito, ese **CALL** simplemente lo que hace es pasar nuestras partes del **Serial** a un valor **HEXADECIMAL** para posteriormente hacer cálculos con este. Empezamos por el recuadro **ROJO**, pasaremos la **PARTE 2** de nuestro **Serial** **"22222"** a un valor **HEXA** (**0x22222**), por eso en **ECX** tenemos nuestra parte del **Serial** que cuando hagamos esa **CALL** retornará en **EAX** nuestra valor **HEXA** de la parte del **Serial** para luego entrar a otro **LOOP** donde hará unos cálculos con nuestro valor **HEXA**, o sea nuestra **PARTE 2** del **Serial**, que será una suma en **012A140C ADD EBX,ECX** y si miramos en los **REGISTROS**, **EBX** vale **0** porque está listo para recibir el valor de la suma mientras se ejecuta el **LOOP**. Luego seguimos con el recuadro **AZUL** que simplemente es para pasar unas de las partes del **Serial**, que gracias a las observaciones que puse sabemos que es la **PARTE 5** del **Serial** para luego hacer la comparación y tomar el **Salto decisivo**. Hagamos el **LOOP** para ver cuánto termina valiendo **EBX**.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

012A1408	. 41	INC ECX		
012A1409	. 0FAFC8	IMUL ECX, EAX		
012A140C	. 03D9	ADD EBX, ECX		
012A140E	. 83FE 05	CMP ESI, 5		
012A1411	. 7C F0	JL SHORT RKG1.012A1403		
012A1413	. 804D E4	LEA ECX, DWORD PTR SS:[EBP-1C]		
012A1416	. 81E3 FFFF0F0	AND EBX, 0FFFFFFF		
012A141C	. E8 FFFCFFFF	CALL RKG1.012A1126		
012A1421	. 3BD8	CMP EBX, EAX		
012A1423	. 0F85 7A05000	JNZ RKG1.012A19A3		
012A1429	. 83FB 01	CMP EBX, 1		

Registers (FPU)
 EAX 00022222
 ECX 006AAAA4
 EDX 00022222
EBX 02155534
 ESP 0023F9E8
 EBP 0023FCD8
 ESI 00000005
 EDI 0023FCE0 ASCII "

TRABAJA CON QUINTO SEGMENTO. 5 CARACTERES.
 COMPARA CALCULO PRIMER SEGMENTO = QUINTO SEGMENTO.
 Salto decisivo

EBX 02155534, que sería nuestro valor **CÁLCULO-3**. Ahí podemos ver algo muy relevante, y es que si observas y recuerdas muy bien te darás cuenta que los últimos 5 valores de **EBX** son la **PARTE 5** de nuestro **Serial**, "55534", pues ese **LOOP** que acabamos de pasar nos da nuestro **CÁLCULO-3**, que a partir de la **PARTE 2** de nuestro **Serial** nos crea la **PARTE 5** del **Serial**; pues ahí está el truco el **CÁLCULO-3=PARTE 5**, en las observaciones que puse en el Olly hago referencia que el **CALCULO PRIMER SEGMENTO=QUINTO SEGEMENTO**, ahí puse mal eso porque en realidad es la **PARTE 2** del **Serial**. Entonces podemos suponer que los otros cálculos que hicimos son en realidad partes del **Serial** verdadero. Con lo que sabemos ahora podemos plantear que el **Serial** verdadero se origina a partir de los 10 primeros caracteres, los cuales son tomados en dos partes de 5 caracteres (por eso yo lo puse en partes) y a partir de estas dos partes se calculan los otros 15 caracteres que vendrían siendo mis otras tres partes de mi **Serial**. Hasta aquí podemos plantear la estructura de nuestro **Serial** válido así:

PARTE 1-PARTE 2-PARTE 3 (CÁLCULO-?)-PARTE 4 (CÁLCULO-?)-PARTE 5 (CÁLCULO-3)

La **PARTE 1** y la **PARTE 2** las debemos generar nosotros en nuestro **KeyGen** de manera aleatoria y con lo que sabemos hasta aquí la única condición es que deben ser números y letras mayúsculas que sean parte de la familia de los **HEXADECIMALES**, y con estas dos partes originamos las otras tres.

Sabemos que ya tenemos los tres cálculos y que **PARTE 5 (CÁLCULO-3)**, así que solo nos queda saber a qué partes pertenecen **CÁLCULO-1** y **CÁLCULO-2**, por eso en mi estructura de **Serial** tengo como **PARTE 3 (CÁLCULO-?)-PARTE 4 (CÁLCULO-?)** en incógnitas. En nuestro **KeyGen** debemos programar las rutinas que generan los cálculos, **CÁLCULO-1**, **CÁLCULO-2** y **CÁLCULO-3**. Listo, lleguemos hasta **012A1416** para ver cómo el panorama se nos aclara más.

012A140E	. 83FE 05	CMP ESI, 5		
012A1411	. 7C F0	JL SHORT RKG1.012A1403		
012A1413	. 804D E4	LEA ECX, DWORD PTR SS:[EBP-1C]		
012A1416	. 81E3 FFFF0F0	AND EBX, 0FFFFFFF		
012A141C	. E8 FFFCFFFF	CALL RKG1.012A1126		
012A1421	. 3BD8	CMP EBX, EAX		
012A1423	. 0F85 7A05000	JNZ RKG1.012A19A3		
012A1429	. 83FB 01	CMP EBX, 1		

Registers (FPU)
 EAX 00022222
 ECX 0023FCBC ASCII "55534"
 EDX 00022222
 EBX 02155534
 ESP 0023F9E8
 EBP 0023FCD8
 ESI 00000005

TRABAJA CON QUINTO SEGMENTO. 5 CARACTERES.
 COMPARA CALCULO SEGUNDO SEGMENTO = QUINTO SEGMENTO.
 Salto decisivo

La flecha **ROJA** nos muestra que en **ECX** tenemos la **PARTE 5** del **Serial** la cual será pasada a valor **HEXA** con el **CALL RKG1.012A1120** que será retornado en **EAX** y la flecha **VERDE** nos muestra que vamos a **XOREAR EBX** para dejarlo ya limpio para la comparación **012A1421 CMP EBX, EAX**. Lleguemos a la comparación.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

012A1413	. 804D E4	LEA ECX,DWORD PTR SS:[EBP-1C]		Registers (FPU)
012A1416	. 81E3 FFFF0F0	AND EBX,0FFFFFFF		EAX 00055534
012A141C	. E8 FFFCFFFF	CALL RKGM1.012A1120	TRABAJA CON QUINTO SEGMENTO. 5 CARACTERES.	ECX 000221F3
012A1421	. 3B08	CMP EBX,EAX	COMPARA CALCULO SEGUNDO SEGMENTO - QUINTO SEGMENTO	EDX 00055534
012A1423	. 0F85 7A05000	JNZ RKGM1.012A19A3	Salto desicivo	EBX 00055534
012A1429	. 83FB 01	CMP EBX,1		ESP 0023F9E8
				EBP 0023FCD8
				ESI 00000005
				EDI 0023FCE0 ASCII "1111"

Ahí lo tenemos, se hace un **AND EBX,0FFFFFFF** y con eso obtenemos el **Serial** limpio, bueno no el **Serial** si no el valor con el cual va a ser comparado nuestra parte del **Serial** convertido en un valor **HEXA**, y como tenemos igualdad pasamos la primera validación y luego siguen esas validaciones de lista negra que con este **Serial** pasaremos sin inconvenientes. Lleguemos hasta la última validación de la lista negra en la dirección **01291724** para ver qué sigue después de esa chorrera de validaciones.

Address	Hex dump	Disassembly	Comment
01291724	. 0F85 7902000	JNZ RKGM1.012919A3	
0129172A	. 804D DC	LEA ECX,DWORD PTR SS:[EBP-24]	
0129172D	. E8 EEF9FFFF	CALL RKGM1.01291120	TRABAJA CON TERCER SEGMENTO. 5 CARACTERES.
01291732	. 8B8D 1CFDFFF	MOV ECX,DWORD PTR SS:[EBP-2E4]	MUEVE EL VALOR DEL PRIMER SEGMENTO XOREADO
01291738	. 81E1 FFFF0F0	AND ECX,0FFFFFFF	
0129173E	. 3BC8	CMP ECX,EAX	TERCER SEGMENTO = PRIMER SEGMENTO XOR 53135
01291740	. 0F85 5D02000	JNZ RKGM1.012919A3	COMPRUEBA TERCER SEGMENTO DE NUESTRO SERIAL
01291746	. 804D D4	LEA ECX,DWORD PTR SS:[EBP-2C]	
01291749	. E8 D2F9FFFF	CALL RKGM1.01291120	TRABAJA CON CUARTO SEGMENTO. 5 CARACTERES.
0129174E	. 8B8D 24FDFFF	MOV ECX,DWORD PTR SS:[EBP-2DC]	
01291754	. 81E1 FFFF0F0	AND ECX,0FFFFFFF	
0129175A	. 3BC8	CMP ECX,EAX	CUARTO SEGMENTO = RUTINA CALCULO PRIMERSEGMENTO.
0129175C	. 0F85 4102000	JNZ RKGM1.012919A3	COMPRUEBA CUARTO SEGMENTO DE NUESTRO SERIAL

Esto es una maravilla, aquí por fin sabremos a qué partes pertenecen nuestros **CÁLCULO-1** y **CÁLCULO-2**. En las observaciones que coloqué se explica qué se hace en cada recuadro, cuando ustedes estén traceando por aquí lo verán más claro. En el recuadro **ROJO** se compara el **CÁLCULO-1** con la **PARTE 3** de nuestro **Serial**, entonces **CÁLCULO-1= PARTE 3**. Y en el recuadro **AZUL** podemos ver que la comparación la haremos con la **PARTE 4** del **Serial** con nuestro **CÁLCULO-2**, así que **CÁLCULO-2= PARTE 4**. Notemos que nuestros **CÁLCULO-1** y **CÁLCULO-2** se les hace un **AND EBX,0FFFFFFF** y es para dejarlos únicamente con los 5 valores **HEXA** a comparar. Ahora debemos recordar cuáles eran los valores, **CÁLCULO-1=0x42024** y **CÁLCULO-2=0x27BCD**. Entonces con nuestra estructura de **Serial**, reemplazamos con los nuevos valores.

PARTE 1-PARTE 2-PARTE 3 (CÁLCULO-?)-PARTE 4 (CÁLCULO-?)-PARTE 5 (CÁLCULO-3)

11111-22222-42024-27BCD-55534

Reiniciemos todo y metamos nuestro nuevo **Serial**, "**11111-22222-42024-27BCD-55534**", y paramos en **01291740** para ver si tenemos la igualdad para pasar esas validaciones con las partes **3** y **4** de nuestro **Serial**.

0129172A	. 804D	LEA ECX,DWORD PTR SS:[EBP-24]		Registers (FPU)
0129172D	. E8 EE	CALL RKGM1.01291120	TRABAJA CON TERCER SEGMENTO. 5 CARACTERES.	EAX 00042024
01291732	. 8B8D	MOV ECX,DWORD PTR SS:[EBP-2E4]	MUEVE EL VALOR DEL PRIMER SEGMENTO XOREADO	ECX 00042024
01291738	. 81E1	AND ECX,0FFFFFFF		EDX 00042024
0129173E	. 3BC8	CMP ECX,EAX	TERCER SEGMENTO = PRIMER SEGMENTO XOR 53135	EBX 00055534
01291740	. 0F85	JNZ RKGM1.012919A3	COMPRUEBA TERCER SEGMENTO DE NUESTRO SERIAL	ESP 001BFBA8
01291746	. 804D	LEA ECX,DWORD PTR SS:[EBP-2C]		EBP 001BF500

Perfecto, la **PARTE 3** del **Serial** es correcta y pasamos esa validación, solo nos queda validar la **PARTE 4**, así que traciemos hasta **0129175C**.

Realist KeyGenMe 1 by Bakasura (MVC++) (KeyGen)(OllyDBG v1.10)

Address	Disassembly	Comment	Registers (FPU)
01291746	LEA ECX, DWORD PTR SS:[EBP-2C]		
01291749	CALL RKGMI.01291120	TRABAJA CON CUARTO SEGMENTO. 5 CARACTERES.	EAX 00027BCD
0129174E	MOV ECX, DWORD PTR SS:[EBP-2DC]		ECX 00027BCD
01291754	AND ECX, 0FFFFFFF		EDX 00027BCD
0129175A	CMP ECX, EAX	CUARTO SEGMENTO = Rutina calculo primersegmento.	EBX 00055534
0129175C	JNZ RKGMI.012919A3	COMPRUEBA CUARTO SEGMENTO DE NUESTRO SERIAL	ESP 001BFBA8

¡Ayy mamita! Son iguales, eso es lo que necesitamos, entonces la **PARTE 4** de nuestro **Serial** es correcto. Sigamos traceando que un par de instrucciones más tenemos otra validación. Tracemos hasta **01291762**.

```

01291762 . 807D CMP BYTE PTR SS:[EBP-14],41 → DEBE SER UNA LETRA COMO PRIMER CARACTER EN PRIMER SEGMENTO.
01291766 . 0F8C JL RKG1.012919A3
0129176C . 807D CMP BYTE PTR SS:[EBP-8],42 → ULTIMO CARACTER DEL SEGMENTO2 DEBE SER "B"
01291770 . 0F85 JNZ RKG1.012919A3
01291776 . 57 PUSH EDI
01291777 . FF15 CALL DWORD PTR DS:[<< KERNEL32.1] String
0129177D . 83F8 CMP EAX,19 strlenA
01291780 . 0F8C JL RKG1.012919A3 LONGITUD DE NUESTRO SERIAL DEBE SER 0x19=25
01291786 . 68 C0 PUSH RKG1.01291FC0
0129178B . 51 PUSH EAX
0129178C . 8B0D MOV ECX,DWORD PTR DS:[<<MSUCP14 MSUCP140.?cout@std@@@3U?$basic_ostream@DU?$char_traits@D@std@@@1@A
01291792 . BA 40 MOV EDX,RKG1.01294340 ASCII "Serial Correcto! \n/"
01291797 . F8 84 CALL RKG1.01291D70
Stack SS:[001BFEB4]=31 ('1')

```

Address	Hex	dump	ASCII
001BFEB4	31 31 31 31 31 00 18 00	32 32 32 32 32 00 29 01	11111.+.22222.)0
001BFEB4	1B 0C C3 2B 10 FF 1B 00	32 1B 29 01 31 31 31 31	44444.+.33333.)0

Podemos ver que tenemos esas dos validaciones más por pasar y que se nos convierten en un par de condiciones que deben cumplir nuestro **Serial** en la **PARTE 1** y **PARTE 2**, pero ojo aquí, porque al cambiar algo de la **PARTE 1** y/o **PARTE 2** nuestro **Serial** será otro completamente distinto porque a partir de esas **2** partes se originan las últimas **3**, y es aquí en donde entra en juego mi **KeyGen** ya con todo lo que sabemos podemos hacer uno que origine un **Serial** a partir de lo que sabemos, entonces voy a suponer mis primeras dos partes cumpliendo esas dos condiciones y que a partir de ahí me origine las otras tres. También podemos hacerlo con la aplicación y luego ver y anotar los valores originados como una especie de **SelfKey** pero no lo recomiendo porque lo mejor es ir avanzando en hacer nuestro **KeyGen**.

A1111-2222B-?-?-?

Yo ya tengo mi **KeyGen** finalizado pero lo cambiaré un poco para que trabaje a partir de mis dos primeras partes.

```
Seg1 = "A1111" '--> PRIMERA PARTE DEL SERIAL  
Seg2 = "2222B" '--> SEGUNDAA PARTE DEL SERIAL
```

'El SEGMENTO3 es el primero que se calcula. Es bien sencillo solo hace un XOR al SEGMENTO1.

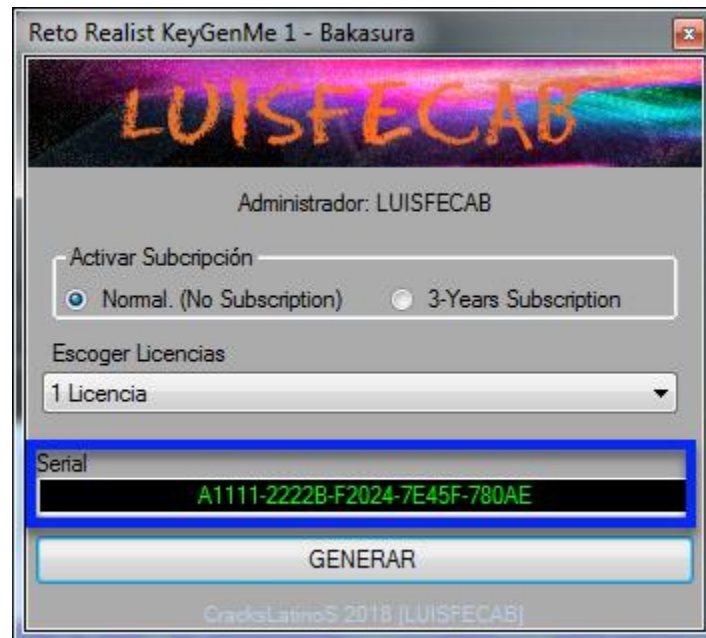
```
Seg3 = Hex(("&H" + Seg1 Xor &H53135) And &HFFFFF)  
'*****  
'  
  
'  
'*****  
'
```

'SEGMENTO1 y SEGMENTO2 para originar SEGMENTO5.

```
EBX = 0  
For Each L In Seg1  
    ECX = Asc(L) + 1  
    ECX = ECX * ConverterNum(Seg2, 16)  
    EBX += ECX  
Next
```


Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Ahí les muestro el inicio del código que de ahí en adelante originará nuestro **Serial**, lo pruebo a ver qué me sale.



Aquí lo relevante es el **Serial** como ven la **PARTE 1** y **PARTE 2** son las que puse hace rato y a partir de esas originé el resto de mi serial.

A1111-2222B-F2024-7E45F-780AE

A probarlo en nuestras dos últimas validaciones para ver si nuestro **KeyGen** hace la tarea bien.

01291762	.	807D	CMP	BYTE PTR SS:[EBP-14], 41	DEBE SER UNA LETRA COMO PRIMER CARACTER EN PRIMER SEGMENTO.
01291766	..	0F8C	JL	RKGM1.012919A3	
0129176C	.	807D	CMP	BYTE PTR SS:[EBP-8], 42	ULTIMO CARACTER DEL SEGMENTO2 DEBE SER "B"
Stack SS:[0016FC34]=41 ('A')					

COMPARA CON 0x41 (A). JL SALTA SI ES MENOR, PERO NUNCA SERÁ MENOR YA QUE TENDREMOS DESDE 0x41 (A) HASTA 0x46 (F). POR ESO NO TOMAMOS EL SALTO

0129175C	..	0F85	JNZ	RKGM1.01291903	COMPRUEBA CUARTO SEGMENTO DE NUESTRO SERIAL
01291762	.	807D	CMP	BYTE PTR SS:[EBP-14], 41	DEBE SER UNA LETRA COMO PRIMER CARACTER EN PRIMER SEGMENTO.
01291766	..	0F8C	JL	RKGM1.012919A3	
Jump is NOT taken					
012919A3=RKGM1.012919A3					

Perfecto, no tomamos el salto así que hasta aquí funciona de maravilla. No saltará porque compara con **0x41 (A)**. **JL** salta si es menor el valor de **[EBP+14]**, pero nunca será menor ya que tendremos **0x41 (A)** hasta **0x46 (F)**. Por eso no tomamos el salto. Ahora solo nos falta la última validación, mirémosla para saber si estamos bien.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

0129176C	. 807D	MOV	BYTE PTR SS:[EBP-8],42	ULTIMO CARACTER DEL SEGMENTO2 DEBE SER "B"
01291770	. 0F85	JNZ	RKGM1.012919A3	
01291776	. 57	PUSH	EDI	String
Stack SS:[0016FC40] 42 ('B')				

AL SER IGUALES NO SALTA

0129176C	. 807D	MOV	BYTE PTR SS:[EBP-8],42	ULTIMO CARACTER DEL SEGMENTO2 DEBE SER "B"
01291770	. 0F85	JNZ	RKGM1.012919A3	
01291776	. 57	PUSH	EDI	String
01291777	. FF15	CALL	DWORD PTR DS:[<<KERNEL32.1	strlenA
Jump is NOT taken				
012919A3=RKGM1.012919A3				

Perfecto, ya tenemos un **Serial** que funciona, solo queda seguir traceando para ver por dónde seguimos.

0129178C	. 8B0D	MOV	ECX,DWORD PTR DS:[<<MSUCP140.?cout@std@03U?%basic_	
01291792	. BA 40	MOV	EDX,RKGM1.01294340	ASCII "Serial Correcto! \o/"
01291797	. E8 D4	CALL	RKGM1.01291D70	
0129179C	. 83C4	ADD	ESP,4	
01294340=RKGM1.01294340 (ASCII "Serial Correcto! \o/")				
EDX=0016FC51, (ASCII "11112222BF20247E45F780AE")				

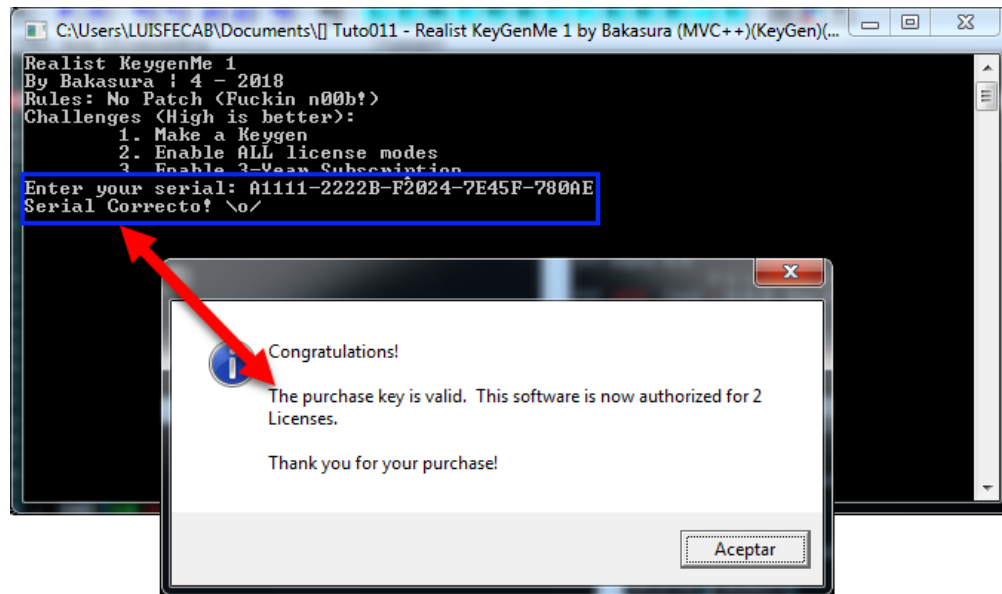
Ya cargamos el mensaje "**Serial Correcto**", con esto podemos decir que nuestro **KeyGen** funciona muy bien pero sabemos que el reto pide algo más, que no es complicado, solo debemos tracear y tracear hasta pillar esas nuevas condiciones que deben tener las dos primeras partes del **Serial**.

012917DA	. 8A4D	MOV	CL, BYTE PTR SS:[EBP-B]	TOMA SEGUNDO CARACTER DEL SEGMENTO2. CONTROLA CANTIDAD LICENCIA
012917DD	. 83C4	ADD	ESP,0C	
012917E0	. 8AC1	MOV	AL,CL	
012917E2	. 2C 61	SUB	AL,61	
012917E4	. 3C 05	CMP	AL,5	SALTA SI AL<=5. UNSIGNED
012917E6	. 76 24	JBE	SHORT RKGM1.0129180C	
012917E8	. 8AC1	MOV	AL,CL	
012917EA	. 2C 41	SUB	AL,41	
012917EC	. 3C 05	CMP	AL,5	SALTA SI AL<=5. UNSIGNED. CON LETRA SALTA - CON NUMERO NO SALTA
012917EE	. 76 14	JBE	SHORT RKGM1.01291804	
012917F0	. 8AC1	MOV	AL,CL	
012917F2	. 2C 30	SUB	AL,30	
012917F4	. 3C 09	CMP	AL,9	
012917F6	. 76 04	JBE	SHORT RKGM1.012917FC	
012917F8	. 33F6	XOR	ESI,ESI	
012917FA	. EB 17	JMP	SHORT RKGM1.01291813	
012917FC	. 0FBF	MOVSX	ESI,CL	
012917FF	. 83EE	SUB	ESI,30	
01291802	. EB 0F	JMP	SHORT RKGM1.01291813	
01291804	. 0FBF	MOVSX	ESI,CL	
01291807	. 83EE	SUB	ESI,37	
0129181F	. 6A FF	PUSH	-1	
01291821	. 68 58	PUSH	RKGM1.01294358	ASCII " plus a 3-Year Subscription"
01291826	. 6A 32	PUSH	32	
01291828	. 50	PUSH	EAX	
01291829	. FFD3	CALL	EBX	<&api-ms-win-crt-string-l1-1-0.strncpy_s>
0129182B	. 83C4	ADD	ESP,10	
0129182E	. 807D	CMP	BYTE PTR SS:[EBP-C],42	COMPARA PRIMER CARACTER DEL SEGMENTO2 CON "B"
01291832	. 74 15	JE	SHORT RKGM1.01291849	SI ES "B". ACTIVA SUSCRIPCION POR 3 YEARS.

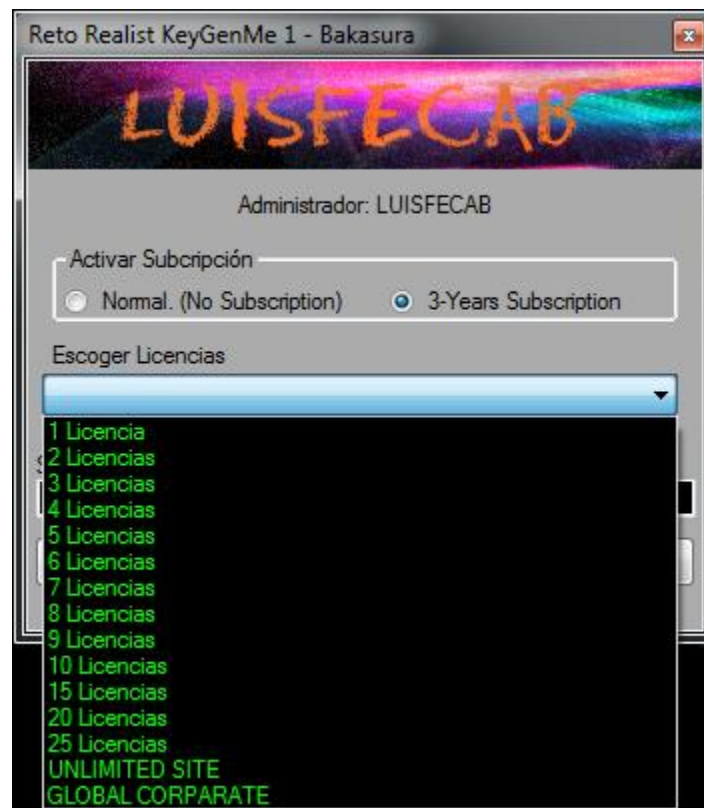
Listo, cuando lleguemos a **012917DA MOV CL, BYTE PTR SS:[EBP-B]** determinaremos la cantidad de **Licencias**, todas esas comparaciones es para saber esa cantidad y eso depende del segundo carácter de la **PARTE 2** de nuestro **Serial**, luego en **0129182E CMP BYTE PTR SS:[EBP-C],42** podemos poner nuestra **Licencias** con **suscripción de tres años** y eso ocurre cuando en la **PARTE 2** de nuestro **Serial** el primer carácter es **"B"**. Listo, hemos resuelto todo, ahora si miramos nuestro serial **"A1111-2222B-F2024-7E45F-780AE"**, podemos inferir mirando la **PARTE 2** que no tendremos la **suscripción**

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

de tres años y que tendremos 2 **Licencias**. Miremos si es verdad, ya por fin terminamos. Presionemos <F9> para conocer al <CHICO BUENO> a ver qué nos dice.



Perfecto amigos, como dijimos dos **Licencias** sin suscripción. Hallar el resto de **Licencias** les toca a ustedes como práctica, solo deben ir cambiando el segundo carácter de la **PARTE 2** de nuestro **Serial**. Lo que respecta al **KeyGen**, ya con lo que sabemos lo podemos programar.



PARA TERMINAR

Tremendo reto, cuando por fin pude entender que era con valores **HEXA**, la cosa se me puso cuesta abajo, después ya era cuestión de agudizar el ojo y tracear para ir pillando cómo funcionaban las rutinas para validar el **Serial**. Es un muy buen reto para mejorar nuestras habilidades con los **KeyGen**, claro está que he encontrado unas rutinas muy difíciles de seguir y descifrar en otras aplicaciones que me han derrotado.

No tengo mucho que decir o aclarar, creo que el tuto me quedó bien explicado, que no me quedó nada en el tintero, solo felicitar y agradecer a **Bakasura** por este buen reto que me puso a tracear y tracear de lo lindo.

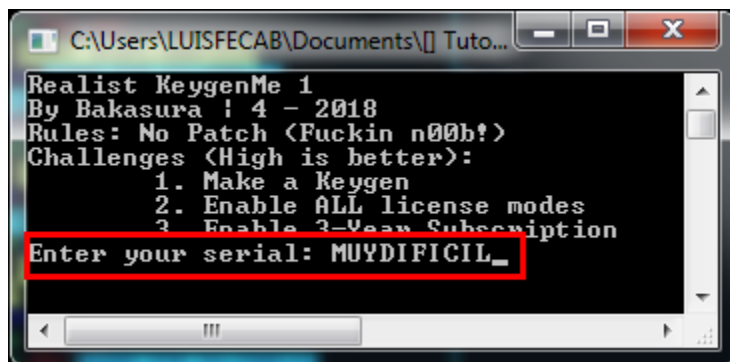
Recuerden que si he dicho algo ambiguo o erróneo les estaría muy agradecido si me lo hacen notar. No olviden también que si encuentran un bug en el **KeyGen** lo pueden corregir y también informármelo.

Saludos a todos,

@LUISFECAB

FE DE ERRATAS

Aquí estoy con algo de pena escribiendo esto, que más que una corrección es un complemento a la solución que planteamos en este tutorial, que así como hicimos el **KeyGen** funciona muy bien, pero anoche no pude dormir y me puse a pensar que algo me faltaba revisar en este reto y analizando en mi mente pensé *¿Qué tal si ingreso mi serial "MUYDIFICIL" y veo qué retorna el CALL RKGM1.012A1120?*, recordemos que ese **CALL** es el que pasa mi **Serial** a valores **HEXA** para hacer nuestros cálculos. Mi **Serial** "MUYDIFICIL" es ideal porque tiene un par de letras que están en el rango **HEXADECIMAL** y otras que no, y además tiene una longitud de **10** con lo cual tendría la **PARTE 1** y **PARTE 2** del **Serial**, y desde ahí ver qué es lo que hace la aplicación con estas. Sé que las dos primeras partes del **Serial** tienen que cumplir unas condiciones que son validadas después, aquí lo que me interesa es el **CALL RKGM1.012A1120** y los cálculos que generan las otras tres partes.



Como ya conozco las direcciones de interés en donde el **CALL RKGM1.012A1120** pasa nuestras partes a valores **HEXA** pues hasta halla voy.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

00B11329	. 83C4 10	ADD ESP,10			Registers (FPU)
00B1132C	. 8D4D EC	LEA ECX,DWORD PTR SS:[EBP-14]			EAX 00000000
00B1132F	. E8 ECFDFFFF	CALL RKGM1.00B11120	TRABAJA CON PRIMER SEGMENTO. 5 CARACTERES.		ECX 0030F95C ASCII "MUYDI"
00B11334	. 35 35310500	XOR EAX,53135			EDX FFFFFFFF
00B11339	. 33C9	XOR ECX,ECX			EBX 7FFDF000
00B1133B	. 8985 1CFDFFF	MOV DWORD PTR SS:[EBP-2E4],EAX	GUARDA VALOR XOREADO.		ESP 0030F680

En la flecha VERDE ya pasamos a ECX 0030F95C ASCII "MUYDI" que es nuestra PARTE 1 del Serial y estamos listos a ejecutar el CALL RKGM1.012A1120 y ver qué nos retorna en EAX, ver la flecha ROJA.

00B11329	. 83C4 10	ADD ESP,10			Registers (FPU)
00B1132C	. 8D4D EC	LEA ECX,DWORD PTR SS:[EBP-14]			EAX 00000000
00B1132F	. E8 ECFDFFFF	CALL RKGM1.00B11120	TRABAJA CON PRIMER SEGMENTO. 5 CARACTERES.		ECX FFFFFFFE8
00B11334	. 35 35310500	XOR EAX,53135			EDX 00000000
00B11339	. 33C9	XOR ECX,ECX			EBX 7FFDF000
00B1133B	. 8985 1CFDFFF	MOV DWORD PTR SS:[EBP-2E4],EAX	GUARDA VALOR XOREADO.		ESP 0030F680

Ahí lo tenemos después de ejecutar ese CALL tenemos que EAX 0000000D y si miramos nuestra PARTE 1, "MUYDI", entonces ese CALL solo pasa la letra "D" a un valor HEXA cosa que ya sabíamos. Lo que quería era averiguar si hacía algo especial con aquellas letras que están fueran del rango HEXADECIMAL pero como vemos en EAX simplemente las desecha, sigamos la flecha ROJA. Con esto nuevo que acabamos de averiguar se expande un poco más las alternativas de generar nuestro Serial ya que la PARTE 1 y PARTE 2 no deben ser conformadas únicamente con valores HEXADECIMALES si no todas las letras, solo debemos tener en cuenta las condiciones para generar las diferentes Licencias. Luego si seguimos la flecha VERDE vemos que hará nuestro primer cálculo.

00B1132C	. 8D4D EC	LEA ECX,DWORD PTR SS:[EBP-14]			Registers (FPU)
00B1132F	. E8 ECFDFFFF	CALL RKGM1.00B11120	TRABAJA CON PRIMER SEGMENTO. 5 CARACTERES.		EAX 00053138
00B11334	. 35 35310500	XOR EAX,53135			ECX 00000000
00B11339	. 33C9	XOR ECX,ECX			EDX 00000000
00B1133B	. 8985 1CFDFFF	MOV DWORD PTR SS:[EBP-2E4],EAX	GUARDA VALOR XOREADO.		EBX 7FFDF000
00B11341	. 33FF	XOR EDI,EDI			ESP 0030F680

Ya conocemos esto, entonces tenemos CÁLCULO-1=PARTE 3=53138. Ya con eso podemos ir armando un serial.

MUYDI-FICIL-53138-PARTE 4 (CÁLCULO-2)-PARTE 5 (CÁLCULO-3)

Debemos seguir con nuestro segundo cálculo, el cual es el más largo y que utiliza la PARTE 1 y PARTE 2 para realizarlo. Este cálculo no hace uso del CALL RKGM1.012A1120, si no que trabaja directamente con los caracteres del Serial para luego decidir qué hacer dependiendo del carácter que va cargando. Creo que esta es la parte más engorrosa para programarla en el KeyGen pero con unas cuantas traceadas uno logra entender lo que hace.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

Address	Hex	dump	Disassembly	Comment
0124134E	0000		ADD BYTE PTR DS:[EAX],AL	
01241350	> 8A5430		MOV DL,BYTE PTR SS:[EBP+EDI-14]	TRABAJA CON PRIMER SEGMENTO. 5 CARACTERES.
01241354	. 8D4A 9		LEA ECX,DWORD PTR DS:[EDX+6]	
01241357	. 80F9 0		CHP CL,5	
0124135A	.. 77 08		JR SHORT RKG1.01241364	SALTA SI AL>5. SIEMPRE SALTARA PORQUE NESECEITA LETRA MINUSCULA PARA NO SALTAR.
0124135C	. 0FBF2		MOVSX ESI,DL	
0124135F	. 83EE 5		SUB ESI,57	
01241362	. EB 22		JMP SHORT RKG1.01241366	
01241364	> 8AC2		MOV AL,DL	
01241366	. 2C 41		SUB AL,41	
01241368	. 3C 05		CHP AL,5	
0124136A	.. 77 08		JR SHORT RKG1.01241374	SALTA SI AL>5. SI ES NUMERO. VALORES HEXA.
0124136C	. 0FBF2		MOVSX ESI,DL	
0124136F	. 83EE 3		SUB ESI,37	
01241372	. EB 12		JMP SHORT RKG1.01241386	PASA A LA SEGUNDA PARTE DEL SERIAL.
01241374	> 8AC2		MOV AL,DL	
01241376	. 2C 30		SUB AL,30	
01241378	. 3C 09		CHP AL,9	
0124137A	.. 77 08		JR SHORT RKG1.01241384	
0124137C	. 0FBF2		MOVSX ESI,DL	
0124137F	. 83EE 3		SUB ESI,30	
01241382	. EB 02		JMP SHORT RKG1.01241386	
01241384	> 33F6		XOR ESI,ESI	
01241386	> 8A4C30		MOV CL,BYTE PTR SS:[EBP+EDI-C]	TRABAJA CON SEGUNDO SEGMENTO. 5 CARACTERES.
0124138A	. 8AC1		MOV AL,CL	
0124138C	. 2C 61		SUB AL,61	
0124138E	. 3C 05		CHP AL,5	
01241390	.. 77 08		JR SHORT RKG1.0124139F	SALTA SI AL>5. SIEMPRE SALTARA PORQUE NESECEITA LETRA MINUSCULA PARA NO SALTAR.
01241392	. 0FBED1		MOVSX EDX,CL	
01241395	. 83EA 5		SUB EDX,57	
01241398	. EB 22		JMP SHORT RKG1.012413BC	
0124139A	. 8AC1		MOV AL,CL	
012413C9	. 0F1F		???	Unknown command
012413CB	. 8000 0		ADD BYTE PTR DS:[EAX],0	
012413CE	. 8000 0		ADD BYTE PTR DS:[EAX],AL	
012413D0	. 8BC1		MOV EAX,ECX	
012413D2	. C1E1 0		SHL ECX,4	
012413D5	. 2BC8		SUB ECX,EAX	
012413D7	. 83EB 0		SUB EBX,1	
012413DA	.. 75 F4		JNZ SHORT RKG1.012413D6	
012413DC	> 33D6		XOR EDX,ESI	
012413DE	. 47		INC EDI	
012413DF	. 0FAFD1		IMUL EDX,ECX	
012413E2	. 0195 2		ADD DWORD PTR SS:[EBP-2DC],EDX	GUARDA VALOR DEL CALCULO. PRIMERA Y SEGUNDA SECCION.
012413E8	. 83FF 0		CHP EDI,5	
012413EB	.. ^ 0FB2 5		JR RKG1.01241350	SALTA SI EDI<5

Como vemos es un poco más largo pero que lo podemos seguir, aquí dentro de este **LOOP** se hace la tarea del **CALL RKG1.012A1120**, lo primero que se hace es averiguar si tenemos un carácter dentro del rango **HEXADECIMAL** y si es así lo utiliza como parte de sus cálculos dentro de ese **LOOP**, de lo contrario lo desecha y coloca a cero los registros que utiliza para almacenar esos cálculos; contándoles un poco ese **LOOP**, por ejemplo si tenemos valores **HEXA**, entonces esos registros guardarán esos cálculos de lo contrario los **XOREA** contra ellos mismos para ponerlos a cero. Los registros utilizados para almacenar esos valores son **ESI** para la **PARTE 1** y **EDX** para la **PARTE 2**. Creo que eso les ayudará a entender este **LOOP** y así puedan programar su **KeyGen**. Bien, ahora hagamos el **LOOP** y miremos el valor obtenido.

00B113DF	. 0FAFD1	IMUL EDX,ECX	
00B113E2	. 0195 24FDFFF	ADD DWORD PTR SS:[EBP-2DC],EDX	GUARDA VALOR DEL CALCULO. PRIMERA Y SEGUNDA SECCION.
00B113E8	. 83FF 05	CHP EDI,5	
00B113EB	.. ^ 0FB2 5FFFFF	JR RKG1.00B11350	SALTA SI EDI<5
00B113F1	. 8D4D F4	LEA ECX,DWORD PTR SS:[EBP-C]	
00B11350=RKG1.00B11350			
Address	Hex dump	ASCII	
0030F694	D0 B6 00 00 30 C0 07 00 0B 00 00 00 00 00 00 00	3A..0L..J.....	

Recordando este viene siendo nuestro **CÁLCULO-2=PARTE 4=0B6D0**. Noten que deben ser 5 caracteres. Vamos bien, ya completamos otra parte de nuestro Serial.

MUYDI-FICIL-53138-0B6D0-PARTE 5 (CÁLCULO-3)

Pasemos a nuestro tercer y último cálculo para ver en qué nos afecta los caracteres no **HEXADECIMALES**.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)

00B113F6	. E8 25FDFFFF	CALL RKG1.00B11120	TRABAJA CON SEGUNDO SEGMENTO. 5 CARACTERES.	Registers (FPU) EAX 0000000F ECX 00000CE8 EDX 0000000F EBX 00000000 ESP 0030F680 EBP 0030F970 ESI 00000000
00B113FB	. 8BD0 20FDFFFF	MOV EDI,DWORD PTR SS:[EBP-2E0]	TRABAJA CON PRIMER SEGMENTO. ORIGINA 5 SEGMENTO.	
00B11401	. 33DB	XOR EBX,EBX		
00B11403	> 0FB0C37	MOVSX ECX,BYTE PTR DS:[EDI+ESI]		
00B11407	. 46	INC ESI		
00B11408	. 41	INC ECX		
00B11409	. 0FAFC8	IMUL ECX,EAX		
00B1140C	. 03D9	ADD EBX,ECX		
00B1140E	. 83FE 05	CMP ESI,5		
00B11411	. ^ 7C F0	JL SHORT RKG1.00B11403		

En nuestro último cálculo también se trabaja con las dos primeras partes pero es mucho más sencillo. El **CALL RKG1.012A1120** pasará a valores **HEXA** la **PARTE 2** del **Serial**, "**FICIL**". Acabo de notar algo muy importante con este **CALL**, ya lo daba todo por hecho y lo pasa con <F8> evitándome la pereza de tracearlo. Mucha atención con esto, si vemos la flecha **ROJA**, **EAX 0000000F**, pues bien tomó nuestra "**F**" pero no tomó la "**C**", este **CALL** sí que me tiene sorpresas guardadas. Pues resulta que este **CALL** empieza a buscar el primer carácter **HEXADECIMAL** y si lo encuentra lo va pasando a valor **HEXA** pero mucho ojo, si el siguiente carácter ya no es **HEXADECIMAL** deja de buscar no importando que luego si halla caracteres **HEXADECIMALES**, mejor dicho caracteres **HEXADECIMALES** que no estén seguidos al primero que halle se joden y quedan fuera de la fiesta. Si vemos nuestra **PARTE 1**, "**MUYDI**", observamos que solo tenemos un carácter "**D**" y por eso no pillamos esa sorpresita antes. Debemos agregar en nuestro **KeyGen** todo eso que hemos descubierto. Hagamos este **LOOP** para hallar nuestra última parte.

00B113F1	. 8D4D F4	LEA ECX,DWORD PTR SS:[EBP-C]	Registers (FPU) EAX 0000000F ECX 00000456 EDX 0000000F EBX 00001743 ESP 0030F680 EBP 0030F970 ESI 00000005
00B113F4	. 33F6	XOR ESI,ESI	
00B113F6	. E8 25FDFFFF	CALL RKG1.00B11120	
00B113FB	. 8BD0 20FDFFFF	MOV EDI,DWORD PTR SS:[EBP-2E0]	
00B11401	. 33DB	XOR EBX,EBX	
00B11403	> 0FB0C37	MOVSX ECX,BYTE PTR DS:[EDI+ESI]	
00B11407	. 46	INC ESI	
00B11408	. 41	INC ECX	
00B11409	. 0FAFC8	IMUL ECX,EAX	
00B1140C	. 03D9	ADD EBX,ECX	
00B1140E	. 83FE 05	CMP ESI,5	
00B11411	. ^ 7C F0	JL SHORT RKG1.00B11403	

Ahí tenemos nuestro último **CÁLCULO-3=PARTE 5=01743**. Con esto completamos nuestro **Serial**, "**MUYDI-FICIL-53138-0B6D0-01743**". Con esto demostramos que podemos usar todas las letras. A Este Serial le falta cumplir algunas condiciones para ser tomado como verdadero.

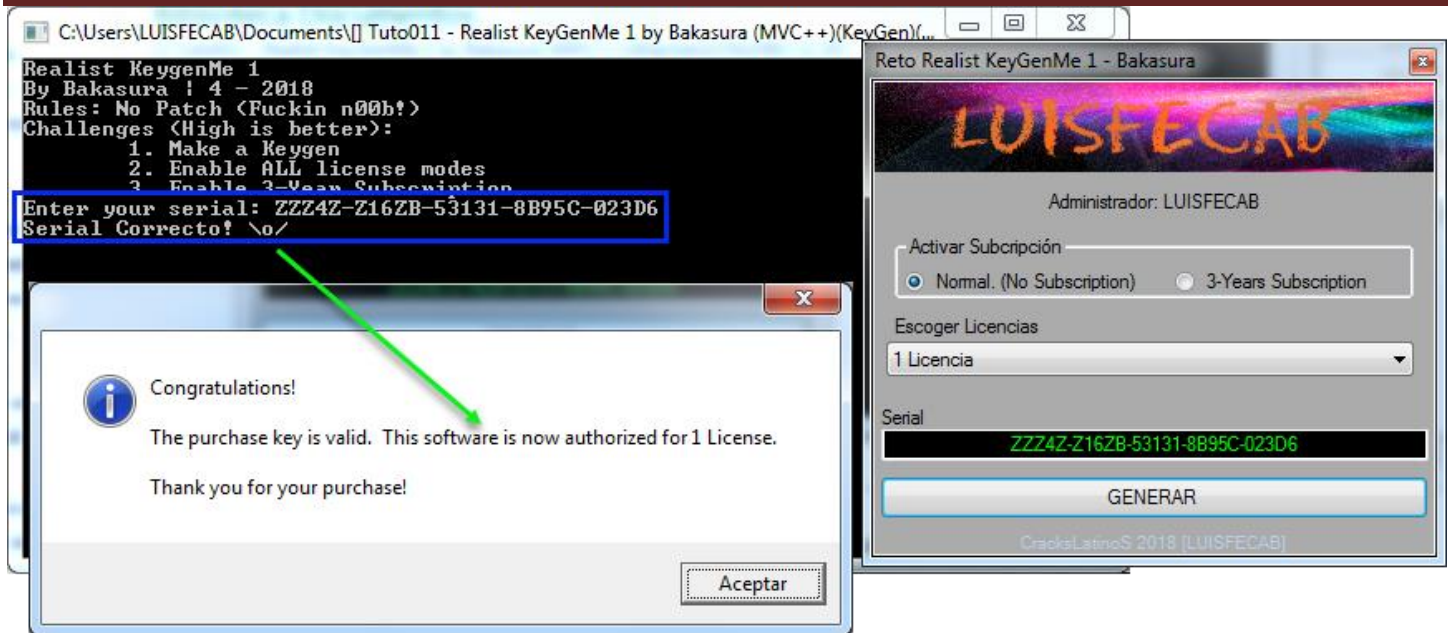
```
'Generamos aleatoriamente el SEGEMENT01 y SEGEMENT02.
'Condiciones:
'(1) Primer caracter del SEGEMENT01 debe ser una letra.
'(2) Caracter final del SEGEMENT02 deber ser "B"
'(3) Segundo caracter determina las licencias.
'(4) Si primer caracter del SEGEMENT02 es "B" -> 3-Years Subscription
```

Más sin embargo podemos probarlo y ver hasta dónde llega, y ver qué lo hace fallar.

002B176C	. 807D F8 42	CMP BYTE PTR SS:[EBP-8],42	ULTIMO CARACTER DEL SEGMENT02 DEBE SER "B"
002B1770	. 0F85 20020000	JNZ RKG1.002B19A3	
002B1776	. 57	PUSH EDI	
002B1777	. FF15 00402B00	CALL DWORD PTR DS:[402B0000]	String
002B177D	. 83F8 19	CMP EAX,19	!strlenA
002B1780	. 0F8C 10020000	JNZ RKG1.002B19A3	LONGITUD DE NUESTRO SERIAL DEBE SER 0x19=25

Falló al no cumplir la condición **2**, último carácter de la **PARTE 2** debe ser la "**B**". Bueno, solo nos queda cuadrar nuestro **KeyGen** para que genere el **Serial** con todos estos nuevos parámetros. Ya con esto, creo que ahora si puedo dar por terminado el tutorial, y quién lo iba a pensar, yo diciendo que no había dejado nada en el tintero y miren, terminé escribiendo casi cuatro páginas más.

Realist KeyGenMe 1 by Bakasura (MVC++)(KeyGen)(OllyDBG v1.10)



Funciona muy bien, hemos generado un **Serial** que tiene puras "Z" y fue aceptado sin problemas para **1 Licencia** y sin suscripción. Listo amigos, ahora si terminado. Saludos y espero sea de ayuda para quien lo lee.