

Victima	Crackme1.exe
Protección	Serial
Herramientas	IDA 5.2 versión 32bit, Calculadora de Windows
Objetivo	Serial

Introducción

Hola a todos en esta la edición 12 del concurso. Nuevamente estamos ante el crackme de un compañero de la lista más concretamente de MCKSys. Aunque según el es facilón por se su primer crackme, por el simple hecho de ser de un listero que menos que echarle un vistazo.

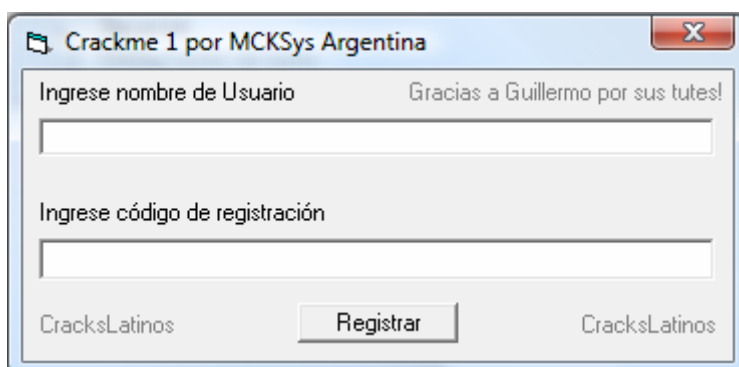
Analizando con IDA

Bueno el ejecutable sabemos que está hecho en ese magnífico compilador que tanto me gusta de VB (ironía) simplemente viendo el icono característico, también sabemos por el autor que no tiene código anti-debug, anti-dump ni nada.

El programa se suministra junto con una dll llamada dll.dll así que bueno algo debe de tener jejeje



Si lo ejecutamos vemos que tenemos 2 Edit para meter un nombre, un serial y el botón para comprobación



Bueno sin más metemos el programa para analizar en IDA y estamos aquí:

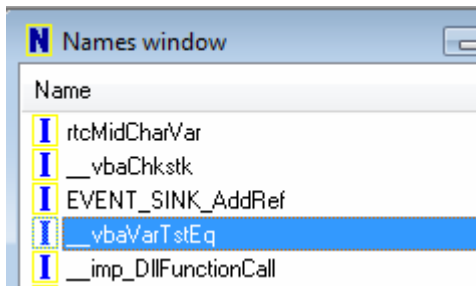
```
.text:00401270 ; ..... ; .text:00401B34↓o ...
.text:00401270 jmp ds:EVENT_SINK_Release
.text:00401276 ; [00000006 BYTES: COLLAPSED FUNCTION ThunRTMain. PRESS KEYPAD "+" TO EXPAND]
.text:0040127C ; -----
.text:0040127C
.text:0040127C public start
.text:0040127C start:
.text:0040127C push offset aUb56Ub6es_dll ; "UB5!6&UB6ES.DLL"
.text:0040127C call ThunRTMain
.text:00401281
```



El problema de los VB es que IDA no los analiza bien.

También hay que tener en cuenta que este crackme tiene un truquito que precisamente dificulta su análisis estático como ya veremos.

Al ser un VB presumiblemente usará funciones como `__vbaVarTestEq`. Si vamos a la ventana de nombre la vemos ahí



Hacemos doble click y en la ventana de desensamblado vemos que hay 2 entradas

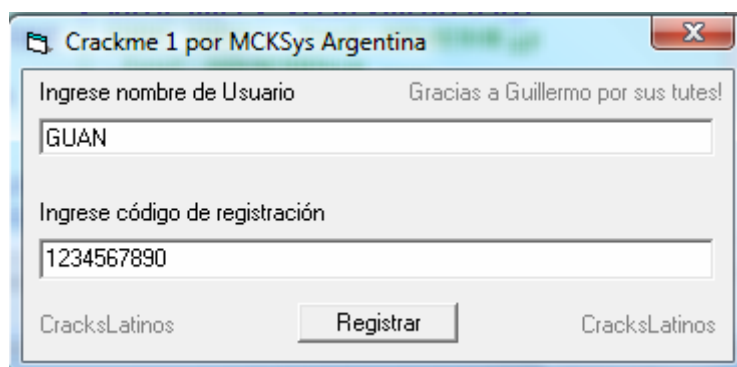
```
extrn __vbaVarTstEq:dword ; CODE XREF: .text:004027E9↓p
; .text:00402910↓p
```

Lo que haremos es poner un BreakPoint en ambas direcciones, cargaremos el depurador y veremos que tenemos cuando paremos

```
.text:004027E2 lea    eax, [ebp-9Ch]
.text:004027E8 push   eax
.text:004027E9 call   ds:vbaVarTstEq
.text:004027EF mov    [ebp-0DCh], eax
.text:004027F5 lea    ecx, [ebp-3Ch]

.text:00402909 lea    edx, [ebp-9Ch]
.text:0040290F push   edx
.text:00402910 call   ds:vbaVarTstEq
.text:00402916 mov    [ebp-0DCh], eax
```

Lanzamos el depurador, en este caso el win32_remote.exe y luego damos F9 para lanzar el depurador de IDA. El programa arranca y ponemos los siguientes datos:



Pulsamos Registrar y para IDA.

```
.text:004027E2 lea    eax, [ebp-9Ch]
.text:004027E8 push   eax
.text:004027E9 call   ds:vbaVarTstEq
.text:004027EF mov    [ebp-0DCh], eax
.text:004027F5 lea    ecx, [ebp-3Ch]
.text:004027F8 call   ds:__vbaFreeObj
.text:004027FE lea    ecx, [ebp-5Ch]
```

Uno será para el Nombre y la otra presumiblemente será para el serial así que damos a RUN y para nuevamente.

```
.text:00402908 push    ecx
.text:00402909 lea    edx, [ebp-9Ch]
.text:0040290F push    edx
.text:00402910 call   ds:_vbaVarTstEq
.text:00402916 mov    [ebp-0DCh], eax
.text:0040291C lea    ecx, [ebp-3Ch]
```

Lo que se está haciendo en estas comprobaciones es verificar si se han introducidos datos en el Edit del Nombre y del Serial.

Como lo hemos hecho seguimos traceando con cuidado con F8 hasta que llegamos aquí

```
.text:004029AF
.text:004029AF loc_4029AF:
.text:004029AF call   sub_40211C
```

Entramos con F7 y seguimos traceando F8 y vemos que llegamos al modulo Dll.dll

```
DLL.dll:002F100C dll_Proc01:
DLL.dll:002F100C push    ecx
DLL.dll:002F100D mov    ecx, [esp+4]
DLL.dll:002F1011 mov    eax, ds:DWORD_2F3000
DLL.dll:002F1016
DLL.dll:002F1016 loc_2F1016:
DLL.dll:002F1016 xor    byte ptr [eax+ecx], 0C7h
DLL.dll:002F101A dec    eax
DLL.dll:002F101B cmp    eax, 0
DLL.dll:002F101E jnz    short loc_2F1016
DLL.dll:002F1020 xor    byte ptr [ecx], 0C7h
DLL.dll:002F1023 mov    eax, 1
DLL.dll:002F1028 pop    ecx
DLL.dll:002F1029 retn
```

¿Qué hace esto?

Pues es una rutina básica de descryptación mediante el uso del comando XOR. Esta DLL simplemente da apoyo para descryptar la zona caliente del código. Es una de la causas por la que el análisis de IDA en estático es tan pobre, ya que la parte interesante está encryptada.

Al salir de la rutina llegamos aquí

```
.text:004029AF loc_4029AF:
.text:004029AF call   sub_40211C
.text:004029B4 call   ds:_vbaSetSystemError
.text:004029B6 and    [eax], dl
.text:004029B8 inc    eax
.text:004029BA mov    ecx, [esi]
.text:004029BB push   cs
.text:004029BC push   esi
```

Entramos con F7 y seguimos trazando con F8 hasta llegar aquí

```

.text:00402A13 lea    eax, [ebp-5Ch]
.text:00402A16 push   eax
.text:00402A17 call  ds:rtcTrimVar
.text:00402A1D mov    dword ptr [ebp-94h], 20h
.text:00402A27 mov    dword ptr [ebp-9Ch], 8002h
.text:00402A31 lea    ecx, [ebp-5Ch]
.text:00402A34 push   ecx
.text:00402A35 lea    edx, [ebp-6Ch]
.text:00402A38 push   edx
.text:00402A39 call  ds:__vbaLenVar
.text:00402A3F push   eax
.text:00402A40 lea    eax, [ebp-9Ch]
.text:00402A46 push   eax
.text:00402A47 call  ds:__vbaVarTstNe
.text:00402A4D mov    [ebp-0DCh], eax

```

Vemos que toma el tamaño de un String y luego compara el valor obtenido con __vbaVarTstNe

Parados sobre la llamada a la API __vbaLenVar, tenemos en ecx el puntero a la cadena de la cual calcular el tamaño, vamos a ver cual es.

```

15 20 00 00 E0 4E E2 75 08 00 17 00 D9 BB 6D 72
FC ED 5A 00 40 07 4D 03 08 00 00 00 01 00 00 00

```

La estructura de una cadena/variable en VB está forma por una estructura de 4 DWORD. El 1º DWORD indica a VB el tipo de dato, en este caso sabemos que es una cadena de caracteres. En este caso el valor remarcado en rojo es la dirección en memoria donde se encuentra la cadena en formato UNICODE.

```

E4 2B BE 15 F9 C4 00 18 14 00 00 00 31 00 32 00 0+¥$''-.T¶...[]-2.
33 00 34 00 35 00 36 00 37 00 38 00 39 00 30 00 3.4.5.6.7.8.9.0.

```

En este caso está tomando el serial

Seguimos traceando y nos quedamos en la API __vbaVarTstNe para ver cual debe de ser el tamaño del serial.

Una vez parado mostramos el valor de eax:

```

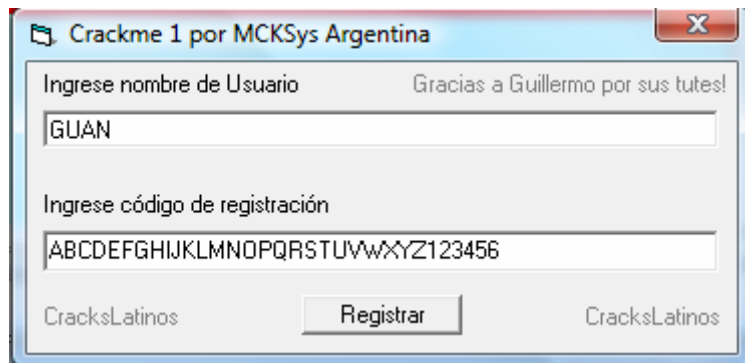
00 00 00 00 F8 77 10 10 02 80 00 00 00 00 00 00
20 00 00 00 AC F9 17 00 00 00 00 00 00 00 00 00

```

Como anteriormente el valor se encuentra en eax+8 (el valor remarcado), en este caso el valor no es un puntero si no un entero, más concretamente 20h => 32 dec.

Nuestro serial solo tiene 10 posiciones por lo que no pasamos el primer filtro.

Dejamos un único BP en la dirección 00402A39 para no perder el tiempo, damos F9, aceptamos el Cartel de Somos muy malos y ponemos el siguiente serial:



Pulsamos Registrar y estamos en el BP, pasamos la API con F8 y vemos que en EAX el valor es 0, vamos bien jejeje

Seguimos traceando con cuidado y llegamos a otra llamada que lleva a DLL.dll para descryptar la siguiente zona caliente.

```
.text:00402A84 loc_402A84:
.text:00402A84
.text:00402A84 call sub_40215C
```

Hacemos la misma operación, F7 para entrar F8 hasta que salimos, llegamos a __vbaSetSystemError, entramos con F7 y seguimos traceando con F8 hasta salir de la API. Seguimos traceando y llegamos aquí:

```
:00402B03 push    edx
:00402B04 push    1
:00402B06 lea    eax, [ebp-5Ch]
:00402B09 push    eax
:00402B0A lea    ecx, [ebp-7Ch]
:00402B0D push    ecx
:00402B0E mov    ebx, ds:rtcMidCharVar
:00402B14 call   ebx ; rtcMidCharVar
:00402B16 lea    edx, [ebp-7Ch]
:00402B19 push    |edx
```

Lo que hace es tomar caracteres de nuestro serial. Si seguimos traceando llegamos a esta API

```
00402EAF push    ecx
00402EB0 call   ds:__vbaVarAdd
00402EB6 push    eax
00402EB7 call   ds:__vbaStrVarMove
```

Con la variable __vbaVarAdd va añadiendo esos caracteres que va extrayendo con rtcMidCharVar creando una segunda cadena.

Podemos ir viendo como va quedando la cadena visualizando el puntero que se muestra en eax+8 una vez ejecutada la API.

```
41 00 46 00 48 00 4C 00 58 00 00 00 ... 00.F.H.L.X...
```

El crackme saca 2 subcadenas de 8 caracteres cada uno usando el mismo este mismo método obtenemos:

Cadena 1º: AFHLXG6J

Cadena 2º: BMN5W3Z

Bueno como no repetimos caracteres en el serial se identifica rápidamente como se ha creado cada cadena:

Cadena 1:

1º	6º	8º	12º	24º	7º	32º	10º
----	----	----	-----	-----	----	-----	-----

Cadena 2:

2º	13º	14º	31º	23º	29º	26º	4º
----	-----	-----	-----	-----	-----	-----	----

Si seguimos traceando llegamos nuevamente a una call a la DLL.dll

```
:004038CB add esp, 18h  
:004038CE call sub_40210C
```

Asi que hacemos como siempre y llegamos aquí:

```
-----  
:004038D9 mov ecx, [ebp-20h]  
:004038DC push ecx  
:004038DD call ds:__ubaLenBstr  
:004038E3 mov [ebp-0E4h], eax  
:004038E9 mov eax, 1  
:004038EE mov [ebp-28h], eax  
:004038F4
```

Bueno hemos llegado a la última parte de la comprobación.

Lo que se hace es sencillamente ir cogiendo los caracteres de la Cadena 1.

Busca su índice en la cadena que se muestra a continuación (los caracteres válidos)

```
aAbcdefghijklmn: ; DATA XREF: .text:004038FD↓  
; .text:004039B2↓  
unicode 0, <abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNopqrstuvwxyz12345>  
unicode 0, <67890>,0
```

A ese índice le suma 0Bh => 11 dec.

Toma el índice del 1º carácter de la Cadena 2 y comprueba que se índice coincide con la suma si no da nos saca el cartel, en caso contrario vuelve a comprobar el siguiente carácter.

Ejemplo: Nuestro carácter de la cadena 1º corresponde con A u tiene como índice 1Bh (posición 27) si le sumamos 0B resulta que el 1º Carácter de la Cadena 2 debe de corresponder con la posición 26h (posición 36) que corresponde con la letra L.

Como no es nos saca el cartel de chico malo.
Con esto un serial bueno sería:

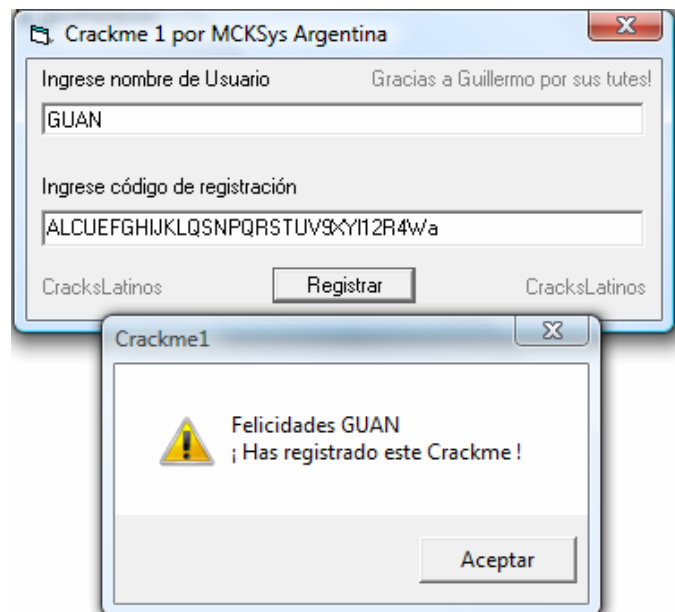
Cadena 1	Cadena 2
A	L
F	Q
H	S
L	W
X	9
G	R
6	¿?
J	U

Como la posición del carácter 6 + 11 posición sobre pasa la cadena de caracteres válidos lo que hacemos es cambiar el 6 por la 'a' (por ejemplo) y por lo tanto sería '1' en la cadena 2.

Con esto un posible serial válido sería

ALCUEFGHIJKLQSNPQRSTUUV9XYI12R4Wa

Si la probamos:



Pues ya lo tenemos jejeje el que quiera que le haga un KeyGen con los datos del estudio.

Hasta la próxima.

