

# Solucionario Crackme “Décimo Aniversario Hispasec”

Autor: Alejo Murillo (alexismm2@gmail.com)

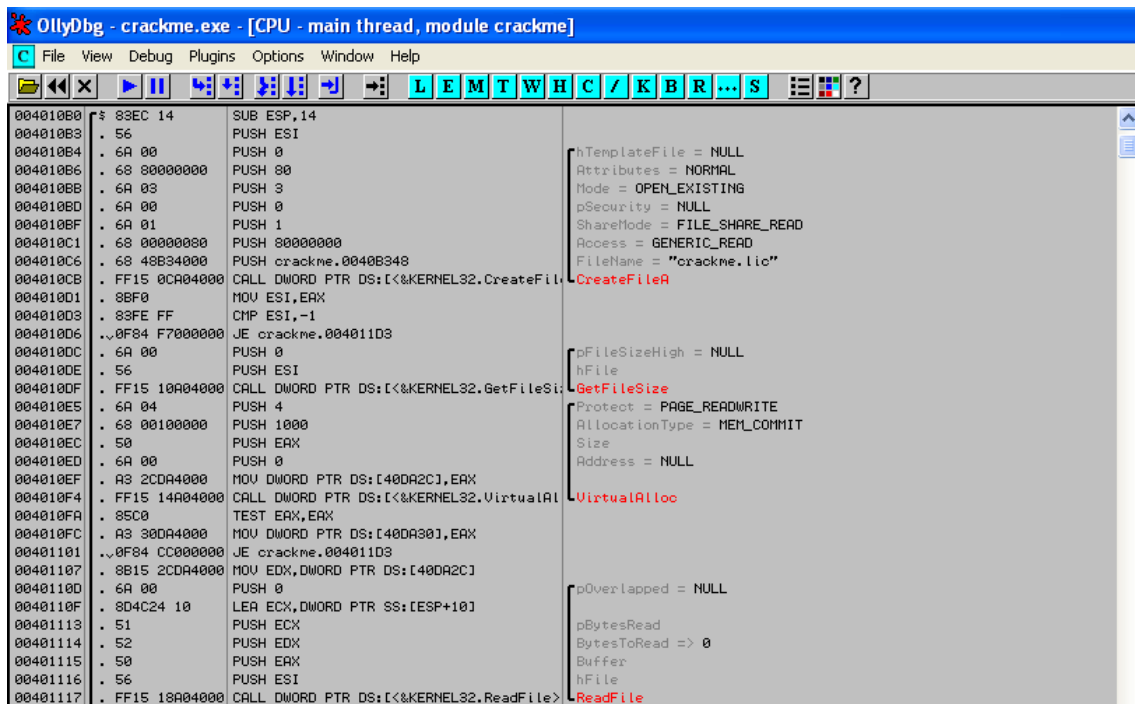
Nivel: Básico

Tiempo invertido: 4h (incluyendo documentación)

## 1. Acercándonos al objetivo

Una vez descargado el fichero, procedemos a ejecutarlo y vemos que no muestra ningún mensaje. Utilizaremos OllyDbg para debuggear el fichero y ver las instrucciones máquina.

Si se ejecuta paso por paso, llegaremos hasta un módulo bastante interesante (0x004010B0)



The screenshot shows the OllyDbg interface with the assembly window on the left and the API call window on the right. The assembly window displays instructions from address 004010B0 to 00401117. The API call window shows the details of the `CreateFileA` call, including parameters like `hTemplateFile = NULL`, `Attributes = NORMAL`, `Mode = OPEN_EXISTING`, `ShareMode = FILE_SHARE_READ`, `Access = GENERIC_READ`, and `FileName = "crackme.lic"`. Other API calls shown include `GetFileSize`, `VirtualAlloc`, and `ReadFile`.

Como puede observarse, el ejecutable intenta abrir un fichero de nombre ‘crackme.lic’ (llamada a la API `CreateFileA`) y procede a leer su contenido en un buffer (`VirtualAlloc + ReadFile`).

A continuación, crearemos un fichero ‘crackme.lic’ con contenido aleatorio en el mismo directorio donde reside el ejecutable y lo ejecutaremos a ver qué pasa...

```
D:\>crackme.exe
Keep trying
D:\>
```

Perfecto!!!, parece que nos vamos acercando al auténtico reto ☺

Si continuamos viendo el código del ejecutable, nos encontramos con el siguiente código

0040111D	. 68 34DA4000	PUSH crackme.0040DA34	pCriticalSection = crackme.0040DA34
00401122	. FF15 1CA04000	CALL DWORD PTR DS:[<&KERNEL32.InitializeCriticalSection]	InitializeCriticalSection
00401128	. 8B35 20A04000	MOV ESI,DWORD PTR DS:[<&KERNEL32.CreateThread]	kernel32.CreateThread
0040112E	. 6A 00	PUSH 0	pThreadId = NULL
00401130	. 6A 00	PUSH 0	CreationFlags = 0
00401132	. 6A 00	PUSH 0	pThreadParam = NULL
00401134	. 68 00104000	PUSH crackme.00401000	ThreadFunction = crackme.00401000
00401139	. 6A 00	PUSH 0	StackSize = 0
0040113B	. 6A 00	PUSH 0	pSecurity = NULL
0040113D	. FFD6	CALL ESI	CreateThread
0040113F	. 68 E8030000	PUSH 3E8	
00401144	. 894424 14	MOV DWORD PTR SS:[ESP+14],EAX	
00401148	. FF15 08A04000	CALL DWORD PTR DS:[<&KERNEL32.Sleep>]	Sleep
0040114E	. 6A 00	PUSH 0	pThreadId = NULL
00401150	. 6A 00	PUSH 0	CreationFlags = 0
00401152	. 6A 00	PUSH 0	pThreadParam = NULL
00401154	. 68 00104000	PUSH crackme.00401000	ThreadFunction = crackme.00401000
00401159	. 6A 00	PUSH 0	StackSize = 0
0040115B	. 6A 00	PUSH 0	pSecurity = NULL
0040115D	. FFD6	CALL ESI	CreateThread
0040115F	. 6A FF	PUSH -1	Timeout = INFINITE
00401161	. 894424 18	MOV DWORD PTR SS:[ESP+18],EAX	
00401165	. 6A 01	PUSH 1	WaitForAll = TRUE
00401167	. 8D4424 18	LEA EAX,DWORD PTR SS:[ESP+18]	
0040116B	. 50	PUSH EAX	pObjects
0040116C	. 6A 02	PUSH 2	nObjects = 2
0040116E	. FF15 24A04000	CALL DWORD PTR DS:[<&KERNEL32.WaitForMultipleObjects]	WaitForMultipleObjects
00401174	. 8B5424 10	MOV EDX,DWORD PTR SS:[ESP+10]	
00401178	. 8B35 28A04000	MOV ESI,DWORD PTR DS:[<&KERNEL32.GetExitCodeThread]	kernel32.GetExitCodeThread
0040117E	. 8D4C24 04	LEA ECX,DWORD PTR SS:[ESP+4]	
00401182	. 51	PUSH ECX	pExitCode
00401183	. 52	PUSH EDX	hThread
00401184	. FFD6	CALL ESI	GetExitCodeThread
00401186	. 8B4C24 14	MOV ECX,DWORD PTR SS:[ESP+14]	
0040118A	. 8D4424 08	LEA EAX,DWORD PTR SS:[ESP+8]	
0040118E	. 50	PUSH EAX	pExitCode
0040118F	. 51	PUSH ECX	hThread
00401190	. FFD6	CALL ESI	GetExitCodeThread
00401192	. 837C24 04 00	CMPL DWORD PTR SS:[ESP+4],0	
00401197	. 74 0E	JE SHORT crackme.004011A7	
00401199	. 837C24 08 00	CMPL DWORD PTR SS:[ESP+8],0	
0040119E	. 74 07	JE SHORT crackme.004011A7	
004011A0	. 68 54B34000	PUSH crackme.0040B354	ASCII "Good job!"
004011A5	. EB 05	JMP SHORT crackme.004011AC	
004011A7	. 68 60B34000	PUSH crackme.0040B360	ASCII "Keep trying!"
004011AC	. E8 5A000000	CALL crackme.0040120B	

Como puede observarse, el programa genera dos hilos hijo (*CreateThread*) que ejecutan el mismo código (0x00401000).

Una vez ejecutados esos hilos, el proceso padre recoge los resultados (*GetExitCodeThread*) y verifica que ambos hilos hayan devuelto un valor distinto de cero. Si esta condición se cumple, mostrará el mensaje deseado: “*Good job!*”

Sin embargo, algo aparentemente tan sencillo puede ser peligroso, puesto que la ejecución en paralelo de 2 procesos puede complicar el análisis de este crackme. En concreto, hay que prestar especial atención a las llamadas a *Sleep*, ya que ‘duermen’ el proceso o hilo un tiempo concreto de milisegundos (ver llamada en 0x00401148, que duerme el proceso padre 1 segundo)

A continuación miramos el código que será ejecutado por los hijos (0x401000):

00401000	. 51	PUSH ECX	
00401001	. 53	PUSH EBX	
00401002	. 55	PUSH EBP	
00401003	. 8B2D 00A04000	MOV EBP,DWORD PTR DS:[<&KERNEL32.EnterC	ntdll.RtlEnterCriticalSection
00401009	. 56	PUSH ESI	
0040100A	. 8B35 08A04000	MOV ESI,DWORD PTR DS:[<&KERNEL32.Sleep>	kernel32.Sleep
00401010	. 57	PUSH EDI	
00401011	. 8B3D 04A04000	MOV EDI,DWORD PTR DS:[<&KERNEL32.LeaveC	ntdll.RtlLeaveCriticalSection
00401017	. C64424 13 00	MOV BYTE PTR SS:[ESP+13],0	
0040101C	. 8D6424 00	LEA ESP,DWORD PTR SS:[ESP]	
00401020	> 68 34DA4000	PUSH crackme.0040DA34	
00401025	. FFDF	CALL EBP	
00401027	. A1 50DA4000	MOV EAX,DWORD PTR DS:[40DA50]	Bytes leidos
0040102C	. 3B05 2CDA4000	CMP EAX,DWORD PTR DS:[40DA2C]	Terminamos de leer el fichero
00401032	..v73 13	JNB SHORT crackme.00401047	Terminamos de leer el fichero?
00401034	. 8B0D 30DA4000	MOV ECX,DWORD PTR DS:[40DA30]	
0040103A	. 8A1C01	MOV BL,BYTE PTR DS:[ECX+EAX]	Leemos el fichero
0040103D	. 83C0 01	ADD EAX,1	
00401040	. A3 50DA4000	MOV DWORD PTR DS:[40DA50],EAX	Incrementamos contador
00401045	..vEB 03	JMP SHORT crackme.0040104A	
00401047	> 80CB FF	OR BL,0FF	
0040104A	> 68 34DA4000	PUSH crackme.0040DA34	
0040104F	. FFD7	CALL EDI	
00401051	. 80FB 0A	CMP BL,0A	BL = caracter leído
00401054	..v73 37	JNB SHORT crackme.0040108D	Si caracter <0x0a
00401056	. 0FB64424 13	MOVZX EAX,BYTE PTR SS:[ESP+13]	contador (inicialmente 0)
0040105B	. 8D1400	LEA EDX,DWORD PTR DS:[EAX+EAX*4]	aux2 = contador*5
0040105E	. 0FB6C3	MOVZX EAX,BL	
00401061	. 8A8450 60CF40	MOV AL,BYTE PTR DS:[EAX+EDX*2+40CF60]	aux=tabla[aux2+caracter_leido]
00401068	. 3C 06	CMP AL,6	
0040106A	. 884424 13	MOV BYTE PTR SS:[ESP+13],AL	contador = aux
0040106E	..v75 09	JNZ SHORT crackme.00401079	
00401070	> 68 CE070000	PUSH 7CE	
00401075	. FFD6	CALL ESI	Si aux=6 => SLEEP() 2 segundos
00401077	..vEB 04	JMP SHORT crackme.0040107D	
00401079	> 3C 09	CMP AL,9	
0040107B	..v74 1F	JE SHORT crackme.0040109C	Si aux = 9 ==> Clave OK
0040107D	> E8 67010000	CALL crackme.004011E9	
00401082	. 99	CDB	
00401083	. B9 32000000	MOV ECX,32	
00401088	. F7F9	IDIV ECX	
0040108A	. 52	PUSH EDX	
0040108B	. FFD6	CALL ESI	
0040108D	> 80FB FF	CMP BL,0FF	
00401090	..^75 8E	JNZ SHORT crackme.00401020	
00401092	. 5F	POP EDI	Fichero incorrecto, hilo devuelve ERROR
00401093	. 5E	POP ESI	
00401094	. 5D	POP EBP	
00401095	. 33C0	XOR EAX,EAX	
00401097	. 5B	POP EBX	
00401098	. 59	POP ECX	
00401099	. C2 0400	RETN 4	
0040109C	> 5F	POP EDI	Fichero correcto, hilo devuelve OK
0040109D	. 5E	POP ESI	
0040109E	. 5D	POP EBP	
0040109F	. B8 01000000	MOV EAX,1	
004010A4	. 5B	POP EBX	
004010A5	. 59	POP ECX	
004010A6	. C2 0400	RETN 4	

Antes de analizar en detalle el código, comentaré los 3 datos básicos utilizados por el módulo:

- Buffer del fichero 'crackme.lic'
- Tabla de correspondencia (0x40CF60): utilizada para caracteres < 0x0a
- Contador: contiene el último valor obtenido de la tabla de correspondencia

Además, podemos apreciar que nuestro objetivo es que el código devuelva un valor distinto de cero (en el proceso padre se verificaba que el valor de retorno no fuera 0).

Por lo tanto se ha de encontrar un fichero que permita ejecutar las siguientes instrucciones 0x0040109C ... 00x004010A6, que devuelve un retorno distinto de cero (MOV EAX,1)

## 2. Tirando del hilo ;p

Si queremos que el hilo devuelva un valor distinto de 0, se deberá realizar un salto a ese código, por lo que se deberá cumplir la condición de la instrucción 0x0040107B

00401051	. 80FB 0A	CMP BL,0A	BL = caracter leído
00401054	√73 37	JNB SHORT crackme.0040108D	Si caracter <0x0a
00401056	. 0FB64424 13	MOVZX EAX,BYTE PTR SS:[ESP+13]	contador (inicialmente 0)
0040105B	. 8D1480	LEA EDX,DWORD PTR DS:[EAX+EAX*4]	aux2 = contador*5
0040105E	. 0FB6C3	MOVZX EAX,BL	
00401061	. 8A8450 60CF60	MOV AL,BYTE PTR DS:[EAX+EDX*2+40CF60]	aux=tabla[aux2+caracter_leído]
00401068	. 3C 06	CMP AL,6	
0040106A	. 884424 13	MOV BYTE PTR SS:[ESP+13],AL	contador = aux
0040106E	√75 09	JNZ SHORT crackme.00401079	
00401070	. 68 CE070000	PUSH 7CE	
00401075	. FFD6	CALL ESI	Si aux=6 => SLEEP() 2 segundos
00401077	√EB 04	JMP SHORT crackme.0040107D	
00401079	> 3C 09	CMP AL,9	
0040107B	√74 1F	JE SHORT crackme.0040109C	Si aux = 9 ==> Clave OK

Esta es la parte del código crítica para validar el fichero. Podemos ver una llamada de 2 segundos a SLEEP (instrucción 0x00401075), pero por ahora no le prestaremos atención.

A continuación se muestra un pseudocódigo del programa:

```

contador = 0
while (caracter=getChar()) {
    if (carácter > 0x0a)
        hacerCosasRaras()
    continue

    aux2 = contador*10
    valor_nuevo = matriz[carácter+aux2]
    contaror = valor_nuevo

    if (contador == 6)
        sleep(2000ms)

    if (contador == 9)
        return OK
}
return ERR

```

En el código vemos que se utiliza una matriz o array de correspondencia, que se encuentra almacenada en el bloque de memoria 0x40CF60.

Address	Hex dump	ASCII
0040CF60	05 01 03 02 01 02 04 03	!@#00000
0040CF68	05 04 00 00 00 00 00 02	!+. . . . 0
0040CF70	00 00 00 00 02 03 03 00	. . . . 000.
0040CF78	00 01 01 04 04 05 02 03	.0000000
0040CF80	03 01 06 04 04 05 05 00	0000000.
0040CF88	00 00 05 01 02 03 04 08	. . 000000
0040CF90	08 05 05 05 05 04 04 05	00000000
0040CF98	05 00 00 00 06 06 06 06	! . . . 0000
0040CFA0	06 07 08 08 06 06 07 07	! . 000000
0040CFA8	09 08 08 07 07 08 08 08	. 00 . 0000
0040CFB0	08 07 07 08 08 08 08 07	0 . . 0000 .
0040CFB8	07 07 00 00 00 00 00 00	. . . . . . . .
0040CFC0	00 00 00 00 00 00 00 00	. . . . . . . .
0040CFC8	00 00 00 00 00 00 00 00	. . . . . . . .

Como podemos observar, la tabla o matriz posee diferentes valores que van modificando el estado interno del programa.

Nuestro objetivo es obtener un valor para la variable *contador* igual a 9. Para ello utilizaremos lápiz y papel, simulando un algoritmo de backtracking.

#### **Paso 4**

Objetivo: contador = 9

Requisitos:  $\text{matriz}[\text{carácter\_leído} + (\text{valor\_anterior} * 10)] == 9$

En la matriz, únicamente la dirección de memoria 0x0040CFA8 posee ese valor, por lo que tenemos que cuadrar los valores de carácter\_leído y valor anterior para que su suma (offset) sea 72 en decimal o 0x48. Este valor se obtiene restando 0040CF60 (valor deseado) a 0040CFA8 (inicio de valores tabla en memoria).

Puesto que la variable valor\_anterior es un múltiplo de 10, los valores requeridos serán los siguientes:

Valor\_anterior = 7

carácter\_leído = 2 (0x02)

$\text{matriz}[2+(7*10)] = \text{matriz}[72] = 9$

Ya tenemos el último byte que debe leer este código (0x02). Si seguimos realizando las cuentas 'hacia atrás' iremos sacando el resto de valores

#### **Paso 3**

Objetivo: contador = 7

Requisitos:  $\text{matriz}[\text{carácter\_leído} + (\text{valor\_anterior} * 10)] == 7$

Revisando la matriz, vemos que la matriz posee un valor adecuado 7 en el offset 65, que permite cumplir las dos condiciones (múltiplo de 10 y que el carácter leído sea menor de 0x0a).

Realizando los cálculos obtenemos la siguiente ecuación

Valor\_anterior = 6

carácter\_leído = 5 (0x05)

$\text{matriz}[5+(6*10)] = \text{matriz}[65] = 7$

## Paso 2

Objetivo: contador = 6

Requisitos:  $\text{matriz}[\text{carácter\_leído} + (\text{valor\_anterior} * 10)] == 6$

Revisando la matriz, vemos que en el offset 34 se encuentra el valor deseado. El offset debe cumplir dos condiciones (múltiplo de 10 y que el carácter leído sea menor de 0x0a).

Realizando los cálculos obtenemos la siguiente ecuación

$$\text{Valor\_anterior} = 3$$

$$\text{carácter\_leído} = 4 \text{ (0x04)}$$

$$\text{matriz}[4+(3*10)] = \text{matriz}[34] = 9$$

## Paso 1

Objetivo: contador = 3

Requisitos:  $\text{matriz}[\text{carácter\_leído} + (\text{valor\_anterior} * 10)] == 6$

Revisando la matriz, vemos que en el offset 02 se encuentra el valor deseado. El offset debe cumplir dos condiciones (múltiplo de 10 y que el carácter leído sea menor de 0x0a).

Realizando los cálculos obtenemos la siguiente ecuación

$$\text{Valor\_anterior} = 0 \text{ (**caso inicial**)}$$

$$\text{carácter\_leído} = 2 \text{ (0x02)}$$

$$\text{matriz}[2+(0*10)] = \text{matriz}[2] = 3$$

Con este cálculo manual hemos obtenido los caracteres que debería leer el programa para que se cumpliera la condición necesaria ( $\text{valor\_final} == 9$ ). Los bytes obtenidos son:

0x02 0x05 0x04 0x02

Puesto que hemos realizado un algoritmo de backtracking (de la meta a la condición de inicio), hay que invertir el resultado. Si se parchea el ejecutable (que solo lance un hilo), da como bueno un fichero con estos 4 bytes:

0x02 0x04 0x05 0x02

Sin embargo, hemos omitido la parte más interesante: la ejecución de 2 hilos diferentes y las llamadas a Sleep().

### 3. Race Condition

Puesto que el proceso padre crea 2 hilos, es necesario prestar atención a las llamadas a Sleep. En el código mostrado únicamente tenemos 2 llamadas:

#### 1) Proceso padre :

Una vez creado el primer hijo, duerme durante un segundo (0x3E8 == 1000 ms). Una vez pasado ese tiempo, crea el otro hilo hijo

00401128	. 8B35 20A04000	MOV ESI,DWORD PTR DS:[<&KERNEL32.CreateThread	kernel32.CreateThread
0040112E	. 6A 00	PUSH 0	pThreadId = NULL
00401130	. 6A 00	PUSH 0	CreationFlags = 0
00401132	. 6A 00	PUSH 0	pThreadParm = NULL
00401134	. 68 00104000	PUSH crackme.00401000	ThreadFunction = crackme.00401000
00401139	. 6A 00	PUSH 0	StackSize = 0
0040113B	. 6A 00	PUSH 0	pSecurity = NULL
0040113D	. FFD6	CALL ESI	CreateThread
0040113F	. 68 E8030000	PUSH 3E8	
00401144	. 894424 14	MOV DWORD PTR SS:[ESP+14],EAX	
00401148	. FF15 08A04000	CALL DWORD PTR DS:[<&KERNEL32.Sleep>]	Sleep
0040114E	. 6A 00	PUSH 0	pThreadId = NULL
00401150	. 6A 00	PUSH 0	CreationFlags = 0
00401152	. 6A 00	PUSH 0	pThreadParm = NULL
00401154	. 68 00104000	PUSH crackme.00401000	ThreadFunction = crackme.00401000
00401159	. 6A 00	PUSH 0	StackSize = 0
0040115B	. 6A 00	PUSH 0	pSecurity = NULL
0040115D	. FFD6	CALL ESI	CreateThread
0040115F	. 6A FF	PUSH -1	Timeout = INFINITE
00401161	. 894424 18	MOV DWORD PTR SS:[ESP+18],EAX	
00401165	. 6A 01	PUSH 1	WaitForAll = TRUE
00401167	. 8D4424 18	LEA EAX,DWORD PTR SS:[ESP+18]	
0040116B	. 50	PUSH EAX	pObjects
0040116C	. 6A 02	PUSH 2	nObjects = 2
0040116E	. FF15 24A04000	CALL DWORD PTR DS:[<&KERNEL32.WaitForMulti	WaitForMultipleObjects

#### 2) Hilos hijos:

Si el valor obtenido de la matriz es igual a 6 (0x00401068), entonces se dormirá el hilo durante 2 segundos (0x00401075)

00401051	. 80FB 0A	CMP BL,0A	BL = caracter leído
00401054	√ 73 37	JNB SHORT crackme.0040108D	Si caracter < 0x0a
00401056	. 0FB64424 13	MOVZX EAX,BYTE PTR SS:[ESP+13]	contador (inicialmente 0)
0040105B	. 8D1480	LEA EDX,DWORD PTR DS:[EAX+EAX*4]	aux2 = contador*5
0040105E	. 0FB6C3	MOVZX EAX,BL	
00401061	. 8A8450 60CF60	MOV AL,BYTE PTR DS:[EAX+EDX*2+40CF60]	aux=tabla[aux2+caracter_leído]
00401068	. 3C 06	CMP AL,6	
0040106A	. 884424 13	MOV BYTE PTR SS:[ESP+13],AL	contador = aux
0040106E	√ 75 09	JNZ SHORT crackme.00401079	
00401070	. 68 CE070000	PUSH 7CE	
00401075	. FFD6	CALL ESI	Si aux=6 => SLEEP() 2 segundos
00401077	√ EB 04	JMP SHORT crackme.0040107D	
00401079	> 3C 09	CMP AL,9	
0040107B	√ 74 1F	JE SHORT crackme.0040109C	Si aux = 9 ==> Clave OK

Este código se ejecutará, puesto que para poder obtener el valor 9 de la matriz (condición deseada) es necesario obtener previamente el valor de 6. Este valor dormirá este hilo, alternando la ejecución con el hilo2.

A continuación se resume la ejecución de los hilos. Lo que se ha de conseguir es que cada hilo, independientemente de cuándo sea dormido/despertado, lea los siguientes bytes del fichero:

0x02,0x04,0x05,0x02

A continuación se muestra la ejecución ideal:

Hilo1

```
Variable = 0
Lee byte 0x02 (variable=3)
Lee byte 0x04 (variable =6) => Sleep(2seg)
...
...
...
Lee byte 0x05 (variable = 7)
Lee byte 0x02 (variable = 9) => return OK!!!
...
...
```

Hilo2

```
...
...
...
Variable = 0
Lee byte 0x02 (variable =3)
Lee byte 0x04 (variable =6) => Sleep(2 seg)
...
...
Lee byte 0x05 (variable = 7)
Lee byte 0x02 (variable = 9) => return OK!!!
```

Es decir, resolveremos el reto si generamos un fichero con los siguientes 8 bytes:

0x02,0x04,0x02,0x04,0x05,0x02,0x05,0x02

00000000h: 02 04 02 04 05 02 05 02 ; .....

Si ejecutamos el crackme lo habremos resuelto:

