

=====
Reto Panda - Prueba #3 - SOLUCIÓN
=====

-- Fecha Publicación: 21.04.2009 --

Román Medina-Heigl Hernández

[<roman@rs-labs.com>](mailto:roman@rs-labs.com)

ÍNDICE DE CONTENIDOS

--[0x01 - Introducción]	3
--[0x02 - Herramientas utilizadas].....	4
--[0x03 - Análisis inicial].....	5
--[0x04 - Los hilos “secundarios”]	11
--[0x05 - Primer intento].....	15
--[0x06 – Las cien funciones].....	17
--[0x07 - Solución “aproximada”]	21
--[0x08 - Afinando la solución: funciones de tipo 2]	23
--[0x09 - Resolviendo las funciones de tipo 3].....	26
--[0x0a - Solución definitiva]	32
--[0x0b - Feedback & Greetz].....	33

--[0x01 - Introducción]

En este paper trataré de resumir mi solución a la última prueba del concurso que Panda Security ha celebrado recientemente (1/Abr - 9/Abr). A día de hoy todavía no se han publicado los resultados del reto por lo que aún no hay ganadores ni soluciones oficiales a las distintas pruebas. Este solucionario se ofrece sin garantía alguna y podría no ser 100% correcto. Tampoco pretende ser un howto detallado, simplemente trataré de esbozar mi solución y el razonamiento que seguí en la resolución de la prueba.

El concurso se ubicó en:

<http://www.retopanda.es/>

Constaba de 3 pruebas diferentes, todas ellas ejercicios de **ingeniería inversa** (“*crackmes*”), y donde la dificultad era creciente. Me ceñiré a la **prueba 3** (la última y supuestamente más difícil, correspondiente al primer premio del concurso).

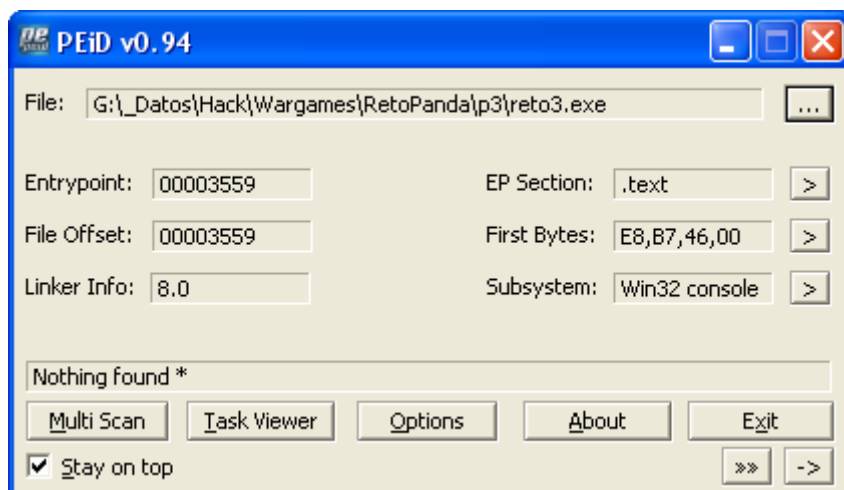
--[0x02 - Herramientas utilizadas]

En mayor o menor medida se utilizaron las siguientes:

- IDA Pro
<http://www.hex-rays.com/idapro/>
La herramienta por excelencia para análisis estático.
- OllyDbg
<http://www.ollydbg.de/>
Uno de los mejores debuggers para Windows.
- UltraEdit
<http://www.ultraedit.com/>
Excelente editor (ascii y hexadecimal).
- PEiD
<http://peid.has.it/>
Detección de packers, rutinas de crypto y compiladores.
- TrID
<http://mark0.net/soft-trid-e.html>
Otro identificador de tipos de fichero basado en firmas binarias.
- MinGW
<http://www.mingw.org/>
Gcc para Windows, etc.
- Crank
<http://crank.sourceforge.net>
Criptoanálisis y cifrados básicos.

--[0x03 - Análisis inicial]

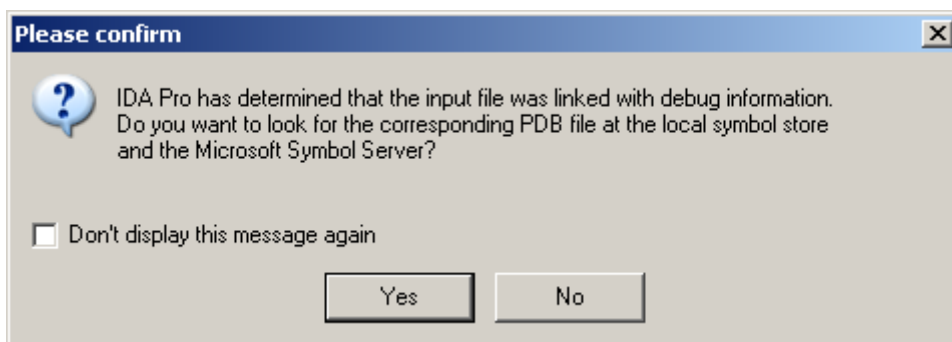
Nos bajamos el binario: **reto3.exe**. Lo analizamos y posteriormente ejecutamos (nos aseguran que no contiene virus; no está de más pasarlo por VirusTotal y como no, recomendable lanzarlo en un entorno controlado: Vmware):



```
c:\ Símbolo del sistema - reto3.exe
G:\_Datos\Hack\Wargames\RetoPanda\p3>md5sum reto3.exe
9c16d4d15c10c61e72e3223473b0f190 *reto3.exe
G:\_Datos\Hack\Wargames\RetoPanda\p3>trid reto3.exe
TrID/32 - File Identifier v2.02 - (C) 2003-06 By M.Pontello
Definitions found: 3555
Analyzing...
Collecting data from file: reto3.exe
60.8% (.EXE) Win32 Executable MS Visual C++ (generic) (31206/45/13)
16.6% (.EXE) Win32 Executable Generic (8527/13/3)
14.7% (.DLL) Win32 Dynamic Link Library (generic) (7583/30/2)
 3.9% (.EXE) Generic Win/DOS Executable (2002/3)
 3.8% (.EXE) DOS Executable Generic (2000/1)
G:\_Datos\Hack\Wargames\RetoPanda\p3>reto3.exe
-
```

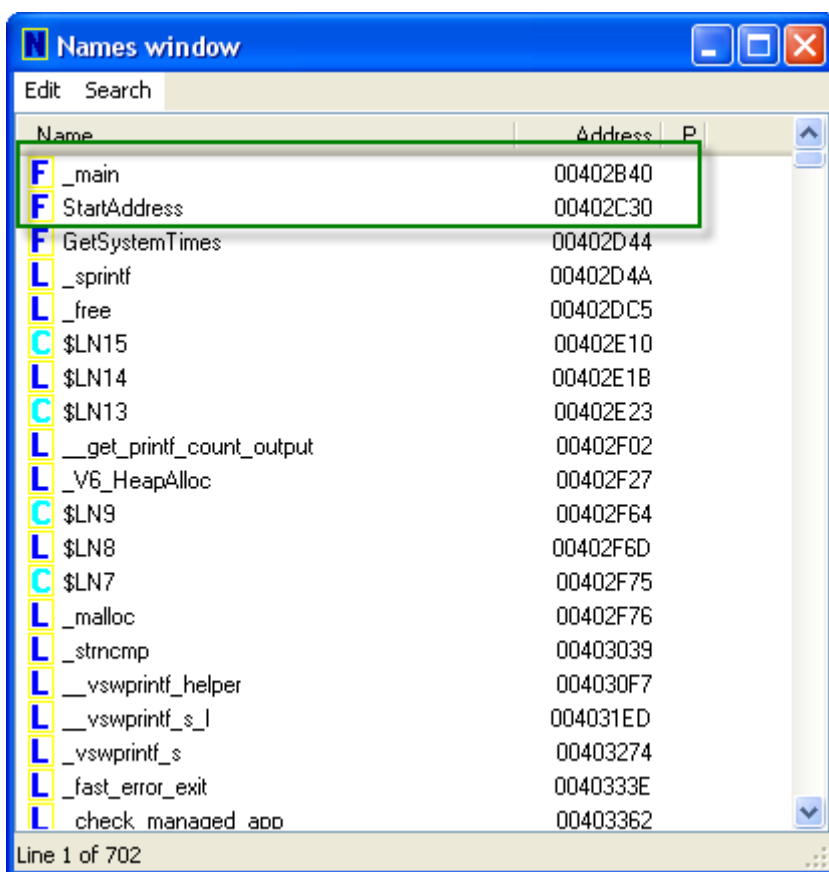
El programa no acepta entrada de usuario ni emite ningún mensaje al ser ejecutado. Las utilidades de identificación lanzadas tampoco son nada definitivas. Por ahora, ninguna pista :-/

Cargamos el todo-poderoso IDA-Pro. Lo primero que nos sorprende (y alegra) es:

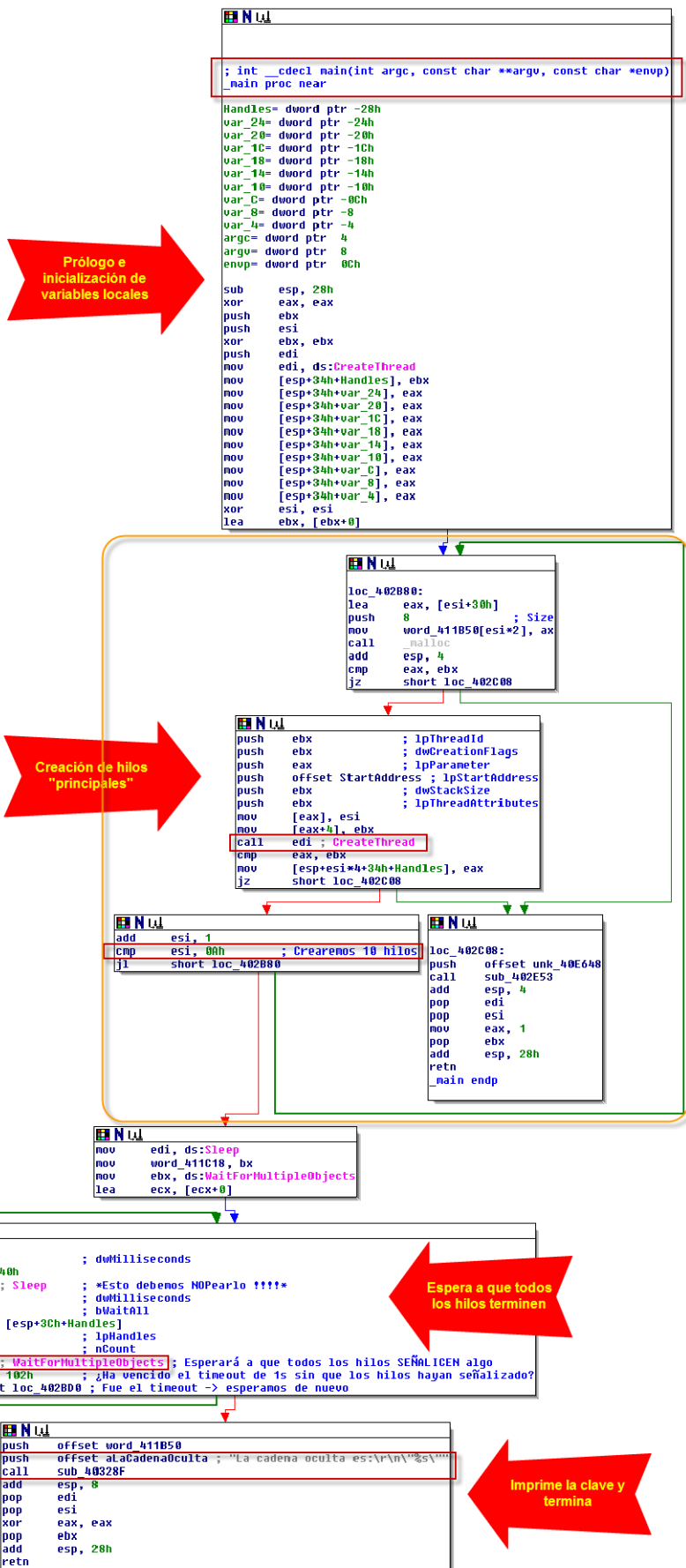


Esto es, el binario contiene información de depuración (lo que facilitará su análisis) y nos pregunta si queremos buscar/cargar los símbolos correspondientes. Por supuesto, asentimos. Además, el binario es pequeño (76 kbytes) y no está empaquetado. A juzgar por estas “facilidades”, todo apunta a que la prueba está pensada para que se resuelva (al menos en parte) mediante análisis estático y seguro que esconderá otras dificultades y “sorpresas”.

Una vez finalizada la carga comprobamos la ventana de “nombres” y nos fijamos en las dos primeras funciones:



Realmente la ejecución comenzará en el “entry point” (00403559), desde donde se llamará a diferentes rutinas de inicialización propias de un ejecutable compilado con Visual Studio (___security_init_cookie –el binario utiliza /gs-, __tmainCRTStartup, ...), pero no nos interesan. Partiremos de la rutina principal o “**main**”, cuya panorámica es la siguiente (tranquilos, en breve entramos en detalle y con un zoom suficiente para no quedarnos sin vista ☺):



Veámoslo desde el principio. Primero, la parte del prólogo de la función e inicialización de variables locales. IDA nos muestra el prototipo de la función en formato C en la cabecera (lo cual es de agradecer). Se inicializan las variables locales, casi todas a cero (utilizando los típicos trucos de optimización: “xor eax, eax” para poner eax a 0 y luego mover eax a las diferentes variables en pila). Nótese el “CreateThread”, que ya lo tenemos preparado en edi ☺



```

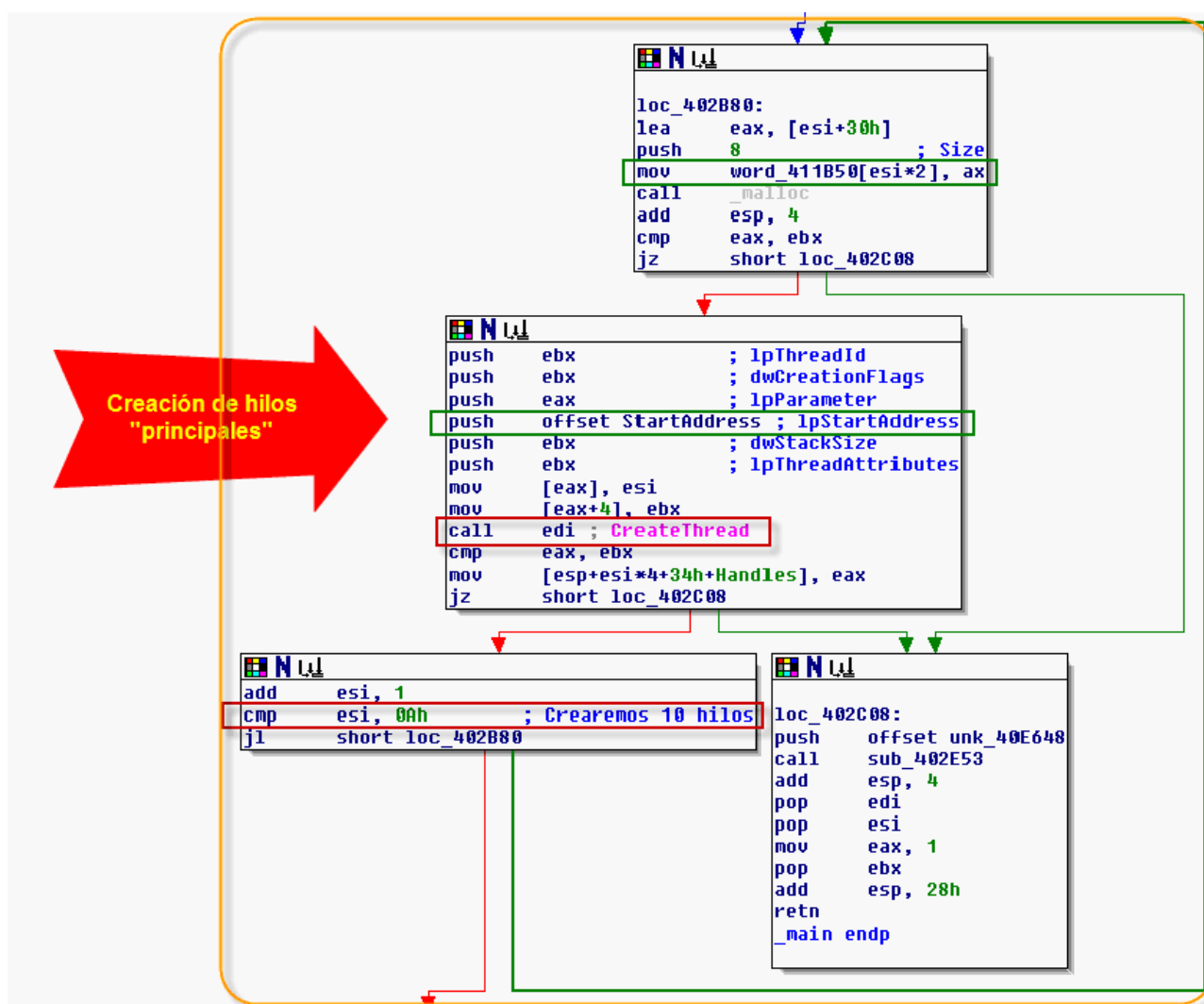
; int __cdecl main(int argc, const char **argv, const char *envp)
_main proc near

Handles= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
argc= dword ptr 4
argv= dword ptr 8
envp= dword ptr 0Ch

sub     esp, 28h
xor     eax, eax
push   ebx
push   esi
xor     ebx, ebx
push   edi
mov     edi, ds:CreateThread
mov     [esp+34h+Handles], ebx
mov     [esp+34h+var_24], eax
mov     [esp+34h+var_20], eax
mov     [esp+34h+var_1C], eax
mov     [esp+34h+var_18], eax
mov     [esp+34h+var_14], eax
mov     [esp+34h+var_10], eax
mov     [esp+34h+var_C], eax
mov     [esp+34h+var_8], eax
mov     [esp+34h+var_4], eax
xor     esi, esi
lea    ebx, [ebx+0]
    
```

Después tenemos el bucle de creación de hilos. En concreto, se crearán 10 threads que yo he llamado “principales” ya que (estoy adelantando acontecimientos) cada uno de estos hilos lanzará a su vez más hilos (lo veremos más adelante), que llamaré “secundarios” (para distinguirlos).

Además, en 411B50 tenemos una variable global que contendrá un array con 100 elementos de 2 bytes (1 word) cada uno. Cada word contendrá un carácter de la frase secreta que debemos averiguar (ocupa doble porque el carácter es Unicode). Cada hilo se corresponderá con un word de este array e inicialmente se asigna el número de hilo a dicha word. La rutina que se ejecuta para cada hilo reside en StartAddress y se le pasa como parámetro el número de hilo.



Una vez lanzados los 10 hilos principales (que ya estudiaremos en profundidad lo que hacen), el programa principal simplemente espera a que todos ellos terminen. El tiempo de sondeo es de 1 segundo (3E8h milisegundos) pero nos fijamos también que se introduce un sleep de algo más de 16 minutos (exactamente 1000 segundos o F40240h milisegundos). Este sleep no aporta nada realmente y más tarde lo anularemos (reduciremos su valor), para que no moleste ☺

Finalmente, cuando todos los hilos han terminado, se imprimirá la frase secreta: “La cadena oculta es: ...”. Queda claro que el objetivo de la prueba es averiguar dicha cadena.

```

mov     edi, ds:Sleep
mov     word_411C18, bx
mov     ebx, ds:WaitForMultipleObjects
lea     ecx, [ecx+0]

```

```

loc_402BD0:                ; dwMilliseconds
push    0F4240h
call    edi ; Sleep      ; *Esto debemos NOPearlo !!!!*
push    3E8h             ; dwMilliseconds
push    1                 ; bWaitAll
lea     ecx, [esp+3Ch+Handles]
push    ecx              ; lpHandles
push    esi              ; nCount
call    ebx ; WaitForMultipleObjects ; Esperará a que todos los hilos SEÑALICEN algo
cmp     eax, 102h        ; ¿Ha vencido el timeout de 1s sin que los hilos hayan señalado?
jz      short loc_402BD0 ; Fue el timeout -> esperamos de nuevo

```

```

push    offset word_411B50
push    offset alaCadenaOculta ; "La cadena oculta es:\r\n\''%s\''"
call    sub_40328F
add     esp, 8
pop     edi
pop     esi
xor     eax, eax
pop     ebx
add     esp, 28h
retn

```

Espera a que todos los hilos terminen

Imprime la clave y termina

--[0x04 - Los hilos “secundarios”]

Hemos visto que cada hilo principal ejecuta la rutina “StartAddress” pasándole como parámetro el número de hilo. Analicemos ahora dicha rutina. Comienza así:

```

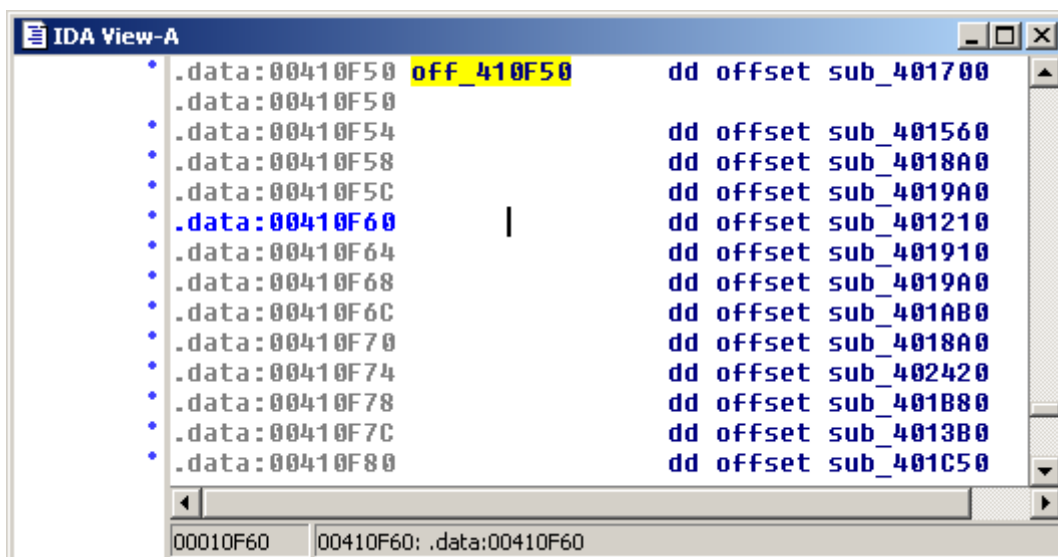
; DWORD __stdcall StartAddress(LPVOID)
StartAddress proc near

Handles= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr 4

sub     esp, 28h
xor     eax, eax
push   esi
mov     esi, [esp+2Ch+arg_0]
mov     [esp+2Ch+var_24], eax
mov     [esp+2Ch+var_20], eax
mov     [esp+2Ch+var_1C], eax
mov     [esp+2Ch+var_18], eax
mov     [esp+2Ch+var_14], eax
mov     [esp+2Ch+var_10], eax
mov     [esp+2Ch+var_C], eax
mov     [esp+2Ch+var_8], eax
mov     [esp+2Ch+var_4], eax
mov     eax, [esi] ; [esi] = num. hilo (0 - 9)
push   edi
push   eax ; Num. hilo
mov     eax, off_410F50[eax*4] ; Tabla con 100 punteros a función
xor     edi, edi
mov     [esp+34h+Handles], edi
call    eax ; sub_401700 ; Cada hilo llama a una función determinada
mov     ecx, [esi]
add     dword_411C1C, 1
add     esp, 4
mov     word_411B50[ecx*2].ax ; Cada hilo tiene 1 word asociado
cmp     [esi+4], edi ; ¿Es 0? (0=Hilo creado desde main)
jz     short loc_402C94

```

Lo más importante es que se llama a una función que depende del número de hilo. Para ello se parte de un array de 100 punteros a función (410F50) y se utiliza como índice precisamente el número de hilo. Comprobamos que efectivamente esta dirección contiene punteros a función:

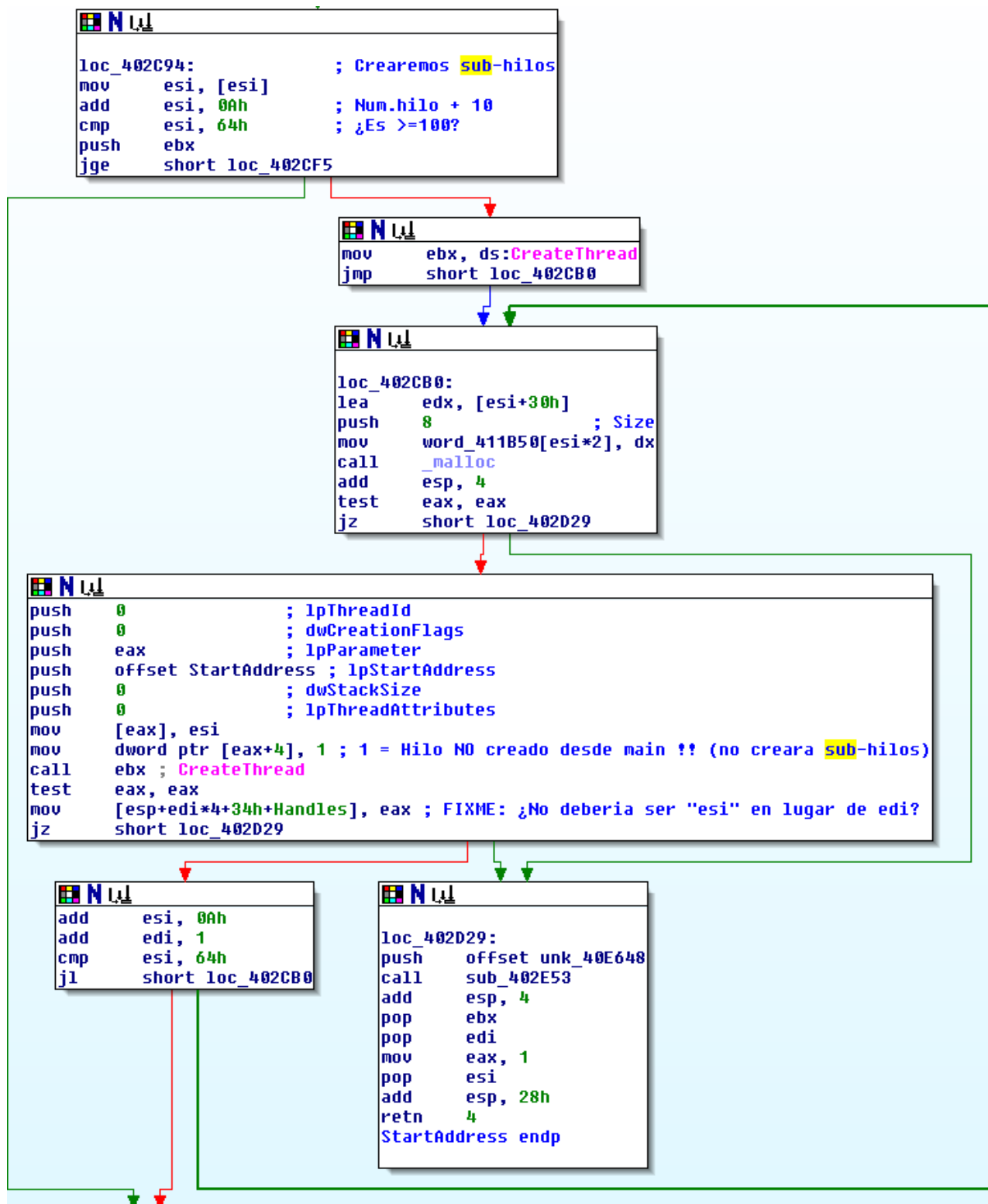


Volveremos a este punto más adelante en el documento.

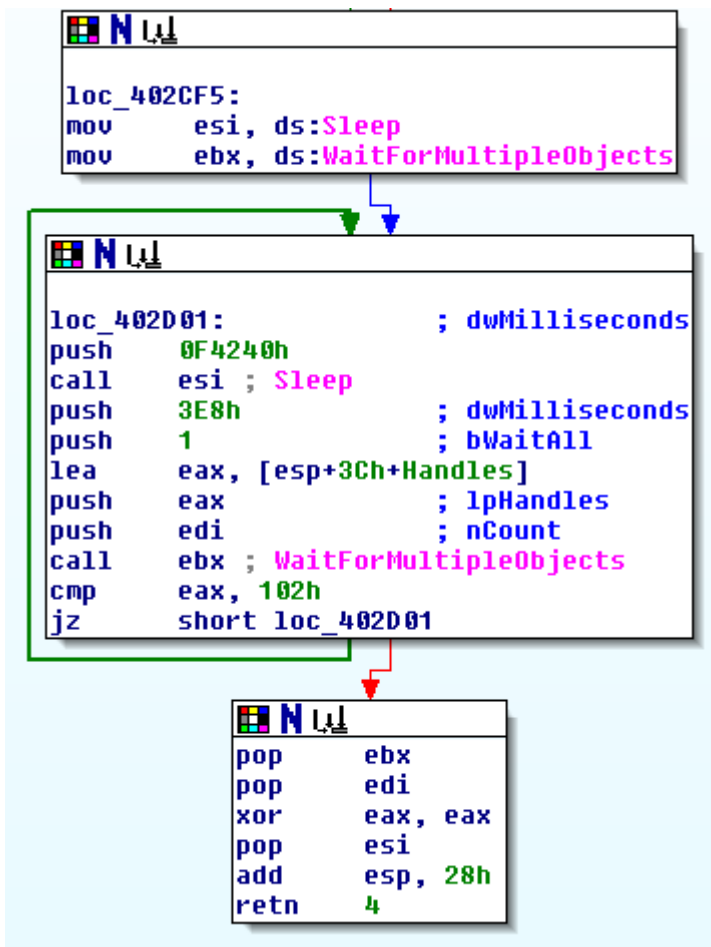
Siguiendo con el análisis del primer fragmento de la función “StartAddress”, el valor que inicialmente recibirá esta función estará entre 0 y 9 (ambos inclusive), que son los 10 hilos creados desde “main”. También cabe recalcar que cuando llamamos a una de las 100 funciones “disponibles” (en realidad son menos, algunos punteros se repiten), siempre se le pasa como parámetro el número de hilo (en algún caso se utilizará este valor, en otros no; esto es importante, puesto que quiere decir que una misma función puede dar lugar a una letra diferente, en función del hilo).

En el segundo fragmento se puede observar como se crean 9 hilos más (“secundarios”), por cada hilo “principal”. Por ejemplo, el primer hilo principal (0) da lugar a los secundarios 10, 20, 30, ..., 90. Vemos que utiliza un flag para distinguir entre hilo principal y secundario (para no entrar en un bucle, esto es, que un hilo “secundario” no de lugar a hilos “terciarios” y así sucesivamente).

Pero en definitiva, lo que nos importa es que al final tendremos un total de 100 hilos (10 principales y 90 secundarios) y que para cada uno se ejecuta una función de un array de punteros a la que se le pasa como parámetro el número de hilo.



El tercer y último fragmento nos resultará familiar puesto que se asemeja al final de la función “main”. Su cometido es que cada hilo principal finalice única y exclusivamente cuando lo hayan hecho los hilos secundarios correspondientes:



También se introduce un “sleep” de ~16 minutos que no vendría mal eliminar.

--[0x05 - Primer intento]

Conocemos ya cómo funciona grosso modo el crackme. También tenemos claro el objetivo del mismo y la ubicación en memoria de la frase secreta que deberemos averiguar. A simple vista (sin entrar a analizar todavía las 100 funciones del array) parece que parcheando algunas llamadas a “sleep” lo tendremos resuelto (veremos que no es tan fácil) así que nuestra primera aproximación consistirá en cargar el binario con OllyDbg e ir haciendo modificaciones sobre el mismo.

Lo primero que se nos ocurre es eliminar los sleeps. Tenemos diferentes formas de hacerlo. Quizás la más conservadora es mantener la llamada a sleep pero disminuyendo el valor que se le pasa como parámetro. Por ejemplo, pasándole un cero:

The screenshot displays the OllyDbg interface with the following components:

- Disassembly Window:** Shows assembly instructions from address 00402B40 to 00402BE8. Key instructions include:
 - 00402B80: `PUSH 0` (highlighted in red)
 - 00402B81: `PUSH EBX`
 - 00402B82: `PUSH EBX`
 - 00402B83: `MOV DWORD PTR DS:[EAX],ESI`
 - 00402B84: `MOV DWORD PTR DS:[EAX+4],E`
 - 00402B85: `CALL EDI`
 - 00402B86: `CMP EAX,EBX`
 - 00402B87: `JE SHORT 00402C08`
 - 00402B88: `MOV DWORD PTR SS:[ESI+4+ESI]`
 - 00402B89: `JE SHORT 00402C08`
 - 00402B8A: `ADD ESI,1`
 - 00402B8B: `CMP ESI,0A`
 - 00402B8C: `JL SHORT 00402B80`
 - 00402B8D: `MOV EDI,DWORD PTR DS:[<&KERNEL32.Sleep>`
 - 00402B8E: `MOV WORD PTR DS:[411C18],BX`
 - 00402B8F: `MOV EBX,DWORD PTR DS:[<&KERNEL32.WaitFo`
 - 00402B90: `LEA ECX,[ECX]`
 - 00402B91: `PUSH 0F4240`
 - 00402B92: `CALL EDI`
 - 00402B93: `PUSH 3E8`
 - 00402B94: `PUSH 1`
 - 00402B95: `LEA ECX,[ARG.5]`
 - 00402B96: `PUSH ECX`
 - 00402B97: `PUSH ESI`
 - 00402B98: `PUSH EBX`
 - 00402B99: `CALL EBX`
 - 00402B9A: `CMP EAX,102`
 - 00402B9B: `JE SHORT 00402BD0`
- Assemble Dialog Box:** Opened at address 00402BD0, showing the instruction `PUSH 0`. Options `Keep size` and `Fill rest with NOPs` are checked.
- CPU Registers Window:** Shows `Arg1 = 8` and `ret03.00402F76`.
- Registers Window:** Shows `Time = 1000000. ms`, `KERNEL32.Sleep`, `Timeout = 1000. ms`, `WaitAll = TRUE`, `HandleList`, `Count`, and `KERNEL32.WaitForMultipleObjects`.

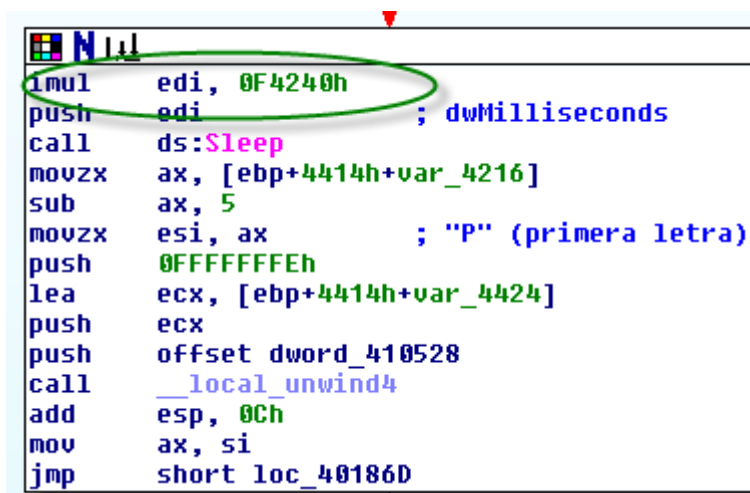
Pero tras eliminar los sleeps (tanto de main como de StartAddress) y ejecutar de nuevo el programa el comportamiento es similar: no se obtiene salida en pantalla, el programa se queda aparentemente bloqueado. Si lo dejamos un buen rato corriendo (~1h) y luego miramos el contenido de la memoria, concretamente donde sabemos que irá la frase secreta, se observa cómo se comienza a formar la cadena oculta:

--[0x06 – Las cien funciones]

Si revisamos con IDA las funciones que aparecen listadas en el array de punteros veremos que todas ellas tienen algún tipo de “impedimento” que imposibilita que la función retorne en el momento (que sería lo deseable, para que todos los hilos terminen en un tiempo razonablemente corto y el programa acabe devolviendo la frase secreta).

Podemos clasificar las funciones en tres tipos diferentes:

- **Tipo 1.** Aquellas que simplemente introducen algún sleep no deseado. Éstas son simples de “apañar”: bastará con NOPear los sleeps (o trucarlos tal y como hicimos con los sleeps de main y StartAddress). Veamos un ejemplo:
 - ❖ **401700:** la primera función (correspondiente al primer hilo, esto es, al primer carácter de la frase secreta). Estudiemos sólo un fragmento:



```
imul edi, 0F4240h
push edi
call ds:Sleep
movzx ax, [ebp+4414h+var_4216]
sub ax, 5
movzx esi, ax
push 0FFFFFFEh
lea ecx, [ebp+4414h+var_4424]
push ecx
push offset dword_410528
call __local_unwind4
add esp, 0Ch
mov ax, si
jmp short loc_40186D
```

En este caso hay un sleep cuyo valor de espera depende de “edi” y que es multiplicativo (por un factor de ~16 minutos [=0F4240h milisegundos]). Si “edi” valiera 1 ya tendríamos que esperar bastante (los 16 minutos) pero si encima este valor fuera mayor el efecto podría resultar devastador. ¿Qué contiene “edi” en este caso particular (401700)? Acudimos al comienzo de la función, donde podemos leer:

```

push    ebp                ; Rutina Hilo 0
lea     ebp, [esp-4414h]
mov     eax, 4414h
call    __alloca_probe
push    0FFFFFFEh
push    offset unk_40EB40
push    offset __except_handler4
mov     eax, large fs:0
push    eax
sub     esp, 10h
mov     eax, dword_410528
xor     [ebp+4414h+var_441C], eax
xor     eax, ebp
mov     [ebp+4414h+var_4], eax
push    ebx
push    esi
push    edi
push    eax
lea     eax, [ebp+4414h+var_4424]
mov     large fs:0, eax
mov     [ebp+4414h+var_442C], esp
mov     edi, [ebp+4414h+arg_0]
mov     [ebp+4414h+noobject], 0FFFFFFFh
xor     esi, esi

```

Es decir, “edi” contiene el primer (y único) argumento pasado a la función, esto es, el número de hilo. Casualmente, en este caso la función corresponde al hilo 0, por lo que edi vale cero y tenemos un sleep(0) que no sería necesario ni parchear ☺

Sin embargo, código similar se repite en otras funciones (otros hilos) del crackme dónde edi ya no va a ser nulo, y que por tanto habrá que tener en cuenta (esto es, ¡será necesario parchear los IMULs!).

- **Tipo 2.** Aquellas que dependen de un instante de tiempo (fecha y hora) indeterminado. Para identificar dicho instante sólo se dispone de un “hash” lo que obliga a emplear fuerza bruta (no basta con NOPear la comparación con el hash, es necesario obtener el instante original ya que el resultado de la función depende de este último). Veamos un ejemplo:
 - ❖ **401AB0:** función correspondiente al hilo 7 (entre otros). A continuación copio el fragmento más descriptivo de esta función. En él se puede observar cómo se realizan ciertas llamadas a funciones que dependen de la hora (GetSystemTime, GetLocalTime, GetSystemTimeAsFileTime) o incluso de los tiempos de proceso (GetSystemTimes) aunque en este caso realmente la función importante es GetSystemTime, ya que es la estructura SystemTime la que se utiliza para generar el hash. A modo simplificado lo que se hace es:
 - Obtener la fecha y hora UTC (GetSystemTime). Se almacenará en una variable local tipo estructura SystemTime.
 - Poner a 0 los milisegundos y los segundos del valor anterior (para que la fuerza bruta sea factible: “sólo” habrá que obtener fecha, hora y minuto).
 - Calcular el hash (MD5) de la estructura SystemTime completa.
 - Compararlo con un hash precalculado, o mejor dicho, con un fragmento del mismo de 8 bytes (16 dígitos hexadecimales).

- Si los primeros 8 bytes del hash que nosotros calculamos coinciden con el fragmento del hash precalculado la función continúa hasta terminar y habremos obtenido el carácter de la frase secreta que andábamos buscando.
- Si no, espera 1 segundo y vuelve a leer la hora, repitiendo todo el proceso de nuevo.

```

loc_401AE0:
lea     eax, [esp+0C0h+SystemTime]
push   eax           ; lpSystemTime
call   edi ; GetSystemTime ; Fecha y hora (UTC)
lea     ecx, [esp+0C0h+var_A0]
push   ecx           ; lpSystemTime
call   ebx ; GetLocalTime ; Fecha y hora (Local)
lea     edx, [esp+0C0h+SystemTimeAsFileTime]
push   edx           ; lpSystemTimeAsFileTime
mov     [esp+0C4h+SystemTime.wMilliseconds], si
mov     [esp+0C4h+SystemTime.wSecond], si
mov     [esp+0C4h+var_92], si
mov     [esp+0C4h+var_94], si
call   ds:GetSystemTimeAsFileTime ; Fecha y hora (UTC) en otro formato
lea     eax, [esp+0C0h+UserTime]
push   eax           ; lpUserTime
lea     ecx, [esp+0C4h+KernelTime]
push   ecx           ; lpKernelTime
lea     edx, [esp+0C8h+IdleTime]
push   edx           ; lpIdleTime
call   GetSystemTimes
push   8003h         ; Algid -> *MD5*
lea     eax, [esp+0C4h+Dest]
push   eax           ; Dest
lea     ecx, [esp+0C8h+SystemTime]
push   10h          ; dwDataLen
push   ecx           ; pbData
call   sub_401000
add     esp, 10h
push   3E8h         ; dwMilliseconds
call   ds:Sleep
push   10h          ; MaxCount
lea     edx, [esp+0C4h+Dest]
push   offset Str2  ; "50D1EB43BF2928AD"
push   edx           ; STR1
call   _strncmp
add     esp, 0Ch
test   eax, eax
jnz    short loc_401AE0
    
```

Para el cálculo del hash se llama a la función de 401000, cuyo prototipo es:
*int __cdecl sub_401000(BYTE *pbData, DWORD dwDataLen, char *Dest, ALG_ID Algid)*

Dicha función hace uso de RSAENH.DLL (Microsoft Enhanced Cryptographic Provider). El valor 8003h (Algid)¹ se corresponde con el algoritmo MD5. En algunas otras funciones del crackme se utilizará SHA1 (8004h).

- **Tipo 3.** Son similares al tipo 2 pero esta vez se basan en los tiempos de procesador (utilizan GetSystemTimes). Implementan tanto MD5 como SHA1 y también habrá que emplear fuerza bruta para su resolución, aunque en este caso la forma de resolución será diferente (de ahí que las hayamos diferenciado). Por ejemplo, esto sería un fragmento de **402720**:

```

loc_402764:
lea     eax, [esp+0A8h+var_90]
push   eax           ; lpUserTime
lea     ecx, [esp+0ACh+var_98]
push   ecx           ; lpKernelTime
lea     edx, [esp+0B0h+pbData]
push   edx           ; lpIdleTime
call   GetSystemTimes
push   3E8h          ; dwMilliseconds
call   edi           ; Sleep
mov     ecx, [esp+0A8h+var_9C]
sub     ecx, [esp+0A8h+IdleTime.dwHighDateTime]
mov     esi, [esp+0A8h+var_94]
sub     esi, [esp+0A8h+KernelTime.dwHighDateTime]
mov     eax, 0CCCCCCCdh
mul     ecx
shr     edx, 3
mov     [esp+0A8h+var_9C], edx
mov     eax, 0CCCCCCCdh
mul     esi
push   8003h         ; Algid
lea     eax, [esp+0ACh+Dest]
push   eax           ; Dest
lea     ecx, [esp+0B0h+pbData]
shr     edx, 3
push   18h           ; dwDataLen
push   ecx           ; pbData
mov     dword ptr [esp+0B8h+pbData], 11CAFCEh
mov     [esp+0B8h+var_98], 136A339h
mov     [esp+0B8h+var_90], 1255852h
mov     [esp+0B8h+var_8C], 59h
mov     [esp+0B8h+var_94], edx
call   sub_401000
push   10h           ; MaxCount
lea     edx, [esp+0BCh+Dest]
push   offset a74f72ff73b2dd6 ; "74F72FF73B2DD6EC"
push   edx           ; Str1
call   strncmp
add     esp, 1Ch
test   eax, eax
jnz    loc_402764

```

¹ [http://msdn.microsoft.com/en-us/library/aa375549\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa375549(VS.85).aspx)

--[0x07 - Solución “aproximada”]

Lo primero que se me ocurrió fue parchear el binario de forma que:

- o Se anularan todos los sleeps (sobre todo los de las funciones del tipo 1).
- o Los “strncmp” (funciones tipo 2 y 3) siempre fueran “exitosos” (devolvieran 0)

El binario resultante produjo una salida similar a:

```
G:\_Datos\Hack\Wargames\RetoPanda\p3>reto3_4_1_8.exe
La cadena oculta es:
"ProtectéoD4a@adDæt4odrwÆeÆ04ÆiÇ`are04TrozaDÆ04{ackerÆ04Æiag4aD_4ot{er4QDterDet4
t{reatÆ74444PaD_a4444"
```

Cada vez que lo ejecutaba obtenía un resultado diferente (lo cual era congruente con la dependencia de fecha/hora y tiempos de proceso, parámetros que lógicamente variaban con cada ejecución). En realidad había unos caracteres fijos, que eran los correspondientes a las funciones de tipo 1 (que no dependían ni de la hora ni de los tiempos de proceso). Todo encajaba aunque por ahora el mensaje secreto resultaba ilegible.

Sin embargo, había ocasiones en que la frase secreta “parecía” más clara:

```
G:\_Datos\Hack\Wargames\RetoPanda\p3>reto3_4_1_8.exe
La cadena oculta es:
"ProtectfoD aQalDut \lr|ueu0 uiÇ`are0 TrozaDu0 {acceru0 uiai aDS ot{er jDterDet
t{reatu! PaDSa "
```

Asumí que el carácter espaciador era correcto en el mensaje anterior. De los demás caracteres, sólo los de tipo 1 eran correctos (y puede que algunos otros de los de tipo 2 y 3 –como el espaciador-, pero eso sería de pura casualidad) así que por claridad realicé una nueva modificación en el binario de forma que las funciones de tipo 2 y 3 devolvieran siempre un “*”. Esto es lo que obtuve:

```
G:\_Datos\Hack\Wargames\RetoPanda\p3>reto3_4_1_9.exe
La cadena oculta es:
"Protect*o**a*a***t***r**e*****are**Tro*a*****ac*er*****a**a***ot*er***ter*et*
t*reat*****Pa**a*****"
```

Donde al sustituir los caracteres espaciadores (que habíamos intuido en el penúltimo paso) daría lugar a:

```
G:\_Datos\Hack\Wargames\RetoPanda\p3>reto3_4_1_9.exe
La cadena oculta es:
"Protect*o* a*a***t **r**e** *****are* Tro*a*** *ac*er** **a* a** ot*er **ter*et
t*reat** Pa**a "
```

Esto ya tenía mejor pinta. Ahora bastaría con tener un poco de imaginación y paciencia, y asignar a cada función de tipo 2 y 3 un carácter. Con la ayuda de la herramienta criptográfica “Crank” (aunque se podía haber hecho perfectamente a mano), fui sustituyendo los caracteres desconocidos, asumiendo que las posiciones de la cadena correspondientes a una misma función en el array de funciones representaban siempre el mismo carácter (a modo de cifrado por

transposición). La hipótesis anterior era fácil de deducir/comprobar²: por ejemplo, el primer carácter (la “P”) se corresponde con la primera función del array (401700); si miramos al final del texto cifrado hay otra “P” (“Pa**a”) que también corresponde con la función 401700. Vale, la “P” ya la teníamos, pero pensemos en lo siguiente: ¿qué pasa si el “Pa**a” del final fuera en realidad “Panda” (como se puede intuir)? Querría decir que la función 402420 es una “n” y la 402190 una “d”, por lo que utilizando el array de punteros a función podemos reemplazar otros caracteres del texto:

```
"Protect*on a*a*n*t **r**e** ****are* Tro*an** *ac*er** **a* and ot*er *n*ternet
t*reat** Panda "
```

Viendo lo anterior ya era sencillo ir completando y resultaba evidente que la frase secreta podía ser:

```
"Protection against viruses, spyware, Trojans, hackers, spam and other internet
threats. Panda "
```

Si buscamos en Internet comprobamos además que el texto anterior coincide con uno de los lemas de Panda. Parecía claro que habíamos terminado con éxito la prueba. ¿O no? No podía ser, demasiado sencillo... algo no cuadraba. Llamaban la atención dos detalles:

- la T de “Trojans” estaba en mayúsculas y sabíamos que era correcta (pues era de tipo 1).
- el lema de Panda incluía la palabra “Internet” en vez de “internet”, es decir, la primera “I” era mayúscula.

Por tanto, nos preguntamos... ¿y si alguno de los caracteres de tipo 2 o 3 que hemos “deducido” no es 100% correcto (puede haber una mayúscula/minúscula cambiada, o por ejemplo, que “threats.” fuera realmente “threats!”)? **En este momento no había forma de saber si nuestra solución actual era totalmente correcta** y la solución a la prueba se debía enviar por correo electrónico a Panda debiendo ser única (sólo se aceptaría, según las reglas del concurso, la primera solución recibida; el resto se descartaría... ¡Qué mala uva!).

² Se trataba de una aproximación que era cierta en la mayoría de los casos (posteriormente se comprobaría que existían caracteres que no sólo dependían de la función empleada sino además de su posición, esto es, del número de hilo). Esto lo descubrí en una ocasión en la que se me ocurrió parchear el número de hilo e introducir siempre un 0 (la idea era ahorrarme tener que parchear todos esos IMUL que dependían del número de hilo).

--[0x08 - Afinando la solución: funciones de tipo 2]

No fue fácil resistir la tentación de enviar la solución anterior pero habiendo llegado tan lejos no merecía la pena arriesgarse a que la solución no fuera correcta. Para seguir avanzando pensamos cómo resolver las funciones problemáticas, comenzando por las de tipo 2. Cómo vimos anteriormente, éstas tienen en común que el resultado de la función depende de un instante de tiempo, desconocido a priori, y que para averiguar el mismo te dan el comienzo de un hash MD5 o SHA1.

Pensando en diferentes soluciones y teniendo en cuenta que dependiendo de la función se utilizaban funciones de tiempo diferentes así como algoritmos de hashing distintos, el intentar resolver cada función por separado (implementando diferentes programas de fuerza bruta - parecidos pero cada uno con particularidades añadidas-) parecía más tedioso. El mejor camino a tomar era modificar el tiempo del sistema y esperar a que dichas funciones se acabasen “resolviendo” solas. Llegado a este punto podíamos simplemente poner a correr el crackme y esperar horas / días con la esperanza de que las funciones fueran acabando una a una. Pero así parecía claro que no íbamos a ganar así que... ¿por qué no acelerar el proceso modificando la fecha/hora del sistema?

Para ello nos creamos el siguiente programa en C:

```
#include <stdio.h>
#include <sys/time.h>
#include <winsock2.h>

settimeofday (const struct timeval *tv, const struct timezone *tz)
{
    SYSTEMTIME st;
    struct tm *ptm;
    int res;

    tz = tz; /* silence warning about unused variable */

    ptm = gmtime(&tv->tv_sec);
    st.wYear      = ptm->tm_year + 1900;
    st.wMonth     = ptm->tm_mon + 1;
    st.wDayOfWeek = ptm->tm_wday;
    st.wDay       = ptm->tm_mday;
    st.wHour      = ptm->tm_hour;
    st.wMinute    = ptm->tm_min;
    st.wSecond    = ptm->tm_sec;
    st.wMilliseconds = tv->tv_usec / 1000;

    res = !SetSystemTime(&st);

    // printf ("%d = settimeofday (%x, %x)", res, tv, tz);

    return res;
}

int main(int argc, char **argv) {
    struct timeval t;
    long int start, stop;
    int interval;

    printf("Reto #3 - Panda - by RoMaNSoFt, 2009 - <roman@rs-labs.com>\n");

    if (argc == 4) {
        interval = atoi(argv[3]);
    } else if (argc != 3) {
```

```

        printf("Syntax: %s <offset_hours> <length_hours> [interval_msecs]\n", argv[0]);
        return;
    } else {
        interval=1000;
    }

    printf("Getting current time...\n");

    if ( gettimeofday(&t, NULL) == -1) {
        printf("Cannot read time! Aborted.\n");
        return;
    }

    printf("%ld secs, %ld usecs\n", t.tv_sec, t.tv_usec);

    start = t.tv_sec + 3600 * atol(argv[1]);
    stop = t.tv_sec + 3600 * atol(argv[2]);

    printf("Start: %ld\nStop: %ld\n", start, stop);
    printf("Setting time ... (bruting)\n");

    while (start<=stop) {
        t.tv_sec = start;
        t.tv_usec = 0;
        settimeofday(&t, NULL);
        _sleep(interval);
        start += 60;
    }

    printf("Finished! :-)\n");
}

```

La idea es la siguiente: el programa anterior modificará la fecha/hora del sistema, en incrementos de un minuto. El intervalo de tiempo entre cambio es por defecto de 1 segundo (se puede elegir). Es decir, si ejecutamos el programa anterior, con los parámetros correctos veremos cómo el reloj de sistema va cambiando de valor con cada segundo, pero incrementándose en 1 minuto (el efecto es muy curioso, probadlo). De esta forma, en 60 segundos reales podemos recorrer todos los valores posibles dentro del rango de una hora completa (esto es posible porque el hash a romper se computa siempre con un valor fijo de 0 segundos y 0 milisegundos, como ya vimos anteriormente).

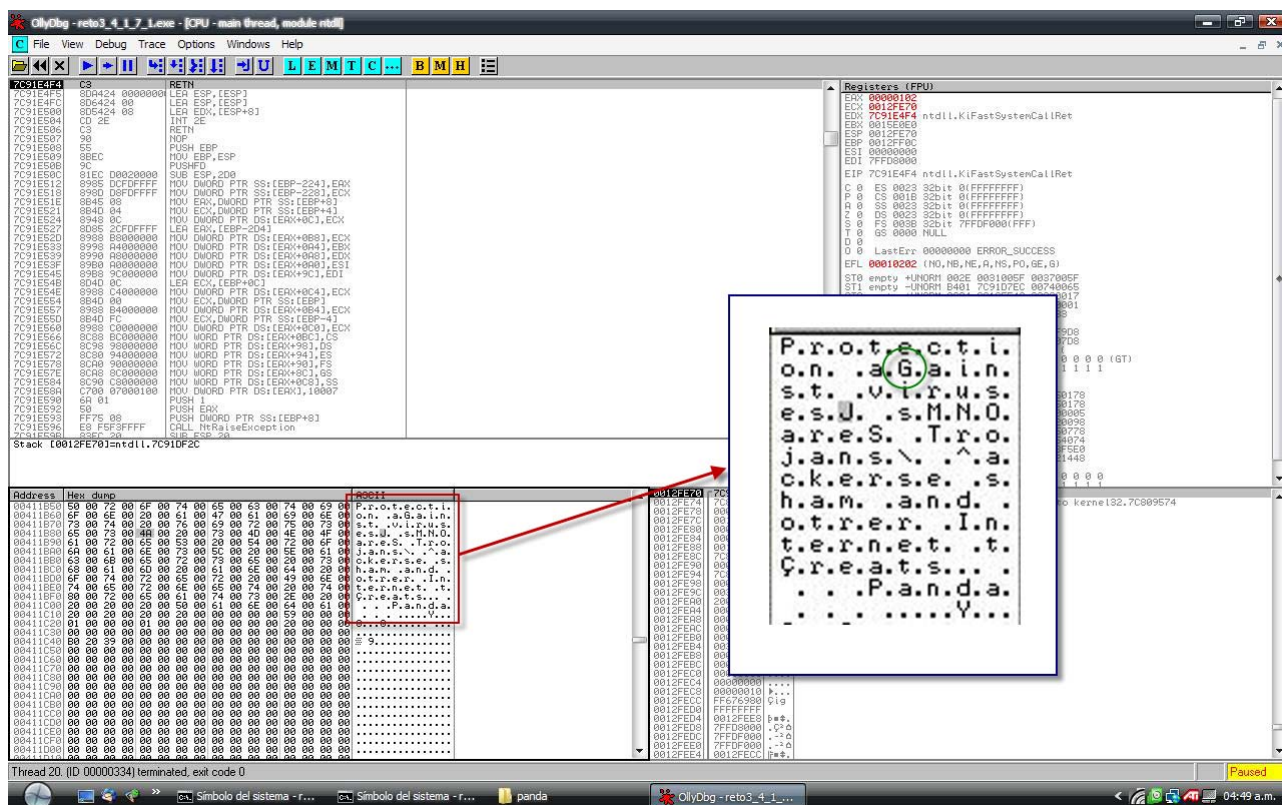
El programa acepta tres parámetros: un offset en horas, una “longitud” también en horas y (opcionalmente) el tiempo entre cambios en milisegundos (por defecto, 1000 [= 1 s]). Lo que hará será leer la fecha/hora actual del sistema, sumar el offset (con esto se calcula la hora de comienzo que utilizaremos en la fuerza bruta) y por último entra en un bucle en el que se incrementa la fecha/hora en 1 minuto, a cada segundo que transcurre (o bien según el tiempo entre cambio que le indiquemos).

La razón de utilizar C (compilado con “gcc” del paquete MinGW) es tratar de optimizar el proceso posterior de fuerza bruta. Al ser más rápido que otros lenguajes basados en scripting, las diferentes funciones del crackme tendrán más tiempo disponible para leer la nueva fecha/hora (que el programa C irá cambiando) y por tanto se supone que podremos minimizar el tiempo de espera de las funciones.

Este último parámetro (tiempo de sleep) es importante y a la vez problemático. Si se anulaban completamente los sleeps de estas funciones de tipo 2, el proceso reto3.exe copaba todo el tiempo de procesador y lo que es peor, al cabo de un rato corriendo acababa fallando y muriendo (quizás tenía algún tipo de “memory leak” cuyo efecto era acelerado por el hecho de suprimir los sleeps). Por otro lado, el tiempo de proceso se va a tener que repartir igualmente entre los threads, aunque

todos ellos intenten acapararlo. En definitiva, lo importante es limitar el tiempo de intervalo de nuestro programa en C; dejándolo como está (1 segundo) y dejando los sleeps también a 1 segundo debería ser suficiente para evitar problemas.

En mi caso realicé diferentes pruebas llegando a reducir el tiempo entre cambios a 50 ms y realizando varias pasadas barriendo el espectro de horas/minutos de los 3 días correspondientes a la prueba 3 del reto. También probé a ampliar dicho espectro. E incluso realicé pruebas con parámetros diferentes en varios ordenadores simultáneamente. La operativa era siempre la misma: lanzar el crackme (modificado con un tiempo de sleep apropiado) y lanzar una o varias veces el programa de fuerza bruta de tiempo; luego hacer un “attach” al proceso con OllyDbg y mirar la posición de memoria donde se forma la frase secreta. En una de esas pruebas conseguí llegar a:



Lo primero que llama la atención es que aparece una “G” mayúscula que no esperábamos (menos mal que no enviamos la primera solución...). Aún así el texto está legible casi en su totalidad. Pero todavía no está completo. ¿Enviamos la solución ya?

--[0x09 - Resolviendo las funciones de tipo 3]

Tampoco quise arriesgarme (aunque estuve tentado y de hecho, en este punto lancé una consulta a Panda para que me confirmaran si de verdad sólo aceptaban la primera solución recibida...) así que decidí ir a por el resto de funciones: las de tipo 3. Éstas coinciden “casualmente” con los caracteres erróneos del paso anterior. Veámoslos a todo color:



Cada color se corresponde con una función distinta:

- Rojo -> 402520 (presumiblemente: ",")
- Naranja -> 402620 (presumiblemente: “p” o “P”)
- Amarillo -> 402720 (presumiblemente: “y” o “Y”)
- Verde -> 402820 (presumiblemente: “w” o “W”)
- Azul -> 402a20 (presumiblemente: “h” o “H”)

Se puede observar que todavía hay grados de libertad (por culpa de las mayúsculas/minúsculas) así que debemos eliminarlos para obtener la solución final.

Esta vez no parecía fácil engañar al sistema: no encontré la forma (al menos sencilla) de modificar los tiempos de proceso que se reciben al llamar a GetSystemTimes. Me fui a la opción más difícil (y que no conseguí hacer funcionar): intentar simular el comportamiento de la función a romper, construyendo la implementación en C de la misma y añadiendo además la lógica de fuerza bruta. Como ejemplo (y sólo para que os hagáis una idea), incluyo el siguiente código en C correspondiente a una de las varias pruebas de concepto que realicé (aunque sinceramente ya no recuerdo si era alguna de las versiones finales o alguna versión preliminar)³:

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

#include "md5.h"

/* sub_402720 - "y" */

void do_md5(unsigned char *src, unsigned char *dst, int len){

    md5_state_t state;
    md5_byte_t digest[16];
    int di;

    md5_init(&state);
    md5_append(&state, (const md5_byte_t *)src, len);
    md5_finish(&state, digest);
    for (di = 0; di < 16; ++di)
        sprintf(&dst[di*2], "%02X", digest[di]);
    dst[32]='\0';
}
```

³ En cualquier caso, insisto: tómese como una simple ilustración ya que no me funcionó. Y si te sientes con fuerzas corrígelo y envíame un parche ;-)

```

    return;
}

int main(int argc, char **argv) {
    long int start, stop;
    int interval;
    long int i, j;
    char hash[1000];
    char *target="74F72FF73B2DD6EC";
    unsigned int src[7];

    printf("Reto #3 - Bruteforce - Panda - by RoMaNSoFt, 2009 - <roman@rs-labs.com>\n");
    printf("Target: %s\n", target);

    /*

pbData= byte ptr -0A0h
var_9C= dword ptr -9Ch
var_98= dword ptr -98h
var_94= dword ptr -94h
var_90= dword ptr -90h
var_8C= dword ptr -8Ch

shr     edx, 3
mov     [esp+0A8h+var_9C], edx
shr     edx, 3
...
mov     dword ptr [esp+0B8h+pbData], 11CAFCEh
mov     [esp+0B8h+var_98], 136A339h
mov     [esp+0B8h+var_90], 1255852h
mov     [esp+0B8h+var_8C], 59h
mov     [esp+0B8h+var_94], edx

*/

    src[0]=0x11CAFCE;
    src[1]=0; /* Parece 0 si lo miro con Olly (guessed) */
    src[2]=0x136A339;
    src[3]=0; /* (edx) Guess it */
    src[4]=0x1255852;
    src[5]=0x59;

    src[6]=0; /* \0 */

//    printf("%08X %08X %08X %08X %08X %08X\n", src[0], src[1], src[2], src[3], src[4], src[5]);

    printf("Cracking...\n");

//    for (src[1]=0; src[1]<=0xffffffff ; src[1]++) {
//        for (src[1]=0xffffffff >> 3; src[1]>=0 ; src[1]--) {
//            src[3]=src[1] >> 3;
//            do_md5((unsigned char *)src, hash, 24);
//            if (!strncmp(hash, target, 16)) {
//                printf("Found -> %08X / %d\n", src[3]);
//                break;
//            }
//        }
//    }

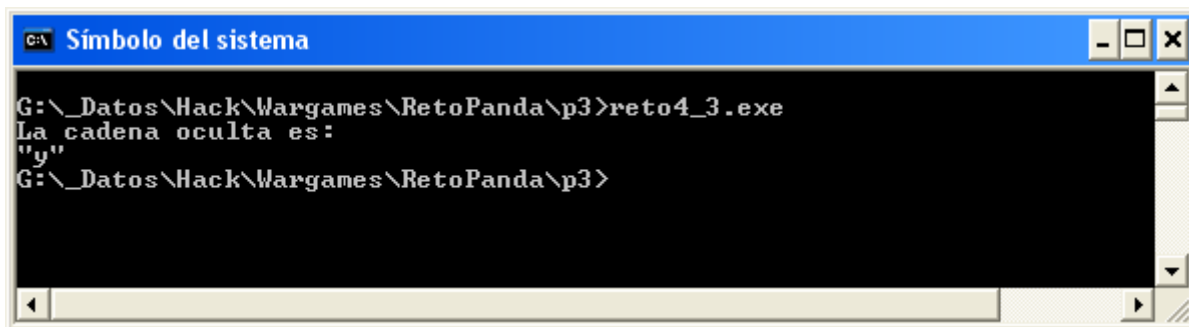
    printf("Finished! :-)\n");
}

```

Para el caso del SHA1 hice una prueba similar, que también resultó infructuosa. Está claro que algo estaba haciendo mal (no resulta tan sencillo portar código de ASM a C y menos con prisas...).

Opté por una vía alternativa (que esta vez sí funcionaría): parchear el binario del reto para que en lugar de llamar a `GetSystemTimes` llamara a una función “custom” (creada por mí) que devolviera secuencialmente 0, 1, 2, 3... Para que el “apaño” funcionara aislé también los caracteres

de forma que el binario sólo calculara un carácter (que lógicamente sería uno de los cinco dudosos). Ya que estaba con la función 402720 (supuesta “y” o “Y”) fui a por ella y obtuve lo siguiente:



¡Bingo! Me devuelve la “y” (casi instantáneamente). Ya hemos eliminado un grado de libertad. Pero antes de seguir veamos más a fondo las **modificaciones** que contiene este binario parcheado (perdonadme si se me pasa alguna):

- **Main.** Anulamos el sleep y forzamos a que únicamente se cree un hilo principal.

<pre> 00402B80 > 8D46 30 LEA EAX,[ESI+30] 00402B83 . 6A 08 PUSH 8 00402B85 . 66:890475 50 MOV WORD PTR DS:[ESI*2+411B50],AX 00402B8D . E8 E4030000 CALL 00402F76 00402B92 . 83C4 04 ADD ESP,4 00402B95 . 3BC3 CMP EAX,EBX 00402B97 *v 74 6F JE SHORT 00402C08 00402B99 . 53 PUSH EBX 00402B9A . 53 PUSH EBX 00402B9E . 50 PUSH EAX 00402B9C . 68 302C4000 PUSH reto4_3.00402C30 00402BA1 . 53 PUSH EBX 00402BA2 . 53 PUSH EBX 00402BA3 . 8930 MOV DWORD PTR DS:[EAX],ESI 00402BA5 . 8958 04 MOV DWORD PTR DS:[EAX+4],EBX 00402BA8 . FF07 CALL EDI 00402BAA . 3BC3 CMP EAX,EBX 00402BAC . 8944B4 0C MOV DWORD PTR SS:[ESI*4+ESP+0C],EAX 00402BB0 *v 74 56 JE SHORT 00402C08 00402BB2 . 83C6 01 ADD ESI,1 00402BB5 * 83FE 01 CMP ESI,1 00402BB8 * ^ 7C C6 JCL SHORT 00402B80 00402BBA . 8B3D 30D0400 MOV EDI,DWORD PTR DS:[<&KERNEL32.Sleep> 00402BC0 . 66:891D 181C MOV WORD PTR DS:[411C18],BX 00402BC7 . 8B1D 34D0400 MOV EBX,DWORD PTR DS:[<&KERNEL32.WaitFor 00402BCD . 8D49 0A LEA ECX,[ECX] 00402BD0 > 6A 00 PUSH 0 00402BD2 . 90 NOP 00402BD3 . 90 NOP 00402BD4 . 90 NOP 00402BD5 . FF07 CALL EDI 00402BD7 . 68 E8030000 PUSH 3E8 00402BDC . 6A 01 PUSH 1 00402BDE . 8D4C24 14 LEA ECX,[ARG.5] 00402BE2 . 51 PUSH ECX 00402BE3 . 56 PUSH ESI 00402BE4 . FF03 CALL EBX 00402BE6 . 3D 02010000 CMP EAX,102 00402BEB * ^ 74 E3 JE SHORT 00402BD0 00402BED . 68 501B4100 PUSH OFFSET reto4_3.00411B50 00402BF2 . 68 6CE64000 PUSH OFFSET reto4_3.0040E66C 00402BF7 . E8 93060000 CALL 0040328F </pre>	<pre> [Arg1 = 8 reto4_3.00402F76 [Time = 0 KERNEL32.Sleep Timeout = 1000. ms WaitAll = TRUE HandleList Count KERNEL32.WaitForMultipleObjects UNICODE "La cadena oculta es: %s" </pre>
--	---

Sólo 1 hilo principal (en lugar de 10...)

No esperas

- **StartAddress.** Procedemos de forma similar a “main” lo que en este caso se traduce en anular el sleep y evitar la creación de hilos secundarios. Para esto último, nos saltaremos el bucle de creación de hilos. En definitiva, **nuestro programa parcheado únicamente generará un hilo** y por tanto devolverá sólo un carácter.

```

00402C94 > 8B36 MOV ESI,DWORD PTR DS:[ESI]
00402C96 > 83C6 0A ADD ESI,0A
00402C99 > 83FE 64 CMP ESI,64
00402C9C > 53 PUSH EBX
00402C9D > EB 56 JMP SHORT 00402CF5
00402C9F > 8B1D 3D004000 MOV EBX,DWORD PTR DS:[<&KERNEL32.Create
00402CA5 > EB 09 JMP SHORT 00402CB0
00402CA7 > 8DA424 000000 LEA ESP,[ESP]
00402CAE > 8BFF MOV EDI,EDI
00402CB0 > 8D56 30 LEA EDX,[ESI+30]
00402CB3 > 6A 08 PUSH 8
00402CB5 > 66:891475 50 MOV WORD PTR DS:[ESI*2+411B50],DX
00402CB8 > E8 B4020000 CALL 00402F76
00402CC2 > 83C4 04 ADD ESP,4
00402CC5 > 85C0 TEST EAX,EAX
00402CC7 > 74 60 JE SHORT 00402D29
00402CC9 > 6A 00 PUSH 0
00402CCB > 6A 00 PUSH 0
00402CCD > 50 PUSH EAX
00402CCE > 68 302C4000 PUSH reto4_3.00402C30
00402CD3 > 6A 00 PUSH 0
00402CD5 > 6A 00 PUSH 0
00402CD7 > 8930 MOV DWORD PTR DS:[EAX],ESI
00402CD9 > C740 04 0100 MOV DWORD PTR DS:[EAX+4],1
00402CE0 > FF03 CALL EBX
00402CE2 > 85C0 TEST EAX,EAX
00402CE4 > 8944BC 0C MOV DWORD PTR SS:[EDI*4+ESP+0C],EAX
00402CE8 > 74 3F JE SHORT 00402D29
00402CEA > 83C6 0A ADD ESI,0A
00402CED > 83C7 01 ADD EDI,1
00402CF0 > 83FE 64 CMP ESI,64
00402CF3 > 7C BB JL SHORT 00402CB0
00402CF5 > 8B35 3D004000 MOV ESI,DWORD PTR DS:[<&KERNEL32.Sleep>
00402CF8 > 8B1D 3D004000 MOV EBX,DWORD PTR DS:[<&KERNEL32.WaitFo
00402D01 > 6A 00 PUSH 0
00402D03 > 90 NOP
00402D04 > 90 NOP
00402D05 > 90 NOP
00402D06 > FF06 CALL ESI
00402D08 > 68 E8030000 PUSH 3E8
00402D0D > 6A 01 PUSH 1
00402D0F > 8D4424 14 LEA EAX,[ARG.5]
00402D13 > 50 PUSH EAX
00402D15 > 57 PUSH EDI
00402D17 > FF03 CALL EBX
00402D1C > 3D 02010000 CMP EAX,102
00402D1E > 74 E3 JE SHORT 00402D01
    
```

Se salta el bucle de creación de hilos secundarios (antes había un JGE aquí)

Arg1 = 8
reto4_3.00402F76

Time = 0

No esperas

KERNEL32.Sleep
Timeout = 1000. ms
WaitAll = TRUE

HandleList
Count
KERNEL32.WaitForMultipleObjects

- **Array de punteros a función.** Al lanzar únicamente un hilo sólo se va a utilizar el primer puntero. Lo modificamos para que se ejecute la función cuyo carácter deseamos obtener.

Address	Hex dump	ASCII
00410F50	20 27 40 00 20 26 40 00 20 27 40 00 20 28 40 00	'e. &e. 'e. (e.
00410F60	20 24 40 00 10 19 40 00 A0 19 40 00 B0 1A 40 00	*e. >e. &e. &e.
00410F70	A0 18 40 00 20 24 40 00 80 1B 40 00 B0 13 40 00	&e. &e. C+e. &e.
00410F80	50 1C 40 00 B0 13 40 00 10 1D 40 00 20 24 40 00	PLe. &e. &e. &e.
00410F90	C0 1D 40 00 A0 19 40 00 80 1B 40 00 80 1E 40 00	L+e. &e. C+e. C+e.
00410FA0	10 1D 40 00 10 1D 40 00 10 1D 40 00 10 1D 40 00	&e. &e. P+e. L+e.
00410FB0	10 12 40 00 10 12 40 00 10 12 40 00 10 12 40 00	&e. L+e. &e. C+e.
00410FC0	C0 1D 40 00 10 1D 40 00 10 1D 40 00 10 1D 40 00	&e. &e. 'e. (e.
00410FD0	B0 13 40 00 10 13 40 00 10 13 40 00 10 13 40 00	&e. &e. &e. &e.
00410FE0	80 1B 40 00 10 1B 40 00 10 1B 40 00 10 1B 40 00	C+e. &e. 'e. &e.
00410FF0	20 29 40 00 10 19 40 00 10 19 40 00 10 19 40 00)e. &e. &e. L+e.
00411000	20 25 40 00 10 15 40 00 10 15 40 00 10 15 40 00	&e. C+e. *e. &e.
00411010	10 19 40 00 10 19 40 00 10 19 40 00 10 19 40 00	&e. -+e. &e. 'e.
00411020	C0 1D 40 00 10 1D 40 00 10 1D 40 00 10 1D 40 00	L+e. &e. C+e. L+e.
00411030	20 26 40 00 10 16 40 00 10 16 40 00 10 16 40 00	&e. &e. L e. C+e.
00411040	B0 13 40 00 20 24 40 00 80 21 40 00 80 1B 40 00	&e. &e. se. e+e. C+e.
00411050	A0 18 40 00 A0 19 40 00 20 2A 40 00 10 12 40 00	&e. &e. *e. &e.
00411060	60 15 40 00 80 1B 40 00 60 22 40 00 20 24 40 00	'e. C+e. 'e. &e.
00411070	A0 19 40 00 10 12 40 00 60 15 40 00 20 24 40 00	&e. &e. 'e. &e.
00411080	10 12 40 00 A0 19 40 00 80 1B 40 00 A0 19 40 00	&e. &e. C+e. &e.
00411090	20 2A 40 00 60 15 40 00 10 12 40 00 B0 13 40 00	*e. 'e. &e. &e.
004110A0	A0 19 40 00 C0 1D 40 00 20 23 40 00 80 1B 40 00	&e. L+e. &e. C+e.
004110B0	80 1B 40 00 80 1B 40 00 80 1B 40 00 80 17 40 00	C+e. C+e. C+e. &e.
004110C0	B0 13 40 00 20 24 40 00 90 21 40 00 B0 13 40 00	&e. &e. e+e. &e.
004110D0	80 1B 40 00 80 1B 40 00 80 1B 40 00 80 1B 40 00	C+e. C+e. C+e. C+e.

Primer puntero del array de funciones: 402720

- **Función sustituto de GetSystemTimes.** He aquí el quid de la cuestión así que mucha atención... Esta función la escribí desde cero en ASM y “emula” a GetSystemTimes. Recibe tres parámetros (lpIdleTime, lpKernelTime, lpUserTime)⁴ y la primera vez que es invocada escribirá un 0 en todos ellos. La siguiente vez escribirá un 1, la siguiente un 2 y así

⁴ Son punteros a estructuras FILETIME.

sucesivamente⁵. De esta forma hemos implementado, como el que no quiere la cosa, un simple brute-forcer en ASM. Nótese que hemos necesitado una variable estática (para almacenar el valor actual –que iremos incrementando- y que perdure durante las sucesivas llamadas a la función) y que hemos utilizado una dirección de memoria conocida (411B80) pero que sabemos que en este momento ya no va a ser útil por lo que se puede machacar sin problemas. De la misma forma, la función la hemos ubicado en 401700, que era donde residía originalmente la función correspondiente al primer hilo original (el carácter “P”) y que en este caso sabemos que tampoco va a resultar útil.

00401700	55	PUSH EBP	reto4_3.00401700(guessed Arg1,Arg2,Arg3)
00401701	89E5	MOV EBP,ESP	
00401703	51	PUSH ECX	
00401704	53	PUSH EBX	
00401705	90	NOP	
00401706	90	NOP	
00401707	A1 801B4100	MOV EAX,DWORD PTR DS:[411B80]	Variable "estática"
0040170C	31C9	XOR ECX,ECX	
0040170E	8B5D 08	MOV EBX,DWORD PTR SS:[ARG.1]	
00401711	890B	MOV DWORD PTR DS:[EBX],ECX	
00401713	8943 04	MOV DWORD PTR DS:[EBX+4],EAX	
00401716	8B5D 0C	MOV EBX,DWORD PTR SS:[ARG.2]	
00401719	890B	MOV DWORD PTR DS:[EBX],ECX	
0040171B	8943 04	MOV DWORD PTR DS:[EBX+4],EAX	
0040171E	8B5D 10	MOV EBX,DWORD PTR SS:[ARG.3]	
00401721	890B	MOV DWORD PTR DS:[EBX],ECX	
00401723	8943 04	MOV DWORD PTR DS:[EBX+4],EAX	
00401726	40	INC EAX	
00401727	A3 801B4100	MOV DWORD PTR DS:[411B80],EAX	
0040172C	5B	POP EBX	
0040172D	59	POP ECX	
0040172E	C9	LEAVE	
0040172F	C2 0C00	RETN 0C	

Sustituto de
GetSystemTimes()

- 402720 (hilo que tratamos de “brute-forcear”). Lo hemos trucado para que llame a nuestra función sustituto de GetSystemTimes.

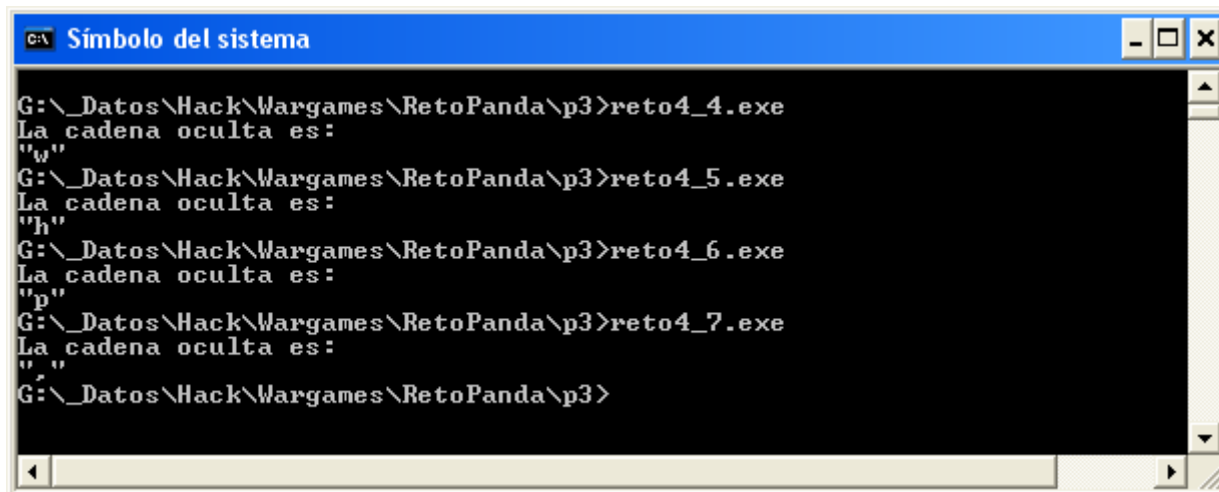
00402720	55	PUSH EBP	
00402721	8BEC	MOV EBP,ESP	
00402723	83E4 F8	AND ESP,FFFFFFF8	QWORD (8.-byte) stack alignment
00402726	81EC A0000000	SUB ESP,0A0	
0040272C	A1 28054100	MOV EAX,DWORD PTR DS:[410528]	
00402731	33C4	XOR EAX,ESP	
00402733	898424 9C0000	MOV DWORD PTR SS:[LOCAL.1],EAX	
0040273A	56	PUSH ESI	
0040273B	57	PUSH EDI	
0040273C	8D4424 30	LEA EAX,[LOCAL.30]	Arg3 => OFFSET LOCAL.30
00402740	50	PUSH EAX	
00402741	8D4C24 2C	LEA ECX,[LOCAL.32]	Arg2 => OFFSET LOCAL.32
00402745	51	PUSH ECX	
00402746	8D5424 28	LEA EDX,[LOCAL.34]	Arg1 => OFFSET LOCAL.34
0040274A	52	PUSH EDX	
0040274B	E8 B0EFFFFF	CALL 00401700	reto4_3.00401700
00402750	8B5D 30004000	MOV EDI,DWORD PTR DS:[\Kernel32.Sleep>	
00402756	33C0	XOR EAX,EAX	
00402758	894424 20	MOV DWORD PTR SS:[LOCAL.34],EAX	
0040275C	894424 28	MOV DWORD PTR SS:[LOCAL.32],EAX	
00402760	894424 30	MOV DWORD PTR SS:[LOCAL.30],EAX	
00402764	8D4424 18	LEA EAX,[LOCAL.36]	Arg3 => OFFSET LOCAL.36
00402768	50	PUSH EAX	
00402769	8D4C24 14	LEA ECX,[LOCAL.38]	Arg2 => OFFSET LOCAL.38
0040276D	51	PUSH ECX	
0040276E	8D5424 10	LEA EDX,[LOCAL.40]	Arg1 => OFFSET LOCAL.40
00402772	52	PUSH EDX	
00402773	E8 88EFFFFF	CALL 00401700	reto4_3.00401700
00402778	90	NOP	
00402779	90	NOP	
0040277A	90	NOP	
0040277B	90	NOP	
0040277C	90	NOP	
0040277D	90	NOP	
0040277E	90	NOP	
0040277F	8B4C24 0C	MOV ECX,DWORD PTR SS:[LOCAL.39]	
00402783	2B4C24 24	SUB ECX,DWORD PTR SS:[LOCAL.33]	
00402787	8B7424 14	MOV ESI,DWORD PTR SS:[LOCAL.37]	
0040278B	2B7424 2C	SUB ESI,DWORD PTR SS:[LOCAL.31]	
0040278F	B8 C0CCCCC0	MOV EAX,CCCCCCC0	
00402794	F7E1 ..	MUL ECX	

"Fake" GetSystemTimes()

⁵ La estructura FILETIME consta de dos variables de tipo DWORD: dwLowDateTime y dwHighDateTime. Realmente escribiremos cero siempre en la variable “baja” y nuestro valor (que iremos incrementando) irá a parar a la variable “alta”. Lo hacemos así porque las funciones de tipo 3, de forma análoga a las de tipo 2, resetean la parte menos significativa antes de obtener el hash (entendemos que con el propósito de hacer factible la fuerza bruta).

Interesante, ¿verdad? ☺

Si repetimos el mismo proceso con el resto de caracteres obtendremos 4 nuevos ejecutables parcheados cuyo resultado se muestra a continuación:



```
G:\_Datos\Hack\Wargames\RetoPanda\p3>reto4_4.exe
La cadena oculta es:
"w"
G:\_Datos\Hack\Wargames\RetoPanda\p3>reto4_5.exe
La cadena oculta es:
"h"
G:\_Datos\Hack\Wargames\RetoPanda\p3>reto4_6.exe
La cadena oculta es:
"p"
G:\_Datos\Hack\Wargames\RetoPanda\p3>reto4_7.exe
La cadena oculta es:
" "
G:\_Datos\Hack\Wargames\RetoPanda\p3>
```

Y por tanto, se comprueba que estos caracteres, correspondientes a las funciones de tipo 3, no contienen mayúscula alguna. En este caso, sí que nos podíamos haber ahorrado esta tediosa parte pero... ¿quién lo iba a saber? ☺

--[0x0a - Solución definitiva]

En cualquier caso, tras el arduo camino recorrido, en el que hemos ido despejando incógnitas – una a una- obtenemos nuestra recompensa. La frase secreta es:

```
"Protection aGainst viruses, spyware, Trojans, hackers,  
spam and other Internet threats.   Panda   "
```


--[0x0b - Feedback & Greetz]

Si te ha gustado este solucionario, por favor, ¡dímelo! Si has encontrado algún error, como no, también me gustaría saberlo. ¡ESCRÍBEME!

Greetz: !dSR, 48bits, SexyPandas, SbD, Maligno, juju666 y al staff de Panda Security por este entretenido concurso ☺

RoMaNSoFt

[<http://www.rs-labs.com/>]

-----[EOF]-----