

Analisi del Malware – Relazione Homework 3

Studente: Andrea Pepe Matricola: 0315903

OBIETTIVO

Analizzare il programma eseguibile “hw3.exe” utilizzando principalmente *Ghidra* e *OllyDbg* per determinare il codice di sblocco che rende funzionale il programma.

ANALISI DI BASE

Il file eseguibile oggetto dell’analisi è in formato Portable Executable (PE), dunque un eseguibile per sistemi operativi Windows. Come prima operazione di analisi statica di base, è stata effettuata una ricerca delle stringhe utilizzate dal programma, in particolare usando il comando *strings* su sistema operativo Linux. Si riporta di seguito il risultato ottenuto, eliminando le stringhe non significative:

```
!This program cannot be run in DOS mode.
.text
P`.data
P`.data
.rdata
0@.bss
.idata
.CRT
.tls
.rsrc
D$rTimef
D$v bef
D$zore
D$~Wind
ows
shut
down
D$@days
D$Ehour
D$Kminu
D$Otes
D$SUnlo
D$Wck c
D$[ode:
Escape from Windows v5 - M. Cesati, 2010, 2021
WARNING: there will be no shutdown without the proper unlock code!
 0 seconds
%2ld seconds
Stop
Sorry, try again!
Shutdown time has come!
However, the unlock code is wrong.
If you want the full version of this wonderful tool,
you can get the unlock code for just ten bucks!
Ask to the nearest teacher around you!
EDIT
CreateWindow failed
BUTTON
SeShutdownPrivilege
```

InternalError
%s=%lu/0x%lx
DEBUG %s:%d
InternalError
Unknown error
_matherr(): %s in %s(%g, %g) (retval=%g)
Argument domain error (DOMAIN)
Argument singularity (SIGN)
Overflow range error (OVERFLOW)
The result is too small to be represented (UNDERFLOW)
Total loss of significance (TLOSS)
Partial loss of significance (PLOSS)
Mingw-w64 runtime failure:
Address %p has no image-section
 VirtualQuery failed for %d bytes at address %p
 VirtualProtect failed with code 0x%x
 Unknown pseudo relocation protocol version %d.
 Unknown pseudo relocation bit size %d.
GCC: (tdm64-1) 9.2.0
AdjustTokenPrivileges
LookupPrivilegeValueA
OpenProcessToken
TextOutA
CloseHandle
CreateFileA
CreateFileMappingA
DeleteCriticalSection
EnterCriticalSection
ExitProcess
FatalAppExitA
GetCurrentProcess
GetCurrentProcessId
GetCurrentThreadId
GetFileInformationByHandle
GetLastError
GetModuleFileNameA
GetProcAddress
GetStartupInfoA
GetSystemTimeAsFileTime
GetTickCount
InitializeCriticalSection
IsDebuggerPresent
LeaveCriticalSection
LoadLibraryA
MapViewOfFile
QueryPerformanceCounter
SetUnhandledExceptionFilter
Sleep
TerminateProcess
TlsGetValue
UnhandledExceptionFilter
VirtualProtect
VirtualQuery
__getmainargs
__initenv
__lconv_init
__p__acmdln
__p__fmode
__set_app_type
__setusermatherr
_amsg_exit
_cexit
_initterm

```
_iob
_onexit
_vsnprintf
abort
calloc
exit
fprintf
free
fwrite
malloc
memcpy
signal
strlen
strncmp
vfprintf
BeginPaint
CreateWindowExA
DefWindowProcA
DispatchMessageA
DrawTextA
EndPaint
ExitWindowsEx
GetClientRect
GetDlgItemInt
GetDlgItemTextA
GetMessageA
GetWindowLongA
KillTimer
LoadCursorA
LoadIconA
MessageBoxA
MoveWindow
PostQuitMessage
RedrawWindow
RegisterClassExA
SendDlgItemMessageA
SetDlgItemInt
SetDlgItemTextA
SetTimer
SetWindowLongA
ShowWindow
TranslateMessage
ADVAPI32.dll
GDI32.dll
KERNEL32.dll
msvcrt.dll
USER32.dll
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="asInvoker"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
  <compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
    <application>
      <!--The ID below indicates application support for Windows Vista -->
      <supportedOS Id="{e2011457-1546-43c5-a5fe-008deee3d3f0}"/>
      <!--The ID below indicates application support for Windows 7 -->
      <supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/>
      <!--The ID below indicates application support for Windows 8 -->
```

```

    <supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}"/>
    <!--The ID below indicates application support for Windows 8.1 -->
    <supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}"/>
    <!--The ID below indicates application support for Windows 10 -->
    <supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}"/>
  </application>
</compatibility>
</assembly>

```

Dalla stringa “GDI32.dll” e dal nome di alcune API come, ad esempio, “GetMessageA” e “TranslateMessage” si desume che il programma abbia una GUI.

Inoltre, sono molto interessanti i nomi di alcune API che potrebbero essere state usate per ostacolare il reversing del codice del programma, adottando misure anti-debugging; in particolare: *GetTickCount*, *IsDebuggerPresent*, *LoadLibraryA*, *GetProcAddress*, *QueryPerformanceCounter*, *SetUnhandledExceptionFilter*, *UnhandledExceptionFilter*.

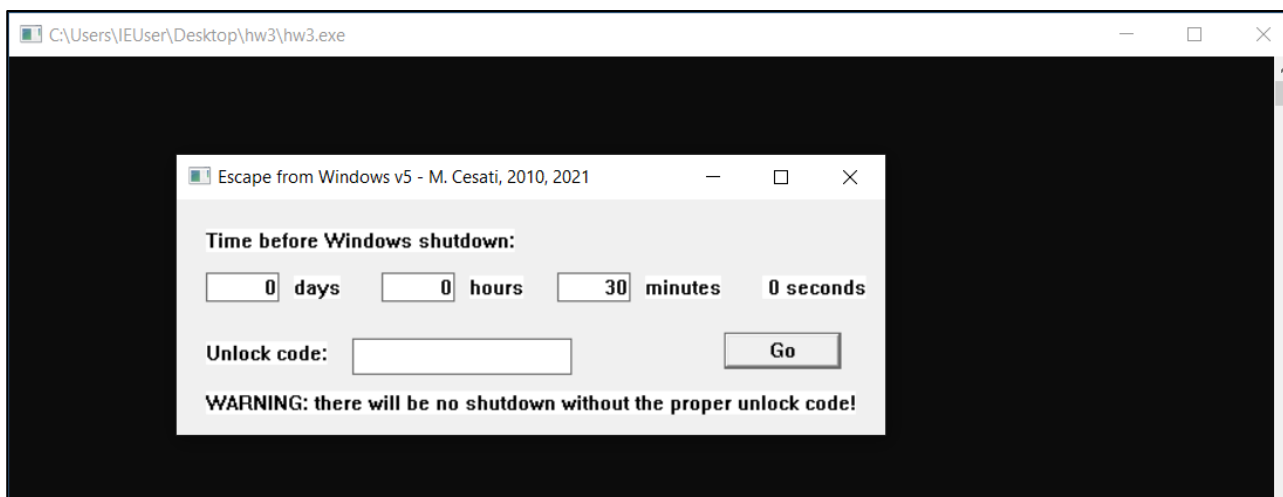
Infine, viene stampato tra le stringhe anche il contenuto di un file manifest in XML che descrive la compatibilità dell’e eseguibile con le diverse versioni dei sistemi operativi Windows: sono supportati Windows Vista, 7, 8, 8.1 e 10. Infatti, usando il software *Resource Hacker*, tale file manifest risulta essere presente nella sezione dedicata alle risorse del file eseguibile.

Si è passati ad analizzare le testate del file eseguibile tramite l’utilizzo di *PE-Bear*: dalle informazioni riguardanti i *SECTION_HEADERS* si è confrontata la raw size e la virtual size di ogni sezione. Sostanzialmente, i valori erano quasi sempre simili, fatta eccezione per la sezione *.tls*: infatti, come visibile dallo stesso *PE-Bear*, sono presenti nella TLS directory i riferimenti a 2 *tls callbacks*. Esse potrebbero essere usate per eseguire del codice prima che il debugger riesca a fermarsi sull’entry point del programma.

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
> .text	400	2A00	1000	2874	60500060	0	0	0
> data	2E00	400	4000	3B0	60500020	0	0	0
> .data	3200	200	5000	1CC	C0600040	0	0	0
> .rdata	3400	800	6000	704	40300040	0	0	0
> .bss	0	0	7000	4C8	C0600080	0	0	0
> .idata	3C00	C00	8000	A6C	C0300040	0	0	0
> .CRT	4800	200	9000	34	C0300040	0	0	0
> .tls	4A00	200	A000	8	C0300040	0	0	0
> .rsrc	4C00	600	B000	4E8	C0300040	0	0	0

Pe-Bear: analisi dei Section Hdrs

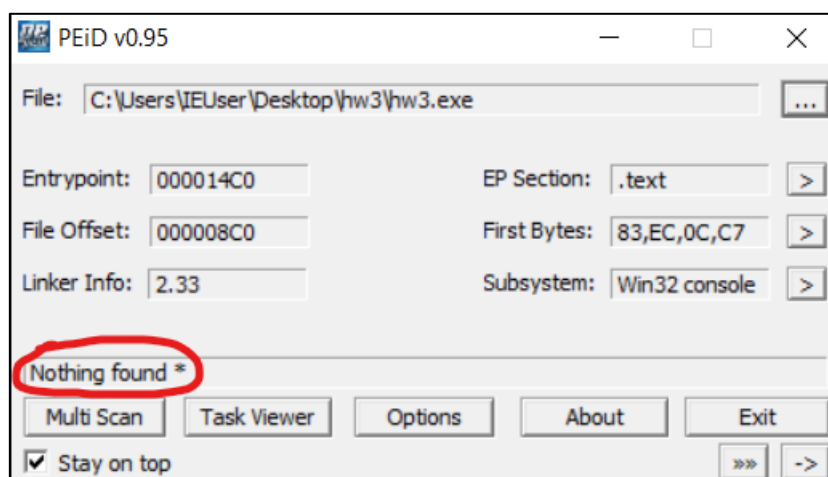
Si è proceduto con l’analisi dinamica di base e come prima cosa è stato lanciato l’e eseguibile su una macchina virtuale con sistema operativo Windows 10 per osservarne il comportamento: viene aperta una finestra per programmare lo shutdown di Windows; non inserendo alcun codice di sblocco, allo scadere del countdown, appare una message box con un messaggio d’errore e il programma termina senza effettuare lo spegnimento del sistema.



Esecuzione del programma

Analizzando il processo in esecuzione tramite *ProcessExplorer* e *ProcessMonitor*, si nota che vengono effettuate diverse operazioni sul registro di sistema, ma non sembrano significative. Utilizzando *RegShot*, per fare un confronto tra lo status del registro di sistema prima di lanciare l'eseguibile e quello dopo averlo lanciato, se ne ha la conferma.

Un'analisi tramite *PEiD*, conferma che l'eseguibile non è stato compresso o offuscato tramite l'uso di un packer, così come era desumibile anche dalle informazioni ottenute usando *PE-Bear*:



Analisi con PEiD: uso di packer non individuato

Provando a lanciare il programma all'interno di *OllyDbg*, si nota che la finestra per programmare lo spegnimento di Windows non viene mostrata e, anziché rimanere running eseguendo il message loop, il programma termina. È evidente che sono presenti delle misure anti-debugging. Ci si pone quindi come obiettivo quello di individuare tali misure tramite l'uso dell'analisi statica avanzata con *Ghidra* combinata con *OllyDbg*, in maniera tale da inibirle e poter così effettuare l'analisi dinamica avanzata con il debugger.

ANALISI AVANZATA

Cercando nella sezione degli import l'API `TranslateMessage()` e guardandone i riferimenti, è stato individuato il message loop e quindi la funzione *WinMain*:

```
2 WPARAM WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
3
4 {
5     ATOM AVar1;
6     app_data *lpParam;
7     HWND pHVar2;
8     BOOL BVar3;
9     MSG local_68;
10    WNDCLASSEX local_4c;
11
12    getFileInfo();
13    local_4c.lpszClassName = s_HW3_00405090;
14    local_4c.hInstance = hInstance;
15    local_4c.lpfnWndProc = WndProc;
16    local_4c.style = 8;
17    local_4c.cbSize = 0x30;
18    local_4c.hIcon = LoadIconA((HINSTANCE)0x0, (LPCSTR)0x7f00);
19    local_4c.hIconSm = LoadIconA((HINSTANCE)0x0, (LPCSTR)0x7f00);
20    local_4c.hCursor = LoadCursorA((HINSTANCE)0x0, (LPCSTR)0x7f00);
21    local_4c.lpszMenuName = (LPCSTR)0x0;
22    local_4c.cbClsExtra = 0;
23    local_4c.cbWndExtra = 0;
24    local_4c.hbrBackground = (HBRUSH)0x5;
25    AVar1 = RegisterClassExA(&local_4c);
26    if (AVar1 == 0) {
27        return 0;
28    }
29    lpParam = init_data(targetFun);
30    pHVar2 = CreateWindowExA(0, s_HW3_00405090, s_Escape_from_Windows_v5_-_M_Cesa_00405060, 0xcf0000,
31        -0x80000000, -0x80000000, 500, 200, (HWND)0x0, (HMENU)0x0, hInstance, lpParam);
32    showWindowIfNoDebugger(pHVar2, nShowCmd);
33    while( true ) {
34        BVar3 = GetMessageA((LPMSG)&local_68, (HWND)0x0, 0, 0);
35        if (BVar3 == 0) break;
36        TranslateMessage(&local_68);
37        DispatchMessageA(&local_68);
38    }
39    return local_68.wParam;
40 }
```

Decompilato della funzione WinMain() in cui figurano già informazioni individuate in un secondo momento

Ciò ha permesso di individuare facilmente la Window Procedure, ridenominata *WndProc*. Tuttavia, prima di inizializzare la struttura dati `WNDCLASSEX`, viene invocata una funzione, la quale è stata ridenominata *getFileInfo*. In questa funzione, tramite l'API *GetModuleFilenameA*, si ottiene il path del file eseguibile del processo corrente. Se tale path esiste, si ottiene un handle aperto al file tramite l'invocazione a *CreateFileA*, che viene usato come parametro nella chiamata a *GetFileInformationByHandle*. In questo modo si ottengono in una struttura dati informazioni sul file e, in particolare, viene salvata in una variabile globale ridenominata *nSizeFileLow* la dimensione in bytes del file. A questo punto, viene invocata l'API *CreateFileMappingA* per creare ed ottenere un handle ad un file mapping object per il file eseguibile. Infine, tramite la *MapViewOfFile*, viene mappato la vista del file mapping object nell'address space del processo chiamante e il valore di ritorno, che corrisponde all'indirizzo iniziale della vista mappata, viene salvato in una variabile globale, ridenominata *startin-gAddress*.

È riportato di seguito il decompilato della funzione appena descritta:

```

void getFileInfo(void)
{
    DWORD DVar1;
    HANDLE hFile;
    BOOL BVar2;
    _BY_HANDLE_FILE_INFORMATION file_info;
    CHAR local_110 [268];

    startingAddress = (LPVOID)0x0;
    nSizeFileLow = 0;
    DVar1 = GetModuleFileNameA((HMODULE)0x0,local_110,260);
    if (DVar1 != 0) {
        hFile = CreateFileA(local_110,GENERIC_READ,FILE_SHARE_READ,(LPSECURITY_ATTRIBUTES)0x0,
                           OPEN_EXISTING,FILE_FLAG_SEQUENTIAL_SCAN | FILE_ATTRIBUTE_NORMAL,(HANDLE)0x0)
        ;
        if (hFile != (HANDLE)INVALID_HANDLE_VALUE) {
            BVar2 = GetFileInformationByHandle(hFile,(LPBY_HANDLE_FILE_INFORMATION)&file_info);
            if ((BVar2 == 0) || (file_info.nFileSizeHigh != 0)) {
                CloseHandle(hFile);
                return;
            }
            nSizeFileLow = file_info.nFileSizeLow;
            hFile = CreateFileMappingA(hFile,(LPSECURITY_ATTRIBUTES)0x0,PAGE_READONLY | SEC_IMAGE,0,0,
                                       (LPCSTR)0x0);
            if (hFile != (HANDLE)0x0) {
                startingAddress = MapViewOfFile(hFile,FILE_MAP_READ,0,0,0);
            }
        }
    }
    return;
}

```

Decompilato della funzione getFileInfo()

In seguito, nella *WinMain*, viene invocata una funzione (*init_data*) che svolge il compito di inizializzare la struttura dati fondamentale contenente i dati dell'applicazione. Dunque, analizzandola, si individuano alcuni campi di tale struttura dati. Tuttavia, oltre a fare ciò, invoca un'altra funzione, la quale usa *LoadLibraryA* per caricare la dll *kernel32.dll* e *GetProcAddress* per ottenere l'indirizzo dell'API *OutputDebugStringA*. Tale indirizzo viene salvato in una variabile globale.

```

app_data * __cdecl init_data(undefined *param_1)
{
    global_app_data.current_time = 0;
    global_app_data.timer_clock = 1000;
    global_app_data.target_time = 1800;
    _vsnprintf(global_app_data.warning_msg,128,
               "WARNING: there will be no shutdown without the proper unlock code!");
    _vsnprintf(global_app_data.seconds,16," 0 seconds");
    global_app_data.isTimerRunning = 0;
    global_app_data.functionToBeCalled = param_1;
    OutputDebugStringA_addr = (undefined *)getOutputDebugStringA_addr();
    return &global_app_data;
}

```

Decompilato della funzione init_data()

L'analisi della funzione *getOutputDebugStringA_addr* è stata complicata dall'utilizzo di misure anti-disassembling; in particolare, è stata utilizzata la tecnica del disassemblaggio impossibile con il seguente schema:

EB	FF	C0	48
----	----	----	----

- EB FF → jmp -1; la prossima istruzione partirà dal byte FF;
- FF C0 → inc eax; incrementa l'accumulatore di 1;

- 48 → dec eax; decrementa l'accumulatore di 1.

Dopo aver opportunamente istruito Ghidra per facilitare il disassemblaggio della funzione, il de-compilato è il seguente:

```

6 void getOutputDebugStringA_addr(void)
7
8 {
9     HMODULE hModule;
10    CHAR local_2d [44];
11
12    local_2d[0] = 'k';
13    local_2d[4] = 'e';
14    local_2d[1] = 'e';
15    local_2d[2] = 'r';
16    local_2d[3] = 'n';
17    local_2d[5] = 'l';
18    local_2d[6] = '3';
19    local_2d[7] = '2';
20    local_2d[8] = '.';
21    local_2d[9] = 'd';
22    local_2d[11] = 'l';
23    local_2d[10] = 'l';
24    local_2d[12] = '\0';
25    hModule = LoadLibraryA(local_2d);
26    local_2d[0] = '0';
27    local_2d[9] = 'u';
28    local_2d[4] = 'u';
29    local_2d[1] = 'u';
30    local_2d[12] = 't';
31    local_2d[5] = 't';
32    local_2d[2] = 't';
33    local_2d[3] = 'p';
34    local_2d[6] = 'D';
35    local_2d[7] = 'e';
36    local_2d[8] = 'b';
37    local_2d[16] = 'g';
38    local_2d[10] = 'g';
39    local_2d[11] = 'S';
40    local_2d[13] = 'r';
41    local_2d[14] = 'i';
42    local_2d[15] = 'n';
43    local_2d[17] = 'A';
44    local_2d[18] = '\0';
45    GetProcAddress(hModule, local_2d);
46    return;
47 }

```

Decompilato della funzione getOutputDebugStringA_addr()

Procedendo nell'analisi della *WinMain*, si nota che, dopo aver creato la finestra, viene invocata la funzione ridenominata *showWindowIfNoDebugger*:

```

2 WPARAM __cdecl showWindowIfNoDebugger(HWND param_1,int param_2)
3
4 {
5     ATOM AVar1;
6     BOOL BVar2;
7     WPARAM WVar3;
8     app_data *lpParam;
9     HWND param_1_00;
10    WPARAM in_stack_ffffff88;
11    HINSTANCE pHStack20;
12    int iStack8;
13
14    BVar2 = IsDebuggerPresent();
15    if (BVar2 == 0) {
16        WVar3 = ShowWindow(param_1,param_2);
17        return WVar3;
18    }
19    ExitProcess(0);
20 }

```

Decompilato della funzione showWindowIfNoDebugger()

Tale funzione, invocando l'API *IsDebuggerPresent*, controlla se l'eseguibile è lanciato sotto il controllo di un debugger: se sì, termina il programma invocando *ExitProcess(0)*, altrimenti effettua la *ShowWindow* della finestra precedentemente creata. Risulta dunque ovvio che, per poter effettuare un'analisi tramite debugger, bisogna di certo inibire la chiamata ad *IsDebuggerPresent* oppure modificare il risultato che restituisce.

Prima di effettuare una patch dell'eseguibile, si decide di proseguire ulteriormente con l'analisi statica alla ricerca di eventuali ulteriori misure anti-debugger. Quindi, si passa ad analizzare la *WndProc*, di cui di seguito si riporta parte del decompilato.

```
LVar3 = GetWindowLongA(hWnd, GWL_USERDATA);
checkBeingDebugged((int *)&uMsg);
if (uMsg == WM_SIZE) {
    MoveWindow(*(HWND *) (LVar3 + 0xac), 0x14, 0x32, 0x32, 0x14, 0);
    MoveWindow(*(HWND *) (LVar3 + 0xb0), 0x8c, 0x32, 0x32, 0x14, 0);
    MoveWindow(*(HWND *) (LVar3 + 0xb4), 0x104, 0x32, 0x32, 0x14, 0);
    MoveWindow(*(HWND *) (LVar3 + 0xb8), (lParam & 0xffffU) - 0x6e, ((uint)lParam >> 0x10) - 0x46, 0x50,
        0x19, 0);
    MoveWindow(*(HWND *) (LVar3 + 0xbc), 0x78, 0x5f, 0x96, 0x19, 0);
    RedrawWindow(hWnd, (RECT *)0x0, (HRGN)0x0, 5);
    changeAsmCode(LVar3);
    return 0;
}
```

Operazioni iniziali della WndProc e gestione di WM_SIZE

Prima di discernere che tipo di messaggio è stato ricevuto dalla finestra, viene invocata una funzione, ridenominata in seguito *checkBeingDebugged*, che prende come parametro in input l'indirizzo di *uMsg*, ovvero il parametro della *WndProc* che identifica proprio il tipo di messaggio. Si riporta di seguito il disassemblato di tale funzione:

	undefined int *	undefined __cdecl AL:1 Stack[0x4]:4 checkBeingDebugged	checkBeingDebugged(int * uMsgPtr)	<RETURN> uMsgPtr	XREF[1]: 00401dc0(R) XREF[1]: WndProc:00401e1a(c)
00401dc0	8b 44 24 04	MOV	EAX, dword ptr [ESP + uMsgPtr]		
00401dc4	64 8b 15	MOV	EDX, dword ptr FS:[0x30]		
	30 00 00 00				
00401dcb	8b 52 02	MOV	EDX, dword ptr [EDX + 0x2]		
00401dce	83 e2 07	AND	EDX, 0x7		
00401dd1	0f 95 c2	SETNZ	DL		
00401dd4	0f b6 d2	MOVZX	EDX, DL		
00401dd7	01 10	ADD	dword ptr [EAX], EDX		
00401dd9	c3	RET			

Disassemblato di checkBeingDebugged()

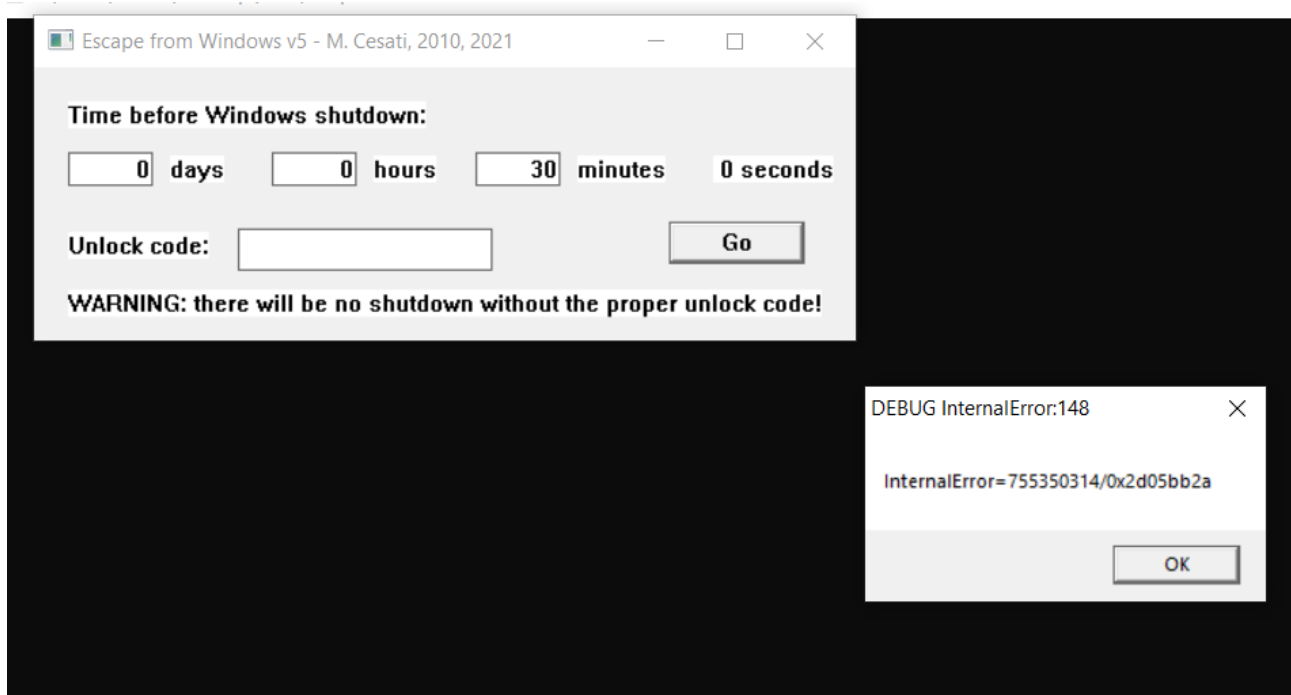
Con FS: [0x30], si accede alla struttura dati PEB (Process Environment Block); dopodiché, si prende il valore del campo presente ad offset 2 di tale struttura, ovvero il valore del campo *BeingDebugged*. Tale campo è pari a 1 se il processo è eseguito sotto il controllo di un debugger, 0 altrimenti. Quindi, nel caso in cui il processo sia debuggato, la funzione *checkBeingDebugged* modifica il valore di *uMsg* aggiungendogli 1 e cambiando così il comportamento dell'applicazione. Quindi, anche la chiamata a questa funzione va inibita per poter analizzare correttamente l'eseguibile con il debugger.

PRIMA PATCH

Come prima patch effettuata all'eseguibile, si inibisce il controllo effettuato invocando *IsDebuggerPresent* sostituendo la CALL alla API con l'istruzione XOR EAX, EAX. In questo modo, si azzerava il registro in cui veniva salvato il valore di ritorno di *IsDebuggerPresent* e la *showWindowIfNoDebugger* effettuerà sempre la *showWindow*, senza mai chiamare *ExitProcess*.

Inoltre, viene negata nella *WndProc* la chiamata alla funzione *checkBeingDebugged* semplicemente sostituendo i bytes dell'istruzione di CALL con delle NOP.

Provando a lanciare l'eseguibile patchato su macchina virtuale (senza utilizzo del Debugger), si nota che la finestra dell'applicazione appare normalmente e sembra funzionante; tuttavia, appare anche una message box che segnala un errore e dopo pochi secondi il processo viene terminato.



Message box segnalante un errore interno

Inoltre, provando a lanciare l'eseguibile patchato con *OllyDbg*, si nota che la finestra dell'applicazione appare, ma non è disegnata correttamente e il debugger va in crash terminando l'esecuzione. È molto probabile che ciò sia dovuto a qualche altra misura anti-debugger presente; d'altronde, avendo scoperto precedentemente che il programma usa *LoadLibraryA* e *GetProcAddress* per ottenere l'indirizzo di *OutputDebugStringA*, ci si aspetta che tale API venga usata da qualche parte per ostacolare l'analisi tramite debugger.

Seguendo l'esecuzione del programma a piccoli passi, impostando opportuni breakpoints, ci si accorge che il debugger va in crash non appena viene invocata la *ShowWindow* all'interno della funzione *showWindowIfNoDebugger*. Da ciò si desume che il problema possa essere presente all'interno della *WndProc* e, in particolare, nella gestione dei messaggi WM_SIZE e/o WM_CREATE, ricevuti dalla finestra proprio quando si cerca di mostrarla.

Si è analizzata prima la gestione di WM_SIZE, in quanto più breve e veloce da analizzare. L'unica funzione sospetta poteva essere la FUN_00404000, successivamente ridenominata *changeAsmCode*.

undefined		changeAsmCode(undefined4 param_1)	
undefined	AL:1	<RETURN>	
undefined4	Stack[0x4]:4	param_1	
			XREF[2]: 00404000(R), 0040402e(W)
			XREF[2]: 004001ac(*), WndProc:004020f0(c)
changeAsmCode			
00404000	8b 4c 24 04	MOV	ECX,dword ptr [ESP + param_1]
00404004	31 d2	XOR	EDX,EDX
00404006	8d b4 26	LEA	ESI,[ESI]
	00 00 00 00		
0040400d	8d 76 00	LEA	ESI,[ESI]
LAB_00404010			
00404010	8b 04 95	MOV	EAX,dword ptr [EDX*0x4 + callOutputDebugStringA] = 64B87A2Bh
	20 50 40 00		
00404017	35 2b fa	XOR	EAX,0x89a3fa2b
	a3 89		
0040401c	c1 c8 09	ROR	EAX,0x9
0040401f	89 04 95	MOV	dword ptr [EDX*0x4 + callOutputDebugStringA],EAX = 64B87A2Bh
	20 50 40 00		
00404026	83 c2 01	ADD	EDX,0x1
00404029	83 fa 0e	CMP	EDX,0xe
0040402c	75 e2	JNZ	LAB_00404010
0040402e	89 4c 24 04	MOV	dword ptr [ESP + param_1],ECX
00404032	e9 e9 0f	JMP	callOutputDebugStringA
	00 00		

Disassemblato della funzione changeAsmCode(): deoffusca delle istruzioni e alla fine le esegue effettuando una jump alla label ridenominata callOutputDebugStringA

Questa funzione effettua un loop in cui, attraverso operazioni di XOR e di ROR, cambia i bytes a partire da una certa label (in seguito ridenominata “callOutputDebugStringA”) in poi. Dopodiché, effettua una jump a tale label. Per scoprire che cosa realmente fa questa funzione, se ne segue il flusso con il debugger e il codice deoffuscato ed eseguito risulta essere il seguente, in cui è evidente una chiamata ad *OutputDebugStringA*:

00405020	83EC 4C	SUB ESP,4C	
00405023	31C0	XOR EAX,EAX	
00405025	8D76 00	LEA ESI,DWORD PTR DS:[ESI]	
00405028	C64404 1F 25	MOV BYTE PTR SS:[ESP+EAX+1F],25	
0040502D	C64404 20 73	MOV BYTE PTR SS:[ESP+EAX+20],73	
00405032	83C0 02	ADD EAX,2	
00405035	83F8 20	CMP EAX,20	
00405038	75 EE	JNZ SHORT hw3p1.00405028	
0040503A	8D4424 1F	LEA EAX,DWORD PTR SS:[ESP+1F]	
0040503E	C64424 3F 00	MOV BYTE PTR SS:[ESP+3F],0	
00405043	890424	MOV DWORD PTR SS:[ESP],EAX	
00405046	8B4424 50	MOV EAX,DWORD PTR SS:[ESP+50]	
0040504A	FF90 CC000000	CALL DWORD PTR DS:[EAX+CC]	KERNEL32.OutputDebugStringA
00405050	83EC 04	SUB ESP,4	
00405053	83C4 4C	ADD ESP,4C	
00405056	C3	RETN	
00405057	90	NOP	
00405058	0000	ADD BYTE PTR DS:[EAX],AL	
0040505A	0000	ADD BYTE PTR DS:[EAX],AL	
0040505C	0000	ADD BYTE PTR DS:[EAX],AL	
0040505E	0000	ADD BYTE PTR DS:[EAX],AL	
00405060	45	INC EBP	
00405061	73 63	JNB SHORT hw3p1.004050C6	
00405063	61	POPAD	
00405064	70 65	J0 SHORT hw3p1.004050C8	
00405066	2066 72	AND BYTE PTR DS:[ESI+72],AH	
00405069	6F	OUTS DX,DWORD PTR ES:[EDI]	I/O command
0040506A	6D	INS DWORD PTR ES:[EDI],DX	I/O command
0040506B	2057 69	AND BYTE PTR DS:[EDI+69],DL	
0040506E	6E	OUTS DX,BYTE PTR ES:[EDI]	I/O command
0040506F	64:6F	OUTS DX,DWORD PTR ES:[EDI]	I/O command
00405071	77 73	JA SHORT hw3p1.004050E6	
00405073	2076 35	AND BYTE PTR DS:[ESI+35],DH	
00405076	202D 204D2E20	AND BYTE PTR DS:[202E4D20],CH	
0040507C	43	INC EBX	
DS:[004070EC]=770F9DA0 (KERNEL32.OutputDebugStringA), JMP to KERNELBA.OutputDebugStringA			

Chiamata ad *OutputDebugStringA*("%%s%%s%%s%%s%%s%%s%%s%%s%%s")

Il parametro passato alla *OutputDebugStringA* è la stringa “%%s%%s%%s%%s%%s%%s%%s%%s%%s”, costruita tramite un loop scrivendo i bytes 0x25 (‘%’) e 0x73(‘s’).

Tale chiamata è quella che fa andare in crash il debugger e va quindi patchata.

SECONDA PATCH

Per evitare che venga chiamata *OutputDebugStringA*, si decide di sostituire la call a *changeAsmCode* all'interno della gestione di WM_SIZE nella *WndProc* semplicemente con delle NOP.

Tuttavia, lanciando il programma con la seconda patch senza l'uso del debugger, compare sempre la message box che segnala un internal error e il programma termina dopo pochi secondi. Lanciando l'eseguibile sotto il controllo del debugger, questa volta la finestra per programmare lo shutdown di windows appare disegnata correttamente; tuttavia, anche in questo caso viene mostrata la message box con il messaggio d'errore e, provando ad eseguire senza breakpoint il programma, esso non rimane attivo come ci si aspetterebbe (dovrebbe eseguire il message loop), bensì raggiunge lo stato di terminato.

È evidente che ci sia qualche ulteriore controllo che termini il processo. Inoltre, eseguendo tramite debugger, la terminazione del processo non sembra essere effettuata dal main thread, in quanto si entra nel message loop, ma improvvisamente, dopo un po' di tempo, il flusso si interrompe e viene terminato il programma.

Dunque si continua ad analizzare la *WndProc* staticamente, concentrandosi sulla gestione del messaggio WM_CREATE.

```
if (uMsg == WM_CREATE) {
    hMenu = (HMENU)0x1;
    hInstance = (HINSTANCE)GetWindowLongA(hWnd, GWL_HINSTANCE);
    SetWindowLongA(hWnd, -0x15, *(LONG *)lParam);
    iVar1 = *(int *)lParam;
    *(HWND *) (iVar1 + 0xa8) = hWnd;
    do {
        if ((int)hMenu < 4) {
            pHVar4 = CreateWindowExA(0, "EDIT", (LPCSTR)0x0, 0x50802002, 0, 0, 0, 0, hWnd, hMenu, hInstance,
                                     (LPVOID)0x0);
            *(HWND *) (iVar1 + 0xa8 + (int)hMenu * 4) = pHVar4;
            if (pHVar4 == (HWND)0x0) goto LAB_00401f86;
        }
        else {
            pHVar4 = CreateWindowExA(0, "EDIT", (LPCSTR)0x0, 0x50800020, 0, 0, 0, 0, hWnd, (HMENU)0x5, hInstance,
                                     (LPVOID)0x0);
            *(HWND *) (iVar1 + 0xbc) = pHVar4;
            pHVar4 = CreateWindowExA(0, "BUTTON", "Go", 0x50000001, 0, 0, 0, 0, hWnd, (HMENU)0x4, hInstance,
                                     (LPVOID)0x0);
            *(HWND *) (iVar1 + 0xb8) = pHVar4;
            if (pHVar4 != (HWND)0x0) {
                CalculateTime();
                funSetTimer(hWnd);
                CalculateFingerprint();
                return 0;
            }
        }
LAB_00401f86:
        FatalAppExitA(0, "CreateWindow failed");
    }
    hMenu = (HMENU)((int)&hMenu->unused + 1);
} while( true );
```

Decompilato della WndProc(); gestione del messaggio WM_CREATE

Ci sono 3 funzioni degne di nota:

- *CalculateTime()*: è una funzione chiamata per calcolare il tempo dello shutdown e aggiornare i valori delle sottofinestre dei giorni, ore e minuti. Non presenta nulla di strano.

- *funSetTimer()*
- *CalculateFingerprint()*

La *funSetTimer* permette di individuare un ulteriore campo della struct contenente i dati usati dall'applicazione, ovvero il timerID del timer associato alla finestra dell'app. Inoltre, permette di individuare la *TimerProc*.

```
void __cdecl funSetTimer(HWND hWnd)
{
    global_app_data.timerID = SetTimer(hWnd,0,global_app_data.timer_clock,TimerProc);
    return;
}
```

Decompilato della funzione funSetTimer()

Siccome, per quanto descritto prima, si sospetta che la terminazione del programma sia opera di un thread diverso dal thread main, il thread della *TimerProc* sembra essere un buon candidato.

```
void TimerProc(HWND param_1)
{
    uint uVar1;
    int iVar2;

    iVar2 = global_app_data.current_time;
    uVar1 = global_app_data.current_time + 1;
    global_app_data.current_time = uVar1;
    if (global_app_data.isTimerRunning != 0) {
        CalculateTime();
    }
    RedrawWindow(param_1,(RECT *)0x0,(HRGN)0x0,5);
    if ((global_app_data.isTimerRunning != 0) && ((uint)global_app_data.target_time <= uVar1)) {
        (*(code *)global_app_data.functionToBeCalled)(&global_app_data);
    }
    if ((iVar2 & 7U) == 0) {
        errorWindowAndExitProcess(&global_app_data);
        return;
    }
    return;
}
```

Decompilato della TimerProc()

Analizzando la *TimerProc*, si individuano due informazioni interessanti: in primis, viene individuato il campo della struct globale in cui è salvato l'indirizzo della funzione da invocare al momento dello scadere del countdown programmato per lo spegnimento di Windows (campo ridenominato "functionToBeCalled"). Tornando indietro alla funzione *init_data* e alla *WinMain*, se ne individua il valore assegnato e si ridenomina la funzione come "*targetFun*"; tuttavia, il suo disassemblaggio sembra essere stato ostacolato da delle tecniche anti-disassembling e si decide di focalizzarsi su questa funzione in seguito, e di dare priorità all'inibizione delle tecniche anti-debugger per poter così effettuare analisi dinamica avanzata.

La seconda informazione che si desume dall'analisi della *TimerProc* è una chiamata ad una funzione, ovvero la FUN_004042A0 (successivamente ridenominata *errorWindowAndExitProcess*). Entrando con *Ghidra* nel codice di tale funzione, si rileva che il disassemblaggio è errato a causa di una tecnica anti-disassembler, in particolare la tecnica dell'*always-true condition*: sono presenti dei jump

condizionali sempre presi, ma il disassembler, analizzando anche il caso in cui non venissero presi, riconosce delle istruzioni di tipo CALLF che depistano il disassemblaggio. Ripulendo il codice disassemblato, disassemblandolo nuovamente in maniera opportuna e sostituendo i bytes mai eseguiti che portavano ad indentificare le CALLF con delle NOP, si ottiene una disassemblato pulito della funzione e, quindi, anche un decompilato:

```
void __cdecl errorWindowAndExitProcess(app_data *param_1)
{
    int iVar1;
    uint extraout_EAX;
    dword dVar2;
    uint extraout_EDX;
    uint uVar3;
    bool bVar4;
    UINT in_stack_ffffff0;
    char acStack268 [128];
    char acStack140 [132];

    dVar2 = param_1->fingerprint;
    if (dVar2 != correct_fingerprint) {
        iVar1 = DAT_00407100 + 1;
        bVar4 = DAT_00407100 == 0;
        uVar3 = correct_fingerprint;
        DAT_00407100 = iVar1;
        if (bVar4) goto LAB_0040432c;
        do {
            ExitProcess(in_stack_ffffff0);
            dVar2 = extraout_EAX;
            uVar3 = extraout_EDX;
LAB_0040432c:
            _vsprintf(acStack268,0x80,"%s=%lu/0x%lx","InternalError",dVar2 ^ uVar3,dVar2 ^ uVar3);
            _vsprintf(acStack140,0x80,"DEBUG %s:%d","InternalError",0x94);
            in_stack_ffffff0 = 0x4043a6;
            MessageBoxA((HWND)0x0,acStack268,acStack140,0);
        } while( true );
    }
    return;
}
```

Decompilato della funzione errorWindowAndExitProcess(); il campo fingerprint della struct e la variabile globale correct_fingerprint sono stati individuati in un momento successivo

Si nota che, se la condizione di un if-statement risulta essere vera, si eseguono delle operazioni, tra cui la creazione della message box che riporta l'internal error vista in precedenza e viene invocata la *ExitProcess* che termina il processo.

Bisogna effettuare una ulteriore patch che neghi l'invocazione della *errorWindowAndExitProcess* nella *TimerProc*.

TERZA PATCH

Si sostituiscono i 5 bytes nella *TimerProc* corrispondenti alla CALL a *errorWindowAndExitProcess* con 5 NOP.

Lanciando ora l'eseguibile patchato, non appare più la message box segnalante l'internal error e il programma non termina automaticamente dopo pochi secondi. Anche l'esecuzione del programma controllata dal debugger sembra non presentare problemi: il programma rimane running una volta lanciato e si riesce ad interagire con la finestra dell'applicazione. Si può dunque procedere con l'analisi dinamica avanzata per cercare di scovare il codice di sblocco che rende funzionale il programma.

Dunque, partendo dalla *WinMain*, si segue su *Ghidra* la *targetFun*, ovvero la funzione chiamata quando scade il timeout associato alla finestra dell'applicazione.

undefined	undefined	undefined FUN_004040e0(undefined4 param_1)	
undefined4	AL:1	<RETURN>	
	Stack[0x4]:4	param_1	
	FUN_004040e0		XREF[1]: 004040ec (R)
			XREF[1]: WinMain:004025b8(*)
004040e0	57	PUSH	EDI
004040e1	56	PUSH	ESI
004040e2	53	PUSH	EBX
004040e3	83 ec 60	SUB	ESP,0x60
004040e6	8b 15 04	MOV	EDX,dword ptr [DAT_00407104]
	71 40 00		= ??
004040ec	8b 44 24 70	MOV	EAX,dword ptr [ESP + param_1]
004040f0	8b b8 a8	MOV	EDI,dword ptr [EAX + 0xa8]
	00 00 00		
004040f6	85 d2	TEST	EDX,EDX
004040f8	74 02	JZ	LAB_004040fa+2
	LAB_004040fa+2		XREF[0,1]: 004040f8(j)
004040fa	9a 42 c7	CALLF	0x0:SUB_1040c742
	40 10 00 00		
00404101	00 00	ADD	byte ptr [EAX],AL
00404103	a1 04 71	MOV	EAX,[DAT_00407104]
	40 00		= ??
00404108	85 c0	TEST	EAX,EAX
0040410a	74 02	JZ	LAB_0040410c+2
	LAB_0040410c+2		XREF[0,1]: 0040410a(j)
0040410c	9a 42 ff	CALLF	0x4082:SUB_1815ff42
	15 18 82 40		
00404113	00 8d 54	ADD	byte ptr [EBP + 0xc72c2454],CL
	24 2c c7		
00404119	44	INC	ESP
0040411a	24 04	AND	AL,0x4
0040411c	28 00	SUB	byte ptr [EAX],AL
0040411e	00 00	ADD	byte ptr [EAX],AL
00404120	89 54 24 08	MOV	dword ptr [ESP + 0x8],EDX
00404124	89 04 24	MOV	dword ptr [ESP],EAX
00404127	ff 15 ec	CALL	dword ptr [->ADVAPI32.DLL::OpenProcessToken]
	81 40 00		
0040412d	8b 1d 24	MOV	EBX,dword ptr [->USER32.DLL::PostQuitMessage]
	83 40 00		= 00008822
00404133	83 ec 0c	SUB	ESP,0xc
00404136	85 c0	TEST	EAX,EAX
00404138	0f 84 42	JZ	LAB_00404280

Disassemblato della *targetFun()*: è stata usata la tecnica anti-disassembling dell'*always-true condition*, infatti *DAT_00407104* è sempre pari a 0

Ripulendo il codice mal disassemblato da *Ghidra* e rieffettuandone il disassembling correttamente in maniera guidata, si riesce ad analizzare anche staticamente la *targetFun*. Si nota che sono presenti chiamate ad API quali *GetCurrentProcess*, *OpenProcessToken*, *LookupPrivilegeValueA* e *AdjustTokenPrivileges* che sostanzialmente hanno lo scopo di prepararsi ad effettuare lo shutdown del sistema, in particolare si ottiene un access token con i privilegi necessari per effettuare lo spegnimento del sistema.

Successivamente, invocando l'API *GetDlgItemTextA*, viene salvato in un buffer l'input che l'utente ha inserito nella edit box del codice di sblocco necessario per far funzionare correttamente l'applicazione; viene inoltre salvata in una variabile anche la lunghezza di tale input.

Dopodiché, viene eseguito un loop in cui, ancora una volta, vengono effettuate delle operazioni di XOR e ROR di bytes a partire da un indirizzo di una funzione, chiamata *FUN_00405c0*. Terminato il ciclo viene invocata tale funzione, passandole 3 parametri: la stringa inserita in input dall'utente, la lunghezza della stringa e l'indirizzo di una label. Seguendo la label, forzando *Ghidra* a disassemblare, si scopre che si tratta di una funzione in cui, ancora una volta, sono state inserite delle misure anti-

disassemblaggio con la tecnica dell'*always-true condition*. Ripulendo il codice, si scopre che tale funzione non fa altro che effettuare lo shutdown del sistema operativo invocando l'API *ExitWindowsEx* con parametri 5 (EWX_SHUTDOWN | EWX_FORCE) e 0 (nessuna motivazione indicata per lo shutdown). Dunque, tale funzione viene ridenominata *shutdown*.

```

2 void __cdecl targetFun(app_data *param_1)
3
4 {
5     HWND hDlg;
6     HANDLE ProcessHandle;
7     BOOL BVar1;
8     DWORD DVar2;
9     UINT inputLen;
10    int iVar3;
11    HANDLE local_40;
12    _TOKEN_PRIVILEGES local_3c;
13    CHAR userInput [30];
14
15    hDlg = param_1->handles[0];
16    param_1->isTimerRunning = 0;
17    ProcessHandle = GetCurrentProcess();
18    BVar1 = OpenProcessToken(ProcessHandle,0x28,&local_40);
19    if (BVar1 == 0) {
20        PostQuitMessage(0);
21    }
22    LookupPrivilegeValueA((LPCSTR)0x0,"SeShutdownPrivilege",(PLUID)local_3c.Privileges);
23    local_3c.PrivilegeCount = 1;
24    local_3c.Privileges[0].Attributes = 2;
25    AdjustTokenPrivileges
26        (local_40,0,(PTOKEN_PRIVILEGES)&local_3c,0,(PTOKEN_PRIVILEGES)0x0,(PDWORD)0x0);
27    DVar2 = GetLastError();
28    if (DVar2 == 0) {
29        inputLen = GetDlgItemTextA(hDlg,5,userInput,0x1e);
30        iVar3 = 0;
31        do {
32            *(uint *)(FUN_004050c0 + iVar3 * 4) =
33                (*(uint *)(FUN_004050c0 + iVar3 * 4) ^ 0x89a3fa2b) >> 9 |
34                (*(uint *)(FUN_004050c0 + iVar3 * 4) ^ 0x89a3fa2b) << 0x17;
35            iVar3 = iVar3 + 1;
36        } while (iVar3 != 0x34);
37        FUN_004050c0(userInput,inputLen,shutdown);
38    }
39    WrongUnlockCode();
40    PostQuitMessage(0);
41    return;
42 }

```

Decompilato della funzione *targetFun()*; nella figura è evidenziato il punto in cui viene effettuato il deoffuscamento del codice della funzione *FUN_004050c0* e la sua successiva invocazione

```

void shutdown(void)
{
    ExitWindowsEx(5,0);
    Sleep(0xffffffff);
    return;
}

```

Decompilato della funzione *shutdown()*

INDIVIDUAZIONE DEL CODICE DI SBLOCCO

Dunque, per scoprire il codice di sblocco, non rimane che seguire con *OllyDbg* il codice della `FUN_004050c0` una volta che è stato ricostruito correttamente. Si setta un breakpoint nella *Timer-Proc*, al punto in cui viene invocata la *targetFun*, per seguirne l'esecuzione. Finita la procedura di ricostruzione del codice della `FUN_004050c0`, quando tale funzione viene invocata, si fa uno *step into* per riuscire a vederne il codice:

004050c0	83EC 1C	SUB ESP,1C
004050c3	A1 E0A04000	MOV EAX,DWORD PTR DS:[40A0E0]
004050c8	8B5424 20	MOV EDX,DWORD PTR SS:[ESP+20]
004050cc	85C0	TEST EAX,EAX
004050ce	74 02	JE SHORT hw3p3.004050d2
004050d0	E8 63C70424	CALL 24451838
004050d5	3F	AA
004050d6	282F	SUB BYTE PTR DS:[EDI],CH
004050d8	A5	MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
004050d9	C74424 04 5D4731	MOV DWORD PTR SS:[ESP+4],4F3D475D
004050e1	C74424 08 3F0000	MOV DWORD PTR SS:[ESP+8],3F
004050e9	A1 E0A04000	MOV EAX,DWORD PTR DS:[40A0E0]
004050ee	85C0	TEST EAX,EAX
004050f0	74 02	JE SHORT hw3p3.004050f4
004050f2	E8 63837C24	CALL 248CD45A
004050f7	24 09	AND AL,9
004050f9	74 04	JE SHORT hw3p3.004050ff
004050fb	83C4 1C	ADD ESP,1C
004050fe	C3	RET
004050ff	0FB60424	MOVZX EAX,BYTE PTR SS:[ESP]
00405103	3202	XOR AL,BYTE PTR DS:[EDX]
00405105	3C 0C	CMP AL,0C
00405107	75 F2	JNZ SHORT hw3p3.004050fb
00405109	0FB64424 01	MOVZX EAX,BYTE PTR SS:[ESP+1]
0040510e	3242 01	XOR AL,BYTE PTR DS:[EDX+1]
00405111	3C 5A	CMP AL,5A
00405113	75 E6	JNZ SHORT hw3p3.004050fb
00405115	0FB64424 02	MOVZX EAX,BYTE PTR SS:[ESP+2]
0040511a	3242 02	XOR AL,BYTE PTR DS:[EDX+2]
0040511d	3C 61	CMP AL,61
0040511f	75 DA	JNZ SHORT hw3p3.004050fb
00405121	0FB64424 03	MOVZX EAX,BYTE PTR SS:[ESP+3]
00405126	3242 03	XOR AL,BYTE PTR DS:[EDX+3]
00405129	3C C0	CMP AL,C0
0040512b	75 CE	JNZ SHORT hw3p3.004050fb
0040512d	0FB64424 04	MOVZX EAX,BYTE PTR SS:[ESP+4]
00405132	3242 04	XOR AL,BYTE PTR DS:[EDX+4]
00405135	3C 2E	CMP AL,2E
00405137	75 C2	JNZ SHORT hw3p3.004050fb
00405139	0FB64424 05	MOVZX EAX,BYTE PTR SS:[ESP+5]
0040513e	3242 05	XOR AL,BYTE PTR DS:[EDX+5]
00405141	3C 13	CMP AL,13
00405143	75 B6	JNZ SHORT hw3p3.004050fb
00405145	0FB64424 06	MOVZX EAX,BYTE PTR SS:[ESP+6]
0040514a	3242 06	XOR AL,BYTE PTR DS:[EDX+6]
0040514d	3C 00	CMP AL,00
0040514f	75 AA	JNZ SHORT hw3p3.004050fb
00405151	0FB64424 07	MOVZX EAX,BYTE PTR SS:[ESP+7]
00405156	3242 07	XOR AL,BYTE PTR DS:[EDX+7]
00405159	3C 70	CMP AL,70
0040515b	75 9E	JNZ SHORT hw3p3.004050fb
0040515d	0FB64424 08	MOVZX EAX,BYTE PTR SS:[ESP+8]
00405162	3242 08	XOR AL,BYTE PTR DS:[EDX+8]
00405165	3C 1E	CMP AL,1E
00405167	75 92	JNZ SHORT hw3p3.004050fb
00405169	A1 E0A04000	MOV EAX,DWORD PTR DS:[40A0E0]
0040516e	85C0	TEST EAX,EAX
00405170	74 02	JE SHORT hw3p3.00405174
00405172	E8 63FF5424	CALL 249550DA
00405177	28A1 E0A04000	SUB BYTE PTR DS:[ECX+40A0E0],AH
0040517d	85C0	TEST EAX,EAX
0040517f	0FB8 76FFFFFF	JE hw3p3.004050fb
00405185	E8 6383C41C	CALL 1D04D4ED
0040518a	C3	RET

Codice della `FUN_004050c0` visto da *OllyDbg* dopo essere stato deoffuscato dall'applicazione

In una prima iterazione, si nota che il codice controlla se il valore corrispondente alla lunghezza dell'input sia pari a 9 e, nel caso in cui così non fosse, la funzione ritorna al chiamante senza effettuare lo shutdown.

Dunque, in una seconda iterazione, si prova ad inserire come codice di sblocco la stringa "123456789", lunga 9 caratteri, e si continua ad analizzare l'esecuzione tramite debugger. Si nota che viene caricato volta per volta in EAX un singolo byte presente sullo stack e se ne fa lo XOR con il corrispondente carattere dell'input dell'utente. Il risultato dello XOR viene confrontato con uno specifico byte che, ad esempio, nel caso del primo carattere deve essere pari a 0C (in esadecimale).

Quindi, il confronto sulla correttezza del codice di sblocco non viene effettuato “in chiaro”, ma viene “cifrato” l’input dell’utente effettuando lo XOR con una “chiave” di 9 bytes e viene controllata la correttezza di tale “ciphertext”.

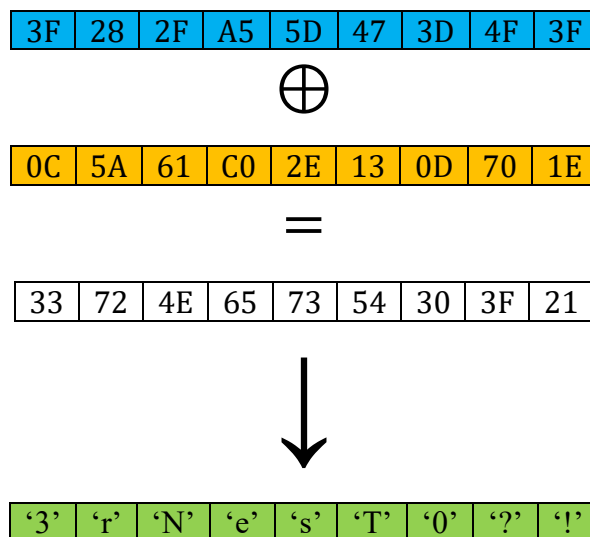
Guardando sullo stack, si riesce ad individuare facilmente la “chiave” che viene usata:

0060FBE4	AS2F283F	
0060FBE8	4F3D475D	
0060FBEC	0000003F	
0060FBF0	001D03CC	
0060FBF4	0060FC46	ASCII "123456789"

La "chiave" caricata sullo stack: 3F 28 2F A5 5D 47 3D 4F 3F

È analogamente semplice riuscire ad individuare il “ciphertext” atteso, guardando alle singole istruzioni di CMP presenti nella funzione FUN_004050c0. Dunque, si ha:

- chiave = 3F 28 2F A5 5D 47 3D 4F 3F
- ciphertext = 0C 5A 61 C0 2E 13 0D 70 1E
- plaintext \oplus chiave = ciphertext \rightarrow ciphertext \oplus chiave = plaintext



Consultando la tabella di codifica ASCII, si è convertito il plaintext ottenuto al corrispettivo valore in caratteri e il codice di sblocco risulta essere la stringa “3rNesT0?!”.

ULTERIORI ANALISI SUL PROGRAMMA

Nella parte della *WndProc* in cui viene gestito il messaggio WM_CREATE, era stata precedentemente individuata una funzione chiamata subito dopo la *funSetTimer*: si decide di approfondire l’analisi di tale funzione che, successivamente, è stata ridenominata *CalculateFingerprint*.

```

void CalculateFingerprint(void)
{
    int iVar1;
    uint *puVar2;

    if (startingAddress != 0) {
        puVar2 = (uint *) (startingAddress + 1024);
        global_app_data.fingerprint = 0;
        iVar1 = (nSizeFileLow >> 2) - 256;
        while (iVar1 != 0) {
            global_app_data.fingerprint = global_app_data.fingerprint ^ *puVar2;
            puVar2 = puVar2 + 1;
            iVar1 = iVar1 - 1;
        }
    }
    return;
}

```

Decompilato della funzione CalculateFingerprint()

Questa funzione utilizza i valori *startingAddress* e *nSizeFileLow* che erano stati precedentemente calcolati tramite l'invocazione della funzione *getFileInfo* analizzata precedentemente. Sostanzialmente, ciò che la funzione fa è sommare allo *startingAddress* il valore 1024 e iterare 4 bytes alla volta, fino alla fine del programma, calcolando ad ogni iterazione lo XOR dei 4 bytes considerati in quell'iterazione con il risultato ottenuto all'iterazione precedente. Sostanzialmente, in questo modo, non ottiene nient'altro che un fingerprint del file eseguibile. Tale fingerprint, viene salvato in uno dei campi della struct globale contenente le informazioni principali usate dall'applicazione.

Molto probabilmente, i 1024 bytes che vengono scartati nel calcolare il fingerprint dell'eseguibile, corrispondono ai bytes della testata. Infatti, come è ben visibile analizzando il file con *PEview*, la testata ha dimensione esattamente di 1024 bytes, dopo i quali, ad indirizzo relativo 0x400, inizia la sezione testo.

pFile	Raw Data	Value
00000400	C3 8D B4 26 00 00 00 00 8D B4 26 00 00 00 90	...&.....&.....
00000410	83 EC 1C 31 C0 66 81 3D 00 00 40 00 4D 5A C7 05	...1.f.=..@.MZ...
00000420	6C 74 40 00 01 00 00 00 C7 05 68 74 40 00 01 00	!t@...ht@...
00000430	00 00 C7 05 64 74 40 00 01 00 00 00 C7 05 10 71	...dt@...q...
00000440	40 00 01 00 00 00 75 18 8B 15 3C 00 40 00 81 BA	@...u...<@...
00000450	00 00 40 00 50 45 00 00 8D 8A 00 00 40 00 74 50	@.PE.....@.tP
00000460	A3 0C 70 40 00 A1 74 74 40 00 85 C0 75 32 C7 04	.p@...tt@...u2...
00000470	24 01 00 00 00 E8 36 27 00 00 E8 39 27 00 00 8B	\$....6'....9'...
00000480	15 88 74 40 00 89 10 E8 14 17 00 00 83 3D B4 51	...t@...=...Q...
00000490	40 00 01 74 4B 31 C0 83 C4 1C C3 8D 74 26 00 90	@...tK1.....t&...
000004A0	C7 04 24 02 00 00 00 E8 04 27 00 00 EB CC 66 90	...\$....f...f...
000004B0	0F B7 51 18 66 81 FA 0B 01 74 3D 66 81 FA 0B 02	...Q.f...t=f...
000004C0	75 9E 83 B9 84 00 00 00 0E 76 95 8B 91 F8 00 00	u.....v.....
000004D0	00 31 C0 85 D2 0F 95 C0 EB 86 8D B6 00 00 00 00	...1.....
000004E0	C7 04 24 00 2A 40 00 E8 24 1E 00 00 31 C0 83 C4	...\$.@...\$.1...
000004F0	1C C3 8D B6 00 00 00 00 83 79 74 0E 0F 86 5E FF	...yt...^...
00000500	FF FF 8B 89 E8 00 00 00 31 C0 85 C9 0F 95 C0 E9	...1.....
00000510	4C FF FF FF 8D B4 26 00 00 00 8D 74 26 00 90	L...&...t&...
00000520	83 EC 2C A1 60 74 40 00 C7 44 24 10 04 70 40 00	...t@...DS...p@...
00000530	A3 04 70 40 00 A1 0C 71 40 00 C7 44 24 08 14 70	...p@...q@...DS...p
00000540	40 00 89 44 24 0C C7 44 24 04 18 70 40 00 C7 04	@...DS...DS...p@...
00000550	24 1C 70 40 00 E8 6E 26 00 00 83 C4 2C C3 66 90	...p@...n&...f...
00000560	8D 4C 24 04 83 E4 F0 31 C0 FF 71 FC 55 89 E5 57	...L\$....1...q.U...W
00000570	56 8D 55 A4 53 89 D7 51 B9 11 00 00 00 83 EC 78	V.U.S...Q.....x
00000580	8B 35 74 74 40 00 F3 AB 85 F6 0F 85 A0 02 00 00	...5tt@.....
00000590	64 A1 18 00 00 00 8B 35 5C 82 40 00 8B 78 04 31	d.....5\...@...x.1
000005A0	DB EB 19 8D 74 26 00 90 39 C7 0F 84 18 02 00 00	...t&...9.....
000005B0	C7 04 24 E8 03 00 00 FF D6 83 EC 04 89 D8 F0 0F	...\$....
000005C0	B1 3D C0 74 40 00 85 C0 75 DE A1 C4 74 40 00 31	...t@...u...t@...1

PEview: indirizzo relativo iniziale della sezione .text del file eseguibile. Corrisponde a 0x400.

Quindi, con l'individuazione del campo "fingerprint", la struct *app_data* utilizzata dall'applicazione risulta essere stata analizzata in tutti i suoi campi ed è riassunta dalla figura riportata di seguito:

DataType	Name	Comment
int	current_time	time counter
int	timer_clock	timer clock unit, set to 1000 ms
int	timerID	timer's identifier
int	target_time	time to wait until shutdown
int	isTimerRunning	indicates if the countdown is active or not
pointer	functionToBeCalled	pointer to the target function called when the timer expires
char[128]	warning_msg	warning message
char[16]	seconds	strings representing the seconds of the countdown
HWND[6]	handles	handles to main window and subwindows
UINT	fingerprint	fingerprint/checksum of the exe file, calculated doing the XOR 4 bytes at the time excluding the first 1024 bytes

Struct "app_data" e i campi che la compongono

Rianalizzando la funzione *errorWindowAndExitProcess*, ci si rende conto che il fingerprint calcolato e salvato nella struct viene confrontato con un valore salvato in una variabile globale, ridenominata *correct_fingerprint* e solo nel caso in cui i due valori fossero diversi viene mostrata la message box che riporta l'internal error e il processo viene terminato. Tale flusso è stato controllato anche usando il debugger analizzando il file eseguibile della patch numero 2.

004042BE	8B9424 30010000	MOV EAX, DWORD PTR SS:[ESP+130]			
004042C5	8B0D 04714000	MOV ECX, DWORD PTR DS:[407104]			
004042CB	8B15 A4504000	MOV EDX, DWORD PTR DS:[4050A4]			
004042D1	8B80 C0000000	MOV EAX, DWORD PTR DS:[EAX+C0]			
004042D7	85C9	TEST ECX, ECX			
004042D9	74 02	JE SHORT hw3p2.004042DD			
004042DB	9A 4239D075 14A	CALL FAR A114:75D03942	Far call		
004042E2	04 71	ADD AL, 71			
004042E4	40	INC EAX			
004042E5	0085 C074029A	ADD BYTE PTR SS:[EBP+9A0274C0], AL			
004042EB	42	INC EDX			
004042EC	81C4 24010000	ADD ESP, 124			
004042F2	5B	POP EBX			
004042F3	5E	POP ESI			
004042F4	C3	RETN			

Nel registro *eax* è presente il fingerprint salvato nella struct, mentre nel registro *edx* è presente il fingerprint corretto

CONCLUSIONE

In conclusione, il programma usa numerose misure anti-disassembling e anti-debugging. In particolare, per quanto riguarda le ultime, ne sono state individuate quattro:

1. Chiamata ad *IsDebuggerPresent*;
2. Controllo del campo *BeingDebugged* della struttura dati PEB associata al processo;
3. Invocazione di *OutputDebugStringA* con parametro "%s%s%s%s%s%s%s%s%s%s"; inoltre, la libreria contenente l'API viene caricata invocando *LoadLibraryA* e l'indirizzo dell'API viene ottenuto con *GetProcAddress*;
4. Viene calcolato un fingerprint del file eseguibile caricato in memoria e il processo viene terminato se tale fingerprint non corrisponde con quello atteso.

Inibendo tali misure anti-debugger, si è potuto combinare l'analisi statica con l'analisi dinamica avanzata, velocizzando il processo di individuazione del codice di sblocco. Il controllo sulla correttezza del codice di sblocco non viene effettuato confrontando l'input dell'utente con il codice corretto, bensì viene cifrato l'input dell'utente effettuando lo XOR con una "chiave" di 9 bytes (pari alla

lunghezza del codice corretto) e si controlla dunque la correttezza del “ciphertext”. Il codice di sblocco risulta essere la stringa “3rNesT0?!”.