InTether protection system…reversing kernel code, reversing data, coding reversing tools, …basically it's the perfect reversing training field!

Part 2

…now that we know how InTether protection works, we can give a look at .ith files that InTether packager creates…this could be a good file reversing exercise.
As always…if you are looking only for a way to crack InTether protection…you are on the wrong way. This is a reversing tutorial!

***** The OS is Windows 2000 professional with NTFS file system *****
***** And please forget my horrible english *****

Ok here we are again with InTether protection. In the part 1 we have seen how this protection works so I don't repeat nothing here, just make sure to have read the previous tutorial before to start with this.
In this tutorial we concentrate our attention on .ith files. An .ith file is the resulting protected file generated by InTether Packager. It stores the original data file and all settings and it's not a win32 PE image but a 'static' data file.
What we want to do is quite simple: we want to extract the original file from its .ith file (the protected version of the original file). With this purpose we can give a look at the .ith file format and reverse a bit in more of the InTether protection system.
But lets start!
First thing we have to create a simple text file to protected …so just put this text in a txt file called Foo.txt :

Hello World!

The file is 12 bytes. Now we can protect it with InTether Packager. We use only the classic option "File may be read for…" and we set a 10 mins time.
Ok now we have our Foo.ith file and we can give it a look in a hex editor.
Keep in mind that this is my version of Foo.ith, which is encrypted with a encryption key that probably comes from a random seed ...so Foo.ith may be different from your in the 'first look' (but not in the real contents ☺).

```
00000000 2175 3737 3C8E 2E10 4506 51D5 11FE E1BA 2D88 D4F0 !u77<...E.Q.....-...
00000014 E0C6 82AE F297 2462 A683 8A4E 519A BFDD CB4A CC3B ......$b...NQ....J.;
00000028 62E5 22E1 0E4E 79B4 F9EA A825 7A58 49CA DC31 3493 b."..Ny....%zXI..14.
0000003C F6D6 6A37 5BF2 2E19 CAE7 6A5B B38A 1A83 7E08 CA80 ..j7[.....j[....~...
00000050 1579 EA80 EF00 CBB7 5052 15E9 75E9 DDD6 5978 532B .y......PR..u...YxS+
00000064 2E4C 2AF5 7D01 7BBD 0B0A 27B0 D1C1 F9DF 663C 10AC .L*.}.{...'.....f<..
00000078 9E0C 3028 0A5B D1A7 4AA0 80D1 808B CB29 282B 00D8 ..0(.[..J......)(+..
0000008C 3382 F903 AFE8 0537 57EC C7E6 B081 369D 47EC F395 3......7W.....6.G...
000000A0 2E7E 8672 2869 E3CB B9E6 4AAA B6C0 9397 581A 0335 .~.r(i....J.....X..5
000000B4 05D1 21D4 6B5C 4A15 ACE3 93C5 739C 701E 0EED 7E3B ..!.k\J.....s.p...~;
000000C8 EB14 0070 7A6B DC83 1EE6 36DD 1C8F 1667 9891 AC83 ...pzk....6....g....
000000DC FE5F 4193 AC43 64D4 04D4 714A A78B FAA7 2579 5A4A ._A..Cd...qJ....%yZJ
000000F0 8A3D 632D 6EC1 361C 1515 9E02 1A0A 38C0 7BFA 70A3 .=c-n.6.......8.{.p.
00000104 7FF1 9437 5204 D1E7 1D53 1B67 B48C 332B 744B 4ECF ...7R....S.g..3+tKN.
00000118 2175 3737 3C8E 2E10 4506 51D5 11FE E1BA A902 2C73 !u77<...E.Q.......,s
0000012C AA72 946F 0DFD B303 ABA0 F1A7 726C 6421 216D 3265 .r.o........rld!!m2e
00000140 2027 9932 4506 51D5 11FE E1BA 2AEA CC4D 1B36 E5F4  '.2E.Q.....*..M.6..
00000154 2A1D 11D1 A979 6F5D 32BF AFE2 5BBB FD90 2175 3737 *....yo]2...[...!u77
00000168 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
0000017C 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
```

```
00000190 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
000001A4 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
000001B8 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
000001CC 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
000001E0 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
000001F4 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000208 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
0000021C 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000230 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
00000244 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000258 3C8E 2E10 0D61 08E1 88A5 931B 2175 3737 3C8E 2E10 <....a......!u77<...
0000026C 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 0930 73FB !u77<...!u77<....0s.
00000280 BBD2 4585 6DFF E113 7EBA 6E0B 2175 3737 3C8E 2E10 ..E.m...~.n.!u77<...
00000294 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
000002A8 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
000002BC 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
000002D0 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
000002E4 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
000002F8 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
0000030C 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000320 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
00000334 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000348 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
0000035C 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000370 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
00000384 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000398 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
000003AC 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
000003C0 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
000003D4 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
000003E8 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
000003FC 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000410 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
00000424 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000438 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
0000044C 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000460 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
00000474 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 !u77<...!u77<...!u77
00000488 3C8E 2E10 2175 3737 3C8E 2E10 2175 3737 3C8E 2E10 <...!u77<...!u77<...
0000049C 2175 3737 3C8E 2E10 0421 6D32 6520 2799 322D 6B80 !u77<....!m2e '.2-k.
000004B0 FD77 DD28 94ED 5C38 68B1 0D5D 7A21 7537 373C 8E2E .w.(..\8h..]z!u77<..
000004C4 1021 7537 373C 8E2E 10E0 0384 8898 9661 E345 1C43 .!u77<.........a.E.C
000004D8 25F4 9F16 8421 6D32 6520 2799 3245 0651 D511 FEE1 %....!m2e '.2E.Q....
000004EC BA62 9D91 4472 B72A 8778 0000 0000 0000 0000 0000 .b..Dr.*.x..........
00000500 0004 0000 00C1 0800 0000 0000 0087 8080 80D4 C3B2 ....................
00000514 A1                                                 .
```

Ok it is 514h bytes (1301d bytes) and our Foo.txt (12d bytes) is inside this ‘spaghetti’ bytes ☺ .
But reversing is, first of all, investigation…
So lets analyse what these bag of bytes can tell us…

- The file is encrypted. (great intuition MaV ! ☺ …but try to justify this in few words ☺).
- A sequence of bytes is repeated several times : `3C8E 2E10 2175 3737`
  This means that we have a simple XOR encryption and the data file has a lot of DWORD of the same value, may be 0.
- At the offset 4F7h we have the only 00 bytes (in red) of all file. This tell us that probably these last bytes are NOT encrypted.
- We know that the file is encrypted (or at least our Foo.txt) but this Foo.ith file can be read by any InTether Reader on any machine. This tell us that the decryption key (or a seed to generate it) should be stored inside this file. Or, in the best situation for us (and the worst for who use this protection) the decryption key is hardcoded inside the InTether Reader. I suggest to avoid to undervaluate the authors of a program that we are reversing ALWAYS!. So we forget the last situation and we can say that the encryption key or a seed to generate it must be stored inside the file, and this value(s) is not inside a piece of the file encrypted!…may be is in the last bytes ☺.

- The last bytes (not encrypted) could be part of a header (or better say a footer). InTether Reader need to be able to recognize a valid .ith file, other than from its extension.

So from these clues we can anticipate the first step that the Reader should perform: reading Foo.ith with an offset beetwen 4F7h and the EOF (end of file).

But now it's time to collect some 'on-field' clues, so we need our FFsd filter (see the part 1) to monitor the accessing to the file Foo.ith.
We run FFsd and then we run sice (don't forget to hide it from meltice trick). Now we can enable our IRP_MJ_READ bpx and we start to collect info. Everytime our int 3 in FFsd break, I remember that, we have the IRP_MJ_READ parameters in these registers:

EAX = file offset inside the file where fsd have to start reading.
EBX = pointer at the (unicode) file name of to read from.
ECX = length (bytes) to read from the file.
EDI = buffer for bytes read from the file.

For now we just take note of all access at Foo.ith without take care of the bytes read. We write down only Offset, Size and the caller of read operation. It takes about 20 mins but it's important because I show you how a reversing session is first of all analyse and investigation of all clues and not only stepping code in softice…often we can have more information studying a log or a static series of bytes.
Anyway these are all the accesses (read) at Foo.ith:

Num	File	Offset (hex)	Size (hex)	Caller
1	Foo.ith	511	4	InTetherReader.exe
2	Foo.ith	4F9	18	InTetherReader.exe
3	Foo.ith	0	2	Shell32.dll
4	Foo.ith	511	4	Dg32.exe
5	Foo.ith	4F9	18	Dg32.exe
6	Foo.ith	4F5	4	Dg32.exe
7	Foo.ith	4F9	18	Dg32.exe
8	Foo.ith	4DD	18	Dg32.exe
9	Foo.ith	4A5	18	Dg32.exe
10	Foo.ith	4A4	1	Dg32.exe
11	Foo.ith	4F9	18	Dg32.exe
12	Foo.ith	4DD	18	Dg32.exe
13	Foo.ith	4A5	18	Dg32.exe
14	Foo.ith	13C	18	Dg32.exe
15	Foo.ith	118	18	Dg32.exe
16	Foo.ith	0	18	Dg32.exe
17	Foo.ith	4F9	18	Dg32.exe
18	Foo.ith	4DD	18	Dg32.exe
19	Foo.ith	4A5	18	Dg32.exe
20	Foo.ith	154	350	Dg32.exe
21	Foo.ith	4F9	18	Dg32.exe
22	Foo.ith	4DD	18	Dg32.exe
23	Foo.ith	4A5	18	Dg32.exe
24	Foo.ith	13C	18	Dg32.exe
25	Foo.ith	118	18	Dg32.exe
26	Foo.ith	18	100	Dg32.exe
27	Foo.ith	4F9	18	Dg32.exe
28	Foo.ith	4DD	18	Dg32.exe
29	Foo.ith	4BD	10	Dg32.exe
30	Foo.ith	4CD	10	Dg32.exe

…Then dgpdb.sys access to dgperm.db through dgcrypt.sys…after that, dg32.exe goes on with Foo.ith reading:

Num	File	Offset	Size	Caller
31	Foo.ith	4F9	18	Dg32.exe
32	Foo.ith	4DD	18	Dg32.exe
33	Foo.ith	4A5	18	Dg32.exe
34	Foo.ith	13C	18	Dg32.exe
35	Foo.ith	118	18	Dg32.exe
36	Foo.ith	0	18	Dg32.exe
37	Foo.ith	4F9	18	Dg32.exe
38	Foo.ith	4DD	18	Dg32.exe
39	Foo.ith	4A5	18	Dg32.exe
40	Foo.ith	13C	18	Dg32.exe
41	Foo.ith	118	18	Dg32.exe
42	Foo.ith	0	18	Dg32.exe

…At this point we have the main nagscreen, it show the characteristic of our protected file and it permits us to choose where to save the file. We leave the default path and we click on “save&close”…and the accessing at Foo.ith goes on….

Num	File	Offset	Size	Caller
43	Foo.ith	4F9	18	Dg32.exe
44	Foo.ith	4DD	18	Dg32.exe
45	Foo.ith	4A5	18	Dg32.exe
46	Foo.ith	13C	18	Dg32.exe
47	Foo.ith	4F9	18	Dg32.exe
48	Foo.ith	4DD	18	Dg32.exe
49	Foo.ith	4A5	18	Dg32.exe
50	Foo.ith	13C	18	Dg32.exe
51	Foo.ith	118	18	Dg32.exe
52	Foo.ith	0	18	Dg32.exe
53	Foo.ith	511	4	Dgfs.sys
54	Foo.ith	4F9	18	Dgfs.sys
55	Foo.ith	4F5	4	Dgfs.sys
56	Foo.ith	4DD	18	Dgfs.sys
57	Foo.ith	4CD	10	Dgfs.sys
58	Foo.ith	4A5	18	Dgfs.sys
59	Foo.ith	4A4	1	Dgfs.sys
60	Foo.ith	154	350	Dgfs.sys
61	Foo.ith	13C	18	Dgfs.sys

…at this point is the turn of vault.dgv. Dgvault.sys access it through dgcrypt.sys…then…

Num	File	Offset	Size	Caller
62	Foo.ith	130	C	Dgfs.sys
63	Foo.ith	118	18	Dgfs.sys
64	Foo.ith	0	18	Dgfs.sys

…and then the messagebox tells us that our Foo.txt has been saved. Cool!! ☺

Ok now we can analyse all this log in deep and this is what we have:

- **(Num=1)** How we have predicted…the first read operation on Foo.ith is performed at the offset 511h and it read the last 4 bytes (which are probably a marker that identify an ith file and a version).

- **(Num=1/2)** InTetherReader.exe performs only 2 read operation (the first two). This tells us that the reader is just a parent application for an hooker, but it doesn’t have a major role during an extraction/activation of our protected file.

- **(Num=3)** Windows shakes the ass in anytime…shell32.dll checks the first 2 bytes to see if the file is a PE image (MZ) ☺ ☺.

- Dg32.exe is responsible to read and interprete the .ith file before to show the main nagscreen. It use dgpdb to read the file dgperm.db and make sure that we can have access at the content of this .ith file.

- **(Num=53…)** After the main nagscreen, when we choose to save the protected file stored inside our .ith file, dgfs.sys takes the responsibility to access again the .ith file and perform the extraction of the Foo.txt file and save it inside vault.dgv file.

- So…two are the modules that perfom reading operation on our Foo.ith file: dg32.exe and dgfs.sys. Dg32.exe performs a reading and checking task…and if we are 'good guys' it decides to give us a chance to save (open) the file…showing the main nagscreen. Dgfs.sys read the file and build the 118h bytes block (see part 1) inside the vault.dgv file and then write our Foo.txt file inside the same vault.dgv .

- Two are the 'key' reading operations in the log info : **Num=4** and **Num=53**. Why ? Simple…both are the first access that each module (dg32.exe and dgfs.sys) perform on the .ith file and both accessing start to read the 4 bytes at offset 511h (InTether marker) after these first accessing both modules start a sequence of reading accesses at the same offsets, with the only difference that dg32.exe performs some reading access two or three times but always at the same offsets…so the reading offset in dg32.exe are read in dgfs.sys too. In more these offsets start from the higher and goes to the lower , basically from the end of the file to the beginning. Every access start at 4F9 (we can exclude the 511h offset, which we know it's a marker so a module has no need to read it more than one time) for ex. to read at the offset 118h, dg32.exe (or dgfs.sys) start to read at 4F9 and goes down passing always for the same offsets.
  So……what this means is simple…a module that want to read a specific information inside an .ith file it need to pass trough all blocks of information that follow the information requested, this because the footer (that we have identified in the binary image of .ith file), doesn't have an idea where each information is stored inside the file are…

- **(Num=62)** This reading operation is quite suspicious, it reads 0Ch bytes (12d bytes) which are exactly the size of our Foo.txt file. But even if we are sure that this is our file…the offset 130h is valid only for Foo.ith file, for another .ith file this offset will be differente, the reason is explained in previous point…there is nothing inside an .ith file that tell at which offset there is a specific information but all information are in a sequence!

- Often the number of bytes read is 18h. This is an important clue because we know that the footer has no idea where each information are inside the .ith file so we can suppose something like this: The 18h bytes are the size of some information, or 18h bytes are the size of an header for each information stored inside the file.

Ok now we have enough clues to start a 'live' reversing session. What we need it's another .ith file…so we create another text file Foo2.txt and we protect it. Then we just run it like before and we skip the IRP_MJ_READ operations until we find the first read operation on Foo2.ith file performed by dgfs.sys

---

!!! Keep in mind that in the following chunks of code all virtual addresses are in the system address space and may be different from your machine. In more all encrypted values may be different too, due to random encryption seeds. !!!

We step until we return in dgfs.sys and we find the routine to perform every read operations:

```
BC1F09B6      call ntoskrnl!ZwReadFile
              test eax,eax
              jge  BC1F09C4
              xor  eax,eax
              jmp  BC1F09D8
              mov  ecx,[ebp-0C]
              cmp  ecx,[ebp+14]          ; Has been read the right bytes number ?
              jz   BC1F09D3              ; yes
              push 01
              pop  eax
BC1F09D3      mov  edx,[ebp+18]
```

```
            mov  [edx],ecx
BC1F09D8    leave
            ret  0014
```

We return from this routine here:

```
BC1E8199    call BC1F098C              ; read from Foo2.ith (log num=54)
                                       ; Offset = 511h
                                       ; Size = 4
```

D4	C3	B2	A1

```
            test eax,eax
            push esi
            jnz  BC1E81BA              ; read was ok, go on
            … …
BC1E81BA    call BC1F0C00              ; call ZwClose to close the filehandle
            cmp  dword ptr[ebp-08],A1B2C3D4 ; check .ith marker
            jz   BC1E81DD              ; yes it's a valid .ith file, go on
            … …
BC1E81DD    push dword ptr [ebp+14]
            push dword ptr [ebp+10]
            push dword ptr [ebp+0C]
            push dword ptr [ebp+08]
            call BC1E7AD9              ; create Foo2.txt empty
            … …
BC1E67DB    push 18
            call BC1F0E16              ; allocate a 18h bytes buffer
            … …
BC1E6814    lea  ecx,[ebp-30]
            push ecx
            push 18
            push eax
            push dword ptr [ebp-28]
            push dword ptr [ebp-10]
            call BC1F098C              ; read from Foo2.ith (log num=55)
                                       ; Offset = 4F9h
                                       ; Size = 18h
```

00	00	00	00	00	00	00	00	04	00	00	00	C1	08	00	00
00	00	00	00	87	80	80	80								

```
            test eax,eax
            jz   BC1E7957
            mov  esi,[ebp-28]
            push 06
            pop  ecx
            lea  edi,[ebp-6C]
            repz movsd                 ; copy 18h bytes in another buffer
            cmp  dword ptr[ebp-58],80808087 ; check in 18h bytes read if
                                       ; last dword is 80808087
```

00	00	00	00	00	00	00	00	04	00	00	00	C1	08	00	00
00	00	00	00	87	80	80	80								

```
            jnz  BC1E7957
            mov  esi,00001048
            push esi
            call BC1F0E16              ; allocate a 1048h buffer (FF3D2000)
            push esi
            mov  [ebp-34],eax
            call BC1F0E16              ; allocate a second 1048h buffer
                                       ; (FF3D0000)
```

```
                cmp   dword ptr[ebp-60],00000678 ; check in 18h bytes read if the
                                                 ; fourth dword is 'above' 678h
```

00	00	00	00	00	00	00	00	04	00	00	00	C1	08	00	00
00	00	00	00	87	80	80	80								

```
                ja    BC1E65AC
                … …
BC1E65AC        mov   eax,[ebp-64]             ; eax = third dword in 18h bytes read
```

00	00	00	00	00	00	00	00	04	00	00	00	C1	08	00	00
00	00	00	00	87	80	80	80								

```
                mov   [ebp-18],ebx
                sub   [ebp+08],eax             ; [ebp+08] = last offset read in .ith
                                               ; file. Sub 04 to generate the next
                                               ; offset 4F5h
                cmp   [ebp+14],ebx
                jz    BC1E6962
                … …
BC1E6962        lea   ecx,[ebp-30]
                push  ecx
                push  eax
                push  dword ptr [ebp+08]
                lea   eax,[ebp-18]
                push  eax
                push  dword ptr [ebp-10]
                call  BC1F098C                 ; read from Foo2.ith (log num=56)
                                               ; Offset = 4F5h
                                               ; Size = 4
```

C5	00	00	00

```
                test  eax,eax
                jz    BC1E7957
                mov   eax,[ebp-64]             ; eax = third dword in 18h bytes read
```

00	00	00	00	00	00	00	00	04	00	00	00	C1	08	00	00
00	00	00	00	87	80	80	80								

```
                push  0A
                add   [ebp+08],eax             ; update the reading offset in Foo2.ith
                                               ; 4F5h + 4 = 4F9h
                pop   ecx
                xor   eax,eax
                lea   edi,[ebp-010C]
                repz  stosd                    ; prepare 28h (0A*4) bytes buffer clean
                stosb                          ; null termined ? ☺
                mov   edi,[ebp-18]             ; edi = 000000C5 (read from .ith before)
                or    ecx,-1
                imul  edi,edi,29h              ; 000000C5 * 29h = 00001F8Dh
                add   edi,BC1F1740             ; BC1F1740 + 00001F8D = BC1F36CD =
                                               ; offset in dgfs data section
```

BC1F1740 is an offset inside dgfs data section, which is buffer that store a series of 28h bytes alphanumeric strings (null terminated). The value read from ith file is used like an index in this buffer to take a 28h bytes alphanumeric string. This string should be used to generate some kind of key to decrypt something else…but lets go on…

```
                xor  eax,eax
                repnz scasb                 ; scan the alphanumeric string until
                                            ; find the end (00)
                not  ecx                    ; size of the string (29h with 00 byte)
                sub  edi,ecx                ; edi = point start of this string
                lea  edx,[ebp-010C]
                mov  eax,ecx
                mov  esi,edi
                mov  edi,edx
                shr  ecx,2
                repz movsd                  ; copy the 28h bytes string in the
                                            ; previous allocated buffer
                mov  ecx,eax
                lea  eax,[ebp-010C]
                push eax
                and  ecx,3
                lea  eax,[ebp-50]
                repz movsb
                push eax
                call ntoskrl!RtlInitAnsiString ; Init the string in an ansi string
                                            ; variable
                movzx eax,word ptr[ebp-50] ; size of the string (28h)
                push eax
                mov  eax,[BC21AA20]
                push dword ptr [ebp-4C]     ; pointer to the 28h bytes string buffer
                push dword ptr [ebp-14]     ; pointer to the 1048h bytes buffer
                                            ; (FF3D0000)
                call [eax+14]               ; call in dgcrypt.sys
```

Ok a break a second!…the above lines of code smell of a 'creating decryption key' routine ☺.
The selected alphanumeric string (28h bytes) is passed as a parameter to dgcrypt.sys routine , with a pointer at one of the previous allocated 1048h bytes buffer, …so what we need to check carefull is what happen inside dgcrypt now…so lets step in this `call [eax+14]` :

```
BC5E12B6        push ebp
                mov  ebp,esp
                push ebx
                push esi
                push dword ptr [eax+10]
                mov  esi,[eax+0C]           ; pointer 28h bytes string
                xor  ebx,ebx
                push esi
                push dword ptr [eax+08]     ; pointer 1048h bytes buffer
                call BC5E11B6               ; fill the 1048 bytes buffer
                cmp  [eax+10],ebx
                jle  BC5E12DB
                mov  al,[esi]
                mov  ecx,[eax+10]
BC5E12D6        and  bl,al
                dec  ecx
                jnz  BC5E12D6
BC5E12DB        movzx edx,bl
                pop  esi
                pop  ebx
                pop  ebp
                ret  000C
```

A simple routine with a call BC5E11B6 that fills our 1048h bytes buffer. So step in…☺

```
BC5E11B6      push ebp
              mov  ebp,esp
              push ecx
              push ebx
              mov  ebx,[eax+08]          ; pointer 1048h bytes buffer (FF3D0000)
              push esi
              push edi
              lea  eax,[ebx+00000800]    ; eax = FF3D0800
              xor  ecx,ecx
              mov  edx,100
BC5E11CD      mov  esi,[ecx+BC5E4760]    ; BC5E4760 is an offset inside
                                         ; dgcrypt.sys data section and it is a
                                         ; table. It just picks up a dword at
                                         ; each cycle of this loop.
              add  ecx,4                 ; prepare index for the next cycle
              mov  [eax+FFFFF800],esi    ; copy the picked up dword in 1048h
                                         ; bytes buffer (FF3D0000-FF3D03FF).
              mov  esi,[ecx+BC5E4B65]    ; BC5E4B65 is an offset inside
                                         ; dgcrypt.sys data section and it is a
                                         ; table. It just picks up a dword at
                                         ; each cycle of this loop.
              mov  [eax-0400],esi        ; copy the picked up dword in 1048h
                                         ; bytes buffer (FF3D0400-FF3D07FF).
              mov  esi,[ecx+BC5E4F5C]    ; BC5E4F5C is an offset inside
                                         ; dgcrypt.sys data section and it is a
                                         ; table. It just picks up a dword at
                                         ; each cycle of this loop.
              mov  [eax],esi             ; copy the picked up dword in 1048h
                                         ; bytes buffer (FF3D0800-FF3D0BFF).
              mov  esi,[ecx+BC5E535C]    ; BC5E535C is an offset inside
                                         ; dgcrypt.sys data section and it is a
                                         ; table. It just picks up a dword at
                                         ; each cycle of this loop.
              mov  [eax+0400],esi        ; copy the picked up dword in 1048h
                                         ; bytes buffer (FF3D0C00-FF3D0FFF).
              add  eax,4                 ; inc index in 1048h bytes buffer
              dec  edx
              jnz  BC5E11CD              ; Has been filled 100h dword (*4) ??
                                         ; no yet, next cycle
```

Ok in this first loop the 1048h bytes buffer is 'virtualy' splitted in 4 block of 400h bytes each. At each loop four buffer in the data section of dgcrypt.sys are used as container to pick up 4 dwords and copy them in the the 400h blocks. After 100h cycle, 1000h bytes of 1048h bytes buffer are filled. Lets go on…

```
              push 12
              lea  edx,[ebx+000001000]   ; edx = FF3D1000
              pop  esi
              xor  eax,eax
              mov  ecx,esi
              mov  edi,edx
              repz stosb                 ; clean 12h bytes in 1048h bytes buffer
                                         ; FF3D1000-FF3D10012
              xor  ecx,ecx
              mov  [ebp-04],esi          ; Init a counter (12h)
BC5E1218      xor  eax,eax
              mov  dword ptr [ebp+08],00000004 ; Init a counter (4)
BC5E1221      mov  edi,[ebp+0C]          ; edi = pointer string buffer
              movsx esi,cx
              shl  eax,08                ; reverse the dword taken from the
                                         ; string buffer
              movsx esi,byte ptr[edi+esi] ; take a byte from the string buffer
              or   eax,esi
              inc  ecx
              movsx esi,cx
              cmp  esi,[ebp+10]
              jl   BC5E123B
              xor  ecx,ecx
BC5E123B      dec  dword ptr [ebp+08]    ; Has been taken 4 bytes ??
```

```
                jnz   BC5E1218              ; not yet, next cycle
                xor   [edx],eax             ; xor dwords from FFD31000 with the
                                            ; reversed dwords (eax) generated in
                                            ; loop above.
                add   edx,4                 ; inc pointer in FFD31000
                dec   dword ptr [ebp-04]    ; Has been processed 12h dword ??
                jnz   BC5E1218              ; not yet, restart the loop above
```

Ok…in this second loop dgcrypt fills the remaining 48h bytes from FFD31000 to FD31047 and it uses the 28h bytes string passed by dgfs.sys. It takes 4 bytes in each cycle and it reverse them to build a dword, then it use the dword in a xor operation with the dword in the range FFD31000 to FD31047.
So now the 1048h bytes buffer is full but it's not ready to be used yet…lets check …

```
                and   dword ptr [ebp+08],00
                and   dword ptr [ebp+0c],00
                push 09
                lea   esi,[ebx+00001004]    ; esi = FFD31004
                pop   edi                   ; counter (09)
BC5E125B        lea   eax,[ebp+08]          ; pointer value A (0 in first cycle)
                push eax
                lea   eax,[ebp+0C]          ; pointer value B (0 in first cycle)
                push eax
                push ebx                    ; ebx =start 1048h bytes buffer FFD30000
                call BC5E10FA
                mov   eax,[ebp+08]          ; eax = value A processed
                mov   [esi-04],eax          ; save the value A processed in the
                                            ; range FFD31000-FFD310047
                mov   eax,[ebp+0C]          ; eax = value B processed
                mov   [esi],eax             ; save the value A processed in the
                                            ; range FFD31000-FFD310047
                add   esi,8                 ; inc index in the 48h bytes
                dec   edi                   ; dec counter
                jnz   BC5E125B              ; processed all 48h bytes ?? not yet
                                            ; next cycle. With A and B update too
```

→ We go on with this part of code at page 12

Ok stop here, we need to see what happen inside this last call because it performs some other operations on the 48bytes at FFD31000, in more the 2 values that this function receive gives us a clue…it possible that it will be used like a decryption routine too…so I have named these two values: A and B , which are both 0 at the first call in this loop.

```
BC5E10FA        push ebp
                mov   ebp,esp
                push ecx
                mov   eax,[ebp+0C]          ; eax = pointer value A
                mov   ecx,[ebp+10]          ; ecx = pointer value B
                push ebx
                push esi
                mov   esi,[ebp+08]          ; esi=start 1048h bytes buffer FF3D1000
                mov   eax,[eax]             ; eax = value A
                mov   ebx,[ecx]             ; ebx = value B
                push edi
                lea   edi,[esi+00001000]    ; esi = FFD31000
                mov   dword ptr [ebp+08],00000010 ; Init counter (10h)
BC5E111B        xor   eax,[edi]             ; value A xored with dword in the range
                                            ; FFD31000-FFD31047 = MAGIC
                push eax                    ; MAGIC
                push esi
                mov   [ebp-04],eax          ; save MAGIC
                call BC5E10AA
                xor   eax,edx               ; eax = (New)MAGIC
                mov   ebx,[ebp-04]          ; ebx = (Old)MAGIC
                add   edi,04                ; inc pointer in the range FFD3D1000-
                                            ; FFD31047
                dec   dword ptr[ebp+08]     ; dec counter
                jnz   BC5E111B
```

```
mov  edx,[ebp+0C]          ; edx = pointer value A
mov  ecx,eax               ; ecx = FINAL (New) MAGIC
mov  eax,[esi+00001040]    ; eax = prelast dword in 1048h bytes
                           ; buffer = LOSTDWORD1
pop  edi
xor  eax,ecx               ; eax = (FINAL (New) MAGIC xor
                           ; LOSTDWORD1) = value B decrypted
mov  ecx,[esi+00001044]    ; ecx = last dword in the 1048h bytes
                           ; buffer = LOSTDWORD2
xor  ecx,ebx               ; eax = (FINAL (Old) MAGIC xor
                           ; LOSTDWORD1) = value A decrypted
mov  [edx],ecx             ; now value A is replaced with itself
                           ; decrypted
mov  ecx,[ebp+10]          ; pointer value B
pop  ebx
mov  [ecx],eax             ; now value B is replaced with itself
                           ; decrypted
leave
ret
```

Ok briefly, the routine above is nothing else that a decryption routine, but in this situation is used to generate 48h bytes to put in the 1048h bytes buffer in the range FFD31000-FF3D1047. Basically the two pointer that this routine receives are pointers to two dword values (A and B) to decrypt. At the first calling these values are 0 which bring the routine (after a loop of 10h times) to generate two dwords which are the entry values 'decrypted'. When this routine returns, these decrypted values replace the old two values A and B, which are resent down at this decryption routine to generate other two values A and B.
How you can see the decrypted values A and B are copy in the buffer in the last 48h bytes.
Anyway the code above is more clear than my words…just a thing we have to check in the above routine….the `call BC5E10AA`. This routine is responsable to generate a New MAGIC value…so surely it uses the current MAGIC and the 1048h bytes buffer…but to be sure just check it :

```
BC5E10AA    mov  ecx,[esp+08]          ; ecx = MAGIC
            mov  eax,000000FF
            mov  dl,cl
            push ebx
            and  edx,eax               ; edx = low byte of MAGIC
            shr  ecx,8                 ; shift right to have a new low byte
            mov  esi,edx               ; esi = first byte of MAGIC
            mov  cl,dl                 ; take the new low byte of MAGIC
            shr  ecx,8                 ; shift right to have a new low byte
            mov  ebx,ecx               ; ebx = 2 MAGIC high bytes
            and  edx,eax
            push edi
            mov  edi,edx               ; edi = second byte of MAGIC
            mov  edx,[esp+10]          ; start 1048h bytes buffer FF3D0000
            and  ecx,eax               ; ecx = third byte of MAGIC
            shr  ebx,8                 ; ebx = fourth (high) byte of MAGIC
            and  ebx,eax
            mov  eax,[ebx*4+edx]       ; take dword in 1048h bytes buffer
            add  eax,[ecx*4+edx+400]   ; another dword from 1048h bytes buffer
            movzx ecx,di               ; ecx = second byte of MAGIC
            pop  edi
            xor  eax,[ecx*4+edx+800]   ; eax = eax xor dword from 1048h bytes
                                       ; buffer
            movzx ecx,si               ; ecx = first byte of MAGIC
            pop  esi
            pop  ebx
            add  eax,[ecx*4+edx+C00]   ; eax = (New)MAGIC = eax xor dword from
                                       ; 1048h bytes buffer.
```

…Indeed…it use each bytes of the dword MAGIC to pick up dwords in the 1048h bytes buffer and build a new MAGIC value. Easy and clear ☺.
Ok now we can return at the code suspended at page 10:

→ We go on with the part of code suspended at page 10

```
                mov  esi,ebx                ; esi =start 1048h bytes buffer FFD30000
                mov  dword ptr [esp+10],00000004 ; Init counter (4)
BC5E1283        mov  edi,00000080           ; Init counter (80h)
BC5E1288        lea  eax,[esp+0C]           ; eax = pointer value A
                push eax
                lea  eax,[esp+08]           ; eax = pointer value B
                push eax
                push ebx
                call BC5E10FA
                mov  eax,[ebp+08]           ; eax = value A processed
                mov  [esi],eax              ; save the value A processed in the
                                            ; 1048h bytes buffer
                mov  eax,[ebp+0C]           ; eax = value B processed
                mov  [esi+04],eax           ; save the value A processed in the
                                            ; 1048h bytes buffer
                add  esi,8                  ; inc index in the 1048h bytes buffer
                dec  edi                    ; dec counter (80h)
                jnz  BC5E1288               ; processed all 80h (dwords) bytes ??
                                            ; not yet, next cycle. With A and B
                                            ; update too
                dec  dword ptr [ebp+10]     ; dec counter (4)
                jnz  BC5E1283               ; process (80h dwords)*4 ??
                                            ; not yet next 80h loop
                pop  edi
                pop  esi
                xor  eax,eax
                pop  ebx
                leave
                ret
```

This is a second loop similar at the previous (at page 10) the only difference here is that it process 80h*4 dwords (800h bytes) from the start of the 1048h bytes buffer.
With this last calculation dgcrypt has finished to fill the 1048h bytes buffer, which is ready to use in dgfs.sys.
So we return in dgfs.sys code just after the `call [eax+14]` that we have seen at page 8…cool…lets go on now from there because now dgfs access again at our Foo2.ith file:

```
BC1E69E2        call [eax+14]               ; call in dgcrypt.sys to fill the 1048h
                                            ; bytes buffer (FF3D0000)
                … …
BC1E69FC        movzx eax,word ptr[ebp-50] ; eax = size ansi string (28h)
                push eax
                mov  eax,[BC21AA20]
                push dword ptr [ebp-4C]     ; pointer to ansi string
                push dword ptr [ebp-34]     ; pointer at the start of the second
                                            ; 1048h bytes buffer (FF3D2000)
                call [eax+14]               ; call in dgcrypt.sys to fill the second
                                            ; 1048h bytes buffer (FF3D2000)
                … …
```

In the lines above dgfs.sys calls again dgcrypt to fill the second 1048h bytes buffer, in the same way as before…then…

```
BC1E6A49        mov  eax,[ebp-58]           ; eax = last dword of 18h bytes read =
                                            ; 80808087h
```

00	00	00	00	00	00	00	00	04	00	00	00	C1	08	00	00
00	00	00	00	87	80	80	80								

```
                and  eax,7F7F7F7E           ; eax = 5
                cmp  eax,10                 ; the last dword is like an ID to
                                            ; identify the type of the 18h block
                ja   BC1E766D
```

```
                movax eax, byte ptr[eax+BC1E7A5C] ; this ID is used like an index
                                                  ; in a jmp table.
                … …
BC1E6A68        mov  eax,[ebp-5C]                 ; eax = third dword in 18h bytes block
```

00	00	00	00	00	00	00	00	04	00	00	00	C1	08	00	00
00	00	00	00	87	80	80	80								

```
                mov  ecx,[ebp-64]                 ; eax = fifth dword in 18h bytes block
```

00	00	00	00	00	00	00	00	04	00	00	00	C1	08	00	00
00	00	00	00	87	80	80	80								

```
                add  eax,ecx                      ; add the 2 dwords = 4 + 0 = 4
                sub  [ebp+08],eax                 ; sub the value at the offset of the
                                                  ; last reading in .ith 4F9-(4+0) = 4F5
                jmp  BC1E766D
                … …
BC1E766D        sub  dword ptr[ebp+08],18         ; sub 18h bytes (block size) at the
                                                  ; offset in .ith. 4F5-18=4DD = new
                                                  ; offset in .ith
                … …
                lea  eax,[ebp-30]
                push eax
                push 18
                push dword ptr [ebp+08]
                push dword ptr [ebp-28]
                push dword ptr [ebp-10]
                call BC1E794A                     ; read from Foo2.ith (log num=56)
                                                  ; Offset = 4DDh
                                                  ; Size = 18h
```

52	D8	E0	48	5A	F3	2B	20	50	E6	A7	C4	4C	E1	99	AC
AE	A1	44	CA	B2	2F	30	0B								

```
                test eax,eax
                jz   BC1E794A
                mov  esi,[ebp-28]                 ; esi = pointer 18h bytes read
                mov  eax,[BC21AA20]
                push 03
                push esi
                push dword ptr [ebp-14]           ; pointer 1048h bytes buffer FF3D0000
                call [eax+24]                     ; call dgcrypt.sys to decrypt the 18h
                                                  ; bytes block read (03*2 dwords).
```

00	00	00	01	00	00	00	00	00	00	00	00	C1	08	00	00
20	00	00	00	85	80	80	80								

```
                push 06
                lea  edi,[ebp-6C]
                pop  ecx
                repz movsd                        ; copy the 18h bytes block decrypted in
                                                  ; another buffer.
                cmp  dword ptr [ebp-5B],80808081 ; check in 18h bytes decrypted if
                                                  ; last dword is 80808081.
```

00	00	00	00	01	00	00	00	00	00	00	00	C1	08	00	00
20	00	00	00	85	80	80	80								

```
                jnz  BC1E6A49
```

```
            … …
BC1E6A49    mov  eax,[ebp-58]          ; eax = last dword of 18h bytes read =
                                       ; 80808085h
```

00	00	00	00	01	00	00	00	00	00	00	00	C1	08	00	00
20	00	00	00	85	80	80	80								

```
            and  eax,7F7F7F7E          ; eax = 3
            cmp  eax,10                ; the last dword is like a ID to
                                       ; identify the type of the 18h block
            ja   BC1E766D

            movax eax, byte ptr[eax+BC1E7A74] ; this ID is used like an index
                                       ; in a jmp table.
            jmp  [eax*4+BC1E7A5C]
            … …
            push F0                    ; -10h
            pop  eax                   ; eax = -10h
            sub  eax,[ebp-64]          ; eax = -10h - 0
            add  [ebp+08],eax          ; add -10h at the previous offset
                                       ; 4DD + (-10) = 4CD new offset in .ith
            cmp  [ebp+14],ebx
            jz   BC1E6BFF
            … …
BC1E6BFF    lea  eax,[ebp-30]
            push eax
            push 10
            push dword ptr [ebp-30]
            lea  eax,[ebp-00E0]
            push eax
            push dword ptr [ebp-10]
            call BC1E794A              ; read from Foo2.ith (log num=57)
                                       ; Offset = 4CDh
                                       ; Size = 10h
```

33	29	4E	19	78	11	FF	47	6C	A2	A6	9F	BC	4F	67	7F

```
            test eax,eax
            lea  eax,[ebp-00E0]
            push 02
            push esi
            push dword ptr [ebp-14]    ; pointer 1048h bytes buffer FF3D0000
            call [eax+24]              ; call dgcrypt.sys to decrypt the 10h
                                       ; bytes read (02*2 dwords).
```

74	A1	A3	1C	08	4E	BA	70	DE	AB	03	BD	FE	6F	AF	17

```
            push 10
            mov  dword ptr [ebp-00A4],00000001
            pop  eax                   ; eax = 10h
            sub  eax,[ebp-5C]          ; eax = 10h - 20h (fifth dword in the
                                       ; 18h bytes block) = -10h
            add  [ebp+08],eax          ; 4CD + (-10h) = 4BD new offset in .ith
            jmp  BC1E766D
            … …
BC1E766D    sub  dword ptr[ebp+08],18  ; sub 18h bytes (block size) at the
                                       ; offset in .ith. 4BD-18=4A5 = new
                                       ; offset in .ith
            … …


            lea  eax,[ebp-30]
            push eax
            push 18
            push dword ptr [ebp+08]
            push dword ptr [ebp-28]
            push dword ptr [ebp-10]
```

```
        call BC1E794A               ; read from Foo2.ith (log num=58)
                                    ; Offset = 4A5h
                                    ; Size = 18h
```

5F	D8	E0	48	5A	F3	2B	20	9E	BA	20	84	66	41	96	2A
E1	42	56	C9	9D	9D	9E	6C								

```
        test eax,eax
        jz   BC1E794A
        mov  esi,[ebp-28]           ; esi = pointer 18h bytes read
        mov  eax,[BC21AA20]
        push 03
        push esi
        push dword ptr [ebp-14]     ; pointer 1048h bytes buffer FF3D0000
        call [eax+24]               ; call dgcrypt.sys to decrypt the 18h
                                    ; bytes block read (03*2 dwords).
```

00	00	00	00	01	00	00	00	01	00	00	00	C1	08	00	00
50	03	00	00	84	80	80	80								

```
        push 06
        lea  edi,[ebp-6C]
        pop  ecx
        repz movsd                  ; copy the 18h bytes block decrypted in
                                    ; another buffer.
        cmp  dword ptr [ebp-5B],80808081 ; check in 18h bytes decrypted if
                                    ; last dword is 80808081.
```

00	00	00	00	01	00	00	00	01	00	00	00	C1	08	00	00
50	03	00	00	84	80	80	80								

```
        jnz  BC1E6A49
        … …
BC1E6A49    mov  eax,[ebp-58]       ; eax = last dword of 18h bytes read =
                                    ; 80808084h
```

00	00	00	00	01	00	00	00	00	00	00	00	C1	08	00	00
20	00	00	00	84	80	80	80								

```
        and  eax,7F7F7F7E           ; eax = 2
        cmp  eax,10                 ; the last dword is like a ID to
                                    ; identify the type of the 18h block
        ja   BC1E766D


        movax eax, byte ptr[eax+BC1E7A5C] ; this ID is used like an index
                                    ; in a jmp table.
        jmp  [eax*4+BC1E7A5C]
        … …
        mov  esi,350
        push esi
        call BC1F0E16               ; allocate a 350h bytes buffer
        … …
        mov  eax,[ebp-64]           ; eax = third dword in 18h byte block
                                    ; 00000001
        sub  [ebp+08],eax           ; 4A5 - 1 = 4A4 new offset in .ith
        … …
        lea  eax,[ebp-30]
        push eax
        push dword ptr [ebp-64]
        push dword ptr [ebp+08]
        push edi
        push dword ptr [ebp-10]
```

```
        call BC1E794A               ; read from Foo2.ith (log num=59)
                                    ; Offset = 4A4h
                                    ; Size = 1
```

20

```
        … …
        sub  [ebp+08],esi           ; 4F4 - 350 (fifth dword in 18h byte
                                    ; block) = 154 new offset in .ith
        … …
        lea  ecx,[ebp-30]
        push ecx
        push esi
        push dword ptr [ebp+08]
        push eax
        push dword ptr [ebp-10]
        call BC1E794A               ; read from Foo2.ith (log num=60)
                                    ; Offset = 154h
                                    ; Size = 350
```

1F	08	94	8E	E3	2E	1F	8B	7C	70	BE	33	CE	C4	AD	06
4E	9D	3B	C0	16	41	B6	F6	4E	9D	3B	C0	16	41	B6	F6

… … … …

```
        test eax,eax
        jz   BC1E794A
        mov  esi,[ebp-04]           ; esi = pointer 18h bytes read
        mov  eax,[BC21AA20]
        push 6A
        push esi
        push dword ptr [ebp-34]     ; pointer 1048h bytes buffer FF3D0000
        call [eax+24]               ; call dgcrypt.sys to decrypt the 350h
                                    ; bytes read (6A*2 dwords).
```

01	00	00	00	43 C	3A :	5C \	46 F	6F o	6F o	32 2	2E .	74 t	78 x	74 t	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

… … … …

```
        … …
BC1E766D sub  dword ptr[ebp+08],18  ; sub 18h bytes (block size) at the
                                    ; offset in .ith. 154-18=13C = new
                                    ; offset in .ith
        … …
        lea  eax,[ebp-30]
        push eax
        push 18
        push dword ptr [ebp+08]
        push dword ptr [ebp-28]
        push dword ptr [ebp-10]
        call BC1E794A               ; read from Foo2.ith (log num=61)
                                    ; Offset = 13Ch
                                    ; Size = 18h
```

5F	D8	E0	48	5A	F3	2B	20	50	E6	A7	C4	4C	E1	99	AC
61	48	B9	F1	F1	C3	31	DB								

```
        test eax,eax
        jz   BC1E794A
        mov  esi,[ebp-28]           ; esi = pointer 18h bytes read
        mov  eax,[BC21AA20]
        push 03
        push esi
        push dword ptr [ebp-14]     ; pointer 1048h bytes buffer FF3D0000
```

```
        call [eax+24]               ; call dgcrypt.sys to decrypt the 18h
                                    ; bytes block read (03*2 dwords).
```

00	00	00	00	01	00	00	00	00	00	00	00	C1	08	00	00
0C	00	00	00	83	80	80	80								

```
        push 06
        lea  edi,[ebp-6C]
        pop  ecx
        repz movsd                  ; copy the 18h bytes block decrypted in
                                    ; another buffer.
        cmp  dword ptr [ebp-5B],80808081 ; check in 18h bytes decrypted if
                                    ; last dword is 80808081.
```

00	00	00	00	01	00	00	00	00	00	00	00	C1	08	00	00
0C	00	00	00	83	80	80	80								

```
        jnz  BC1E6A49
        … …
BC1E6A49    mov  eax,[ebp-58]       ; eax = last dword of 18h bytes read =
                                    ; 80808083h
```

00	00	00	00	01	00	00	00	00	00	00	00	C1	08	00	00
0C	00	00	00	83	80	80	80								

```
        and  eax,7F7F7F7E           ; eax = 3
        cmp  eax,10                 ; the last dword is like a ID to
                                    ; identify the type of the 18h block
        ja   BC1E766D


        movax eax, byte ptr[eax+BC1E7A74] ; this ID is used like an index
                                    ; in a jmp table.
        jmp  [eax*4+BC1E7A5C]
        … …
        mov  ecx,[ebp-5C]           ; ecx = third dword in 18h bytes block
                                    ; 00000000
        mov  edx,[ebp-64]           ; ecx = third dword in 18h bytes block
                                    ; 0000000C
        sub  [ebp+08],edx           ; 13C - 0C = 130 new offset in .ith
        … …
        lea  eax,[ebp-30]
        push eax
        push dword ptr [ebp-64]
        push dword ptr [ebp+08]
        push edi
        push dword ptr [ebp-10]
        call BC1E794A               ; read from Foo2.ith (log num=62)
                                    ; Offset = 130h
                                    ; Size = 0Ch
```

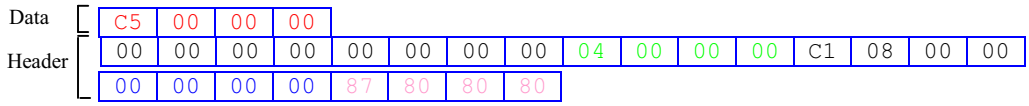
F1	55	AD	F6	BB	4C	D0	31	72	6C	64	21

```
        … …
        mov  eax,esi                ; eax = 0C num bytes read
        shr  eax,3                  ; eax = 0C / 8 = 1
        push eax
        mov  eax,[BC21AA20]
        push dword ptr [ebp-04]     ; pointer 0C bytes to decrypt
        push dword ptr [ebp-14]     ; pointer 1048h bytes buffer FF3D0000
        call [eax+24]               ; call dgcrypt.sys to decrypt the 0Ch
                                    ; bytes read
```

48	65	6C	6C	6F	20	77	6F	72	6C	64	21
H	e	l	l	o		W	o	r	l	d	!

Yeppa!!!… got it…our clean text is in the buffer above…how we have supposed when we have analysed the logged reading operations some pages above, the read operation number 62 read 0Ch bytes, which are our text encrypted.
So…just we need to put all together…to have a complete picture of how is made an .ith file…

An .ith file is marked in the last four bytes with `D4C3B2A1`, which probably defined the version too. The rest of the .ith file is made of blocks of information, each information block is made of an header (18h bytes) and a body (data). But lets see an example :



`87808080` The last dword in the 18h bytes header block is some kind of "ID". It's used has an index in a jmp table, so each block is treath in the specific way inside dg32.exe/dgfs.sys. The first information block (basically the the first to read just before the dword marker) is the 87808080 (in the example above). This information block is not encrypted because the data that it store (4 bytes) are used to generate the decryption key (1048h bytes buffer) to decrypt the others information blocks in the .ith file. Other type of blocks are:

85808080	10h+10h data (we haven't analysed)
84808080	Path_FileName and others data
83808080	Original File
82808080	We haven't analysed
81808080	We haven't analysed, It mark the end of the information blocks too.

`00000000` (THIS IS MY PERSONAL INTERPRETATION – WILL BE GREAT IF INFRAWORKS GUYS CAN CONFIRM OR CORRECT MY DEDUCTION. THANKS)
The fifth dword in the 18h bytes block header is the size of the encrypted data that precede the header. In this information block the value is 00000000, no encrypted data here.

`00000004` (THIS IS MY PERSONAL INTERPRETATION – WILL BE GREAT IF INFRAWORKS GUYS CAN CONFIRM OR CORRECT MY DEDUCTION. THANKS)
The third dword in the 18h bytes block header is the size of the not-encrypted data that precede the header. In this information block the value is 00000004, so 4 bytes of not-encrypted data (000000C5).

`000000C5` The data of the information block.

The meaning of the others fields are off-topic for this tutorial, which is already long enough…but now you easly study any part of an .ith file and experiment others protection options too.
I think that I can close this second part here…now it's a piece a of cake write a simple program to automate the 'extraction' of our Foo2.txt file…☺.

Don't limit yourself to crack a protection….reverse it …Be a curious reverser…crack a protection is not the final goal for a reverser, but the excuse to start …then you can free your brain.

A BIG THANKS TO :

InTether developing group, for the funny nights that I have spent on their protection.
It's been the first No-boring reversing session after a long period. All my respect.

…see ya in the next tut.

MaV3RiCk

maverickluke@hotmail.com