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

Part 1

Back from a long period of silence with a tutorial for all (serious) crackers and reversers that they don't want to waste their time to play with kiddy packers/crypters.
Like always this is a reversing tutorial, so if you're looking for only a way to crack InTether protection…you have opened the wrong one…and probably you are a not a reverser too. I'm sorry.
In the title I have defined InTether protection like the "perfect reversing training field", this because with a real reversing approach it possible have a lot of fun coding tools to better understand not only how this protection works but also how parts of our OS works too.
The tutorial is made of 2 parts because it's quite long and because I want to give you the approach that I have used here, with the background too.
Lets start!

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

First a briefly description of the protection that it has given me the inspiration.

InTether protection system (www.infraworks.com)

This is an extract of the help file, which explain better what is and how it works:

" *With InTether Desktop, you can control exactly who receives your data and what happens to your file once it is received. Using revolutionary technology, InTether Desktop offers the following advantages:*

- *Most secure Digital Property Protection technology available.*
- *Multiple layers of defense and active file access monitoring.*
- *Performance that is independent from file content and content transfer method.*
- *Compatibility and ease-of-use with existing software, readers and players.*
- *Restricted access to InTethered files by using sender-established permissions such as time/date to view and time to delete the document.*
- *Prevention against any unauthorized changing, printing, dissemination, forwarding or screen-capturing of InTethered content.*
- *A self-destruct mechanism that is imposed on those attempting to hack InTethered content.*

… …

- *Content owners will use InTether Desktop's Packager to package their file(s) with customized permission. The package can then be sent to any recipient. InTethered packages can be delivered via email, web, FTP, through shared network drives and on CD-ROM.*
- *Recipients of InTethered content must have the InTether Receiver to read the secured file(s). The InTether Receiver not only allows access to InTethered content, but in extreme cases, the InTether technology will delete the file and any remnants of the InTether Received technology if it detects brute force or unauthorized access attempts to the dat. "*

Simple. In few words it permits to protect any static data (documents, images, movies etc) applying around them a custom protection (limit time to view the file, limit number of trials, or the classic password protection, etc). Someone can say "…Oh yes another pgp clone!" …and in part it's true but there is something in the above words that push Intether ahead, or at least this is what they want to show. So the point that has stimulated my brain was this:

" Prevention against any unauthorized changing, printing, dissemination, forwarding or screen-capturing of InTethered content."

Hmm…this is what I define a coding challenge! Cool! Realise this in a serious and secure way (against the crackers/reversers long hands) is not an easy task. I mean protect a document file with a good encryption algorithm it's easy and (if well done) secure , pgp or kremlin are good example. There is no point to discuss the encryption algorithm used in InTether when it works like a pgp clone and there is no point to reverser that. Instead what is really interesting is to see how in deep they have integrated the protection to prevent (a malicious user like me) doing a copy of a protected document and freely printing,moving,changing or screen capturing it. So lets see this protection:

After the installation we have several links on our desktop:

- **InTether Packager** : it's the main IDE. It creates an InTether protected copy of a data file.
- **InTether Content Manager** : Permit to manage all InTether files active on our disk, basically any opened protected file on our system.
- **InTether UID manager**: Permit to manage all UID (unique reader user identifier) when we want to protect a file and permit only at the specific user (UID) to view the file.
- **Packaged Content directory**: By default all our created InTether files are saved here.
- **Received Content directory**: By default when we open a InTheter protected file it'll be saved in this directory.

Ok, now our interest is obviously on InTether Package IDE to create our first protected copy of a txt file.
First we create a simple txt file called FooA.txt and with a stupid text inside:

Hello World from FooA!

So now we have a file to protect, FooA.txt, with a size of 22bytes. It's important keep in mind every single detail of what we are doing… later you'll understand why.
We run the InTheter Package IDE and we select our text file to protect, we leave the default package saving path and we select only the option "File may be read for…" and we set the time at 10mins. We don't use any other option in this tutorial for now. Then we click on "Create Package". Now our text file is protected. If we open the directory Packaged Content, we have our protected file FooA.ith
OK now lets see what happen when we run our protected file.
A nagscreen tells us the main characteristics of this file, type of protection (time limit) and where we want to save our file on disk . We save it in the root and we click on "Save&open".
Now our txt file is saved in the root directory with its original name FooA.txt and then it is open inside notepad.
If now we close notepad and we open the file FooA.txt just saved, a nagscreen this time tell us how much time we still have to view the file and then if we click on the "open" button the file will be open in notepad.
If we leave the file open for more than 10 mins, automatically notepad will be closed and our FooA.txt will be deleted from disk. But for now we don't want that so close it before that the 10mins expire.
We need to see now if we can do a copy of this file…In the shell, if we try to copy the file in another directory the result is only a file empty (0 bytes), same if inside notepad we try to save the file.
So we can distinguish two situations here, one, when we try to do a copy of our protected file in a shell (explorer) and second, when we try to make a copy using the "save" option inside a program.
In both situation, from a system point of view, the copy operation is the result of a series of I/O requests directed to the file system driver so for the system doesn't exist difference. InTether has needed to monitor any I/O request sent by our text editor and filter the writing requests to prevent us to save a copy of the protected file.
But for InTether it's not enough doing this because two are the situations that we have seen …and prevent a copy (write requests) from inside a shell mean filter any write operations that our shell sends to the file system driver, basically a disaster!
So during our first opening on a protected file, InTether save the file with 0 byte basically our FooA.txt is empty and the real content is stored in another place (obviously encrypted ☺).

Doing this, we can't make a copy in anyway even if InTether filters are unloaded . In more InTether avoids the problem of filtering any window shell (explorer.exe) write operation.
Ok but this is only my logical deduction and we have nothing in our hands to prove this especially because if we look at the properties of our FooA.txt, explorer reports the real size 22 bytes and not 0. But don't forget that any user operation on a file is nothing else that an I/O request sent to the file system driver and InTether filters can trap it and fake it!.
Anyway…the best thing now is to have a confirmation about my theory so we need to check if the file FooA.txt is really 0 bytes, in this way we can take the right direction to find our real file and make it a copy.

The only way to be sure at 100% about our theory is read directly the file from the disk…I mean a "raw access" to see exactly what is really present on our NTFS disk structure for FooA.txt file. In more we can reduce at the minimum any InTether disturb (filtering/faking and who know what else ☺)
Here is required a little theoretical part on the NTFS on-disk data structures…because today I don't think that there are anymore virus writers that use on-disk data structures directly…☺

Ok briefly, our disk is divided in sectors, which are addressable blocks of disk (on x86 the size of each sector is 512bytes).
Clusters are the number of sectors that the system can manage. NTFS internally use only clusters .
So if I have a file of 1050 bytes to save on disk, and the disk has a cluster size of 1 (1 sector x cluster) ,NTFS file system use 3 clusters to save my 1050 bytes or 3 sectors.
NTFS uses Logical Cluster Numbers (LCNs) when it refers at physical location on disk. All clusters, from the start to the end of our disk, are numbered and these numbers are the LCNs.
Basically if we have a LCN and we know the cluster size and the sector size we can calculate the physical disk address.

PhysAddr = LCN * ClusterSize * SectorSize

PhysAddress are used by the disk driver interface.
Internally NTFS use Virtual Cluster Number (VCNs) to number the clusters within a file. Basically VCNs number the clusters in a particular file from 0 to the end of the file. VCNs can be mapped at any LCNs on a disk, this mean that the VCNs of a file can be not necessarily physically contiguous.

Another important NTFS on-disk structure is the Master File Table (MFT). In NTFS all data stored on a disk are presented as files.
The MFT is an array of FileRecords. The size of a FileRecord is stored in the boot sector and on win2K it's always 1Kb.
A filerecord is the 'description' of the file, basically each entry in the MFT describe a file or a directory or a metadafile.
The first 12 entries are fixed and describe NTFS metadafile, 4 are reserved and all the others are used for files and directories. Here a simple scheme of the MFT:

$Mft (MFT)
$MftMirr (MFT mirror)
$LogFile (Log file)
$Volume (Volume file)
$AttrDef (Attribute definition table)
\ (Root directory)
$Bitmap (Vol cluster allocation file)
$Boot (Boot sector)
$BadClus (Bad cluster file)
$Secure (Security settings file)
$UpCase (Uppercase char mapping)
$Extend (Extended metadata dir)
Not used
Not used
Not used
Not used
User file and directory

The first 12 MFT metada files have a name that begins with $. The first MFT entry is the file record for the MFT itself.
I don't explain what are these metada files because we go off topic and they are not important for this tutorial. But I have to explain what are and how are made up in general these MFT filerecord entries especially a MFT filerecord entry which describes an user file.
From a general point of view a MFT entry is made of a header and a series of Attributes. Each of these Attributes describes the file/directory/metadata file that the filerecord refer to.
Included with this tutorial there is the NTFS.inc file which has the definition of some Attributes useful for this tutorial.
Each Attribute can be "resident" or "not resident" . If an Attribute is resident means that all the attribute values are stored inside the attribute structure. If an Attribute is not resident means that some values are stored outside the MFT entry and the Attribute stores the LCNs of the clusters where are stored the values of this non resident attribute. This happen when an Attribute stores values with a big size…a classic example is the AttributeData. For ex. the AttributeData in a MFT entry (that describe a file) is the attribute that stored the real data for a file…the binary image of an executable, or the text of a text file, or the binary image a jpg etc. How you can understand because of the limited size of each MTF entry (1Kb on win2K)…often the data of a file can't fit in its MFT entry…so NTFS creates the AttributeData 'non resident' and it store the LCNs of each cluster used to store the data of this file.
But better to see a real example…I suggest to use the ntfs.inc file to understand better each byte in the following MFT entry:
This is the MFT entry of the file BOOTLOG.TXT on my harddisk:

```
00000000 4649 4C45 2A00 0300 925A 5201 0000 0000 FILE*....ZR.....
00000010 1100 0100 3000 0100 D801 0000 0004 0000 ....0...........
00000020 0000 0000 0000 0000 0400 6100 0000 0000 ..........a.....
00000030 1000 0000 4800 0000 0000 1800 0000 0000 ....H...........
00000040 3000 0000 1800 0000 0049 130F 57A7 C101 0........I..W...
00000050 0049 130F 57A7 C101 8030 C481 A6A7 C101 .I..W....0......
00000060 0000 2290 C5A6 C101 0600 0000 0000 0000 ..".............
00000070 0000 0000 0000 0000 3000 0000 7000 0000 ........0...p...
00000080 0000 1800 0000 0100 5800 0000 1800 0100 ........X.......
00000090 0500 0000 0000 0500 0049 130F 57A7 C101 .........I..W...
000000A0 0049 130F 57A7 C101 0049 130F 57A7 C101 .I..W....I..W...
000000B0 0000 2290 C5A6 C101 006A 0000 0000 0000 ..".....j......
000000C0 E268 0000 0000 0000 2200 0000 0000 0000 .h......".......
000000D0 0B03 4200 4F00 4F00 5400 4C00 4F00 4700 ..B.O.O.T.L.O.G.
000000E0 2E00 5400 5800 5400 5000 0000 9800 0000 ..T.X.T.P.......
000000F0 0000 1800 0000 0200 7C00 0000 1800 0000 ........|.......
00000100 0100 0480 5C00 0000 6C00 0000 0000 0000 ....\...l.......
00000110 1400 0000 0200 4800 0300 0000 0003 1400 ......H.........
00000120 FF01 1F00 0101 0000 0000 0001 0000 0000 ................
00000130 0000 1400 FF01 1F00 0101 0000 0000 0005 ................
00000140 1200 0000 0000 1800 FF01 1F00 0102 0000 ................
00000150 0000 0005 2000 0000 2002 0000 0102 0000 .... ... .......
00000160 0000 0005 2000 0000 2002 0000 0102 0000 .... ... .......
00000170 0000 0005 2000 0000 2002 0000 0000 0000 .... ... .......
00000180 8000 0000 5000 0000 0100 4000 0000 0300 ....P.....@.....
00000190 0000 0000 0000 0000 3400 0000 0000 0000 ........4.......
000001A0 4000 0000 0000 0000 006A 0000 0000 0000 @........j......
000001B0 E268 0000 0000 0000 E268 0000 0000 0000 .h.......h......
000001C0 3120 7290 0021 15B0 1200 0000 0000 0000 1 r..!..........
000001D0 FFFF FFFF 0000 0000 0000 0000 0000 0000 ................
000001E0 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

How you can see the offsets are relative at the start of the MFT entry for better reading.
Lets go to analyse this MFT filerecord entry for the file Bootlog.txt:

Each MFT filerecord start with a NTFS_RECORD_HEADER structure (see the ntfs.inc for the meaning of each byte).

Just after there is the FILE_RECORD_HEADER structure (see the ntfs.inc file for the meaning of each byte).

At the offset 30h (from the start of the MFT filerecord) there is the first Attribute for our file. Each Attribute start with an ATTRIBUTE structure, which is common for resident attribute and for not resident attribute (see the ntfs.inc file for further details). The first dword (AttributeType) defines the type of the Attribute that follow and it can be one of these values:

00000010h	AttributeStandardInformation
00000020h	AttributeAttributeList
00000030h	AttributeFileName
00000040h	AttributeObjectId
00000050h	AttributeSecurityDescriptor
00000060h	AttributeVolumeName
00000070h	AttributeVolumeInformation
00000080h	AttributeData
00000090h	AttributeIndexRoot
000000A0h	AttributeIndexAllocation
000000B0h	AttributeBitmap
000000C0h	AttributeReparsePoint
000000D0h	AttributeEAInformation
000000E0h	AttributePropertySet
000000F0h	AttributeLoggedUtilityStream

So for our MFT entry the first attribute is an AttributeStandardInfomation.
Another Important value inside this ATTRIBUTE structure is the NonResident byte, which tells us if the attribute data is resident inside the MFT entry or external. In our bootlog.txt the first Attribute has this byte 0 so the attribute value is resident.

After the ATTRIBUTE structure we can have or a RESIDENT_ATTRIBUTE if the NonResident flag is 0, or a NONRESIDENT_ATTRIBUTE structure if the NonResident flag is 1. For both structures see the ntfs.inc.
In our example we have that the first Attribute is resident so just after the ATTRIBUTE strucure we have a RESIDENT_ATTRIBUTE

Then we have the data of the specific Attribute. For the first Attribute we have the AttributeStandardInformation. (see ntfs.inc file).

This for any Attribute in the MFT filerecord.
So each Attribute is made of an ATTRIBUTE (–) structure plus or a RESIDENT_ATTRIBUTE (–) or a NONRESIDENT_ATTRIBUTE (–) and the data structure of the specific attribute (–).

But exactly what is stored in the Attribute if the Attribute is marked NonResident ?…lets see our example…we have the last Attribute, at offset 180h ,defined as NonResident and we have a NONRESIDENT_ATTRIBUTE. The type of the Attribute is 00000080h (AttributeData) which is the real data of our file, basically the text stored in the bootlog.txt. Often the real data of a file can't fit in an 1kb MFT entry (like for this file) so what is stored inside the Attribute in the MFT are the LCNs (logical cluster numbers) with the size of each block of clusters, but take the example above, at offset 1C0h we have the resident part of the Attribute:

```
000001C0 3120 7290 0021 15B0 1200 0000 0000 0000 1 r..!..........
```

These bytes have this meaning; the first byte (31h) defines, in the low nibble (4 bits), how many bytes we have to take (just after this value) to have the number of LCNs used to store the text of our bootlog.txt. The high nibble (4 bits) defines how many bytes we have to take (after the bytes of the size) to have the starting LCN.
So we have:

31h = 0011 0001 b → LowNibble = 1 / HighNibble = 3

1 is the number of bytes to take after the 31h to have the number of cluster……20h
3 is the number of bytes to take after the size bytes to have the starting LCN,in our case are … 009072h which define a:

block of clusters : LCNs 009072h to 009092h

This means that at the LCN 009072h on our disk, we have 20h clusters that store our text. But it's not finished…after the 3 bytes of the LCN we have another value used like before…21h…so again here the Low nibble is the number of bytes that we have to take to have the number of clusters BUT the high nibble is a 'delta' value to add at the highest LCN resulting from the previous block. Basically from the previous block we had

LCNs 009072h to 009092h

So we have :

21h = 0010 0001 b → LowNibble = 1 / HighNibble = 2

1 is the number of bytes to take after the 31h to have the number of cluster…15h
2 is the number of bytes to take after the size bytes to add a the LCN 009092h, and in our case are … 12B0h
Which define a :

second block of clusters :LCNs 00A342h to 00A357h

This last operation is performed until we reach a 00 bytes…which mean no more used clusters.
Basically if we read 20h cluster from the LCN 009072h and 15h clusters from the LCN 00A342h we have our bootlog text.

Important is that the delta is a signed value so if the highest bit is set the delta is a negative value.
Now probably you can image how a defragmentation utility works…more blocks we have for a file and more this file is fragmented ☺.
A MFT filerecord entry don't have a fix number of Attributes and the type of these Attributes depend of the characteristic of the file that it describe …files, directories or metadafile.
A dword FFFFFFFFh in the Attribute.Type field , marks the end of the Attributes for that MFT filerecord. (ex. for our bootlog.txt filerecord we have the end of the Attributes at the offset 1D0h).
A final note, how I have said each file, directory or metadata file on our harddisk has a filerecord entry in the MFT, so everytime we save a new file or we create a new directory , the file system try to find the first filerecord not used (bit 0 in FILE_RECORD_HEADER.Flags clear) to use for the new file. Only if there aren't 'not-used' filerecords…NTFS allocates a new MFT filrecord entry.

Ok we can stop here with these NTFS stuff because this is enough for the purpose of this tutorial…and I can't write a tutorial to explain all NTFS on-disk data structures …It's not Christmas time ☺.
Now it's time to write some code to access to our protected text file …do you remember our FooA.txt ? …cool now it's time to read it directly from its MFT filerecord and avoid to have InTether playing dirty filtering tricks…so we can see if my theory is correct….InTether save FooA.txt empty just to say "…hey user! if you want to open this protected file just double click here that after I take care to find it and show it to you for the limited time ☺ ".

Ok attached at this tutorial there is the NTSF.ASM source code (and the compiled version), which is well commented and very simple. It gives us 3 options:

1. Dump the entire MFT (we see later for what it's useful)
2. Dump a file; basically make a copy of the file using the info that I have explained above on the MFT filerecord entries. It requires a MFT entry number, from a MFT dumping (option 1).
3. Dump a MFT filerecord entry. It requires a MFT entry number, from a MFT dumping (option 1).

First thing to do is make a dumping of our MFT, so just click on the "Dump MFT entries…" after few seconds we'll have a file in the same directory called MFTEntries.txt, with each MFT filerecord. We search in the list our FooA.txt and we take note of the MFT entry number. Now we insert the number in the edit box and we click on "Dump MFT entry at…". In the same directory we'll have a XXXX.txt (where is xxxx is the MFT number) file with a binary dumping of the FooA.txt MFT entry.
Now you know what these bytes means ☺ ….what interest us is the Attribute AttributeData (type = 00000080h). I'm sure you can locate it easly now…. How we can see this Attribute is Resident but nothing is stored inside and no definition of clusters blocks too (the attribute is Resident). So this file is 0 bytes for real !! And when in explorer we look at the propriety of FooA.txt , InTether fake the result (fake the I/O request) and return at the shell the size of the file which is obviously taken from another place on our hard disk.

Now the question is ….Where is saved our FooA.txt file ?

Meanwhile you call Sherlock Holmes….I go in the Toilet ☺. Bloody Stella…drink yep! Another!

Back!…ok the first (and logical) thing that comes in mind is this: everytime we run an InTether file protected for the first time, InTether save a dummy copy with its name and size 0 and another copy hidden in same place with the real file (probably encrypted) .
When we open the dummy file, InTether traps this and redirect the reading operation on this hidden file.
So we have to verify this…and to do this we can use our fresh NTFS utility (if you don't like you can use any other monitoring tools and skip over to next part but this is a reversing tutorial so if you want to learn something follow it in every part because all has a sense ☺.
First thing to do is protect another file…we create a simple txt file named FooB.txt with a stupid text inside:

Hello Reversing World from FooB!

Now the size of the file is 32 bytes.
Then we protect the txt file with InTether packager and the same option used before for the FooA.txt file. So time limit of 10mins. At this point we close all applications and we use our NTFS.EXE to do a dumping of our entire MFT.
NTFS utility creates a MFTEntries.txt file in the same directory…now we open the FooB.ith file and in the nagscreen we select "open&save" …after that notepad has opened the text file….again with NTFS utility we

create a dumping of our entire MFT . OK Now we have 2 MFTEntries.txt file…one before that InTether has saved&opened FooB.txt file and one after.
Now if we compare these two files we can have a good picture of all new files that have been created during the InTether save&open operation. I suggest to use "compareit" a very good shareware comparing tool…so this is the result on my harddisk:

Only 3 MFT filerecords has been updated, lets see in details:

895835136 FILE 0 0 PACKAG~1.CFG 895835136 FILE 1 0 MFTENT~1.TXT

The MFT filerecord for the file PACKAG~1.CFG wasn't used when we have taken the first dump and the file itself wasn't "officialy" present on disk but deleted. So windows has used this MFT filerecord to store the filerecord for the new file MFTENT~1.TXT (MFTEntries.txt). Nothing of suspected here ☺
Lets take the second:

895847424 FILE 1 0 FooB.ith 895847424 FILE 0 0 FooB.ith

During the first dumping of our MFT entries, we had the protected file FooB named FooB.ith. After that InTether has saved&open our text file, it has deleted the protected version FooB.ith. So the MFT filerecord is changed from "Used" to "NotUsed". Nothing of suspected here too ☺
So give a look at the third:

895853568 FILE 0 1 is-8JCDO.tmp 895853568 FILE 1 0 FooB.txt

The MFT filerecord for the file is-8JCDO.tmp wasn't used like for the file packag1.cfg above, So windows has used this MFT filerecord to store the filerecord for the new file FooB.txt, our protected text file… …"empty". Again …nothing of suspected here ☺.

Hmm…hmm…so InTether doesn't create a hidden file everytime we run an .ith file…but use one single file probably hidden and created during the installation or at the first run of the InTether Reader.
Cool now we have a better picture of what happen …it's time to look more near at these InTether filter drivers. We know that there is one file system filter driver for sure so we need to know the name of it because I suspect that it's not the only kernel mode module of the InTether protection.
Time for sice now! We load the ntdll.dll export in sice and we set bpx on NtWriteFile, we leave sice and straight away sice breaks on the NtWriteFile in the ntoskrnl.exe.
Now to jump inside the file system driver we can remove the bpx on NtWriteFile and set a bpx on IofCallDriver…in few word the NtWriteFile is the exported kernel mode function called indirectly from the user mode WriteFileA api function (or directly in kernel mode from kernel mode code)…ntoskrnl in turn calls the file system driver with IofCallDriver…so we leave sice and straight away sice break inside IofCallDriver which call the right routine inside the file system…which should be the Ntfs.sys (windows NTFS file system driver). We step inside IofCallDriver until we reach :

```
call  [eax*4+ecx+38]
```

we step in and we should be inside the Ntfs.sys…but because InTether has put its own file system filter driver we are now inside…. gdfs.sys …got it!

Ok now we have the name…and because I'm a curious guy…we go inside our windows registry to give it a look. Indeed under the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services we find, not only gdfs.sys registered but also a series of its "friends" ☺ :

- Dg32
- Dganchor
- Dgbase
- Dgcomm
- Dgcrypt
- Dgfs
- Dggdicom
- Dgguard
- Dgiogrid

- Dgpdb
- Dgtdi
- Dgtimer
- Dgvault
- Dgwatch

He!he!…now something tell me that on our system an I/O operation is intricate like a 'spaghetti plate' ☺. How we can see there is not only a simple file system filter driver…but also a series of services and drivers that work together.
Well…now we can start to look inside these drivers, but what we need is a different approach…we can simple jump in sice and follow an I/O request sent to file system…but we get lost inside the kernel code very soon…so we need something that tell us exactly where we are and what is happen in the system…**a file system filter driver**.
Yes something like the InTether dgfs.sys driver…but adapted for our purpose. Let start to code!

==========

!!! I don't want to explain how to code a 2K driver or how to code a file system filter driver in this tutorial…are not enough hundred of these docs to cover these topics…so I assume that who read from here know how to code that. I limit only my discussion on some parts of the included code. !!!

==========

The complete source code is attached at this tutorial:

- FFsd.c
- FFsd.h
- FFsdLoader.asm

The FFsd driver will be loaded on demand by FfsdLoader. But I haven't include the code to unload it on demand, because unload a file system filter driver is quite dangerous and require to keep track of the IRP queue to avoid unload the filter during an IRP processing etc. This fsd filter is simple enough for our main purpose . So when you want unload FFsd filter just reboot the system.
Our FFsd driver support two IOCTL command:

IOCTL_FFSD_STARTHOOK	Hook the specific drive and start hooking I/O request.
IOCTL_FFSD_ENABLEBPX	Enable an int 3 (trapped by sice) everytime an IRP_MJ_READ has been sent to the file system.

The default hooked drive is the 'C:' so if you don't use that you have to modify the FFsd.c and recompile. I'm sorry for these stupids limits, but there are a lot of lammers around and I repeat all source code is just for reversing and learning! So if you want to use it at least you have to understand where and what you have to change in the source code.
Ok lets see some part of the code:

```
//*****************************************************************************
//
// FFSdFastIoDeviceControl - here we manage the IRP_IO_DEVICE_CONTROL for our
//                           FFsd driver. Basically only 2 ;)
//
//*****************************************************************************
BOOLEAN
FFsdFastIoDeviceControl(
   IN PFILE_OBJECT FileObject,
   IN BOOLEAN Wait,
   IN PVOID InputBuffer,
   IN ULONG InputBufferLength,
   OUT PVOID OutputBuffer,
   IN ULONG OutputBufferLength,
   IN ULONG IoControlCode,
   OUT PIO_STATUS_BLOCK IoStatus,
   IN PDEVICE_OBJECT DeviceObject )

{
   ….    ….
   ….    ….
case IOCTL_FFSD_STARTHOOK:              // Hook drive 'C:' and start filtering

        retval = FFsdHookDrive(0, DeviceObject->DriverObject);
        IoStatus->Information = sizeof(ULONG);
        break;

case IOCTL_FFSD_ENABLEBPX:              // Enable/Disable IRP_MJ_READ bpx

        if (!bpxEnabled) {

                bpxEnabled = TRUE;
        }else{
                bpxEnabled = FALSE;
        }
        IoStatus->Information = sizeof(ULONG);
        break;

default:

        IoStatus->Status = STATUS_INVALID_DEVICE_REQUEST;
        break;
    ….    ….
    ….    ….

   return retval;
}
```

Our FFsd driver registers a FastIO interface so when it receives an IRP_MJ_DEVICE_CONTROL the FfsdFastIoDeviceControl routine is called. Here we manage the two IOCTL passed from FfsdLoader.exe.

```
//*****************************************************************************
//
// FFsdHookRoutine routine - Here pass all IRPs directed to the driver hooked
//                           How u can see we hook only IRP_MJ_READ. It's enough
//                           knowing which files are read and to jump easly from
//                           here, inside the InTether drivers code.
//
//*****************************************************************************
NTSTATUS
FFsdHookRoutine(
   PDEVICE_OBJECT HookDevice,
   IN PIRP Irp)
{

   PIO_STACK_LOCATION      currentIrpStack = IoGetCurrentIrpStackLocation(Irp);
   PIO_STACK_LOCATION      nextIrpStack    = IoGetNextIrpStackLocation(Irp);
   PFILE_OBJECT            FileObject;
   PHOOK_EXTENSION         hookExt;
   NTSTATUS                ns;
   BOOLEAN                 ret;
   PFILE_STANDARD_INFORMATION fileStandardInfo;

   FileObject = currentIrpStack->FileObject;
   hookExt = HookDevice->DeviceExtension;

   // Hook all IRP_MJ_READ and get the file name on which is directed this I/O request.

   if(currentIrpStack->MajorFunction == IRP_MJ_READ) {

         FileObject = currentIrpStack->FileObject;
         ret = FFsdGetPathName(FALSE, FileObject, hookExt, currentIrpStack, Irp, 0, 0, 0);
   }

   *nextIrpStack = *currentIrpStack;

   // We don't use a Completion routine. We look at the result inside InTether's drivers ☺.

   IoSetCompletionRoutine(Irp, FFsdHooked, NULL, FALSE, FALSE, FALSE);
   return IoCallDriver(hookExt->FileSystem, Irp);
}
```

FfsdHookRoutine is the 'heart' of our file system filter driver, but how you can see we are interested only to trap IRP_MJ_READ requests, and then to know on which file is directed the I/O request. In this way we know which file(s) are read when we try to read an InTether protected file.
The IoCallDriver function sends all IRPs and in particulary (from our point of view) the IRP_MJ_READ requests down to the next file system driver in the chain …which will be the InTether filter DGFS.SYS…how we see later, following the IRP request down, bring us after few lines of the oskrnl code to the caller of the read operation…and we 'll see big stuff there if you have the patience to follow me ☺.
Now we give a look at the FfsdGetPathName routine, which give us not only the file name but also all parameters of the read request.

```
//*****************************************************************************
//
// FFsdGetPathName - We ask the filename for this IRP_MJ_READ hooked and then
//                           we give a look at the parameters (bpx enabled)
//
//*****************************************************************************
BOOLEAN
FFsdGetPathName(
   BOOLEAN IsFastIO,
   PFILE_OBJECT fileObject,
   PHOOK_EXTENSION hookExt,
   PIO_STACK_LOCATION curParameters,
   PIRP  irp,
   ULONG ReadOffsetFastIO,
   ULONG Length,
   ULONG Buffer)
{
   ANSI_STRING                fileName;
   PFILE_NAME_INFORMATION fileNameInfo;
   FILE_INTERNAL_INFORMATION fileInternalInfo;
   UNICODE_STRING             UnicodeName;
   ULONG                      ReadOffset;
   ULONG                      ReadLength;
   ULONG                      ReadBuffer;
   BOOLEAN                    retval;

   fileNameInfo = (PFILE_NAME_INFORMATION) ExAllocatePool( NonPagedPool,
                                          MAXPATHLEN*sizeof(WCHAR) );
   RtlZeroMemory(fileNameInfo,MAXPATHLEN*sizeof(WCHAR));

   if (IsFastIO) {

         ReadOffset = ReadOffsetFastIO;
         ReadLength = Length;
         ReadBuffer = Buffer;

    } else {

         ReadOffset = curParameters->Parameters.Read.ByteOffset.LowPart;
         ReadLength = curParameters->Parameters.Read.Length;
         ReadBuffer = irp->UserBuffer;
     }

    // Ask at the win fsd driver the filename from the fileobject this is 'legal' because we are inside an
    //  IRP_MJ_READ so windows fsd  'knows' already this fileobject.

    if(fileNameInfo &&
        FFsdQueryFile(hookExt->FileSystem, fileObject, FileNameInformation, fileNameInfo,
                                                          (MAXPATHLEN1)*sizeof(WCHAR),
                                    FFsdQueryFileComplete,IRP_MJ_QUERY_INFORMATION)) {

        fullUniName.Length = (SHORT) fileNameInfo->FileNameLength;
        fullUniName.Buffer = fileNameInfo->FileName;
        if(NT_SUCCESS(RtlUnicodeStringToAnsiString( &fileName, &fullUniName, TRUE))) {

        // if 'IRP_MJ_READ bpx' is set we can look in sice at each IRP parameters.
        // I have used inlineasm to have these values directly in the 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.
        if (bpxEnabled) {
```

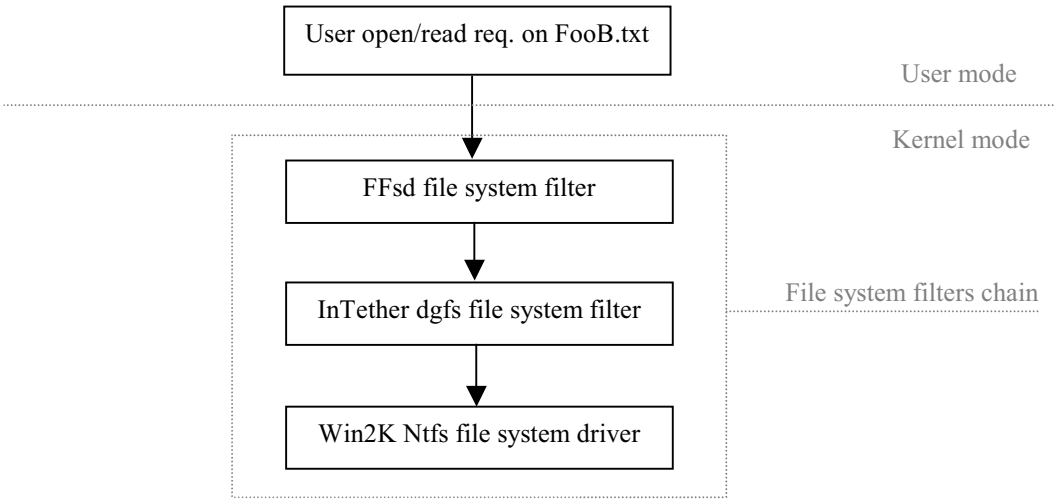
```
                __asm
                {
                        mov ebx,[fullUniName.Buffer]
                        mov eax,ReadOffset
                        mov ecx,ReadLength
                        mov edi,ReadBuffer
                        int 3
                }
            }
        }
    }

    if (fileNameInfo) ExFreePool(fileNameInfo);
    return TRUE;
}
```

Ok now we have our FFsd filter driver ready. We need to have softice loaded to use the IRP_MJ_READ bpx so the first thing to do is take care of a little anti-sice code in one of the InTether modules…if we load sice after few seconds a messagebox tell us that a system debugger has been found and InTether will be stopped. The anti-sice trick is the classic Meltice detection and it is in GD32.EXE …so just hide sice and all will be ok. Now we can run FfsdLoader.exe to enable FFsd filter and then we can load sice. Obviously until we don't enable the IRP_MJ_READ bpx we can see nothing but our driver is working hard….☺.
To enable the bpx, just set the option in the FfsdLoader interface, you have to keep in mind that when the int 3 in our hooking routine is enabled softice breaks at any read operation directed to the windows file system driver…originated in user mode or kernel mode…and in any thread!. This mean that we need to know exactly if an IRP_MJ_READ is generated as a consequence of an open/read request on our InTether protected file. To do so we can code just few lines of code instead of using notepad to open for ex. our FooB.txt file…in this way we use int 3 to isolate any api call used on the file protected…and step over any IRP_MJ_READ generated out of these calls…

```
push    00000000
push    FILE_ATTRIBUTE_NORMAL
push    OPEN_EXISTING
push    00000000
push    00000000
push    GENERIC_READ
push    offset   FooBName
int  3
call    CreateFileA
mov     FileHnd,eax
push    00000000
push    eax
int  3
call    GetFileSize
push    00000000
push    offset  ByteReceived
push    eax
push    offset  DataBuffer
push    dword ptr  FileHnd
int  3
call    ReadFile
int  3
push    dword ptr  FileHnd
call    CloseHandle
```

With the code above we can set a Bpint 3 in sice and we can enable the IRP_MJ_READ bpx in FfsdLoader. We can simple step over (ctrl-d) in sice everytime our FFsd filter break with read operation trapped…until we meet the first int 3 before the createfileA api that open the text file protected….we skip the int 3…and we step over the api function…and straight away FFsd traps the first (of a long series) read requests…but from now we

are sure that our FFsd filter breaks on a InTether requests…we are in! …turn on the stereo…get some drink…send the wife/girl in bed (make sure she is alone ☺) …the tactical phase is complete, time for some on-field action!
Before to dig in sice just a note to refresh the path of any IRP_MJ_READ that our FFsd filter trap:



When the IRP_MJ_READ pass through our FFsd filter we can retrieve the file name on which the read request is directed (with all parameters) and when we see FooB.txt filename we can follow the IRP inside the InTether file system filter and see on which file it redirects the read request and find where it stores our text file FooB. Lets go now…

During the **CreateFileA** api on FooB.txt file our FFsd filter report a series of suspected IRP_MJ_READ requests sent to InTether Gdfsd.sys file system driver (well…directed to windows file system ☺):

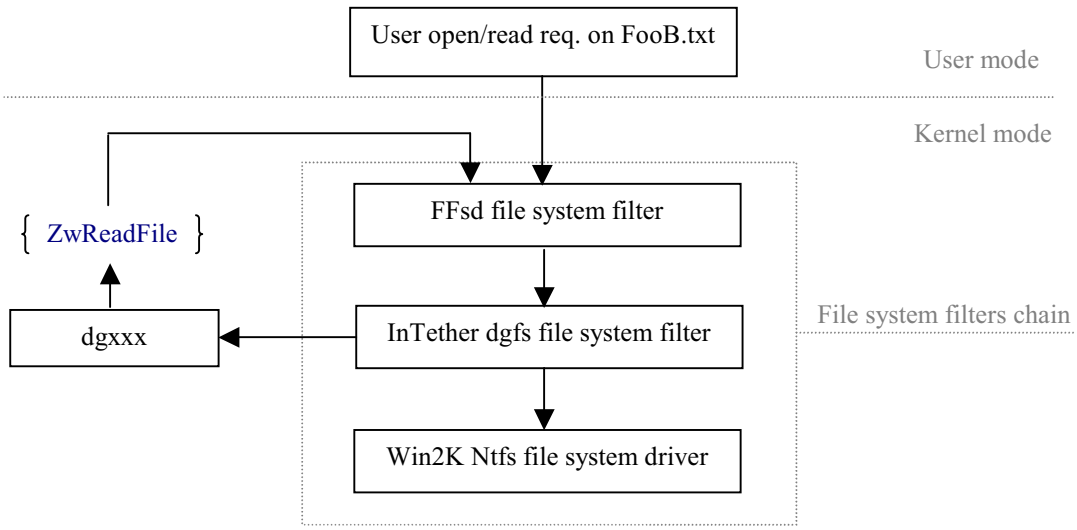
```
…
c:\winnt\system32\dgperm.db      Offset : 38h      Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset: : 6A0h    Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : D08h     Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : D08h     Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : 38h      Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset: : 6A0h    Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : D08h     Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : D08h     Length : 668h bytes
c:\winnt\system32\dg32.exe
c:\winnt\system32\dgimages
c:\winnt\system32\dg32.exe
c:\winnt\system32\dg32.exe
c:\winnt\system32\win32k.sys
… …
```

then the nagscreen tell us how many mins we still have to see the text file, then after clicking on the open button in the dialog:

```
… windows reads some own files…but after again….
c:\winnt\system32\dgperm.db      Offset : 38h      Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset: : 6A0h    Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : D08h     Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : D08h     Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : 38h      Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset: : 6A0h    Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : D08h     Length : 668h bytes
c:\winnt\system32\dgperm.db      Offset : D08h     Length : 668h bytes
c:\winnt\system32\vault.gdv      Offset : 20h      Length : 118h bytes
c:\winnt\system32\vault.gdv      Offset : 138h     Length : 118h bytes
c:\winnt\system32\vault.gdv      Offset : 138h     Length : 118h bytes
c:\winnt\system32\vault.gdv      Offset : 20h      Length : 118h bytes
```

c:\winnt\system32\vault.gdv	Offset : 138h	Length : 118h bytes
c:\winnt\system32\vault.gdv	Offset : 138h	Length : 118h bytes

and then createfile returns and we break at int 3 after the api function.
And this is a good news for us…dgperm.db and vault.dgv are for sure InTether files …and this tell us that even if dgfs is lower than our FFsd filter in the file system filters chain, it's not able to hide the access to its own files. How this happen is explained in the scheme below:



So basically the InTether file system filter (dgfs) call an external driver (dgxxx) which call ntoskrnl functions (ZwReadFile etc) to access to its own files. But the ntoskrnl pass the I/O operation to the top of the file system filters chain…and our FFsd filter traps all read requests on dgperm.db and vault.dgv.

Ok back to our code…we are stopped on the int 3 after the createfilea api function, now we can skip over and execute the getfilesize api to have the size of our FooB text file.
The GetFileSize api is executed without any break by FFsd …so no IRP_MJ_READ requests…Someone can say that this is normal because the GetFileSize api generates an IRP_MJ_QUERY_INFORMATION and not IRP_MJ_READ…and this is right, but wrong here…because I want to remember that every protected files are saved by InTether within one, and unique file…so using an IRP_MJ_QUERY_INFORMATION on this unique file gives the total size of the file.
So this little detail gives us a precious clue…the size of a protected file is read from dgperm.db or vault.dgv …and probably there are others delicious info there ☺.
Ok go on, we skip the int 3 and we arrive at ReadFile api…we execute the function and FFsd breaks for these read requests:

c:\winnt\system32\vault.gdv	Offset : 20h	Length : 118h bytes
c:\winnt\system32\vault.gdv	Offset : 138h	Length : 118h bytes
c:\winnt\system32\vault.gdv	Offset : 138h	Length : 118h bytes
c:\winnt\system32\vault.gdv	Offset : 40B00h	Length : 20h bytes

and then breaks the int 3 after the ReadFile api …so if we check the buffer we have the FooB text.
Cool…now we have enough clues to understand what InTether does…lets put all these clues together:

1. During the open request 2 files have been accessed : "dgperm.db" and "dgvault.dgv".
2. dgperm.db has been read in block of 668h bytes and it has an header of 38h bytes.
3. vault.dgv has been read in block of 118h bytes and it has an header of 20h bytes.
4. Both are hidden file.
5. Both they have been read twice.
6. One of these 2 files store the size of each files protected.
7. During the ReadFile api no one IRP_MJ_READ is generated for a "FooB.txt" fileobject.

8. vault.dgv seems store the protected file. We know that the last read operation (during the ReadFile api) read 20h bytes (32d) from vault.dgv…this is the size of our FooB.txt and after that there are no more read requests in the system until we return from the ReadFile api.

Ok what we want to do is make a copy of our FooB.txt file, so now in this part 1 of this tutorial we concentrate to analyse in details what happen inside the read operation. Basically to extract a protected file we need to know where it's stored (even if we know now that vault.dgv is the container), the offset where start each file inside the container and (if they are encrypted) …how decrypt them.
Just a note the virtual addresses of the code that follow are just for better reading jmps ..but don't trust on it they are surely different on your pc, except for the ntoskrnl code which is mapped from the VA 80000000h on all win2k system ☺.
So we can restart the little code to open and read FooB file again …this time, during the ReadFile api function, we start to step the code after the first trapped IRP_MJ_READ.

We have seen that the first read request got it in ReadFile is :

c:\winnt\system32\vault.gdv Offset : 20h Length : 118h bytes

so sice break inside our FFsd here :

FFsd.sys

```
F09AC419      mov  ebx, [ebp-0C]  ; ebx = pointer to filename buffer
              mov  eax, [ebp+14]  ; eax = offset to read
              mov  ecx, [ebp+1C]  ; ecx = length to read (bytes)
              mov  edi, [ebp+08]  ; edi = buffer that receives data
              int  3
              push esi            ; ← we are here
              call [ntoskrnl!ExFreePool]
              pop  edi
              pop  esi
              mov  al,01
              pop  ebx
              leave
              ret
```

We are in our code (FfsdGetPathName) and we have trapped the IRP_MJ_READ on vault.dgv…and we have all parameters in the registers. This request will read 118h bytes at the offset 20h (skip the header) with the buffer pointer in EDI…we just keep on eye on this buffer (D EDI) to see what InTether read .
Ok we can step on and return from our FfsdGetPathName…and we land here:

FFsd.sys

```
F09AC464      push 09
              mov  edx,ebx
              pop  ecx
              rep  movsd
              mov  eax, [ebx+60]
              and  dword ptr [eax-04],00
              and  byte ptr [eax-21],00
              sub  eax,24
              mov  dword ptr [eax+1C],F09AC280
              mov  eax, [ebp+0C]
              mov  ecx, [eax+04]
              call [ntoskrnl!IofCallDriver]
              pop  edi
              pop  esi
              pop  ebp
              ret  0008
```

We are inside our FfsdHookRoutine, the code above copy the irp stacklocation in the next irp stack location for the below (our driver) driver, then the IRP_MJ_READ is passed down to the next filter driver…InTether dgfs.sys. We don't follow the IRP down…we are interested only to have the data read and to know who has read that. So we step over the IofCallDriver …if we look at the buffer in sice we see the 118h bytes read.
We can return from our Hook routine to the kernel…here :

ntoskrnl.exe

```
8041F547      call [eax*4+ecx+38]
              pop  edi                  ; ← we land here
              pop  esi
              ret
```

The above code is the end of the IofCallDriver (which in reality is the IoBuildSynchronousFsdRequest).
We step over and we land for a while in the dgfs.sys filter just to see that it has used IofCallDriver to send the IRP down to the file system…then it returns again in the above code in the kernel…and then we return after the 'main' IofCallDrive used by the kernel to send to our FFsd filter this IRP_MJ_READ…here:

ntoskrnl.exe

```
804BA5E3      call ntoskrnl!IofCallDriver
              cmp  byte ptr [ebp+14],00  ; ← we land here
              mov  [ebp+OC],eax
              jz   804BA61B
              cmp  eax,00000103
              jz   804BA61B
              mov  cl,1
              ……   ……
              ret
```

Now we step the lines above until the ret instruction and after that we land to the end of NtReadFile ntoskrnl function …we exit from this function with the first ret that we meet and we land inside the ExReleaseResourceForThread kernel function…here:

ntoskrnl.exe

```
80461691      mov  esp,ebp
              mov  ecx,[FFDFFF124]
              mov  edx,[ebp+3C]
              mov  [ecx+00000128],edx
              cli
              test dword ptr [ebp+70],00020000
              …    …… ……
              pop  edx
              add  esp,08
              pop  ecx
              sti
              sysexit
              iretd                     ; exit from the int 2Eh (A1h NtReadFile)
```

With the iretd instruction we return from the int 2Eh used to call the NtReadFile (A1h) and we land after the ZwReadFile (ZwReadFile = NtReadFile = A1h) call, inside the InTether code that has called this kernel function …dgcrypt.sys….he!he! got it!.
So now you can follow any IRP_MJ_READ that our FFsd traps, but give a look at the dgcrypt code because I have the feeling that the buffer will be decrypted soon ☺ …

Dgcrypt.sys

```
BC64F3A1      call [ntoskrnl!ZwReadFile]
              jge  BC64F3AF
              xor  eax,eax
              jmp  BC64F3D9
BC64F3AF      cmp  [ebp-10],edi
              jnz  BC64F3AB
              cmp  [BC658964],esi
              lea  eax,[ebp-04]
              push eax
              push esi
              push edi                  ; size buffer to decrypt
              push dword ptr [ebp+OC]   ; pointer buffer to decrypt
              jz   BC64F3CC
              push BC6599E0
```

```
            jmp  BC64F3D1
            push BC658980
BC64F3CC    call BC64F2E4              ; decrypt the buffer
            push 01
            pop  eax
            mov  ecx,[ebp+18]
            mov  edx,[ebp-10]
            pop  edi
            pop  esi
            mov  [ecx],edx
            leave
            ret  0014
```

At the BC64F3CC we have the call that decrypt the data just read from the vault.dgv. We can just skip over the call and look at the buffer decrypted…but before or after we need to take care of reversing this decryption algo if we want to code a program that make copy of InTether protected files…so lets give it a look now :

Dgcrypt.sys

```
BC64F2E4    push ebp
            mov  ebp,esp
            push esi
            mov  esi,[ebp+18]
            push edi
            mov  edi,000007FF
            mov  eax,[esi]             ; eax = offset used to read in vault.dgv
            xor  edx,edx
            add  eax,[ebp+14]
            and  eax,edi               ; Keep the 11 lowest bits (max.7FFh)
                                       ; will be used like index in the buffer
                                       ; key to take a byte.
            cmp  [ebp+10],edx          ; is the size to decrypt 0 ?
            jbe  BC54F319              ; no! go on
            push ebx
BC64F300    mov  ecx,[ebp+0C]          ; ecx = pointer in buffer to decrypt
            mov  ebx,[ebp+08]          ; ebx = point to the buffer for the
                                       ; decryption key.
            add  ecx,edx               ; ecx = point byte to decrypt
            mov  bl,[ebx+eax]          ; bl = byte key from the key buffer
            and  eax,edi               ; eax = make sure that the new index is 11
                                       ; bits (max.7FFh)
            xor  [ecx],bl              ; decrypt one byte in buffer
            inc  eax                   ; increment index for the next pass
            inc  dword ptr [esi]       ; increment index for the next pass in
                                       ; memory too.
            inc  edx                   ; increment index in buffer to decrypt
            cmp  edx,[ebp+10]          ; decrypted all bytes ??
            jb   BC64F300              ; not yet!……loop for the next bytes.
            pop  ebx
            pop  edi
            pop  esi
            pop  ebp
            ret  0014
```

It's a simple decryption algo, It use a xor operation on each byte and use a key buffer of 7FFh bytes.
The offset used to read in the vault.dgv file is used like the starting index in the key buffer, so for example…in the first read operation it reads a 118h bytes block, with the offset 20h (skip header)…so the first byte of the read data is xored with the value at KeyBuffer+20h …and then the 'offset-index' is incremented.
The Key buffer is stored in the data section of dgcrypt.sys…so ready to be read by us later ☺.

Ok now we can look at the decrypted buffer :

```
00000000 4447 5F46 494C 4500 0000 0000 633A 5C46 DG_FILE.....c:\F
00000010 6F6F 412E 7478 7400 0000 0000 0000 0000 ooA.txt.........
00000020 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000030 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000040 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000050 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000060 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000070 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000080 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000090 0000 0000 0000 0000 0000 0000 0000 0000 ................
000000A0 0000 0000 0000 0000 0000 0000 0000 0000 ................
000000B0 0000 0000 0000 0000 0000 0000 0000 0000 ................
000000C0 0000 0000 0000 0000 0000 0000 0000 0000 ................
000000D0 0000 0000 0000 0000 0000 0000 0000 0000 ................
000000E0 0000 0000 0000 0000 0000 0000 0000 0000 ................
000000F0 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000100 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000110 1600 0000 0100 0000                     ........
```

- Just a marker.
- The path+filename of our first protected file.
- Seems the size of the protected file. 16h = 22d.
- Unkwon for now. (explained later)
- Unkwon for now (explained later)

Ok basically each 118h bytes block describes an InTether protected file active on our system…but lets go to see what the code search /check in these bytes…After that the 118h bytes have been decrypted we follow the code and we arrive inside another InTether driver…dgvault :

Dgvault.sys

```
BC40010E      push ebp
              mov  ebp,esp
              mov  eax,[ebp+08]
              push esi
              test eax,eax
              mov  ecx,[eax+04]
              jz   BC400152
              mov  eax,[ebp+0C]          ; num of 118h bytes blocks checked
              cmp  eax,63                ; we can open max. 63h protected files (?)
              ja   BC400152
              mov  esi,[xxxxx]
              test esi,esi
              jz   BC400152
              imul eax,eax,00000118      ; calculate offset in vault.dgv to read
              lea  eax,[ebp+08]
              add  eax,20                ; add the header size
              push edx
              push 00000118
              push eax
              mov  eax,[xxxxxx]
              push esi
              push ecx
              call [eax+0C]              ; call dgcrypt to read/decrypt 118h bytes
              test eax,eax               ; ← we are here after the decryption
              jnz  BC400159
              and  [esi+00000114],eax
              xor  eax,eax
              pop  esi
```

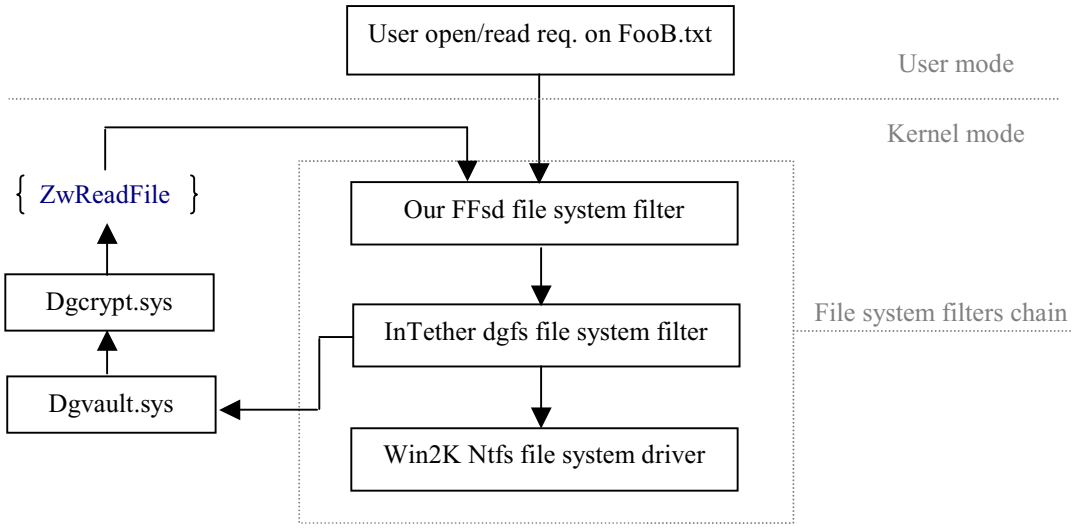
```
            pop  ebp
            ret  000C
```

The routine above prepare the offset within vault.dgv ,the size (118h) and the buffer to pass down to the dgcrypt, which (how we have seen) read and decrypt the 118h byte block….so we return after the `call [eax+0C]` and we can return again in the main routine to see what it'll check in the buffer :

Dgvault.sys

```
BC4001E6    call BC40010E                   ; call the above routine to read
                                            ; vault.dgv
            test eax,eax
            jz   BC400244
            mov  esi,ntoskrnl!RtlInitString
            push edi
            call BC400062                   ; check if the first 8 bytes of the
                                            ; 118h block are 'DG_FILE'.
            test eax,eax                    ; eax = 1 = yes, go on
            jz   BC400244
            lea  eax,[edi+0C]               ; eax = path-name in the 118h
                                            ; bytes block
            push eax
            lea  eax,[ebp+08]
            push eax
            call esi                        ; Init buffer with path-name in the
                                            ; 118h bytes block.
            push dword ptr [ebp+0C]
            lea  eax,[ebp-10]
            push eax
            call esi                        ; Init buffer with path-name of the
                                            ; file that we want read in usermode
                                            ; (…\FooB.txt).
            lea  eax,[ebp-08]
            push 01
            push eax
            lea  eax,[ebp-10]
            push eax
            call ntoskrln!RtlEqualString    ; compare both string
            test eax,eax                    ; al = 1 = equal
            jz   BC400230
            push edi
            call BC400040                   ; It's the right block (same path-
                                            ; name) check dword at the offset
                                            ; 114h in the block.
            test eax,eax                    ; If dword = 1 the file is ok
            jz   BC400256                   ; and we jump out of the loop
BC400230    inc  ebx                        ; if 2 the file is bad (expired?)
            cmp  ebx,64                     ; so prepare to read another block
            jae  BC400244
            push edi
            push ebx
            push dword ptr [ebp+08]
            call BC40010E                   ; call the above routine to read
                                            ; vault.dgv
            test eax,eax
            jnz  BC4001F5
BC400244    or   esi,-1
BC400247    push edi
            call BC4001F5                   ; free strings buffers
            mov  eax,esi
            pop  esi
            pop  esi
            pop  ebx
            leave
            ret  0008
BC400256    mov  esi,ebx
            jmp  BC400247
```

Now we know that dgvault scans each 118h bytes block to find the path-filename of the file that we want to read…when it has found it , the last dword of the 118h bytes block is compared with 1 (good file) if it's 2 (probably mean expired file) and the block is discarded. Then the right block is read a second time…this loop of reading operations is performed twice…after that, the execution return to the code that has started this reading loop….dgfs.sys.
So now we can make a scheme about the InTether kernel modules relationship :



Ok..when dgvault has found the right 118h bytes block (FooB.txt, the second for us)…it simple releases the Mutex (KeReleaseMutex) that protect the vault.dgv accessing and then close the file.

==========

I suggest at the InTether coders to check the code that manage all their syncronization objects because I have the feeling that it's quite dodgy…and may be buggy…the process ID check should be enough to leave go down a write operation that come from a process which has not open any InTether protected file…but unfortunately sometime it fails ☹. I don't know exactly why but I think worth a check there.

==========

At this point it returns in dgfs, which just check (eax ☺) if dgvault has found the right block …which mean that the file protected is inside vault.dgv and is possible read it.
Here the dgfs code…

Dgfs.sys

```
BC3A1433      call [eax+0C]              ; call dgvault to read each 118h
                                         ; bytes block until it finds the
                                         ; right one.
              test eax,eax               ; eax = 1 found
              mov  [ebp-08],eax
              jnz  BC3A1445              ; found it, jump and go on
              mov  dword ptr [esi],C0000001
              jmp  BC3A147
BC3A1445      lea  ecx,[ebp-04]
              push ecx
              push dword ptr [ebp+10]    ; Irp.paramaters.Offset (of our ReadFile)
              push dword ptr [ebp+0C]    ; Irp.parameters.Length (of our ReadFile)
              push dword ptr [ebp+14]    ; Irp.UserBuffer (of our ReadFile)
              push eax
              mov  eax,[BC3CEA2C]
              call [eax+1C]              ; call dgvault to read the file FooB.txt
                                         ; inside vault.dgv
              cmp  dword ptr [ebp+04],00
              mov  edi,eax
              jz   BC3A146A
              mov  dword ptr [esi],C0000001
              lea  eax,[ebp-08]
```

```
                push eax
                mov  eax,[BC3CEA2C]
                call [eax+14]              ; call dgvault last time to free its
                                           ; allocated mem
                …  …  …
```

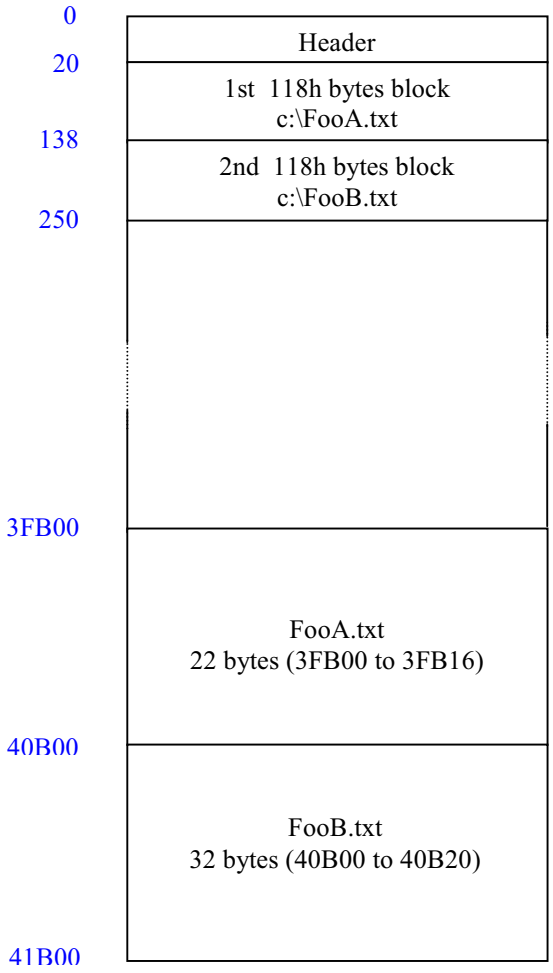
In the code above dgfs calls dgvault for the last time (call `[eax+1C])` to read FooB.txt and satisfy our ReadFile api request. Finally now we can know where or in which way it retrieve the offset inside vault.dg for each file within it.
When we have this information we have all necessary to code our own backup utility for InTether protected file…but give it the last look inside dgvault.sys code:

Dgvault.sys

```
BC4276BE        push dword ptr [BC440020]
                call BC42886A              ; open vault.gdv (ZwCreateFile)
                …     … …
BC4276ED        mov  eax,[ebp+10]          ; Irp.paramaters.Offset (of our ReadFile)
                mov  ebx,[eax+14]          ; Irp.parameters.Length (of our ReadFile)
                mov  ecx,[esi+00000118]    ; size of the file stored in the 118h
                                           ; bytes block
                lea  edx,[ebx+eax]         ; edx = offset + Length to read
                cmp  edx,ecx
                jbe  BC427719              ; if below or equal at the file size jmp
                cmp  eax,ecx
                jbe  BC427708
                and  dword ptr [ebp-08],00
                sub  ecx,eax
                cmp  dword ptr [ebp-08],00
                mov  [ebp+14],ecx
                jz   BC427864
                mov  ebx,ecx
BC427719        mov  ecx,[esi+00000114]    ; A new value for us for FooB's block is
                                           ; 0000001 and it is the number of 1000h
                                           ; blocks (pages) from the point where
                                           ; start the files protected in vault.dgv
                                           ; to the start of our FooB.txt.
                test eax,eax
                mov  [ebp-04],ecx
                jz   BC427771
                …     … …
BC427771        cmp  dword ptr [ebp-08],00
                jz   BC427864
                mov  eax,[ebp-14]
                lea  edx,[ebp+eax+FFFFF000]
                test edx,edx
                jle  BC4277A1
                …     … …
BC4277A1        mov  eax,[eax-14]
                mov  edi,1000h
                test eax,eax
                jz   BC4277B7
                …     … …
BC4277B7        cmp  dword ptr [eax-10],01
                mov  esi,edi
                jae  BC4277C1
                mov  esi,ebx
                shl  ecx,0C                ; ecx = num pages * page size
                lea  edx,[ebp+08]
                push edx
                lea  eax,[eax+ecx+0003FB00] ; eax = 3FB00 (offset in vault.dgv where
                                           ; start the files protected) + ecx =
                                           ; offset start FooB.txt
                push esi
                push eax
                push dword ptr [ebp+0C]
                mov  eax,[BC440058]
                push dword ptr [eax+04]
```

```
mov  eax,[BC44001C]
call [eax+0C]              ; call dgcrypt to read the bytes which are
                           ; our FooB.txt and then decrypt them with
                           ; the same algo see before.
```

Just a little explanation about the algo above…it basically calculate the offset within the file vault.dgv where is stored the requested protected file (our FooB.txt).
The 'key' point is the dword value stored in each 118h bytes block at the offset 10C (before the dword of the size if you look at the block some pages above). In the code this value is retrieved with an index of 114h , but this is not more the original 118h bytes block buffer read by dgcrypt…but it has been copied in a second buffer with 2 dword (values) in more at the start by dgvault…during the last block reading….so no worry 114h index point at the value 10C in the 118h byte block. But what is this value ?? …InTether store the protected files in chunks of 1000h bytes (I have called these chunks 'pages' don't confuse with page of mamory ☺) … starting from a fixed point (offset) which is 0003FB00. So in our vault.dgv we have 2 file stored, FooA.txt (22bytes size) and FooB.txt (32bytes size)…and this is the scheme of our dgvault file :



Even if FooA.txt is only 22 bytes it uses an entire 1000h bytes chunk (page), same for FooB.txt, which is 32 bytes. Each 118h bytes block has a dword value at the offset 10Ch which is the number of 'pages' from the start of the files (3FB00) to the start of the file that the block describe…basically a kind of delta offset but expressed in number of 'pages', from a fixed point.
With this value dgvault calculate the starting offset of our file.

…cool…now we have all information to code a backup utilty…

- We know where InTether stores the active protected files on our system.
- We know where InTether stores information for each protected files.
- We know where to find a specific file inside vault.dgv
- We know the decryption algo (and where is the decryption key in memory) to decrypt blocks and files.

Included there is my InTetbck.exe and InTetKey.sys , to create a copy of an active InTether protected file (…protected with the same options used for our 2 examples FooA.txt and FooB.txt).
The code is very simple and well commented…just a note on the InTetKey code…this is a kernel mode code to read the decryption key from dgcrypt.sys data section in memory. I have chosen to use a 'runtime' dump of the key because the dgcrypt data section is full of key buffers used for others decryption process that we see in the next tutorial.
To read the key buffer we can simple use the 'bad documented' or 'documented in the classic M$hit style' ntoskrl function ZwQuerySystemInformation with SystemModuleInformation (0x0000000Bh) as system_class_information.
With this kernel function we can retrieve the System_Module_Information (see InTetKey.h) for each system module loaded in the system address space.
So we can know where dgcrypt.sys has been mapped in memory and then read its data section.
Simple but full working.
A final note on InTetBck.exe…it require in the same directory a copy of the file vault.dgv. Which is (I remember) in the \winnt\system32 directory and it's an hidden and locked (obviously ☺) file.
…So why don't use the NTFS.exe utility coded at the start of this tutorial, to make a copy of that. ☺

Nothing to see in more…just remember that in this part 1, we have analysed the read operation on a InTether protected file by InTether protection system…with the purpose to understand how it works inside the Win2k 'panties' and to perform a backup copy of our files.
In the part 2 we'll see the 'open' operation on an InTether protected files…and the others dgxxxxx.sys drivers and their relationship.

Stop now…I hope to have kept all in a simple and readable form for everybody …the most funny part of a reversing session is building a 'strategic' way to operate…like in a strategic/tactic game, often this takes more time than just step in the code until you see a 'jmp oep' ☺ …but worth more too.

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 part 2.

MaV3RiCk

maverickluke@hotmail.com