

MrxCls – Stuxnet Loader Driver.

Amr Thabet

Independent Malware Researcher,
Author, Pokas x86 Emulator

Exclusive



InfoSPYWARE

1- Introduction:

Stuxnet is not only a new virus or worm but it represents a new era of malware. This virus has changed the meaning of malware and their goals. You usually hear about a virus annoying people or stealing bank or credit cards, but this is the first time you hear about viruses damaging buildings, physically destroying machines, or even killing people and this is what Stuxnet does.

Stuxnet has gained a lot of attention from malware researchers and media in the past year. It seems to have been specifically created to sabotage Iran's nuclear program. This complex threat uses up to four 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. This allows it to modify the control program which changes the behavior of the machine.

MrxCls is a driver which is dropped by Stuxnet. The mission of this driver is to load the Stuxnet worm on Windows startup in a special way. This driver is extremely complex and contains kernel mode and user mode codes, creation of PE files and mapping of PEs on the fly, which is what we will describe in this article

2- Behind the Scenes:

MrxCls is a very complex project. It includes many features and abilities to load programs secretly without detection by any Antivirus applications particularly behavioral antiviruses.

This virus seems to be a separate project, which wasn't created by the creators of Stuxnet worm. It could have been created by another department within the organization that created Stuxnet.

This driver wasn't modified along with the versions of Stuxnet and it also 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 have allowed it to steal certificates from big companies like Realtek Semi-Conductor Co-Op. This driver is signed with Realtek as a product from this company as you can see in this image.

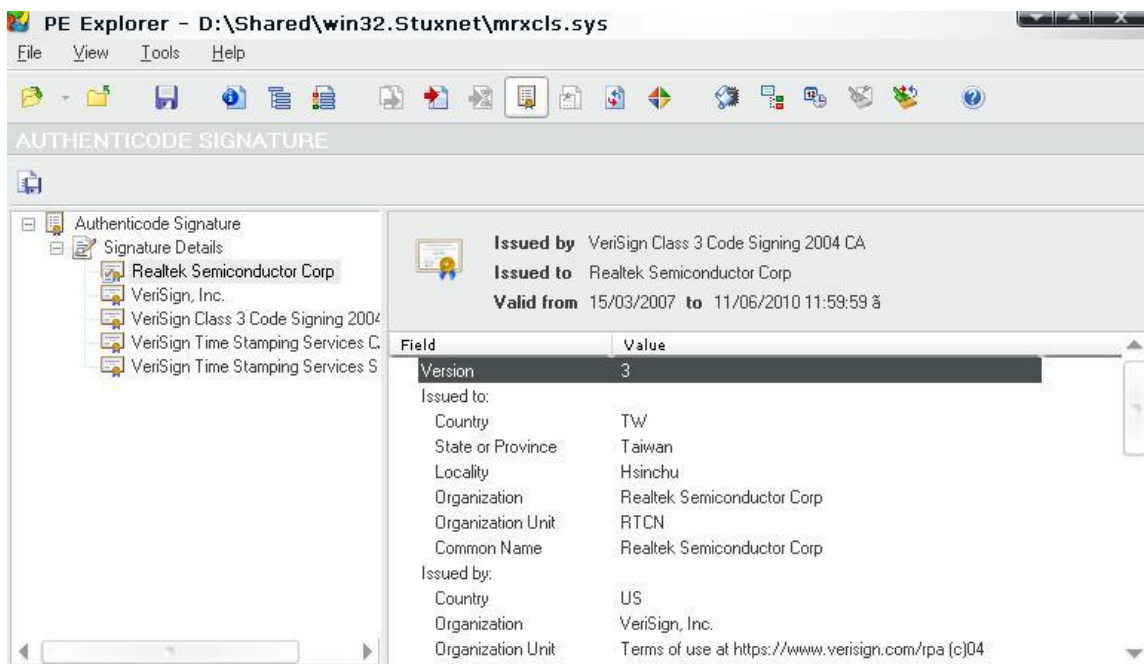


Figure 2-1-1

This level of effort makes us sure that this virus is not a game from some virus writers but a planned crime.

3- Technical Details:

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.

3-1 The Input:

MrxCls takes the parameters from the registry from a key named:
 “HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MRxCls”

It reads the “Data” value in this key as the parameter of the driver.
 This data contains 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;[]\w;.
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...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	\.....

Figure 3-1-1

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.\.....

Figure 3-1-2

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 name of the process that stuxnet needs to inject its file into. 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 to 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 this:

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

This is 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 containing this information:

services.exe → \SystemRoot\inf\oem7A.PNF (stuxnet main dll) and call to Export 1
 S7tgotpx.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 to “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 doing this in 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 this is a brief summary.

3-2 Initialization:

First, stuxnet creates a registry key and adds 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 from “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. After this, 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)

At the end it calls to “PsSetLoadImageNotifyRoutine” to register a function to be called every time a process or a module is loaded into 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.

3-3 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 for “kernel32.dll” (which we’ll talk about later) and if the loaded module is not kernel32, it parses the registry data (which was loaded and decrypted before) and loops through the elements of this data, searching for the name of the process that needs to receive the stuxnet code injection, comparing it with the loaded process’s name.

When it finds a process that needs to receive the stuxnet code injection, it loads the stuxnet file into the process's memory and decrypts it. It then copies a part of its data (containing code) into the process' memory and writes "MZ", "PE" and some additional data into this piece of data.

This piece of data seems to be two PE files (which were created separately before) that are missing 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 code to rewrite and replace them (This is obviously an obfuscation technique). Additionally, the names of all sections were also deleted.

Next, 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 this 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 this, 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 relocatable code 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 call 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 appears to be an attempt to disguise the call to ZwAllocateVirtualMemory with these parameters to avoid detection by antiviruses).

At the end, it creates a buffer with size equal to the size of stuxnet file plus 0x28 bytes and then copies 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 PEXLoader or LoadLibraryW) (8 bytes)

Table 3-3-1

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

Table 3-3-2

Lastly, it writes the place of this buffer (including stuxnet file) into a specific place in the copied PE module (the portion of data that was copied to the process's memory previously).

3-4 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 to kernel32.dll, it jumps to the stage 2.

When stage 2 is reached, 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 processId 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
lea     eax, [ebp+PEDataPtr]

```

Figure 3-4-1

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

```

Then it modifies the immediate of “mov eax,0” with 3rd element of the generic table buffer (InjectedMemory at "MZ" + 0x560) which is the entrypoint of the injected code. The InjectedMemory at "MZ" + 0x2B8 becomes like this:

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

Table 3-4-1

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

3-5 Stage Three : Loading and Executing Stuxnet in The User-Mode

I begin reversing this part by injecting this 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.lstrcpA
\$+24	7C80BB41	kernel32.lstrcpA
\$+28	7C812B7E	RETURN to kernel32.GetVersionExA
\$+2C	7C900000	ntdll.7C900000
\$+30	C07089E0	
\$+34	00000000	

Figure 3-5-1

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 relocatables 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	0FE75E 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	

Figure 3-5-2

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.

4 – Conclusion:

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

MrxCls is the loader driver of stuxnet. It seems to be a separate project because it wasn't modified in most stuxnet versions and also because of its complexity and size.

MrxCls is a huge project. It loads the stuxnet module into a process given by a data written by stuxnet in the registry. Then, it injects stuxnet module into the process in 3 stages from kernel-mode in the first 2 stages to the user-mode in the stage. Then, it calls to a specified export function inside stuxnet module and at last it executes the infected process normally.

5 – About the Author:

I'm Amr Thabet. I'm a Freelance 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 began programming at the age of 14. I have read many books and research in the malware, reversing and antivirus fields and I have been a reverser for almost 4 years.

6 – 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>