Analysis and Diversion of Duqu's Driver

Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier, Aurélien Thierry

Analysis and Diversion of Duqu's Driver

Guillaume Bonfante
Université de Lorraine
LORIA

Jean-Yves Marion
Université de Lorraine
LORIA

Fabrice Sabatier
Inria
LORIA

Aurélien Thierry
Inria
LORIA

E-mail: firstname.lastname@loria.fr

Abstract

The propagation techniques and the payload of ***Duqu*** *have been closely studied over the past year and it has been said that* ***Duqu*** *shared functionalities with* ***Stuxnet****. We focused on the driver used by* ***Duqu*** *during the infection, our contribution consists in reverse-engineering the driver: we rebuilt its source code and analyzed the mechanisms it uses to execute the payload while avoiding detection. Then we diverted the driver into a defensive version capable of detecting injections in Windows binaries, thus preventing further attacks. We specifically show how* ***Duqu****'s modified driver would have detected* ***Duqu****.*

1 Introduction

When it was first discovered in September 2011 by `CrySys` [4], it was said that `Duqu` was close to `Stuxnet` in that they shared infection and propagation mechanisms. `Duqu` is an offensive tool used to steal information: Symantec [7] identified, amongst other payloads, keyloggers, screen recorders, network monitors and service discovery tools. It is kept up to date by Command & Control servers and bundled with an auto-destruction feature typically triggered 36 days after the infection.

The attacks seem to have been successful since the malware was not detected during the operations (some of them lasted a few month) but only post-mortem. Besides, according to Kasperky [5], the latest version of `Duqu` was found in February 2012, long after the malware was first discovered and documented. Thus `Duqu` is a stealthy spyware and the attacks were carefully adapted to each target.

At the High Sec Lab, we develop a malware detector based on morphological analysis [3], which is a technique based on control flow graph comparison. As soon as we put our hands on `Stuxnet` and `Duqu` samples, we wanted to know if our approach, knowing `Stuxnet`, could lead to the detection of `Duqu`. In fact the answer is *yes but*: we recognize `Duqu`'s main DLL as related to `Stuxnet` but only if it is decrypted. The aim of this paper is to present an automatic extraction procedure to bypass stealth techniques such as those used by `Duqu`. The irony of this work is that `Duqu` itself served for this purpose. Let us describe in broad terms our contribution.

Infection timeline. The infection detected by `CrySys` used a malicious Microsoft Word document embedding `Duqu`. Firstly it exploits a 0-day kernel vulnerability (on TrueType fonts [1]), it installs three components: a driver (`nfrd965.sys`), an encrypted DLL (`Duqu`'s main DLL, `NETP191.PNF`) and an encrypted configuration file (`netp192.PNF`).

The second step takes place at the next reboot. The driver monitors processes loaded by the OS and injects `Duqu`'s DLL into a process specified by the configuration file, typically `services.exe`.

During the third and final step, the payload, which is included in the DLL, is activated.

Detecting this malware is challenging because only the driver is unencrypted on the hard drive. The DLL is encrypted, packed with UPX, and is injected immediately after decryption so it is only stored decrypted in RAM. Thus detection could take place when the injection is done, which is right after the decryption of the DLL and before it is activated.

Once installed on a first target, `Duqu` receives orders for further attacks and propagation from Command & Control servers. Each new controlled machine can be configured to connect back to the attacker through some routing tunnel, providing access to restricted areas.

Defensive version. The defense scenario will be the following. We modify `Duqu`'s driver so it monitors loaded processes, stores hashes of their code and detects when an alteration is done.

We began by rebuilding a consistent source code for the driver from the binary, then performed an in-depth analysis of the infection method. Finally we modified the code to build a defensive version capable of monitoring loaded processes and detecting an infection.

2 Rebuilding the driver's source code

We knew `Duqu` shared code with `Stuxnet` since we showed equivalent parts in the drivers' main DLLs [9] and Symantec revealed similarities in the injection technique and some resources. The decompilation of `Stuxnet`'s driver had already be done by Amr Thabet [8] but we decided to decompile `Duqu`'s driver into C code to expose how it operates and the difference between the drivers of `Duqu` and `Stuxnet`. And actually we found singular techniques for injection and anti-detection mechanisms.

2.1 IDA's decompiler

We used IDA's plugin "Hex-Rays Decompiler" [2]. It generates C pseudocode from a binary being analyzed in IDA and authorizes modifications of the C code from within the plugin's GUI. Unfortunately the code generated by IDA was, in our case, not directly usable: an example of the first lines of a function decompiled by IDA is given on Figure 1. Firstly the code cannot be recompiled because some variable types, function arguments and calling conventions were not recognized by the decompiler. Secondly the generated code is hardly readable, partly because some structures and librairies were not identified. Let us detail how to circumvent these issues.

In order to rebuild a consistent source code we followed an incremental protocol based on the decompiler's output. We began with commenting the whole decompiled code except for the entrypoint of the driver. We fixed compilation errors and identified missing structures. Once the entrypoint was readable and could be compiled, we added some more code that was previously commented, fixed it, and so on.

2.2 Structures and types identification

Figure 1 depicts the first lines of the decompiler's output for one of the driver's function. A lot of information is missing, for instance most types are described as `int` so the kind of processing done is unknown. We will give more details on this example to illustrate what has been done to fix the code generated by the decompiler. Some of the constants are going to be of great help: for instance we can combine both of them 0xF750F284 XOR 0xF750B7D4 = 0x00004550 = 'PE\0\0', the text searched is 'PE\0\0' which is obfuscated with a XOR operation.

```
signed int __cdecl sub_12F36(int a1,
                             int a2, int a3)
{
  int v4; // eax@3
  unsigned __int16 v5; // cx@4
  int v6; // ecx@7

  v4 = a2 + *(_DWORD *)(a2 + 60);
  if (*(_DWORD *)v4 ^ 0xF750F284
                    != 0xF750B7D4)
    return 1;
```

Figure 1. First 10 lines of the ParsePE function as decompiled by IDA

At this point we have strong suspicion that this function is used to parse PE files. A quick dive in Microsoft's Visual C++ documentation reveals the structure PIMAGE_NT_HEADERS which first field is *Signature*, whose value is 'PE\0\0' for Windows binaries. Thus the variable `v4`, which is tested against 'PE\0\0', might be of type PIMAGE_NT_HEADERS. So we force this structure and others into IDA's decompiler in place of default `int` type, IDA then finds fields based on their offsets and updates them in the code. In some cases the structure is specific to the analyzed binary (user defined), it is possible to define them manually in the decompiler plugin.

The beginning of the fixed code is presented on Figure 2, it is readable by a C developer: one sees that the function checks whether a file is a PE binary. The following part of the code will parse it and fill a user defined structure with some information (entrypoint, sections...). Besides the compiled binary of this version is very similar to the original binary. Doing this kind of analysis on the whole driver lead us to build an understandable and consistent version of the driver's source code.

3 Functional analysis from the source code

Now that the source code has been rebuilt, let us detail how it operates. There are two main phases,

```
NTSTATUS __cdecl ParsePE(
  __out PEDataPtr pPEData,
  __in PIMAGE_DOS_HEADER BaseAddress,
  __in int flag){
PVOID infosPE;
PIMAGE_DOS_HEADER pDosHeader;
PIMAGE_NT_HEADERS pNtHeader;

pNtHeader = (DWORD)infosPE +
                    infosPE->e_lfanew;
if ((pNtHeader->Signature ^ 0xF750F284)
  != (IMAGE_NT_SIGNATURE ^ 0xF750F284))
    return STATUS_WAIT_1;
```

Figure 2. First 10 lines of the fixed ParsePE function

the first one consists in setting up the driver: it asks the operating system for notifications when a binary is loaded, and initializes stealth mechanisms. The second one is triggered when it receives notifications: the driver infects the target binary by injecting `Duqu`'s DLL into `services.exe`, then activates the payload.

3.1 Initialisation of the driver during boot

Recall that on Windows, the startup order of drivers is determined by their `Group` key in the registry. `Duqu`'s `nfrd965.sys`, belonging to the "network" group, is activated before the hardware abstraction layer (HAL) is loaded into memory.

Once started, `nfrd965.sys` allocates 512 bytes for storing a pointer array of functions shared by various callback routines. Then it decrypts some internal parameters, revealing the name and path of the registry key used for configuring the injection.

If the decryption succeeded, the driver checks its execution mode. If it is in debug or fail-safe mode, the driver halts, otherwise it creates a device named `{624409B3-4CEF-41c0-8B81-7634279A41E5}` and defines a list of control commands that the device can process.

That being done, it registers two callback functions within the kernel's event handler. The first is required by the operating system: it is used to create an access point (`\Device\Gpd0`) and a link (`\DosDevices\GpdDev`) to the driver, and attaches the device to a memory stack. The second function will be called when the driver is initialized or reinitialized, it is inserted in a waiting list of events.

This second function waits until the Windows kernel is totally loaded: it checks if the DLL `hal.dll` is loaded, if not, the function is once again inserted into the events waiting list (for at most 200 times). When the system is ready, an access point `\Device\Gpd1` is created and linked to a request processing function. At that point librairies are available for `Duqu`'s injection.

3.1.1 Stealth techniques

The driver acts like a rootkit because it avoids directly using suspicious system calls. Indeed those system calls might be monitored by an antivirus. Typically the function `ZwAllocateVirtualMemory` can be used to allocate memory space into any process, for instance to inject code into any target process. Besides, in order to hook the entrypoint of a system binary, `Duqu` also needs the function `ZwProtectVirtualMemory` that is deliberately not exported by the kernel. This function modifies the permissions of a memory block and can be used to make a code section writable or execute code in a data section. `Duqu` was built to find `ZwProtectVirtualMemory`'s memory address without using imports.

The two functions are implemented in the Windows kernel (in `Ntoskrnl.exe` or `ntkrnlpa.exe`, depending on the version). The driver inspects every module (DLLs and EXE files) loaded by the OS during boot until it finds one of the two target kernel files.

Once the file is found, the driver uses the function `ParsePE` to examine it closely. It searches, in that kernel file, a call to `ZwAllocateVirtualMemory` (whose address is known from the export table) followed by the opcode `push 104h` and another (near) call to an unknown function. If this pattern, shown in Figure 3, is found, the target address of this last call is considered to be `ZwProtectVirtualMemory`. At this point the memory addresses of both functions are known.

Avoiding hooks. It is firstly checked that the two functions are located inside the kernel's memory addresses and not in user space which would indicate an obvious hook to a monitored function. Secondly an integrity mask applying a logical `AND` is applied on the first 32 bytes of both functions. The mask is the same for the two functions and is shown on Figure 4. If the functions pass both tests, their addresses are considered valid and are kept for a future stealthy use.

3.1.2 Initialization of shared memory

A shared memory space is allocated and used as a link between the driver's callback functions and the kernel. It is to be filled with, amongst others, infection parameters decrypted from the registry and an import table giving access to `kernel32.dll` and kernel functions.

```
(01) PAGE:004ED1AD                                loc_4ED1AD: [...]
(02) PAGE:004ED1BC 50                             push    eax             ; BaseAddress
(03) PAGE:004ED1BD 57                             push    edi             ; ProcessHandle
(04) PAGE:004ED1BE E8 19 8C F1 FF                 call DS:ZwAllocateVirtualMemory
(05) PAGE:004ED1C3 3B C3                          cmp     eax, ebx
(06) PAGE:004ED1C5 8B 4D FC                       mov     ecx, [ebp+BaseAddress]
(07) PAGE:004ED1C8 89 4E 0C                       mov     [esi+0Ch], ecx
(08) PAGE:004ED1CB 7C 2E                          jl      short loc_4ED1FB
(09) PAGE:004ED1CD 38 5D 0B                       cmp     byte ptr [ebp+ProcessHandle+3], bl
(10) PAGE:004ED1D0 74 27                          jz      short loc_4ED1F9
(11) PAGE:004ED1D2 8B 45 D0                       mov     eax, [ebp+var_30]
(12) PAGE:004ED1D5 89 45 F8                       mov     [ebp+ProtectSize], eax
(13) PAGE:004ED1D8 8D 45 F4                       lea     eax, [ebp+OldProtect]
(14) PAGE:004ED1DB 50                             push    eax             ; OldProtect
(15) PAGE:004ED1DC 68 04 01 00 00                 push 104h
(16) PAGE:004ED1E1 8D 45 F8                       lea     eax, [ebp+ProtectSize]
(17) PAGE:004ED1E4 50                             push    eax             ; ProtectSize
(18) PAGE:004ED1E5 8D 45 FC                       lea     eax, [ebp+BaseAddress]
(19) PAGE:004ED1E8 50                             push    eax             ; BaseAddress
(20) PAGE:004ED1E9 57                             push    edi             ; ProcessHandle
(21) PAGE:004ED1EA E8 93 96 F1 FF                 call loc_406882 ; ZwProtectVirtualMemory
(22) PAGE:004ED1EF 3B C3                          cmp     eax, ebx
```

Figure 3. Function calling ZwProtectVirtualMemory.

B8	00	00	00	00	8D	54	24	04	9C
6A	08	E8	00	00	00	00	C2	14	00

ZwAllocateVirtualMemory :

.text:00405DDC	B8	11	00	00	00	mov eax 11h
.text:00405DE1	8D	54	24	04		lea edx [esp+ProcessHandle]
.text:00405DE5	9C					pushf
.text:00405DE6	6A	08				push 8
.text:00405DE8	E8	B9	20	00	00	call sub_407EA6
.text:00405DED	C2	14	00			retn 14h

ZwProtectVirtualMemory :

.text:00406882	B8	89	00	00	00	mov eax 89h
.text:00406887	8D	54	24	04		lea edx [esp+ProcessHandle]
.text:0040688B	9C					pushf
.text:0040688C	6A	08				push 8
.text:0040688E	E8	13	16	00	00	call sub_407EA6
.text:00406893	C2	14	00			retn 14h

Figure 4. Integrity mask applied on ZwAllocateVirtualMemory and ZwProtectVirtualMemory. Values filled in gray are those checked.

This import table will be used by both the executable code that `Duqu` is going to inject and the payload.

The initialization phase ends by setting up a notification triggered each time a module is loaded into memory (through the system call `PsSetLoadImageNotifyRoutine`).

3.2 Code injection

3.2.1 Processing the first notification

Before the injection. The driver is notified each time a module (DLL or EXE file) is loaded into memory. Each time, the driver checks if the Windows version is supported, then it tries to locate the mapped module. To do so it uses the process `id` given as a parameter by the OS. It reads the file's base address from the PEB (Process Environment Block) structure and compares it to the address passed by the operating system. It checks that the configuration file is decrypted in the shared memory and reads the target file field. As explained by `CrySys`' document [4], the target file in that case is `services.exe` so from now on we will focus on that process and the injection into it.

Payload injection. `Duqu`'s driver is now going to inject malicious code into `services.exe`. Thus the payload will be executed by `services.exe` before `services.exe`'s legitimate code.

Once `services.exe` is loaded, the driver determines its entrypoint and allocates memory (with `ZwAllocateVirtualMemory`) in the `.data` segment. It injects two `PE` files with altered headers. Then it restores the missing constants ('`MZ`',

'IMAGE_NT_SIGNATURE', 'IMAGE_PE_i386_MACHINE, and 'IMAGE_PE32_MAGIC') of the first injected code. Finally it proceeds with the addresses relocation and modifies the permissions of services.exe's entrypoint from RX (PAGE_EXECUTE_READ) to RWX (PAGE_EXECUTE_WRITECOPY) using ZwProtectVirtualMemory.

Duqu's nfrd965.sys allocates memory in the services.exe process, its size is 57 bytes plus the size of the decrypted DLL. The payload (the DLL NETP191.PNF) is decrypted and copied there. Next a *handler* is opened on the kernel driver (nfrd965.sys), saved in a shared structure in order to be gathered by the injected code.

3.2.2 Processing the second notification

The driver is not only notified when the main module (process services.exe) is loaded, but also when DLLs linked to that module are loaded. In particular, when kernel32.dll is loaded, the driver looks for the addresses of 10 functions exported by kernel32.dll that will be used by the payload. Trying to be stealthy, the search consists in comparing the hashed names of 10 functions. This processing ends with saving the first 12 bytes of the entrypoint assembly code of services.exe and their replacement by a jump on the first injected (and restored) code. The first instructions of the entrypoint are changed into mov eax,@adresseInjection followed by call eax.

The process services.exe has been altered and is now ready to launch the payload.

3.3 Launching the payload

The operating system finishes the initialization of the services.exe process and proceeds with its execution by passing control to the code at the entrypoint. Actually the system starts the first injected code.

Its first task consists in determining its own absolute memory address (with the instruction sequence *call-pop*) because further processing (read, write, jump) depend on it. During execution the addresses are relocated with respect to the absolute address of the entrypoint.

It then restores the headers of the second injected code so it is a valid PE and fills, within a shared structure, an import table from the 10 functions found in kernel32.dll. Then it creates a handler on ntdll.dll which is stored in a shared structure. It then jumps to the entrypoint of the second injected code.

This additional module adds data from its own PE header (address of the module, number of sections, address of the export table) to the shared structure. Finally these informations are used to map the PE into memory manually: it allocates memory space, copies the PE header, maps sections, loads DLLs, creates the import table, relocates the addresses and finally determines the entrypoint. Then the function relocates Duqu's main decrypted DLL, NETP191.PNF, as a DLL linked to the PE just mapped and calls its entrypoint. Figure 5 sums up the injections done by Duqu into services.exe and the system memory.

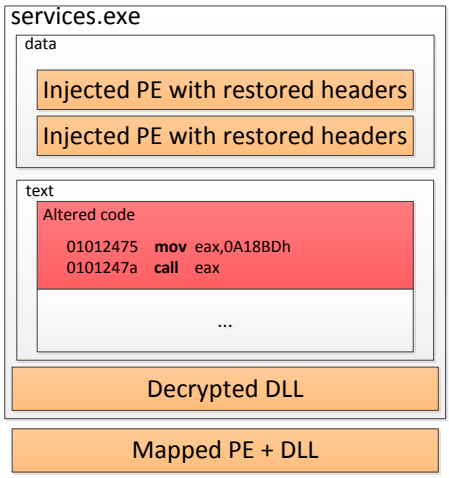

Figure 5. Memory space injected by Duqu into services.exe once the injection is done.

The payload contained in the DLL is now in place and executed. Once it is done it sends a request to the driver through the access point \Device\{624409B3-4CEF-41c0-8B81-7634279A41E5} so it restores the 12 first bytes located at the entrypoint of services.exe. A second request is finally sent to restore the access permission of services.exe's entrypoint.

The injection is now done, control is then passed back to the restored services.exe.

4 Turning the driver

In the previous paragraph we described how the Duqu's DLL is injected in services.exe. Some of these mechanisms, for instance the notifications once a module is loaded, can be used for defensive purposes.

In a nutshell the modified driver will calculate signatures (checksums) on binaries upon notification that

they are loaded into memory. On reception of further notifications, if the checksum has changed, an alert is risen and further actions might be taken. Let's now go into some details of the initialization, memorization and detection phase of the defensive driver. We'll end this paper by a demonstration of the defense granted by the modified driver against an attack by Duqu.

4.1 Initialization phase

The initialization phase of our driver has been greatly simplified. We kept the creation of the access points, removed the search for the ZwProtectVirtualMemory function. We kept the handling of notifications when modules are loaded and also asked for notifications when the system finishes creating a process (function PsSetCreateProcessNotifyRoutine).

4.2 Memorization phase

We saw that the alteration is not done when the first notification is triggered. Thus, if a checksum had been done at that point, the modification of the entrypoint could be detected when the second notification is triggered.

We reused Duqu's checksum function which was originally called for obfuscating function names.

A first notification is received when a process is created. Unfortunately Windows passes only the process id, its parent and whether it was created or destroyed. However we can retrieve the PEB (Process Enviroment Block) structure associated to that process. We used that information to find the memory address of the loaded file.

In order to detect Duqu, we focused on services.exe, we compare the process name to the string "services.exe". If it is the target process, we look for its entrypoint, we calculate its hashed value and store it as an initial signature. The defensive driver is now ready to detect the hook made by Duqu.

4.3 Detection phase

When a module is loaded, the operating system passes control to the defensive driver which looks for the entrypoint of the module. If the loaded module is a DLL, the entrypoint is searched in the PE file of the executable file it is linked to. We reused what was done in Duqu and added the verification of the hash.

The hash of the PE file has been determined and is to be compared to the original signature previously stored. If both checksums are different, we infer that

```
-+* Create process 0x914 *+-
ProcessImageInformation:
  PEB=0x7ffd6000 ImageBaseAddress=0x01000000
  UniqueProcessId=0x914
Entrypoint bytes at 0x01012475:
  0x6a 0x70 0x68 0xe0 0x15 0x00 0x01 0xe8
ProcessImageName: Desktop\services.exe
ProcessImageName: save processID=0x914
CreateProcessNotify: ImageBaseAddress=0x01000000
  EntryPoint=0x01012475
  EntrypointChecksum=0x49af1bf2
```

Figure 6. From WinDbg: The process services.exe is loaded. The defensive driver stores its id (0x914), its entrypoint (0x01012475) and its hash (0x49af1bf2).

Duqu hooked services.exe between the two notifications. Since the entrypoint has been altered, the process services.exe is flagged as suspicious and submitted to further analysis.

4.4 Proof of concept

Aiming to debug and test the original driver and the defensive one, we followed the steps described by Sergei Shevchenko [6] who proposes to rename the Windows calculator calc.exe as services.exe and launch it to watch how the drivers react.

For this example we used two virtual machines using Windows XPSP3 connected by a serial link. The first machine runs Microsoft's WinDBG for kernel debug. The other machine is launched in "kernel debug" mode which allows the debug machine to communicate on a kernel level and debug drivers.

While doing tests we uncovered that Duqu's nfrd965.sys checks if the system is in debug or failsafe mode so we had to patch that out for test purposes, thus allowing debug. We also configured both drivers so they can be launched on demand, it was needed to modify registry keys (the Start parameter has to be set to 3). This configuration provides us with the possibility to choose the launch order of both drivers and services.exe.

We first launched the defensive driver, then Duqu's driver and finally services.exe. In the debugger console, shown on Figure 6, we see when services.exe is launched. The system notifies the defensive driver which outputs information about the loaded module then stores its id, the address of its entrypoint and its initial signature (checksum of the first bytes at the entrypoint).

When the notification for kernel32.dll is triggered to the defensive driver, no modification has been made

```
* Loaded module \WINDOWS\system32\kernel32.dll *
LoadImageNotifyRoutine:
  ImageBaseAddress=0x7c800000 ProcessId=0x914
-> Verify services.exe process:
   Entrypoint at 0x01012475:
     0x6a 0x70 0x68 0xe0 0x15 0x00 0x01 0xe8
-> OK!
* Loaded module \WINDOWS\system32\shell32.dll *
LoadImageNotifyRoutine:
  ImageBaseAddress=0x7c9d0000 ProcessId=0x914
-> Verify services.exe process:
   Entrypoint at 0x01012475:
     0xb8 0xbd 0x18 0x0a 0x00 0xff 0xd0 0xe8
-> Checksum error !!!!
-> Terminating services.exe
```

Figure 7. Detection of the altered entrypoint (0x01012475) of services.exe.

since Duqu's driver will receive the notification afterwards (due to the launch order of drivers). So the checksum succeeds. However there are further notifications, for instance when the linked DLL `shell32.dll` is loaded, the defensive driver checks once again the entrypoint's hash and it has been altered. It is shown on Figure 7. Thus the alteration of `services.exe` is detected and the defensive driver takes further steps to protect the system: it terminates `services.exe`, ending Duqu's attempt to compromise the machine.

5 Conclusion

Similarities between Duqu and Stuxnet lead us to look for a detection method of Duqu when the attack is going on. We described by and large the infection technique of Duqu and how its driver operates, stealthy injecting code into `services.exe` using kernel functions. Thus we rebuilt a source code for Duqu's driver and created a defensive version from this source code. Our modified driver is able to detect the injection made by Duqu and protects the system by terminating the infected process `services.exe`.

Duqu was considered at its times as one of the most sophisticated malware. And the above shows that, indeed, the malware was built with great care. At the same time, it is known that complex systems may be fragile. Usually, it is on the defender side that we make the observation: complex infrastructures offer a lot of entry points to malware. Here, the argument is opposite: it's the malware which was fragile and we exploited this feature to turn it for our own purposes.

References

[1] CVE-2011-3402. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3402.

[2] Hex-Rays Decompiler. http://www.hex-rays.com/products/decompiler/index.shtml.

[3] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Architecture of a Morphological Malware Detector. *Journal in Computer Virology*, 5:263–270, 2009.

[4] L. o. C. o. S. S. (CrySyS). Duqu: A Stuxnet-like malware found in the wild, October 2011.

[5] Kaspersky. The mystery of Duqu: Part Ten, Mar. 2012. https://www.securelist.com/en/blog/208193425/The_mystery_of_Duqu_Part_Ten.

[6] S. Shevchenko. Actually, my name is Duqu - Stuxnet is my middle name. http://baesystemsdetica.blogspot.fr/2012/03/actually-my-name-is-duqu-stuxnet-is-my_4108.html.

[7] Symantec. W32.Duqu: The Precursor to the Next Stuxnet, October 2011.

[8] A. Thabet. Reversing Stuxnet's Rootkit (MRxNet) Into C++, Jan. 2011. http://amrthabet.blogspot.fr/2011/01/reversing-stuxnets-rootkit-mrxnet-into.html.

[9] A. Thierry, G. Bonfante, J. Calvet, J.-Y. Marion, and F. Sabatier. Recognition of binary patterns by Morphological analysis. *REcon*, 2012.