NCC Group Malware Technical Note

Derusbi Server variant (November 2014)

Handling information

This document was produced by the NCC Group Cyber Defence Operations team. The content of this document should be considered proprietary information.

NCC Group gives permission to copy this report for the purposes of disseminating information at TLP WHITE. Please see the US CERT website for full details of the traffic light marking system.

Contents

Introduction

This document provides brief technical details of one of the pieces of malware found during an incident response engagement conducted on behalf of a client in November 2014.

During the overall incident the attackers made a number of attempts to remain persistent in the network including:

- Elevating privileges with a technique similar to the Churrasco IIS6 token kidnapping exploit.
- Dropping ASPXSpy webshells on public facing servers.
- Using the Gh0st RAT.

The most sophisticated persistence attempt included the installation of the Derusbi Server backdoor on a number of machines. This backdoor has previously been reported by both RSA[1] and Novetta[2].

This document outlines the key features of the persistence mechanism, which appears to be different from previously reported samples seen in prior attacks. A number of mechanisms for detecting whether the Derusbi Server backdoor is installed on a computer are also provided.

Initial samples

Two files were obtained from a Windows 2003 Server which had been created by the attacker. Both were owned by an administrative user due to the attackers using pass the hash techniques to impersonate a domain administrator.

No specific changes to the operating system's configuration for persistence were identified. Further analysis showed that the execution of the code contained in the malicious `wmi.dll` relies on DLL load order hijacking. The legitimate `wmi.dll` is located in `%System32%`, however the malicious one is loaded first by `wmiprvse.exe` as both files are located in the same directory.

During the execution of the malware a number of additional DLLs and device drivers are deobfuscated and unpacked. However, although there are several layers to the malware, only two of these files are written to disk. An overview of all files, including those unpacked and executed in memory, is provided in the section Summary of files.

Original file path	MD5
`C:\Windows\system32\wbem\wmi.dll`	`81DF67FCA641A3EEA60072E81CEE039A`
`C:\Windows\system32\wmrkins.tbl`	`76D62E98AF4E9235DCC8AEDDF32840AC`

[1] http://www.emc.com/collateral/white-papers/h12756-wp-shell-crew.pdf
[2] http://www.novetta.com/wp-content/uploads/2014/11/Executive_Summary-Final_1.pdf

Analysis of injection mechanism

Stage 1: WMI.DLL

When loaded the DLL `WMI.DLL` reads `wmrkins.tbl` into heap memory. It then exclusive-ors (XORs) each byte in the file with 0xB5 to decode its contents. Finally it calls a decompression routine using MiniLZO[3] and writes the decompressed data to an executable buffer on the heap. A new thread is then created with code execution commencing at the start of the decompressed code. The usage of MiniLZO in this DLL is consistent with usage elsewhere in the Derusbi Server backdoor.

The code also tries to open `C:\windows\temp\WMI0I0ARK.AX` for write access with the attribute set to hidden. However, it does nothing with this file at this time. The presence of this file can be used to identify machines which have been infected. The code installs a Windows messaging hook using `SetWindowsHookExA`, though this appears to do nothing and the callback function simply passes the message using `CallNextHookEx`.

The DLL exports the same functions as the real `wmi.dll` and essentially proxies them. `DllMain` loads the real `wmi.dll` and stores its base address in a global variable. Each exported function calls a function which uses `GetProcAddress` to find the real code and then jumps to it.

Stage 2: WMRKINS.TBL

The unobfuscated and decompressed code extracted from `wmrkins.tbl` runs directly from memory. At least some parts, if not the entire code stub, have been hand-coded in x86 assembly language. The main purpose is to extract an embedded PE stored within the binary image.

The code contains a PE loader that extracts each section into virtual memory, resolves imports and calls the entry point. It provides the equivalent of `LoadLibrary` without the need to write the PE to disk.

The code is highly similar to a typical exploit payload in that it is position-independent binary with no external dependencies. All necessary APIs are found directly using native implementations of `GetModuleHandle` and `GetProcAddress` using hashes rather than names. The following APIs are resolved and stored in a table:

Library	Function	Offset
ntdll	memcpy	0x0c
ntdll	memset	0x10
ntdll	RtlAllocateHeap	0x18
ntdll	RtlReAllocateHeap	0x14
kernel32	GetProcAddress	0x04
kernel32	GetProcessHeap	0x1c
kernel32	IsBadReadPtr	0x20
kernel32	LoadLibraryA	0x00
kernel32	VirtualAlloc	0x24
kernel32	VirtualFree	0x28
kernel32	VirtualProtect	0x2c

[3] http://www.oberhumer.com/opensource/lzo/

Stage 3: Embedded DLL

The DLL extracted has no exports and all functionality is exposed via the entry point `DllMain`. When loaded it performs the following actions.

1. Creates a mutex called `symantec_srv002`
2. Decompresses an embedded DLL to memory
3. Attempts to open `services.exe` or if that fails `dllhost.exe`
4. Enables debugging privileges for itself
5. Injects the uncompressed DLL into the remote process
6. Injects a loader stub into the remote process
7. Creates a remote thread to call the loader stub

The loader stub is functionally identical to the PE loader in the previous version. There are minor implementation-level differences; functions are resolved by name rather than hash, for example. The embedded DLL is decompressed using the same MiniLZO code as in `wmi.dll`. From its export table we can deduce that the embedded and injected DLL is called `OfficeUt32.dll`.

Stage 4: OFFICEUT32.DLL

This library is the usermode portion of the Derusbi Server variant and has five exported functions:

- `DllRegisterServer`
- `Func`
- `ServiceMain`
- `SvchostPushServiceGlobals`
- `WUServiceMain`

The entry point resolves the module name and stores it in a global variable, creates a thread and exits. The thread created sleeps for 60 seconds then calls another function to spawn a worker thread and exits. The worker thread performs a number of tasks.

1. Initializes a random number generator then enables various privileges: `SeDebugPrivilege`, `SeLoadDriverPrivilege`, `SeShutdownPrivilege` and `SeTcbPrivilege`.
2. Checks whether `\Device\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}` exists and, if not, extracts an embedded kernel driver and loads it. The kernel driver is obfuscated by exclusive-or with a constant 32-bit mask and compressed using LZO. The driver is written to `dump_dumpfve.sys` under the system's `drivers` directory and loaded. This embedded driver matches previous reporting about the Derusbi Server backdoor.
3. Creates another thread which contains the main command handler.
4. The command handler extracts an embedded DLL and writes it to disk using the same obfuscation and compression as the driver. From its export table we can see the DLL is called `Update.dll`.

The code has a number of unusual features. There is extensive use of SSE assembly instructions. Normally this type of assembly code is only emitted by a compiler for highly optimized intrinsic functions such as `memset` and `memcpy` or computational routines that vectorize easily.

The code checks the capabilities of the CPU extensively and in some cases selects routines based on those capabilities.

Several analysed routines which would not appear to benefit from SSE implementation (such as exclusive-or unmasking of embedded files) use SSE assembly instructions. This suggests parts of the program may have been hand-coded is assembly. However, there are features that suggest it is compiled code (failure to reuse dead registers for example). The most likely explanation is that it was compiled using the Intel Compiler which tends to use vectorization more aggressively.

Stage 5: dump_dumpfve.sys

This kernel driver is the Derusbi Server backdoor, which has been widely reported previously.

The PE timestamp is 3rd April 2014 04:24:28 and the driver is signed by "Luzhoushi Huicheng Technology Co.,Ltd.", certificate serial `1F 28 51 FC 5D F4 5D A8 44 65 05 20 39 08 F4 22`. This certificate expired on the 27th September 2014.

The driver installs custom firewall hooks which enable it to listen on any open port and coexist with other network services. A magic "handshake" value is used at the start of connections to enable the attackers to trigger the backdoor.

Detection mechanisms are discussed later in this document.

Stage 6: Update.dll

This library has a single exported function called `Func`. All functionality is exposed through this function.

When called it starts a thread which creates a named pipe called `\\.\pipe\usbpcg`**`NNNN`** where `NNNN` is a decimal number derived from taking the process identifier and calling `ProcessIdToSessionId` on it. It then goes into an infinite loop reading from the pipe and taking appropriate action.

Code to connect to the named pipe can found in `OfficeUt32.dll`.

Identifying suspicious files

Infected machines may be identified by:

- Searching for the MD5 hashes listed above.
- Using Yara rules (below) to search potentially compromised systems.
- Using ssdeep fuzzy hashes (below) to search potentially compromised systems.

Yara rules

The following Yara[4] rules can be used to identify the main files reported in this document. Further rules are available as a separate file.

An entire system can be searched using a command similar to: `yara –r rules.yar c:\`

```
rule wmi_dll {
  meta:
    description = "WMI shim / hook, used to load Derusbi"
    author = "David Cannings, NCC Group"
    tlp = "WHITE"
    md5 = "81DF67FCA641A3EEA60072E81CEE039A"

  strings:
    $str01 = "Function can not be found %hs,Program can not run properly"
    $str02 = "Unable to load %s,Program can not run properly"
    $str03 = "\\temp\\WMI0I0ARK.AX"
    $str04 = "\\wmrkins.tbl"
    $str05 = "RK_Wmi.dll"

    // Legitimate Windows APIs which are hooked by this DLL
    $str06 = "WmiQueryAllDataA"
    $str07 = "QueryAllTracesA"

  condition:
    3 of them
}

rule derusbi_server_kernel_driver {
  meta:
    description = "Strings from a Derusbi Server variant kernel driver"
    author = "David Cannings, NCC Group"
    tlp = "WHITE"
    md5 = "728A72B076EFF5E1A8887F3FF7D5F3BC"

  strings:
    $str01 = "%x:%d->%x:%d, Flag %s%s%s%s%s, seq %u, ackseq %u, datalen %u"
    $str02 = "%x->%x, icmp type %d, code %d"
    $str03 = ", id %d, seq %d"
    $str04 = "%x->%x" fullword
    $str05 = "\\BaseNamedObjects\\EKV0000000000" wide
    $str06  =  "\\Registry\\Machine\\System\\CurrentControlSet\\Control\\Class\\{4D36E972-
E325-11CE-BFC1-08002BE10318}" wide
    $str07 = "{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}" wide

  condition:
    4 of them
}
```

[4] http://plusvic.github.io/yara/

Ssdeep fuzzy hashes

Fuzzy hashes generated using ssdeep are provided below. These have been generated from the two suspicious kernel drivers identified during the investigation.

These may be checked against suspicious files or a whole system using:

```
ssdeep -r -m rules.ssdeep c:\
```

```
ssdeep,1.1--blocksize:hash:hash,filename
384:q8999PUoojRezchW6x3iW4ndpkEYPLqK9y2g1eMb:1TlzcNMIE,"fake_wmi_dll"
768:s9dPsukYL9HhWTT3Lv3jf0M41Lhg3KeSa+Eu5Vz:5NI9HQnfsM41Lhg3LVg,"kernel_driver"
```

Identifying infected machines

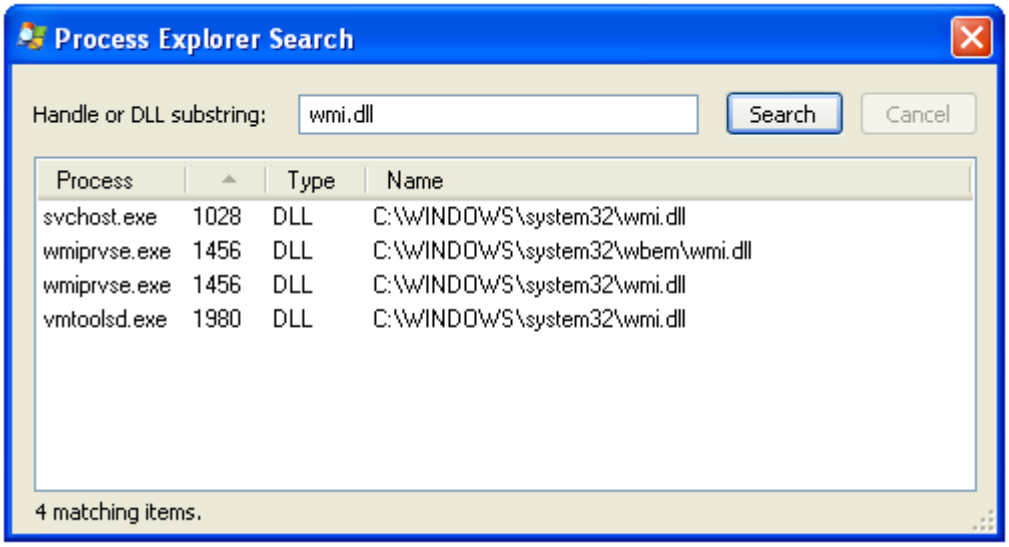
This section provides a number of techniques which can be used to identify an infected machine. A number are specific to the persistence mechanism and should not be relied upon as complete assurance that a machine is not infected.

One simple mechanism is to look for the presence of the file `C:\windows\temp\WMI0I0ARK.AX`, which was always empty during our testing. However, it is possible this file will be used or deleted if the attackers successfully establish communication with the backdoor.

Persistence mechanism - DLL Hijacking

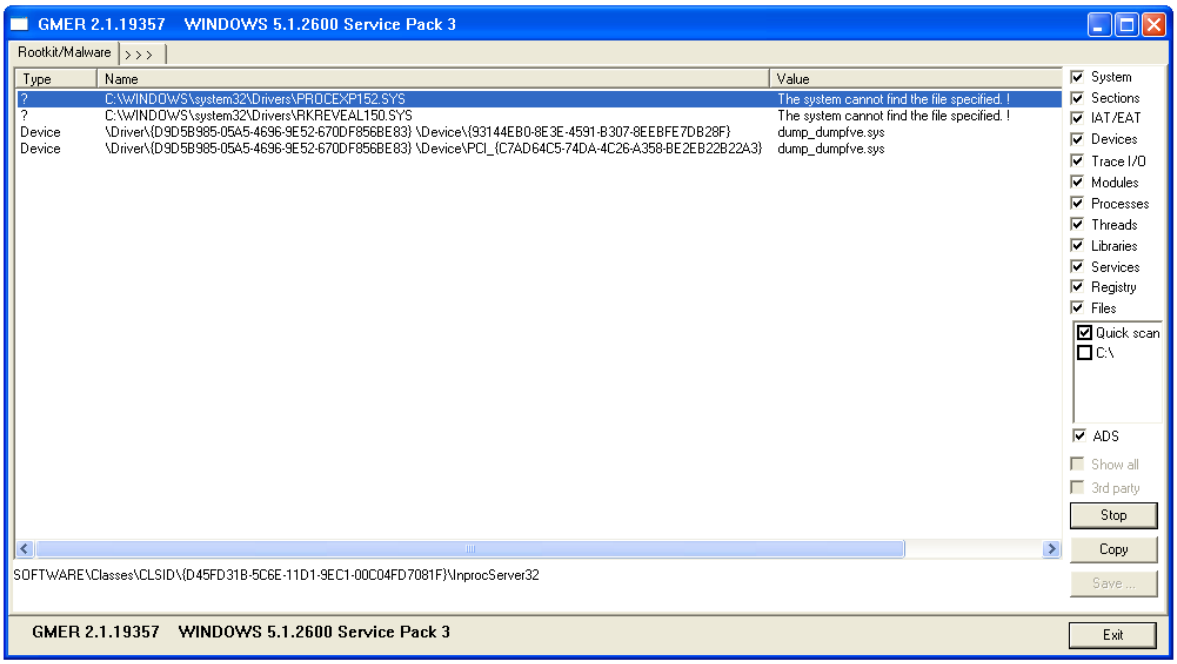
The specific persistence mechanism used in this example can be identified using Process Explorer, when `wmiprvse.exe` is running. However, note that many other persistence mechanisms could be used to load the Derusbi Server malware.

Below is an example of Sysinternals Process Explorer showing the malicious proxy `wmi.dll` loaded in `wmiprvse.exe` (from `%System32%\wbem\`) and the legitimate `wmi.dll` (in `%System32%`).



Identifying driver components - GMER

The GMER tool shows a number of the kernel devices, most likely because the original driver does not exist on disk after it has been loaded.



Identifying driver components – accesschk

The Sysinternals tool accesschk can be used to search for GUIDs associated with the driver. The GUID values used here match previous reporting from Novetta, suggesting that they have not been changed between driver versions.

The relevant objects are:

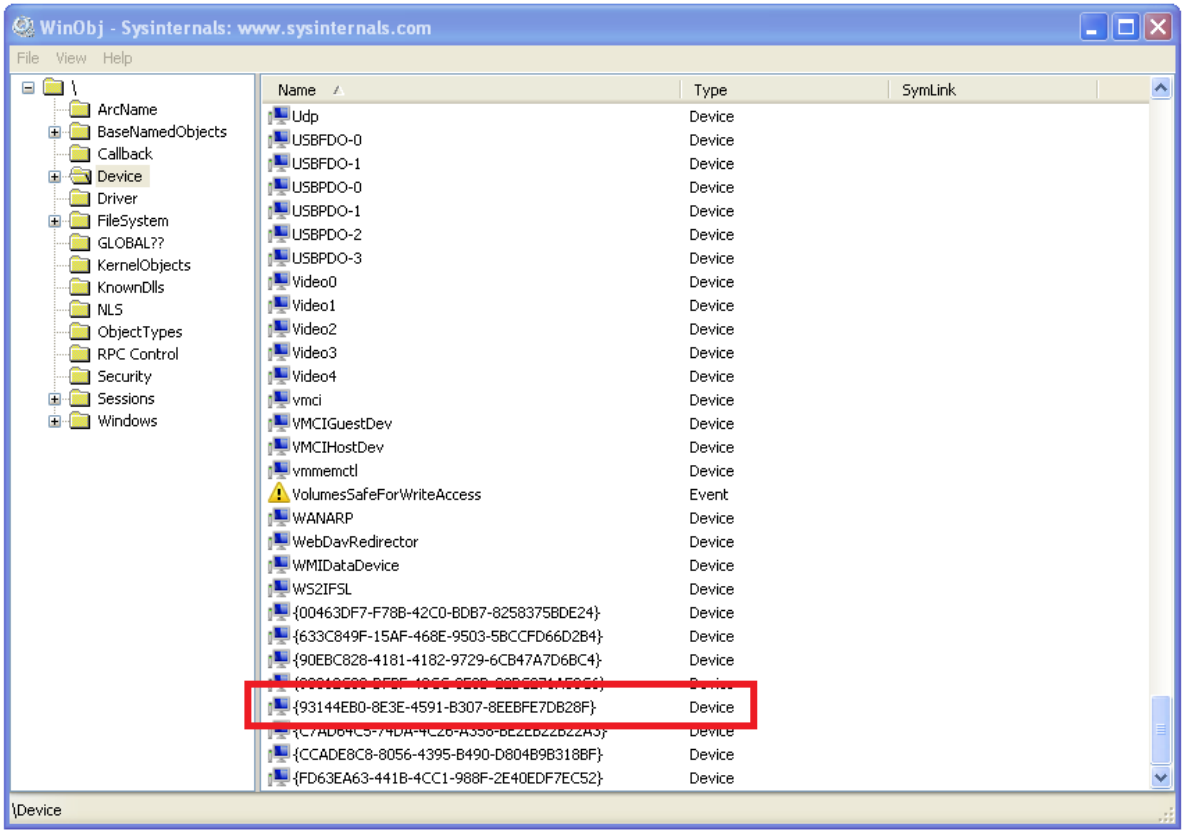
- Device: `\Device\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}`
- Driver: `\Driver\{D9D5B985-05A5-4696-9E52-670DF856BE83}`
- Object: `\BaseNamedObjects\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}`
- Object: `\BaseNamedObjects\{93144EB0-8E3E-4591-B307-8EEBFE7DB28E}`

On an infected machine, the output looks similar to the below:

```
C:\Documents and Settings\Administrator\Desktop\Bundle>accesschk -o -q -u -accep
teula \BaseNamedObjects\{93144EB0-8E3E-4591-B307-8EEBFE7DB28E}
\BaseNamedObjects\{93144EB0-8E3E-4591-B307-8EEBFE7DB28E}
  Type: Event
  RW Everyone

C:\Documents and Settings\Administrator\Desktop\Bundle>accesschk -o -q -u -accep
teula \BaseNamedObjects\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}
\BaseNamedObjects\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}
  Type: Event
  RW Everyone

C:\Documents and Settings\Administrator\Desktop\Bundle>accesschk -o -q -u -accep
teula \Driver\{D9D5B985-05A5-4696-9E52-670DF856BE83}
No matching objects found.


C:\Documents and Settings\Administrator\Desktop\Bundle>accesschk -o -q -u -accep
teula \Device\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}
\Device\{93144EB0-8E3E-4591-B307-8EEBFE7DB28F}
  Type: Device
  RW Everyone
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
  R  NT AUTHORITY\RESTRICTED
```

Identifying driver components – WinObj

The same GUIDs can also be looked for manually using WinObj. Below is an example of one kernel object in `\Device`:



Identifying over the network

NCC Group have written a Python script that will trigger known variants of the Derusbi Server backdoor using the correct "magic" values. This script is available separately.

Below the script is shown when run against an infected machine. Note that the port(s) used for testing need to be open, if Windows Firewall is blocking connections then the packets do not appear to reach the kernel driver.

```
-> % python derusbi-server-trigger.py 192.168.99.50
[+] Sending 64 byte handshake to 192.168.99.50:80
[E] Port not open or timed out reading from socket!
[+] Sending 64 byte handshake to 192.168.99.50:139
[D] Received:
3b12751cc4ed8ae324ea38767854736d4d08d067be548258be66db43c257461241582b5d8c63fa03306f7f5270
5af00aa7465479860732239512aa7d5b4f6825
[D] Values: 1c75123b e38aedc4 7638ea24
[!] Got a valid reply from 192.168.99.50, this machine is potentially infected!
[+] Sending 64 byte handshake to 192.168.99.50:445
[D] Received:
e658fc1b19a703e4b1f837cc9f0d89738a38410a1b64fd15b87c4f63686ff61a723a7b001460990ecd33d3270d
7ff00444203a18b41fa613664f537133780b19
[D] Values: 1bfc58e6 e403a719 cc37f8b1
[!] Got a valid reply from 192.168.99.50, this machine is potentially infected!
```

Identifying firewall hooks – Volatility

NCC Group have produced a Volatility plugin which can be obtained from our Github repository[5].

When run against a memory dump of an infected machine, the output should look similar to the below. Please note that there are legitimate reasons to use the firewall hooking API on Windows (such as the usage in `ipnat.sys`, below) but this is one way of identifying suspicious behaviour.

```
[FWHook] Found tcpip.sys at offset 0x17b7098 with DllBase 0xf117d000
[FWHook] PE Header Offset: 0x0000d8
[FWHook] Image Base Address: 0x010000
[FWHook] FQ Block Address: 0xf11be860
[FWHook] FQ Counter Address: 0xf11be880
[FWHook] FQ Counter Value: 16 0x000010
[FWHook] Final FQ Block Address: 0xf11be870
[FWHook] Total hooks registered 3
[FWHook] ------------------------------------------------------------------------------
[FWHook] Call Out Address: 0xf0582246
[FWHook] Module Base Address: 0xf0580000
[FWHook] Module DLL Name dump_dumpfve.sys
[FWHook] Module Binary Path \??\C:\WINDOWS\system32\drivers\dump_dumpfve.sys
[FWHook] ------------------------------------------------------------------------------
[FWHook] Call Out Address: 0xf103c6dc
[FWHook] Module Base Address: 0xf1027000
[FWHook] Module DLL Name ipnat.sys
[FWHook] Module Binary Path \SystemRoot\system32\DRIVERS\ipnat.sys
[FWHook] ------------------------------------------------------------------------------
[FWHook] Call Out Address: 0xf0582270
[FWHook] Module Base Address: 0xf0580000
[FWHook] Module DLL Name dump_dumpfve.sys
[FWHook] Module Binary Path \??\C:\WINDOWS\system32\drivers\dump_dumpfve.sys
```

[5] https://github.com/nccgroup/WindowsFirewallHookDriverEnumeration

Summary of files

Please note that because some files are never saved to disk they have been recovered through a combination of memory forensics and reverse engineering. Therefore the MD5 values below represent the files which are provided with this report.

Name	`wmi.dll`
MD5	`81DF67FCA641A3EEA60072E81CEE039A`
Linker timestamp	15th August 2014, 08:37:21
Description	DLL hijack for legitimate `wmi.dll`, proxies requests to the real functionality and loads additional code.

Name	`wmrkins.tbl`
MD5	`76D62E98AF4E9235DCC8AEDDF32840AC`
Description	Compressed and XOR obfuscated data, loaded by `wmi.dll`. Contains all other components. Decompresses to `B4845947A31A7A1C4A271D91A369E975`.

Name	*Unnamed (not saved to disk, discussed in* Stage 3: Embedded DLL*)*
MD5	`15444857488C9C93C672A246F41AE313`
Linker timestamp	7th August 2014, 14:13:51
Description	First DLL, unpacked by `wmi.dll` from `WMRKINS.TBL`.

Name	`dump_dumpfve.sys`
MD5	`C2780885305FCB20D2B76D23AA124D51`
Linker timestamp	3rd April 2014, 04:24:28
Description	Derusbi Server variant, kernel driver.

Name	`officeut32.dll`
MD5	`728A72B076EFF5E1A8887F3FF7D5F3BC`
Linker timestamp	3rd April 2014, 04:25:53
Description	Derusbi Server variant, usermode portion.

Name	`Update.dll`
MD5	`70470DFBD248D8A85843B552D3392156`
Linker timestamp	3rd April 2014, 04:25:38
Description	Unpacked and saved to disk by `officeut32.dll`

Document history

Document History			
Issue No.	**Issue Date**	**Issued By**	**Change Description**
0.1	28/11/2014	Pete Beck	Draft for NCC Group internal use
0.2	08/01/2015	David Cannings	Added Yara rules, ssdeep and network scanning script, added detail ready for public release.
0.3	08/01/2015	David Cannings	Changes from QA process integrated
1.0	12/01/2015	David Cannings	Initial release to industry partners, TLP: AMBER.
1.1	06/03/2015	David Cannings	Public release at TLP: WHITE.