CROWDSTRIKE

DEEP PANDA

INTELLIGENCE TEAM REPORT VER. 1.0

CROWDSTRIKE GLOBAL INTELLIGENCE TEAM
http://www.crowdstrike.com
@CrowdStrike | intelligence@crowdstrike.com

EXECUTIVE SUMMARY

EXECUTIVE SUMMARY

In late December 2011, CrowdStrike, Inc. received three binary executable files that were suspected of having been involved in a sophisticated attack against a large Fortune 500 company. The files were analyzed to understand first if they were in fact malicious, and the level of sophistication of the samples.

The samples were clearly malicious and varied in sophistication. All three samples provided remote access to the attacker, via two Command and Control (C2) servers. One sample is typical of what is commonly referred to as a 'dropper' because its primary purpose is to write a malicious component to disk and connect it to the targeted hosts operating system. The malicious component in this case is what is commonly referred to as a Remote Access Tool (RAT), this RAT is manifested as a Dynamic Link Library (DLL) installed as a service. The second sample analyzed is a dual use tool that can function both as a post exploitation tool used to infect other systems, download additional tools, remove log data, and itself be used as a backdoor. The third sample was a sophisticated implant that in addition to having multiple communication capabilities, and the ability to act as a relay for other infected hosts, utilized a kernel mode driver that can hide aspects of the tool from user-mode tools. This third component is likely used for long-term implantation and intelligence gathering. Some AV engines occasionally identify this sample as Derusbi Trojan. CrowdStrike Intelligence Team has seen Trojans from 8 different builder variants of this RAT, including 64-bit versions, used in targeted attacks in 2011 against Defense, Energy/Power, and Chemical Industries in US and Japan.

All of these samples reflect common toolmarks and tradecraft consistent with Chinese based actors who target various strategic interests of the United States including High Tech/Heavy Industry, Non-Governmental Organizations (NGOs), State/Federal Government, Defense Industrial Base (DIB), and organizations with vast economic interests. This report contains an in-depth technical analysis of the samples, detection/remediation/mitigation information, attribution intelligence, and a conclusion aimed at providing the reader with a synopsis of the report.

TECHNICAL ANALYSIS

TECHNICAL ANALYSIS

Dropper Sample (MD5: 14c04f88dc97aef3e9b516ef208a2bf5)

The executable 14c04f88dc97aef3e9b516ef208a2bf5 is commonly referred to as a 'dropper', which is designed with the purpose of extracting from itself a malicious payload and to initialize and install it into a targeted system. In this case, the malicious payload is a Dynamic-Link Library (DLL), which enables an attacker to have full control of the system. This code appears to have been compiled on Wednesday May 4th, 2011 at 11:04:24 A.M. UTC (equivalent to early evening time in China). Note that the timestamp is in UTC, however the relative time of day in China is provided for the benefit of the reader. The sample first resolves several library functions provided by Microsoft using the LoadLibrary() and GetProcAddress() Application Programming Interfaces (APIs). The imported function names are not encrypted; however, the function name is minutely obfuscated by a simple single character substitution:

```
//Obfuscation of GetTempPathA() API function call
strcpy((char *)ProcName, "2etTempPathA");
ProcName[0] = 'G';
```

The dropper invokes the SHGetSpecialFolderPath() API supplying a Constant Special Item ID List (CSIDL) of 'CSIDL_COMMON_DOCUMENTS' to identify the destination folder for the malicious DLL payload. The CSIDL in this case points to: "The file system directory that contains documents that are common to all users. A typical path is C:\Documents and Settings\All Users\Documents."

The dropper attempts to write the malicious payload to one of the following file names, using the first available name in this set:

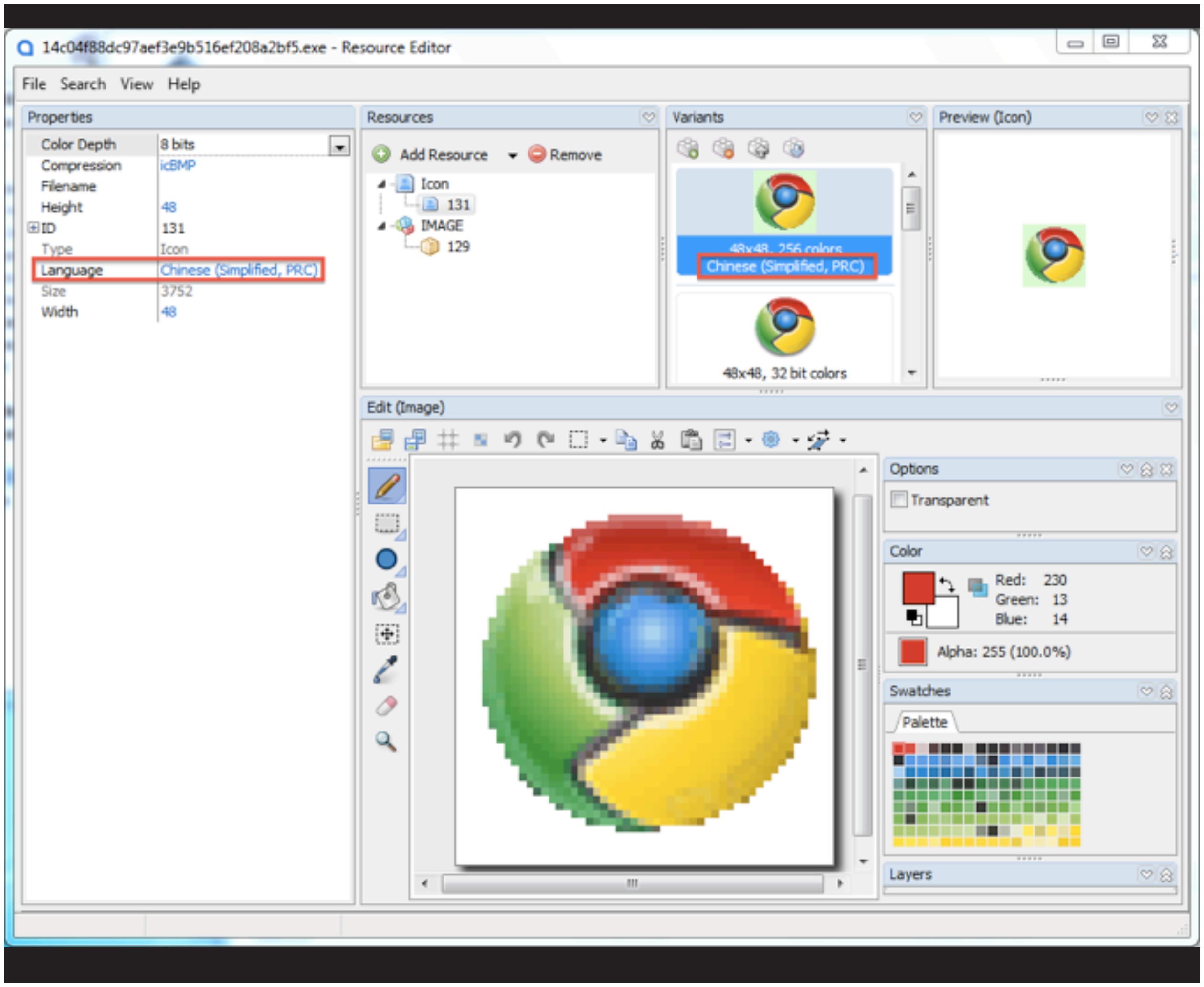
1. infoadmn.dll
2. infoctrs.dll
3. infocardapi.dll

The dropper sets the creation and last written timestamp of the newly created file to the date 2007-03-07 00:00:00; this allows the newly created malicious DLL to blend in with other system files. This is meant to prevent identification during disk forensics using a common investigative technique called a forensic analysis timeline. This date is specified in the dropper code and does not change across multiple infections.

The malicious DLL file that is dropped is hidden in a resource of the dropper binary. This is a relatively common technique used by malware dropper files to optimize the number of files required to infect a machine. The resource language of the malicious DLL is set to "Chinese (Simplified)", this is a compiler artifact which indicates the language setting on the compiler used by the person who built the binary was set to Chinese (Simplified)" at the time the dropper was compiled. The 'MZ' header which denotes a binary executable file of the dropped DLL is initially obfuscated. When the dropper writes the file to disk, the first byte of the file is ‘Z’ which prevents the file from executing or being detected as an executable by many defensive tools. The dropper subsequently opens the dropped file and corrects the header by writing the ‘M’ over the first byte, allowing the file to be executed.

A subroutine to decompress the dropped file is present as ‘dead code’ (code that is not used) in the binary. This subroutine will be invoked on the already closed file handle of the dropped file in the present code version. Since the dropped resource is not compressed, the routine fails. This indicates a low sophistication modification to the original dropper code to make it work with an uncompressed resource.

The final step the dropper performs is to load the dropped DLL into its own process space; it then resolves the export ‘OpenINFOPerformanceData’ from the DLL and invokes it with the dropped DLL’s filename as parameter. This export then implements the actual install logic to maintain persistence and invoke the main routine.



Color Depth	8 bits
Compression	icBMP
Filename	
Height	48
ID	131
Type	Icon
Language	Chinese (Simplified, PRC)
Size	3752
Width	48

48x48, 256 colors
Chinese (Simplified, PRC)
48x48, 32 bit colors

Red: 230
Green: 13
Blue: 14
Alpha: 255 (100.0%)

The dropper binary contains an icon resource that resembles the ‘Google Chrome’ browser icon, the resource language is set to “Chinese (Simplified)”, which is consistent with the builder of the tool having their systems language set to Chinese. The use of the Chrome icon may indicate a possible attempt to socially engineer the intended victim into thinking the dropper was a legitimate file associated with Google.

Backdoor DLL Sample (MD5: 47619fca20895abc83807321cbb80a3d)

This sample is a ‘backdoor’ which is the DLL dropped by the dropper sample file with an MD5 of 14c04f88dc97aef3e9b516ef208a2bf5. This code appears to have been compiled on Wednesday May 4th, 2011 at 10:48:19 A.M. UTC (equivalent to early evening time in China). It is instantiated when it is mapped into the process space of its dropper file, and its’ export named ‘OpenINFOPerformanceData’ is called. This export first attempts to stop a service called “msupdate,” which is not a known Microsoft Windows service despite the appearance. If the service is present, the malware replaces its previous instances or versions of this backdoor. After attempting to disable the existing service, the malware tries to install itself as a service with that same name. During installation, the sample attempts to use documented APIs such as OpenSCManager() and CreateService() to initialize itself as a persistent Windows service. As a precaution, the sample writes settings directly to the Windows Registry to accomplish the same goal if installing the service with the documented APIs fails. The registry change creates the following key:

HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\msupdate\\Parameters

Following this, the subroutine will set the value of the ‘ServiceDLL’ to the module handle of the DLL.

The next key to be changed is:
HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Svchost, which will have the ‘msupdate’ key set to ‘msupdate’.

The export ‘CollectW3PerfData’ is registered as the main function of the DLL. If the installation of the new service is successful, the sample then starts the new service and exits. If the installation fails, the sample spawns a new process using rundll32.exe, this executable will instantiate the DLL and can call a specific exported function. In the case of installation failure, rundll32.exe calls the main functions export ‘CollectW3PerfData’. The rundll32.exe executable is instantiated with a new NULL Security Identifier (SID) (S-1-0-0) with permissions set to grant all access to the file. This allows any user to have complete control over the machine, as rundll32.exe is frequently launched by tasks such as changing the time, wallpaper, or other system settings. This means that after cleaning up the components dropped by the malware, the system remains vulnerable to local attacks by simply overwriting the legitimate rundll32.exe executable with a malicious version and await it’s automatic execution by the Operating System.

The main entry point to the DLL is named ‘CollectW3PerfData,’ as it first creates and displays a fake Window with class “NOD32_%d” where %d is replaced with a pseudo-random number. This may be an attempt to fool some automated dynamic analysis or anti-malware software into believing this is the legitimate ESET AV software. The window is however not visible and implements no specific functionality.

After creating this window, the routine starts the main thread that eventually initiates calling out to the Command and Control (C2). In order to accomplish this task, the newly created thread initializes networking APIs using WSAStartup() and resolves some other APIs dynamically using LoadLibrary() and GetProcAddress(). Once the proper API's have been resolved, the sample then assigns a NULL SID to the rundll32.exe executable and sets the current process' Window Station to "winsta0", which enables the sample to access the real user's desktop if started as service.

The communication to the C2 is handled by a while() loop, with each successive connection attempt causing the loop to invoke the Windows Sleep() API for a time interval of 2 seconds, exponentially increasing in length up to 1024 seconds (17 minutes) and then restarting back to 2 seconds.

Initial C2 Phone Home Beacon

The C2 location in this sample is statically defined as 1.9.5.38:443 (Malaysia: Tmnet, Telekom Malaysia Bhd). While there is 'dead code' that will download the C2 location from an HTTP URL that could be defined in the binary, using the User-Agent string "Google", this code is not activated due to the format of the statically defined C2 location using an IP address. Thus the sample will only attempt to connect directly using a raw socket to the C2 located at 1.9.5.38:443. This indicates the use of a 'boiler plate code' or a builder software package that automates the creation of the malicious sample.

The malicious sample sends an initial beacon to the C2 that includes the following information:

- The computer name as obtained with the GetComputerName() API
- The username of the current Remote Desktop session if currently being executed in a Remote Desktop session or "none" otherwise.
- The currently logged in username in the system as obtained with the GetUserName() API
- The machine's uptime
- The Windows version and Service Pack level
- The amount of available Physical Memory in MB
- Current Remote Desktop sessions as enumerated with WTSEnumerateSessions
- A string identifier statically set to "FBI20111024"

The beacon is encrypted using an XOR/ADD loop using the statically defined key 0x1C and sent to the C2.

The following python function can be used to decode the beacon stings:

```
def decode(crypted):
decoded=""
  for x in crypted:
    decoded+=chr(((ord(x)^(0x1C +1)) + (0x1C +1)) & 0xFF)
  return decoded
```

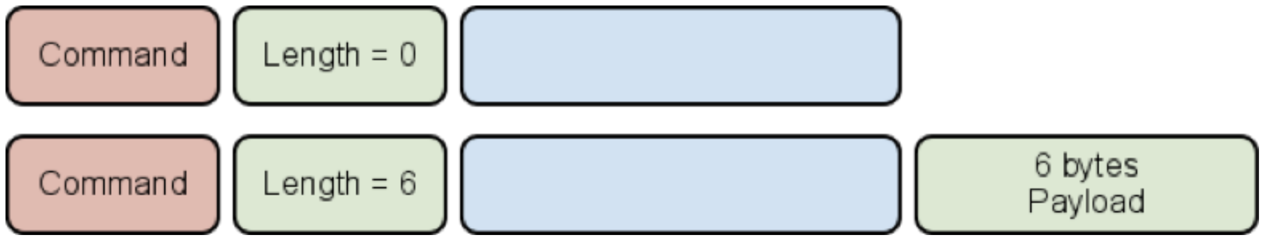
After sending the initial beacon, the routine loops receiving incoming commands and executes them in sequence. When a connection can successfully be established to the C2 server, the sleep timer is reset to two seconds for the next attempt.

Network Protocol and Implementation

The network protocol used by this sample resembles a 'Type-Length-Value' layout in both directions. Each 16 byte request header consists of:

1. A numerical 4-byte (seemingly) little-endian command identifier
2. A 4-byte little-endian payload length
3. 8 bytes remaining are a request header that is typically uninitialized and only used by some commands instead of the arbitrary length payload

Zero or more of specified bytes of additional payload then follows the header.



This inbound payload is received unconditionally and regardless of command type into a fixed-size stack buffer of 408 bytes size. Providing additional payload of any larger size will result in a trivial exploitable stack buffer overflow that allows arbitrary code execution due to the absence of any security features. However, exploitation of this vulnerability is unnecessary due to the already available unauthenticated command execution capabilities of this backdoor.

Certain commands initiate a second connection to the C2 in a separate thread using the same network protocol but providing a different request command identifier than for the initial beacon.

Backdoor Functionality, Supported Commands

The primary aim of this backdoor is remote desktop control functionality comparable to VNC or Remote Desktop over a custom protocol. It allows the adversary to view the main desktop graphically and control the keyboard and mouse. This remote control functionality is implemented as separate messages for mouse clicks, pressed keys, etc. using command identifiers 0x20000002 to 0x20000019. The command 0x22000001 initiates continuous transmission of screen captures to the C2. The screen captures are created using a series of Microsoft Windows Graphic Device Interface (GDI) API calls culminating in a call to GetDIBits().

Command 0x20000001 exits the backdoor and 0x20000000 is issued to completely remove the backdoor from the system.

When command 0x23000004 is received, a temporary new user "_DomainUser_" with password "Dom4!nUserP4ss" is created and added to the local Administrators group. The backdoor is then started under that account and the user is deleted. It would appear this technique is meant to obfuscate the activities of the malicious sample by masking the process creator's user name to appear to be a generic domain user. Note that such an account does not normally exist in an Active Directory environment.

Additionally, the primary C2 connection allows for requests to start additional connections to the C2 implementing the following functionality:

- A process control connection initiated by command 0x25000000 that allows for enumeration and killing of running processes
- A piped command line process connection that allows communication with standard input and output of arbitrary executables; initiated by command 0x23000000
- A file browser connection initiated by command 0x21000000 that allows for
  - Listing directory contents
  - Copying, deleting and moving files
  - Opening files using the ShellExecute() API
  - Downloading and uploading files from / to the C2
- A Remote Desktop session enumerator initiated by command 0x21010000

Post Exploitation Tool Sample (MD5: 2dce7fc3f52a692d8a84a0c182519133)

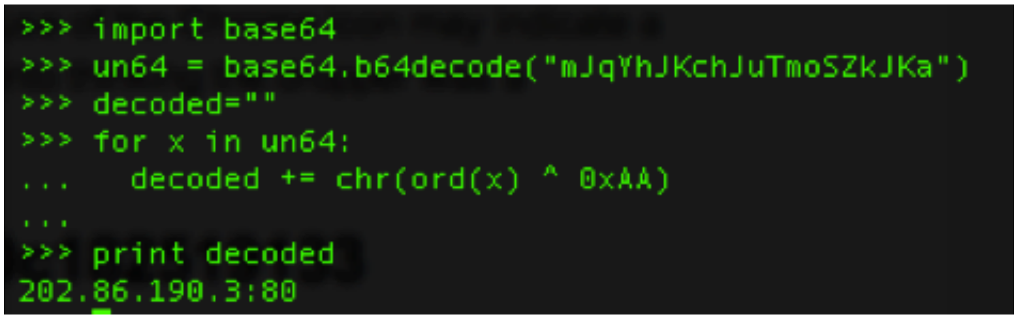
This sample is typical of a post exploitation tool; it is written in .NET 2.0. This code appears to have been compiled on Thursday May 26th, 2011 at 10:21:44 A.M. UTC (early evening time in China). The backdoor functionality can be instantiated either directly from the command line or through commands issued over a network based protocol via the C2. If no arguments are given, a connection to the C2 is initiated to the statically defined IP address. The command line options support post exploitation capabilities such as changing file timestamps, forensic mitigation, privilege escalation, launching the executable, and specifying a specific C2.

One interesting command line option allows the backdoor to filter the contents of specified files to remove content using a regular expression (regex). This command then modifies the creation, modification, and last access timestamps of the modified file to conceal the content modifications. A detailed listing of command line arguments can be viewed in Appendix A. This activity is generally associated with log cleaning to complicate a forensic investigation.

The sample contains an embedded IP address for C2 that is stored in an encrypted format as a string resource:

t = "14mJqYhJKchJuTmoSZkJKa42C7885997B523F63566F407F3834BCC54AAA32524"

The first two bytes of this string represent the base 16 length of the encrypted string, in this case, “0x14.” Following this is a base64 encoded string of the specified length. Once this string has been decoded using base64, the result is then XOR’d with the fixed value of 0xAA yielding the decoded IP Address 202.86.190.3:80 (Hong Kong: TeleOne(HK) Limited).



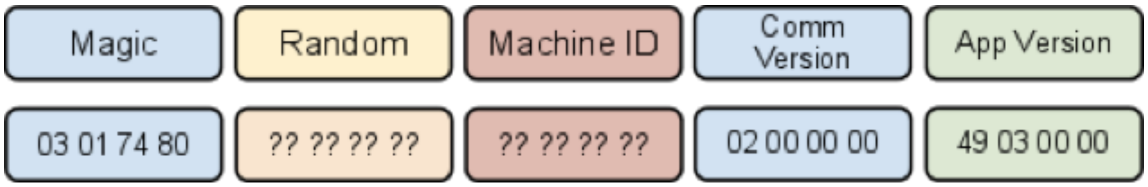
```
>>> import base64
>>> un64 = base64.b64decode("mJqYhJKchJuTmoSZkJKa")
>>> decoded=""
>>> for x in un64:
...   decoded += chr(ord(x) ^ 0xAA)
...
>>> print decoded
202.86.190.3:80
```

Network Protocol and Implementation

There are three components to the protocol:

Authentication is accomplished using a 32 byte packet, this packet consists of:

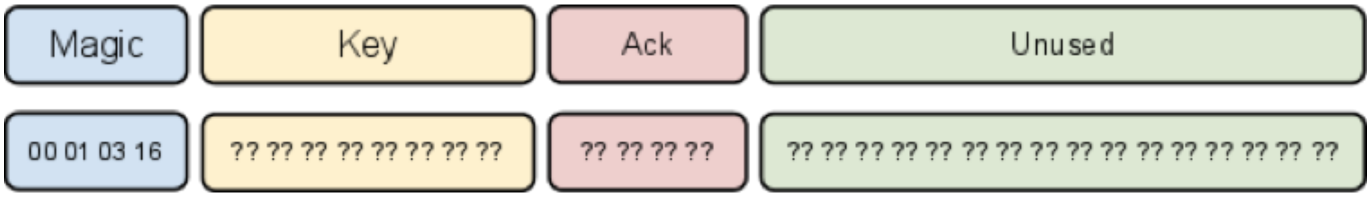
1. A four byte ‘magic’ key which in this sample is statically defined as 0x80740103
2. A four byte random number generated by the rand() function
3. The machine ID comprised of an obfuscated combination of the Machine name and hard drive serial number. The algorithm for generating this is in Appendix B
4. The communication protocol version number, which in this sample is 0x2
5. The version of the malicious sample, in this case it is 841



Magic	Random	Machine ID	Comm Version	App Version
03 01 74 80	?? ?? ?? ??	?? ?? ?? ??	02 00 00 00	49 03 00 00

An example authentication packet sent to the C2 is located in Appendix E

After sending the initial authentication packet, the sample verifies that the first four bytes of the response is equal to a statically defined value, in this sample the value is: 0x16030100. In addition, an 8 byte key is sent to the client which is then RC4 encrypted using the random number generated in step 2 from above as the password. This value is then transformed using a simple algorithm in Appendix F into a 32 byte array. The first 16 bytes of this array are then used as the KEY and the second 16 bytes are used as the IV for setting up AES encryption which is then used to encrypt and decrypt any further communications.



Magic	Key	Ack	Unused
00 01 03 16	?? ?? ?? ?? ?? ?? ?? ??	?? ?? ?? ??	?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

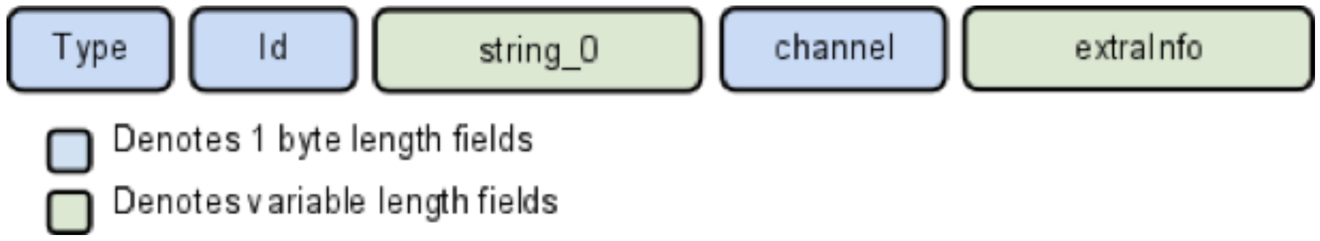
Beacon, this is typical of this type of malicious sample, it allows the operator to separate various infected hosts in a targeted campaign. The beacon for this sample is formatted as XML and consists of:

- The infected machine name
- Current time zone
- Windows version
- Local time/date of the infected machine
- C2 protocol version

An example of an unencrypted beacon:

```
<?xml version=”1.0” encoding=”utf-16”?>
<BasicInfo xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance” xmlns:xsd=”http://www.
w3.org/2001/XMLSchema”>
  <HostName> Infected System Hostname</HostName>
  <int_0>-8</int_0>
  <osVersion>Microsoft Windows NT 6.1.7601 Service Pack 1</osVersion>
  <string_0>12/27/2011 16:34:36</string_0>
  <Version>2</Version>
</BasicInfo>
```

Command handling loop, this is a loop structure that will process and execute commands sent by the C2. The malware sends and receives a heartbeat/keepalive packet every 2 minutes. The command format is derived from a structure consisting of:



These fields are received as a sequence of serialized .NET objects in the order specified. A detailed description of the possible values for commands is in Appendix D. It is important to note that the order in which the application defines them is not the same order as they appear to be coming over the network. Examples of implemented commands include download and upload files, installing new .NET assemblies, calling methods on those assemblies, connecting to new C2 servers and executing processes.
Backdoor DLL Sample (MD5: de7500fc1065a081180841f32f06a537)

Backdoor DLL Sample (MD5: de7500fc1065a081180841f32f06a537)

This sample is a sophisticated backdoor which implements several communications protocols and was developed in C++. This binary is compiled with the /GS flag using Visual Studio 2010, enabling stack buffer overflow detection. This code appears to have been compiled on Sunday October 30, 2011 at 12:43:33 P.M. UTC (late evening time in China). The code contains several Run Time Type Information (RTTI) artifacts that indicate most of the C++ class names were prefixed with the string “PCC_” in the original source code.

Variants of this Trojan are sometimes detected under the name 'Derusbi' by Microsoft, Trend, Sophos and Symantec AV engines.

This sample is a DLL which can be registered as a service and is used to drop a kernel driver and provide an interactive command line shell to the C2. It also is able to bypass User Account Control (UAC) to install itself by using the 'sysprep.exe' Microsoft Windows executable provided by the targeted system. The steps it takes to install itself onto a system are as follows:

1. Copies itself to to %WINDIR%\system32\Msres<3 random characters>.ttf
2. After it copies itself, it will modify the creation time, last access time and last modification time to the current system time when the copy was made but with the year changed to 2005.
3. Adds itself as a service name from the backdoor's configuration under HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\<service>" This defaults to "wuauserv", the legitimate Windows Update service, in the given binary's default configuration.
4. Adds itself to list of services started by 'netsvc' using the service name 'helpsvc'.
5. If McAfee AV is installed, creates a copy of regsvr32.exe named Update.exe and then schedules the copy to be deleted on reboot using the well documented MoveFileExA API.
6. It then calls either the original or copy of regsvr32.exe with the parameters /s /u and the path to the copy of itself it made in Step 1. The /u parameter means "uninstall", which calls DllUnregisterServer, this is an unsophisticated method of DLL entry point obfuscation.
7. DllUnregisterServer installs the driver and initiates the backdoor component.

The sample is capable of 'dropping' an embedded/encrypted kernel driver. If the process "ZhuDongFangYu.exe" is running (AntiVirus360 program from the Chinese 'Quihoo 360 Technology Co., LTD' 360安全卫士), or the username of the DLL's host process context is not 'SYSTEM', the driver is not written to disk. Barring the two aforementioned conditions, the sample decrypts the kernel driver to:

"%sysdir%\Drivers\{6AB5E732-DFA9-4618-AF1C-F0D9DEF0E222}.sys"

Following the decryption and writing of the driver to disk, it is loaded using the ZwLoadDriver() API. The driver is encrypted with a simple four byte XOR key value of 0x2E885DF3; after decryption the file has the MD5 hash of dae6b9b3b8e39b08b10a51a6457444d8.

The malware contains a dynamic configuration stored in the Registry under

"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Rpc\Security"

and loads a default configuration embedded into the binary if such a configuration is not found. The way this default configuration is loaded and parsed indicates that this malware has been built with a 'builder' that takes a template sample and lets an unsophisticated user specify a configuration without recompiling any code.

If the current service name matches a set of pre-defined service names that legitimately exist in Windows, the backdoor then loads the original service's DLL into the address space with LoadLibrary and invokes the ServiceMain export. This effectively hijacks the original service's entry while retaining its functionality.

While there is code in the binary that allows downloading a list of C2 servers from an HTTP URL, the default configuration present specifies 202.86.190.3:80 as a C2 to use, this is the same Hong Kong C2 server as the one used by the post exploitation .NET tool.

C2 Communication Mechanisms

The malware has three distinct C2 protocols two of which can be transmitted over HTTP proxies and one can be bundled in two different 'dual' modes (see 3.), totaling 7 distinct supported C2 mechanisms. The configuration contains the C2 protocol to be used or optionally a self-configuration mode in which the malware attempts the different C2 protocols in a pre-defined order. In self-configuration, a connection via a proxy is attempted if the system wide Internet Explorer settings specify such a proxy. The configuration found in this sample is set to automatic self-configuration, resulting in the following mechanisms being tried in this order:

1. Proprietary binary header (optionally over an HTTP Proxy using CONNECT mechanism); this protocol consists of 64 random bytes being sent to the C2. The C2 then responds with 64 bytes where the first four bytes must match the first four sent bytes to establish a connection successfully. The remaining bytes are discarded. Interestingly, the malware stores the first four bytes rotated right by seven bits and compares that value to the seven bits rotated right version of the server's response, effectively neutralizing the rotation's effect; the purpose of this is unclear.
2. A long-running HTTP POST request to the path "/forum/login.cgi" with a statically defined HTTP request string including HTTP headers (optionally over a HTTP Proxy using CONNECT). The malware requires the response to start with "HTTP/1.0 200" or "HTTP/1.1 200" and an absence of a "Connection: close" header. This one HTTP connection will be used for bi-directional communications, sending chunks of POST payload and receiving chunks of the response, interleaved.
3. Two long-running HTTP requests to the same C2 (optionally over an HTTP Proxy with original request verb), one GET request to "/Photos/Query.cgi?loginid=" followed by a random number and one POST request to "/Catelog/login1.cgi". The GET request serves as a down-stream channel while the POST request serves as a upstream channel.

This demonstrates an attempt to use the most efficient communication channel first, falling back to more legitimate appearing channels as required in order to appear Request For Comment (RFC) compliant with the HTTP protocol.

Additionally, the malware contains a custom DNS client implementation that will use the system's configured DNS server to tunnel C2 traffic over legitimate DNS. Since this C2 mechanism is not attempted in self-configuration and was not configured for this binary, analysis was left out due to time constraints.

After establishing any of the aforementioned channels for arbitrary binary data exchange, the malware will start sending and receiving compressed binary blobs via the channel of choice. The C2's binary data blobs are compressed. No further encryption of the data takes place.

All C2 transport implementations contain code for accepting and handling server-side connections of the respective protocols. However, this code does not appear to be invoked. It appears that the author of this code shares the library that implements these transports for the client with the C2 server.

C2 Command Invocation

The main backdoor thread then reads commands from the chosen C2 protocol and passes them on to any of the following registered handler classes based upon a command ID. The handler class is responsible for parsing the remainder of the command.

PCC_FILE: File Browsing and Data Exfiltration
This handler class for command ID 8 implements generic directory and file browsing using FindFirstFileW() and FindNextFileW() APIs, as well as reading and writing arbitrary files via C2 commands, thus enabling upload and download of arbitrary files. This is typically seen in RATs for searching specific files to exfiltrate. Additionally, this class implements launching of specified executable files via the CreateProcess() API.

PCC_PROXY: TCP Proxy
This handler class implements a generic TCP proxy. It supports establishing TCP connections to other hosts and also listening for incoming connections. The incoming connection contents are forwarded to the C2 and data from the C2 is passed on to connections. It supports up to 1024 parallel connections.

PCC_MISC : Gather and report system information
The malware is capable of gathering various pieces of information from the system, triggered by a command ID 10. The capabilities include recovering authentication credentials from various system and client storage such as Mozilla Firefox, Internet Explorer, and Remote Access Service (RAS). This class also supports gathering intelligence on the infected system including identifying security tools by their process name, proxy accounts, and version numbers for the Operating System (OS) and Internet Explorer.

PCC_SYS: System Management
This handler class provides the attacker with the ability to manage system components including start/stop/delete system services, enumerate/alter registry keys, and manage running processes. This class also provides the ability for the attacker to take a screen shot of the users desktop.

INTERNAL_CMD: Command-Line Shell

This handler class uses the command ID 5 and implements an interactive command line shell accessible from the C2 server, containing a series of built-in commands. If the input is not in this list of built-in commands, the malware attempts to invoke cmd.exe in the background, launching a command or command line utility already present on the system. The standard output channel of that command is provided back to the C2. The supported built-in commands are:

- help | ? (shows a list of built-in commands)
- cd
- dir
- md
- rd
- del
- copy
- ren
- type
- runas
- pid
- cmd
- start
- reboot [-f]
- shutdown [-f]
- clearlog [system] [application] [security]
- wget
- httpurl

Kill Switch / Self-Destruction

The only command that is implemented directly in the main backdoor thread as a subprocedure call and not via a generic command handler class is command ID 256. This command results in the DLL deleting itself and terminating the backdoor process.

Kernel Driver Sample (MD5: dae6b9b3b8e39b08b10a51a6457444d8)

This sample is a packed 32-bit kernel driver extracted by the aforementioned DLL with an MD5 hash of: de7500fc1065a081180841f32f06a537, this sample will only function on a Windows 32-bit kernel. This code appears to have been compiled on Sunday October 9, 2011 at 4:50:31 P.M. UTC (very early morning time of Monday, October 10 in China).

Entrypoint

This section describes how the driver performs its initialization routine.

Multiple Instance Protection

The driver begins by opening a named event in the BaseNamedObjects object directory with the name {8CB2ff21-0166-4cf1-BD8F-E190BC7902DC} using the Windows API ZwOpenEvent(). If the event already exists, the driver fails to load, presumably to avoid a multiple instances of itself. If the event does not exist, the driver then creates it using the Windows API ZwCreateEvent(). The Windows API for creating events (ZwCreateEvent(), or CreateEvent() in user-mode) already provides the ability to “create-or-open” an event, so the use of an initial ZwOpenEvent is superfluous and indicative of relatively limited Windows API knowledge of the author of that part of the code. It is interesting to note that some of the hex digits in the object name are mixed case which is potentially indicative of the code being re-appropriated from another source.

Anti-Debugging Protection

The second component of the entry point performs an anti-debugging technique, calling the function KdDisableDebugger(), which allows the driver to disable usage of the built-in Windows kernel debugging facility that is used by popular kernel debuggers KD and WinDbg. Tools such as Syser Debugger, or debugging through a virtual machine are unaffected by this technique. The sample, rather than importing the KdDisableDebugger() API using conventional methods, looks up the API through MmGetSystemRoutineAddress() instead. All of the other APIs used by the driver are imported normally, so this is not a technique to hide import APIs used throughout the driver. Searching Google for “MmGetSystemRoutineAddress” and “KdDisableDebugger” results in dozens of Chinese language blogs which explain how to use this technique to “Disable WinDbg”.

Hooking

The final step of the entry point is to begin hooking the system, which is done by two helper functions – one is designed to hook the system call table, while the other hooks the network stack.

Network Stack Hooking

The network stack hooking first queries the OS version using RtlGetVersion() or PsGetVersion(). Checking the version is necessary because Windows versions beginning with Vista utilize a redesigned TCP/IP network stack, most hooking operations will require a different implementation for these versions. On versions prior to Windows Vista, the TCP/IP driver creates a \Device\Tcp device object through which most network requests are piped through. On Vista and later, TCP/IP has been split up into multiple components, and IP connection enumeration, which this driver is targeting, is managed by \Device\nsiproxy instead.

In either case, the driver obtains the device object by using IoGetDeviceObjectPointer() and hooks Major Function 14 the IRP_MJ_DEVICE_CONTROL, as this is the function through which all Input Output ConTroLls (IOCTLs) are sent, such as the IOCTL for querying active IP connections.

Network Store Interface (NSI) Hook

The NSI hook, targets IOCTL 0x12001B, which is used by NsiGetObjectAllParameters() in nsi.dll when users typically run commands such as netstat.exe or use any of the IP Helper APIs in iphlpapi.dll. The purpose of the hook is to scan the list of active connections returned to the user, and hide any such connection currently bound to a local TCP port in the range between 40000 and 45000. The hooking is performed by creating a new completion routine associated with any IRP_MJ_DEVICE_CONTROL IRP that matches the IOCTL, attaching to the target process, performing several memory copies to hide the entry, and detaching.

This functionality is nearly identical to the code posted by Edward Sun (aka cardmagic, sunmy1@sina.com, onlyonejazz@hotmail.com, cardcian@mail.ustc.edu.cn, QQ# 28025945) from Hefei, Anhui province (Nanjing Military District) on July 8, 2007, then a China-based researcher at Trend Micro (now working at Kingsoft Chinese AV company; LinkedIn profile page: http://www.linkedin.com/profile/view?id=84082731) at http://forum.eviloctal.com/viewthread.php?action=printable&tid=29604 (See Appendix G). CrowdStrike has no information connecting Mr. Sun to this intrusion activity, his code appears to have been appropriated by the actor to add similar functionality to their code.

TCP Hook

The TCP hook works almost identically to the NSI hook, though instead hooking IOCTL 0x120003 (IOCTL_TCP_QUERY_INFORMATION_EX). This IOCTL has the exact same functionality as the NSI specific IOCTL. This IOCTL was the mechanism used on Windows versions prior to Windows Vista. This hook also filters any connections listening on TCP ports in the range between 40000 and 45000.

System Call Hooking

The system call hooking targets three functions: ZwSaveKey(), ZwQueryValueKey(), and ZwEnumerateValueKey(). The unpacked kernel driver sample hooks these functions by reading the second DWORD at each of these exported functions. Because the system call stub uses the EAX register as an index for the system call ID, and a "mov eax, imm32" instruction" instruction is used, this second DWORD will match the system call ID. It then adds this index to the value of KeServiceDescriptorTable.Base, which is the exported kernel variable (on 32-bit Windows only) which directly points to the system call table. This is one of the simplest ways to do a system call hook, but will not work on 64-bit Windows as this variable is not exported in addition to the protection provided by Microsoft PatchGuard.

The system call hook is then performed by first allocating a Memory Descriptor List (MDL) using the Windows API IoAllocateMdl(), and associating the MDL to a non-paged buffer using MmBuildMdlForNonPagedPool(). Once the MDL is associated to the non-paged buffer, the sample locks the underlying pages using the Windows API MmProbeAndLockPages(). Instead of hooking the entry in the table directly, which is easily detectable, the driver uses the LDASM open-source disassembly engine to analyze the function that is being pointed to by the table, and applying a Detours-style hook directly in the code. It uses the standard "mov cr0, eax" technique, turning off the Write Protect (WP) bit as it does this. When the hook is installed, it writes a special DWORD value, 'KDTR', which allows it to prevent double-hooking or badly-hooking the system call, during unhooking, this value is also checked.

Registry Hooks

In the ZwSaveKey() hook, access to \\REGISTRY\\MACHINE\\SYSTEM is blocked. RegSaveKey() which is the user-mode implementation of the kernel ZwSaveKey() API, is typically used when performing an offline backup of a particular registry key.

hook is the ZwQueryValueKey() hook, which looks for “Parameters” key of a service within the registry at \\REGISTRY\\MACHINE\\SYSTEM\\ControlSet001\\Services\\. It then checks for the values of the “ServiceDll” and “Security” keys, in the latter case it applies an XOR on the data with the value 127. The user-mode component of this malware is a service called “msupdate”, this driver is attempting to hide the service. The user-mode service stores configuration data in the “Security” subkey of the RPC registry key, this component will obfuscate the user-mode configuration data. The driver does not make any efforts to hide its own key, nor does it specifically check for “RPC” before “Security”, which can lead to random data being obfuscated. The final hook, ZwEnumerateValueKey(), is similar in structure to the ZwQueryValueHook key, due to the fact that these APIs provide almost identical functionality when it comes to reading registry values.

In the registry hooking code of the driver, a call is made to ObReferenceObjectByHandle(). This allows the driver to receive the ‘CM_KEY_OBJECT’ which is then used with ObQueryNameString() to get the key/value path. However, no call to ObDereferenceObject() is ever made, which means that all registry objects being sent to these APIs are eventually leaked.

In the registry hook, it was noticed that “CurrentControlSet001” was used as the target, if the target machine was using a “last known good” configuration, or a roaming hardware profile, the registry hook would not function as intended. This is the reason the Microsoft implemented a symbolic link to \\CurrentControlSet which ensures that regardless of the machines configuration any request will access the correct registry key.

MITIGATION & REMEDIATION

MITIGATION / REMEDIATION

This threat actor leaves several key fingerprints which can be used to identify compromised systems. These digital fingerprints are unique to this adversary for this campaign.

Network Signatures

The following network signatures are designed for the popular Open Source IDS called Snort. These signature can be ported to other formats upon request.

Malware #1

```
alert tcp any any <> any any (msg: "BackDoor Beacon Attempt"; content:"|78 7c
71 4c 4a 49 49 49 4A 4C 46|"; classtype:backdoor; sid:123456; rev:27122011;)

alert tcp any any <> any any (msg: "BackDoor Beacon Attempt"; content:"Goo-
gle"; http_uri; classtype:backdoor; sid:123457; rev:27122011;)

alert ip 1.9.5.38 any <> any any (msg: "Malicious Host Detected"; class-
type:backdoor; sid:123460; rev:27122011;)
```

Malware #2

```
alert tcp any any <> any any (msg:"BackDoor Beacon Attempt"; content:"|03 01
74 80 82 21 b5 64 c2 74 22 e3 02 00 00 00 49 03 00 00 00 00 00 00 00 00 00 00
0000 00 00|"; classtype:backdoor; sid:123458; rev:27122011;)

alert ip 202.86.190.3 any <> any any (msg:"Malicious Host Detected"; class-
type:backdoor; sid:123459; rev:27122011;)
```

Malware #3

```
alert tcp any any <> any any (msg: "BackDoor C2"; content: "POST /forum/
login.cgi HTTP/1.1"; content:"User-Agent: Mozilla/4.0"; classtype:backdoor;
sid:123461; rev:27122011;)

alert tcp any any <> any any (msg: "BackDoor C2"; content: "GET /Photos/Query.
cgi?loginid="; classtype:backdoor; sid:123462; rev:27122011;)

alert tcp any any <> any any (msg: "BackDoor C2"; content: "POST /Catelog/
login1.cgi HTTP/1.1"; content:"User-Agent: Mozilla/4.0"; classtype:backdoor;
sid:123461; rev:27122011;)
```

File System Artifacts

The following file system artifacts are indicative of a compromised host:

Dropper/DLL
C:\Documents and Settings\All Users\Documents\infoadmn.dll (TS: 2007-03-07 00:00:00)
C:\Documents and Settings\All Users\Documents\infoctrs.dll (TS: 2007-03-07 00:00:00)
C:\Documents and Settings\All Users\Documents\infocardapi.dll (TS: 2007-03-07 00:00:00)
MD5: 47619fca20895abc83807321cbb80a3d

Post Explotiation Tool
MD5: 2dce7fc3f52a692d8a84a0c182519133

Backdoor
MD5: de7500fc1065a081180841f32f06a537

Kernel Driver:
MD5: dae6b9b3b8e39b08b10a51a6457444d8
"%sysdir%\Drivers\{6AB5E732-DFA9-4618-AF1C-F0D9DEF0E222}.sys"

Registry Artifacts

The following Windows Registry artifacts are indicative of a compromised host:

Dropper/DLL
HKLM\\SYSTEM\\CurrentControlSet\\Services\\msupdate
HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Svchost which will have the 'msupdate' key set to 'msupdate'

Backdoor
HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\Msres<3 character rand>.ttf

Other Artifacts

Dropper/DLL
Username: _DomainUser_ Password:"Dom4!nUserP4ss"

Backdoor
The backdoor may be detected by several different Anti-Virus products under a signature with the name: Derusbi

Kernel Driver
Object: {8CB2ff21-0166-4cf1-BD8F-E190BC7902DC}

ATTRIBUTION

ATTRIBUTION

Attribution in the cyber domain is always a tricky subject when relying solely on malicious samples. Compiler artifacts and language settings can of course be deliberately masked or spoofed. CrowdStrike uses a unique approach of comprehensive threat analysis in order to decipher attributable components. Based on the corroborating evidence discovered in the course of this analysis, it appears there are numerous indications that this is a Chinese-speaking actor.

ZhuDongFangYu.exe is a component of 360 360安全卫士, a Chinese security product available from http://www.360.cn/. This is particularly relevant in this case because the backdoor DLL sample with an MD5 of de7500fc1065a081180841f32f06a537 specifically avoids installing the kernel driver on a system running this tool. Speculatively this may be because this security product detects this rootkit, or the author was attempting to prevent accidental infection on systems running this Anti-Virus product.

The obfuscation of the KdDisableDebugger() function call is seen on several Chinese language forums, and can be seen being reused in several code samples on those forums. As previously mentioned there is no advantage associated with using this call obfuscation, and appears to be reused for no apparent reason other than the attackers have copied code directly from forum code.

While the various network hooking techniques used in the kernel driver may appear novel or well researched, upon close inspection it is actually a line-for-line copy of an existing post from the now-offline 'rootkit.com' by a Chinese language developer. This post is currently mirrored on dozens of Chinese hacking websites.

Similarly the system call hooking is less impressive after searching for "IoAllocateMdl" and "cr0" (bbs.pediy.com/showthread.php?t=77467) which identifies Chinese forum websites with almost identical code to perform system call hooking through MDLs. The ldasm inline hooking is also repeated in numerous postings to Chinese forums. One particular website (http://read.pudn.com/downloads197/sourcecode/windows/system/927802/CCRootkit/RootkitSys/HookSSDT.c__.htm) had an almost identical ldasm loop that tried to identify the exact same code sequences. Open source research of the 4 innocuous kernel APIs "ZwSaveKey ZwQueryValueKey ZwEnumerateValueKey IoAllocateMdl", in concert leads directly to a Chinese website that has a cached rootkit performing similar hooks on the same 3 registry related APIs.

While the driver does not use pool tags for most of its allocations, it does utilize them in the networking hooking code, much like the examples found on the Chinese language forums. This sample uses pool tags: 'tnet,' and 'KDTR'. Although the meaning of the KDTR tag is not

[1] http://bbs.pediy.com/showthread.php?t=125358
http://kost0911.pixnet.net/blog/post/36914183-anti-anti-windbg

obvious, we assess with high confidence that this is a shortened version of: “Kernel DeTouR”, which coincides with the matching functionality of the detour-style inline hook.

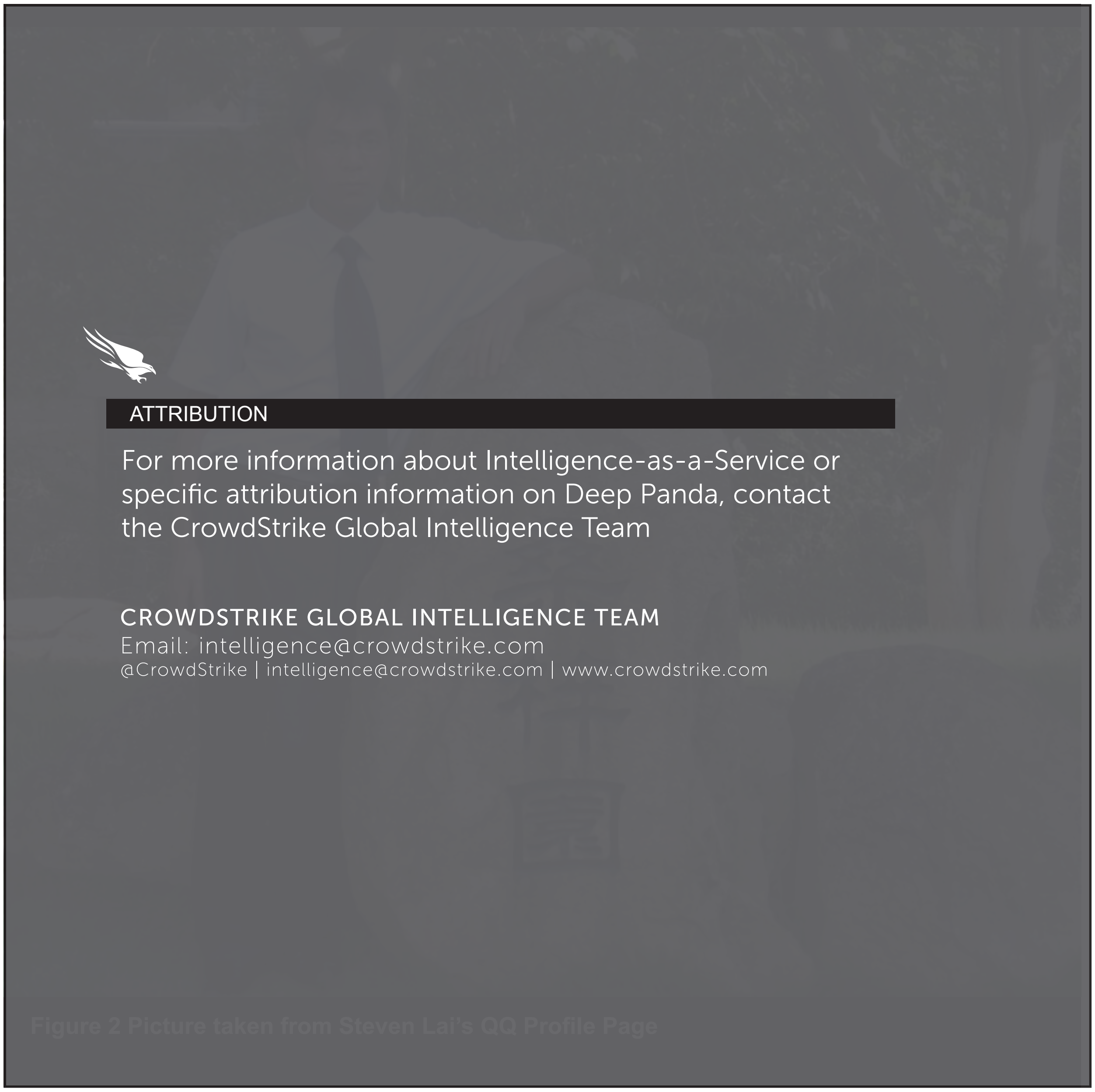
The driver code (MD5: dae6b9b3b8e39b08b10a51a6457444d8) appears to be a combination of various code that is easily searchable on the Internet, and almost always attributed to Chinese language forums and websites. The system call hooking parts of the code appear to be identical to the HookSSDT.c code authored by Steven Lai ‘embedlinux’ and utilized in what the author titled ‘CC Rootkit’ on on August 4, 2008 who’s email address is hqulyc@126.com. This user has a QQ identity of: 5054-3533, QQ is a popular instant messaging chat client used almost exclusively in China. His real name according to his QQ profile (http://user.qzone.qq.com/50543533) appears to be Steven Lai. He was is 28 years old (born September 5, 1983) and lives in Xiamen, Fujian province (Nanjing Military Region). According to his profile, he has worked at Xiamen XOCECO New Technic Co., Ltd. (http://www.likego.com/en/about.asp), a company that builds audio/video systems for transportation systems. Mr. Lai is not being identified as the actor, his code however was used by whomever built the kernel driver utilized by the backdoor and for this reason we are providing the background on this individual.

ATTRIBUTION

For more information about Intelligence-as-a-Service or specific attribution information on Deep Panda, contact the CrowdStrike Global Intelligence Team

CROWDSTRIKE GLOBAL INTELLIGENCE TEAM
Email: intelligence@crowdstrike.com
@CrowdStrike | intelligence@crowdstrike.com | www.crowdstrike.com

Figure 1 Picture taken from Steven Lai's QQ Profile Page



ATTRIBUTION

For more information about Intelligence-as-a-Service or specific attribution information on Deep Panda, contact the CrowdStrike Global Intelligence Team

CROWDSTRIKE GLOBAL INTELLIGENCE TEAM
Email: intelligence@crowdstrike.com
@CrowdStrike | intelligence@crowdstrike.com | www.crowdstrike.com

Figure 2 Picture taken from Steven Lai's QQ Profile Page

[redacted]

ATTRIBUTION

For more information about Intelligence-as-a-Service or specific attribution information on Deep Panda, contact the CrowdStrike Global Intelligence Team

CROWDSTRIKE GLOBAL INTELLIGENCE TEAM
Email: intelligence@crowdstrike.com
@CrowdStrike | intelligence@crowdstrike.com | www.crowdstrike.com

Parts of CC

(http://read. /HookSS-
DT.c__.htm

According t wenku.
baidu.com/

CONCLUSION

CONCLUSION

The samples involved in this incident are typical of attacks commonly associated with the People's Republic of China (PRC). These code samples have a variety of Tools, Techniques, and Procedures (TTPs) that are used to track and identify specific adversary groups. The sophistication of the actor responsible for this incident is difficult to quantify without visibility into the activities that transpired on the victims network. The ability to conduct Incident Response (IR) including forensics, and log analysis, greatly augments this visibility into these aspects of the incident. Some indications as to the adversaries' capabilities can be derived from the captured samples alone.

Dropper/Implant #1

The dropper code (MD5: 14c04f88dc97aef3e9b516ef208a2bf5) does not utilize any techniques that are unique or unusual, and is consistent with tools, techniques, and procedures of attacks targeting proprietary information and generally attributed to the PRC. The presence of dead code and its replacement by a more simple obfuscation method to hide the to-be-dropped dll binary file indicates code reuse on the attacker side. The 'dead code' utilizes a more sophisticated compression algorithm provided by a third party which was rendered useless for some reason. This may have been a result of the attacker modifying an existing tool, or unknowingly using a re-purposed tool. The dropper resources indicate the compiler used to build the tool was running on a system that utilized the Chinese "Simple" language pack and was built on Wednesday May 4th, 2011 at 11:04:24 A.M. UTC (early evening time in China). While this can be deliberately spoofed as a 'false flag' other indicators including the C2 are consistent with this having been the work of a Chinese speaking actor.

The dropped DLL (MD5: 47619fca20895abc83807321cbb80a3d) itself contains functionality that is typical of a Remote Access Tool (RAT) which are commonly used by PRC based actors in data exfiltration attacks. The code quality is not impressive, and contains a trivial stack buffer overflow vulnerability. Despite the buffer overflow, the C2 channel lacks any command authentication or encryption, apart from the initial beacon encryption/obfuscation using a statically compiled XOR key. The sample uses TCP port 443 for communication, but makes no attempt to mimic the SSL protocol typically used on that port number, which would provide enhanced Operational Security (OPSEC). This code appears to have been compiled on Wednesday May 4th, 2011 at 10:48:19 A.M. UTC (early evening time in China).

Post Exploitation Tool

The post exploitation tool (MD5: 2dce7fc3f52a692d8a84a0c182519133) is a dual-use tool, it can be dropped and executed by a client-side exploit, or the adversary can launch it using a variety of command line options. This tool is built in Microsoft .NET framework, which is typically an indication of a less sophisticated attacker, because .NET is easier to develop in but requires the .NET framework be present on the victim machine. The tool appears to have been compiled on Thursday May 26th, 2011 at 10:21:44 A.M. UTC (early evening time in China). The sample utilizes the AES cryptographic algorithm to protect its C2 communications.

Implant #2

Backdoor DLL

This DLL is a moderately sophisticated backdoor with several well designed communication mechanisms not typically seen in these types of implants. The code base for the sample was developed in C++. The code appears to have been compiled on Sunday October 30, 2011 at 12:43:33 P.M. UTC (late evening time in China). This sample has multiple communication capabilities available that makes it far more versatile and stealthy. It implements relatively well thought out protocols including HTTP and DNS. The tool has the ability to automatically down select the most effective communication channel once it has been instantiated, which can help avoid detection from solutions like DNS blacklisting and RFC protocol enforcement. The DLL itself contains traces of the original C++ class names that were utilized in the source code, which in general were prefixed with 'PCC'. The sample supports the ability to act as a generic proxy, this may be intended to proxy C2 traffic for other infected machines in order to minimize the number of systems communicating to the C2, thus enhancing OPSEC. The sample contains 'dead code' which appears to be command and control server classes, this is likely an indicator that the C2 client which would communicate with this sample shares the same communications library which was compiled into this sample.
System Driver
The kernel driver component dropped by the Backdoor DLL bears many tool marks associating it with the CCRootkit package publicly by Steven Lai (a/k/a embedlinux). This kernel mode rootkit implements several hooking techniques that are aimed at preventing a system administrator from detecting the backdoor DLL. The implementation of these techniques has some unique idiosyncrasies that permit direct attribution to the source code Steven Lai posted. This driver attempts to hide a wide swath of TCP ports (40000-45000) for an unknown reason, however it is suspected that this may relate to the potential network relaying capability alluded to for the backdoor dll.

System Driver

The kernel driver component dropped by the Backdoor DLL bears many tool marks associating it with the CCRootkit package publicly by Steven Lai (a/k/a embedlinux). This kernel mode rootkit implements several hooking techniques that are aimed at preventing a system administrator from detecting the backdoor DLL. The implementation of these techniques has some unique idiosyncrasies that permit direct attribution to the source code Steven Lai posted. This driver attempts to hide a wide swath of TCP ports (40000-45000) for an unknown reason, however it is suspected that this may relate to the potential network relaying capability alluded to for the backdoor dll.

APPENDIX

APPENDIX

Appendix A: Command Line Options for Post Exploitation Tool Sample (MD5: 2dce7fc3f52a692d8a84a0c182519133

The following are command line options identified in the sample

- **iu** - impersonate user, iu represents a username and expects the following additional arguments.
    - **id** -domain
    - **ip** -password
- **f** - perform command based on value. Possible values listed below
    - **sh** - Connect to C2.
        - **x** - hostname, connect to http address to download
            - **y** - port
            - **u** - username
            - **w** - password
        - **l** - set up listener
            - **s** - hostname
            - **p** - port
    - **v** - display communication protocol version
    - **dl** - download file
        - **url** - url to download from.
        - **file** - path to save file to.
    - **ul** - upload file
        - **url** - url to upload to.
        - **file** - file to upload.
    - **cl** - replace contents of files in directory p matching wild card pattern m with list of regexes. This command will search line by line a matching file and filter out contents matching the supplied regexes, it will then set the modify/create date to the original file so as to hide the tampering.
        - **p** - target path
        - **m** - file wild card pattern
        - <arguments>
    - **tu** - Copy last access, last modify and creation time from file r. If r does not exist, a default date of 11-30-2005:12:00:00 with the UTC offset of the system applied.
        - **p** - target path
        - **m** - file wild card
        - **r** - reference file.
    - **d** - dump System.IO.FileInfo for file t to console.
        - **t** - path to file
    - **wmi** - perform Windows Management Instrumentation (WMI) command
        - **s** - machine
        - **u** - username
        - **p** - password
        - a - kerberos
        - **m** - can be one of the following 3 items

            - query - run WMI query
                - q - query
            - call - call WMI
                - q - target to call on
                - c - method to invoke
                - <array of arguments to the call>
            - get - do nothing
        - ra - run as
            - ru - username
            - rd - domain
            - rp - password
            - wp - with profile
            - <array of arguments for startprocess>

Appendix B: Algorithm for computing machine ID

```
char ch = 'L';
foreach(char ch2 in Environment.MachineName)
{
    ch = (char)(ch ^ ch2);
}
byte num3 = (byte)ch;
return (GetVolumeSerial() ^ (uint)(((num3 + (num3 * 0x100)) + (num3 * 
0x10000)) +
                      (num3 * 0x1000000)));
```

Appendix C: Remote Commands Supported by .NET Backdoor Post Exploitation Tool Sample

```
public class RcDataCommand
{
   public byte channelHint;
   public RcDataCommandId cmdID;
   public RcDataCommandType cmdType;
   public string extraInfo;
   public string string_0;
}
```

Implemented values for cmdID are as follows:

- 2: CreateShell - Execute application and send output to C2.
- 3: ReadFile - Upload file to C2
- 8: Execute - Execute application and send output to C2

cmdType can be one of the following (Interesting commands explained in detail):

- 0: None
- 1: Command
- 2: Info
- 3: Error
- 4: HeartBeat
- 5: ChannelCreate
- 6: ChannelBind
- 7: ChannelTerminate
- 8: ChannelRequestCreate
- 9: ChannelRequestConfirmationAccept
- 10: ChannelRequestConfirmationDenied
- 11: Object - extraInfo parameter is an XML serialized object.
    - string_0 == 1: download a file or .NET module.
    - string_0 == 2: connect to IP address.
- 12: InitilizeStringTable
- 13: ClearMemoryFiles - clear list of downloaded modules.
- 14: LoadModule - load a previously downloaded module
    - string_0: name of module to load.
- 15: UnloadModule
- 16: ModuleListReport
- 17: CallModuleMethod - Call method in loaded module
    - string_0: contains module name.
    - extraInfo: contains arguments to the method.
- 18: ChannelSpecifiedCommand
- 19: SayGoodBye
- 20: ChannelInitializeError
- 21: RequestAssemblyDependent
- 22: ResponseAssemblyDependent
- 23: ConnectSuccessed
- 24: ConnectFailed
- 25: ChannelRequestTcpConnect
- 26: ChannelRequestUdpConnect
- 100: Customized

string_0 can have one of the following values dependant upon command id and type.

- “new block”
- “seek”
- “set end”
- “finishfile”
- “stop”
- “start copy”
- “ssoi”
- “abort”
- “Evt”
- Other values calculated at runtime.

Appendix D: Raw bytes of example Authentication packet.

03 01 74 80 82 21 b5 64 c2 74 22 e3 02 00 00 00 49 03 00 00 00 00 00 00 00 00 00 00 00
00 00 00

Appendix E: Initialization of KEY and IV for AES

```
        for (int i = 0; i < 0x20; ++i )
        {       for (int i = 0; i < 0x20; ++i )
        {
            buffer[i] = (byte)((i + 8) + ((byte)password[num++]));
            buffer[i] = (byte)(buffer[i] ^ 170);
            num = num % password.Length;
```

Appendix F: Command & Control Servers

C2 Server	Port	Geolocation	Whois	Samples Used In
1.9.5.38	443	Bukit Mertajam, Maylasia	inetnum: 1.9.0.0 - 1.9.255.255 netname: TMNET-AS-AP descr: Tmnet, Telekom Malaysia Bhd. descr: Telekom Malaysia Berhad descr: 44th Floor, Global Data Marketing, TM Global descr: Jalan Pantai Baharu country: MY admin-c: TA35-AP tech-c: TA35-AP mnt-by: AP-NIC-HM mnt-lower: TM-NET-AP mnt-routes: TM-NET-AP status: ALLOCATED PORTABLE remarks: -+-+-+-+-+-+-+-+-+-+-+-+-++-+-+-+-+-+-+-+-+-+-+-+-+-+-+ remarks: This object can only be updated by APNIC hostmasters. remarks: To update this object, please contact APNIC remarks: hostmasters and include your organisation's account remarks: name in the subject line. remarks: -+-+-+-+-+-+-+-+-+-+-+-+-++-+-+-+-+-+-+-+-+-+-+-+-+-+-+ changed: hm-changed@apnic.net 20100610 source: APNIC	47619fca20895abc83807321cbb80a3d

C2 Server	Port	Geolocation	Whois	Samples Used In
202.86.190.3	80	Hong Kong	inetnum: 202.86.190.0 - 202.86.191.255 netname: Tele-One-HK country: HK descr: TeleOne(HK) Limited admin-c: HL13 tech-c: AC612-AP status: ASSIGNED NON-PORTABLE changed: angus@edu.ctm.net 20041122 mnt-by: MAINT-CTM-MO source: APNIC	2dce7fc3f52a692d8a84a0c182519133 de7500fc1065a081180841f32f06a537

Appendix G: Edward Sun's kernel network hook code

```
        标题: [转载]NSI Module Hook : Hiding Port Under Windows Vista [打印本页]
      作者: eviloctal    时间: 2007-7-8 20:53      标题: [转载]NSI Module
Hook : Hiding Port Under Windows Vista

                    原始出处: http://rootkit.com/newsread_print.php?newsid=735
信息来源: 邪恶八进制信息安全团队(www.eviloctal.com)

cardmagic writes: Windows Vista has changed alot on network module, many old
port hiding materials are no longer usable.
In this post, I will share with you a simple code to hide port under Vista,hope
it is useful for some guys .
Actually under Windows Vista, netstat.exe will call InternalGetTcpTable2 which
is exported by Iphlpapi.dll to list all open ports,then InternalGetTcpTable2
will transfer control to NsiAllocateAndGetTable which is exported by nsi.dll,
and finally nsi.dll involve NsiEnumerateObjectsAllParametersEx to interact with
kernel mode module of NSI -- nsiproxy.sys. nsiproxy.sys is almost like a
wrapper of netio.sys, it will then call internal subroutines of netio.sys .
Here ,we will use a relatively easy way -- "NSI Kernel Module Dispatch Routine
Hook" to demostrate the specified port hiding uner Vista. Dispatch routine hook
is an old topic, this time ,we will apply this method to nsiproxy.sys. Please
focus on how to handle the content filtering of NSI:)
Check the following code(Notice: I only tested it under Windows Vista RTM
32bit): 复制内容到剪贴板
```

代码:

```
////////////////////////////////////////////////////////////////////////////////
////////
// Filename: PortHidDemo_Vista.c
//
// Author: CardMagic(Edward)
// Email: [email]sunmy1@sina.com[/email]
// MSN: onlyonejazz at hotmail.com
//
// Description:
//       A Demostration Of Hiding
//       Specified Port Under Windows Vista RTM 32bit.
//    Tested Under Windows Vista Kernel Version 6000 MP (1 procs) Free x86 com-
patible
//
//

#include "stdlib.h"
#include "ntifs.h"

unsigned short htons(unsigned short hostshort);
unsigned long inet_addr(const char *name);
typedef unsigned long DWORD;

#define LOCALHIDEIP "10.28.157.71"
#define LOCALHIDEPORT 139

#define IOCTL_NSI_GETALLPARAM 0x12001B

extern POBJECT_TYPE *IoDeviceObjectType,*IoDriverObjectType;
PDRIVER_OBJECT pNsiDrvObj = 0;
PDRIVER_DISPATCH orgNsiDeviceIoControl = 0;

DWORD gLocalPort=0,gLocalIp=0;

typedef struct _HP_CONTEXT
{
  PIO_COMPLETION_ROUTINE oldIocomplete;
  PVOID oldCtx;
  BOOLEAN bShouldInvolve;
  PKPROCESS pcb;
}HP_CONTEXT,*PHP_CONTEXT;
```

```
}INTERNAL_TCP_TABLE_SUBENTRY,*PINTERNAL_TCP_TABLE_SUBENTRY;

typedef struct _INTERNAL_TCP_TABLE_ENTRY
{
  INTERNAL_TCP_TABLE_SUBENTRY localEntry;
  INTERNAL_TCP_TABLE_SUBENTRY remoteEntry;

}INTERNAL_TCP_TABLE_ENTRY,*PINTERNAL_TCP_TABLE_ENTRY;

typedef struct _NSI_STATUS_ENTRY
{
  char bytesfill[12];

}NSI_STATUS_ENTRY,*PNSI_STATUS_ENTRY;

typedef struct _NSI_PARAM
{
  //
  // Total 3CH size
  //
  DWORD UnknownParam1;
  DWORD UnknownParam2;
  DWORD UnknownParam3;
  DWORD UnknownParam4;
  DWORD UnknownParam5;
  DWORD UnknownParam6;
  PVOID lpMem;
  DWORD UnknownParam8;
  DWORD UnknownParam9;
  DWORD UnknownParam10;
  PNSI_STATUS_ENTRY lpStatus;
  DWORD UnknownParam12;
  DWORD UnknownParam13;
  DWORD UnknownParam14;
  DWORD TcpConnCount;
}NSI_PARAM,*PNSI_PARAM;

unsigned short htons(unsigned short a)
{
  unsigned short b = a;
  b = ( b << 8 );
  a = ( a >> 8 );
  return ( a | b );
};
```

```
unsigned long inet_addrt(const char* name)
{
  int i,j,p;
  int len = strlen(name);
  unsigned long temp_val[4];
  char namesec[10] ;

  for(i = 0,j =0,p =0;i < len;i++)
  {
    memset(namesec,0,10);
    if(&#39;.&#39; == name[i])
    {
      if(p)
        strncpy(namesec,name+p+1,i-p);
      else
        strncpy(namesec,name,i);
      temp_val[j] = atoi(namesec);
      j++;
      p = i;
    }
  }

  strncpy(namesec,name+p+1,i-p);
  temp_val[j] = atoi(namesec);

  return (temp_val[0]|(temp_val[1]<<8)|(temp_val[2]<<16)|(temp_val[3]<<24));
}

NTSTATUS
HPCompletion(
      IN PDEVICE_OBJECT DeviceObject,
      IN PIRP Irp,
      IN PVOID Context
      )
{
  PIO_STACK_LOCATION irpsp = IoGetCurrentIrpStackLocation(Irp);
  PIO_STACK_LOCATION irpspNext = IoGetNextIrpStackLocation(Irp);
  PHP_CONTEXT pCtx = Context;
  PNSI_PARAM nsiParam;
  int i;

  if(NT_SUCCESS(Irp->IoStatus.Status))
  {
```

```
nsiParam = Irp->UserBuffer;
    if(MmIsAddressValid(nsiParam->lpMem))
    {
      //
      // netstat will involve internal calls which will use
      // nsiParam structure
      //
      if(  (nsiParam->UnknownParam8 == 0x38))
      {
        KAPC_STATE apcstate;
        PNSI_STATUS_ENTRY pStatusEntry = (PNSI_STATUS_ENTRY)nsiParam->lpStatus;
        PINTERNAL_TCP_TABLE_ENTRY pTcpEntry = (PINTERNAL_TCP_TABLE_ENTRY)nsi-
Param->lpMem;
        int nItemCnt = nsiParam->TcpConnCount;

        KeStackAttachProcess(pCtx->pcb,&apcstate);
        //
        //make sure we are in the context of original process
        //
        for(i = 0;i < nItemCnt;i ++)
        {
          if((pTcpEntry[i].localEntry.dwIP == gLocalIp)&&(pTcpEntry[i].localEn-
try.Port == gLocalPort))
          {
            //
            //NSI will map status array entry to tcp table array entry
            //we must modify both synchronously
            //
            RtlCopyMemory(&pTcpEntry[i],&pTcpEntry[i+1],sizeof(INTERNAL_TCP_TA-
BLE_ENTRY)*(nItemCnt-i));
            RtlCopyMemory(&pStatusEntry[i],&pStatusEntry[i+1],sizeof(NSI_STA-
TUS_ENTRY)*(nItemCnt-i));
            nItemCnt--;
            nsiParam->TcpConnCount --;
            i--;
          }
        }

        KeUnstackDetachProcess(&apcstate);
      }
    }
  }
```

```
irpspNext->Context = pCtx->oldCtx;
  irpspNext->CompletionRoutine = pCtx->oldIocomplete;

  //
  //free the fake context
  //
  ExFreePool(Context);

  if(pCtx->bShouldInvolve)
    return irpspNext->CompletionRoutine(DeviceObject,Irp,Context);
  else
  {
    if (Irp->PendingReturned) {
      IoMarkIrpPending(Irp);
    }
    return STATUS_SUCCESS;
  }
}

NTSTATUS
ObReferenceObjectByName (
            IN PUNICODE_STRING ObjectName,
            IN ULONG Attributes,
            IN PACCESS_STATE AccessState OPTIONAL,
            IN ACCESS_MASK DesiredAccess OPTIONAL,
            IN POBJECT_TYPE ObjectType,
            IN KPROCESSOR_MODE AccessMode,
            IN OUT PVOID ParseContext OPTIONAL,
            OUT PVOID *Object
            );

NTSTATUS HPUnload(IN PDRIVER_OBJECT DriverObject)
{
  LARGE_INTEGER waittime;

  waittime.QuadPart = -50*1000*1000;
  InterlockedExchange(&(pNsiDrvObj->MajorFunction[IRP_MJ_DEVICE_CONTROL]), or-
gNsiDeviceIoControl);

//
  //delay loading driver to make it more secure
  //
```

```
KeDelayExecutionThread(KernelMode,0,&waittime);

  return STATUS_SUCCESS;
}

NTSTATUS HPDummyDeviceIoControl(
                IN PDEVICE_OBJECT DeviceObject,
                IN PIRP Irp
                )
{
  ULONG     ioControlCode;
  PIO_STACK_LOCATION irpStack;
  ULONG     status;

  irpStack = IoGetCurrentIrpStackLocation(Irp);

  ioControlCode = irpStack->Parameters.DeviceIoControl.IoControlCode;

  if(IOCTL_NSI_GETALLPARAM == ioControlCode)
  {
    if(irpStack->Parameters.DeviceIoControl.InputBufferLength == sizeof(NSI_
PARAM))
    {
      //
      //only care the related I/O
      //
      PHP_CONTEXT ctx = (HP_CONTEXT*)ExAllocatePool(NonPagedPool,sizeof(HP_CON-
TEXT));
      ctx->oldIocomplete = irpStack->CompletionRoutine;
      ctx->oldCtx = irpStack->Context;
      irpStack->CompletionRoutine = HPCompletion;
      irpStack->Context = ctx;
      ctx->pcb = IoGetCurrentProcess();

      if((irpStack->Control&SL_INVOKE_ON_SUCCESS) ==SL_INVOKE_ON_SUCCESS)
        ctx->bShouldInvolve = TRUE;
      else
        ctx->bShouldInvolve = FALSE;
      irpStack->Control |= SL_INVOKE_ON_SUCCESS;
    }
  }
```

```
//
  //call original I/O control routine
  //
  status = orgNsiDeviceIoControl(DeviceObject,Irp);

  return status;
}

NTSTATUS DriverEntry(
         IN PDRIVER_OBJECT DriverObject,
         IN PUNICODE_STRING RegistryPath
          )
{
  int i;
  NTSTATUS status;
  UNICODE_STRING uniNsiDrvName;

#if DBG
    _asm int 3 //debug
#endif

  DriverObject->DriverUnload = HPUnload;

  RtlInitUnicodeString(&uniNsiDrvName,L"\\Driver\\nsiproxy");

  status = ObReferenceObjectByName(&uniNsiDrvName,OBJ_CASE_INSENSI-
TIVE,NULL,0,*IoDriverObjectType,KernelMode,NULL,&pNsiDrvObj);

  if(!NT_SUCCESS(status))
  {
    return STATUS_SUCCESS;
  }

  //
  //store the original dispatch function of NSI driver
  //
  orgNsiDeviceIoControl = pNsiDrvObj->MajorFunction[IRP_MJ_DEVICE_CONTROL];

  gLocalIp = inet_addrt(LOCALHIDEIP);
  gLocalPort = htons(LOCALHIDEPORT);
  //
  //hook NSI dispatch routine
  //
InterlockedExchange(&(pNsiDrvObj->MajorFunction[IRP_MJ_DEVICE_CONTROL]), HPDum-
myDeviceIoControl);

  return STATUS_SUCCESS;
}
```

Appendix H: Command and Control MD5 Correlation

MD5	Command and Control Server
47619fca20895abc83807321cbb80a3d	1.9.5.38:443
2dce7fc3f52a692d8a84a0c182519133	202.86.190.3:80
de7500fc1065a081180841f32f06a537	202.86.190.3:80