

CNIT 581: CYBER FORENSICS OF MALWARE – FINALS

Ibrahim Waziri Jr

PhD in Information Security (CERIAS)

Final Exams

Instructor: **Associate Prof Sam Liles**

Purdue University

Fall 2014

Abstract

This report is CNIT 581: Cyber Forensics of Malware finals. The report contains the analysis of two malware files **pFPpHVqOjicpiWc.000** and **sample.zip**. Both malwares undergoes all the analysis taught during the period of the class. These analyses are the Basic Static Analysis, Basic Dynamic Analysis, Advanced Static Analysis, and Advanced Dynamic Analysis. We decided to call the first malware analyses Part I and the second malware Part II.

The first malware is identified as a Win32.exe file designed for Windows OS, while the second malware is an ELF file (Executable and Linkable Format), which is a standard binary format for UNIX systems. The first malware was successfully analyzed with no problems, but considering the last malware is designed for UNIX systems we ran into multiple problems.

We conclude the analyses with an explanation of the malware functionality and behaviors, which provides a summary of how we classify applications as malwares.

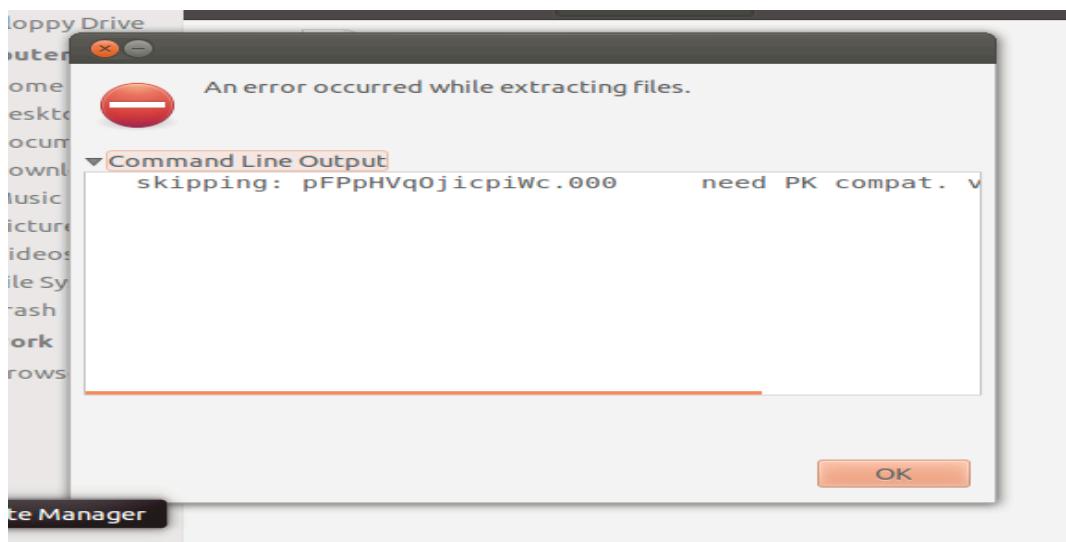
Part I

Malware 1 Analysis

File: **pFPpHVqOjicpiWc.zip**

Extracting the file

The malware file couldn't be extracted on Windows so we have to use a Linux environment. We decided to use Ubuntu 12.04 (The choice of Ubuntu not Kali was based on the available Linux distribution I have). Extracting the file in Ubuntu seems to be a struggle too. It requires a password. However when using the password provided which is "malware" it shows an error "**need PK compat. V5.1**" as shown in Figure 1 below:



It appears an update is required for the file to be extracted. Installing **p7zip-full** package using Ubuntu Terminal seems to solve the problem. As shown in Figure 2 below:

```
ubuntu@ubuntu: ~/Desktop/MalwareFiles
ubuntu@ubuntu:~$ sudo apt-get install p7zip-full
[sudo] password for ubuntu:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  thunderbird-globalmenu
Use 'apt-get autoremove' to remove them.
Suggested packages:
  p7zip-rar
The following NEW packages will be installed:
  p7zip-full
0 upgraded, 1 newly installed, 0 to remove and 383 not upgraded.
Need to get 1,561 kB of archives.
After this operation, 3,995 kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu/ precise/universe p7zip-full amd64 9
0.1~dfsg.1-4 [1,561 kB]
Fetched 1,561 kB in 0s (2,738 kB/s)
Selecting previously unselected package p7zip-full.
(Reading database ... 142857 files and directories currently installed.)
Unpacking p7zip-full (from .../p7zip-full_9.20.1~dfsg.1-4_amd64.deb) ...
Processing triggers for man-db ...
Setting up p7zip-full (9.20.1~dfsg.1-4) ...
```

Figure 2 above shows **p7zip-full** package installed using the Ubuntu terminal by using the command **sudo apt-get install p7zip-full**.

The file was successful extracted as **pFPpHVqOjicpiWc.000** and then copied to windows for analysis.

Basic Static Analysis

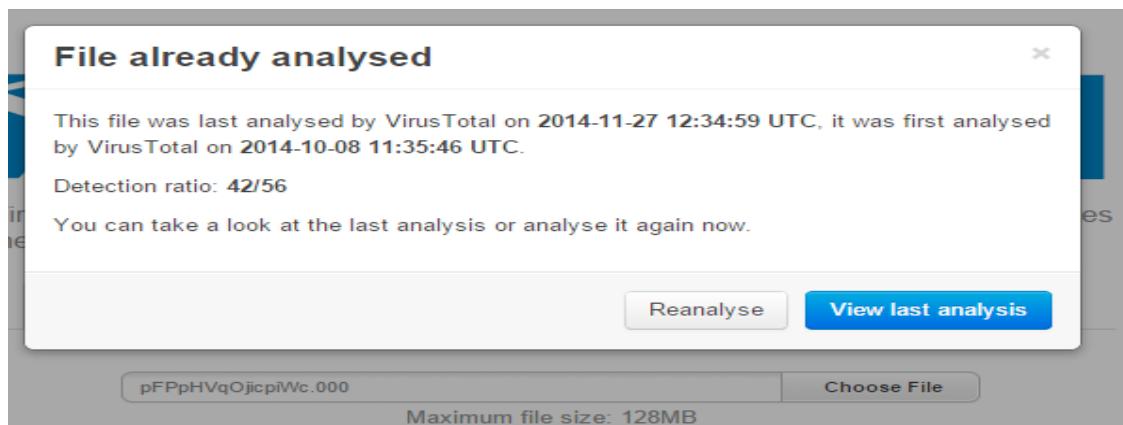
Basic Static Analysis is the analysis of a malware to examine the executable file without viewing the actual instruction. Basic static analysis can confirm whether a file is malicious, provide information about its functionality, and sometimes provide information that will allow you to produce simple network signature. Basic static analysis can be done without running the malware.

Analysis:

We used windows 8.1 with the entire analysis package described installed already.

First thing we did before running the malware was uploading it virustotal.com to see if the file matches existing virus signatures.

As shown in Figure 3 below, we see that the file was previously analyzed



We then reanalyze it to see the result. Reanalyzing the malware shows that the malware matches 43 of 55 virus signatures as shown in figure 4 below:

The screenshot shows the VirusTotal analysis interface. At the top, there are links for Community, Statistics, Documentation, FAQ, About, English, Join our community, and Sign in. Below the header, the VirusTotal logo is displayed. The analysis details are as follows:

- SHA256: 1a3517b6df790fba41c5d3d0a3e69db66b23cf8cb764ce8913be6a3f7eb4fe95
- File name: pFPpHVqOjicpiWc.000
- Detection ratio: 43 / 55
- Analysis date: 2014-12-04 21:51:32 UTC (2 minutes ago)

A circular progress bar indicates the detection ratio, with a red devil icon labeled '1' and a green smiley face icon labeled '0'. Below the analysis details, there are tabs for Analysis (selected), File detail, Additional information, Comments, Votes, and Behavioural information. A table lists the results from various antivirus engines:

Antivirus	Result	Update
ALYac	Trojan.GenericKD.1909383	20141204
AVG	PSW.Generic12.AWOA	20141204
AVware	Trojan.Win32.GenericIBT	20141204

Navigating to the “File Details” tab shows the compilation date as “**2014-10-08 09:14:15**” we could also see the imports of the file, which helps us understand what the malware was intended to do. These imports are KERNEL32.dll, MFC42.dll, MSVCRT.dll, and USER32.dll.

The screenshot shows the VirusTotal File detail page. At the top, there are links for Community, Statistics, Documentation, FAQ, About, English, Join our community, and Sign in. Below the header, there are sections for PE header basic information, PE sections, and PE imports.

PE header basic information:

- Target machine: Intel 386 or later processors and compatible processors
- Compilation timestamp: 2014-10-08 09:14:15
- Entry Point: 0x0001DE4E
- Number of sections: 3

PE sections:

Name	Virtual address	Virtual size	Raw size	Entropy	MD5
.text	4096	376628	376832	6.65	6dc276c4f6601eaa805798d91217f712
.data	380928	41568	45056	5.53	6d64658fefea947971b33bf0eb77871a
.rsrc	425984	3512	4096	3.41	9e9eaa966c0213459083e4151859e4c9

PE imports:

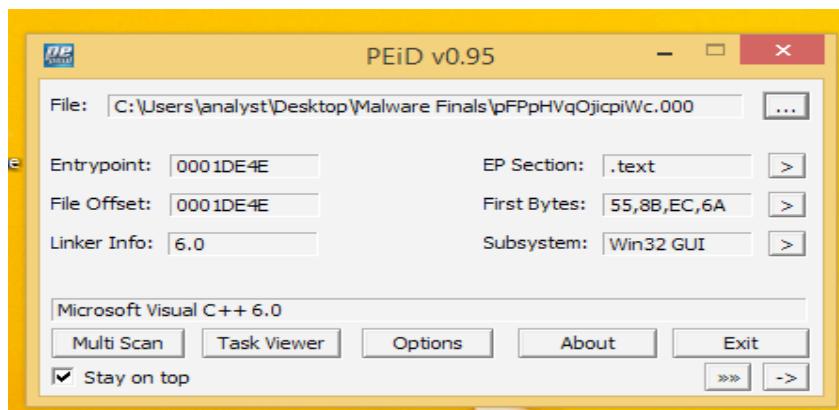
- [+] KERNEL32.dll
- [+] MFC42.DLL
- [+] MSVCRT.dll
- [+] USER32.dll

The “Additional Information” tab shows us the file type, size and what type of operating system it is designed for. Which are: The file is a Win32.exe file, it size is 420.0KB and it is a PE32 executable file designed for windows OS using Intel processor.

File identification	
MD5	32206ea7534debc3af5f0a0a9bd01f14
SHA1	64a3780a143b8c57c2a1ade4fb6d09f4f72a4b2b
SHA256	1a3517b6df790fba41c5d3d0a3e69db66b23cf8cb764ce8913be6a3f7eb4fe95
ssdeep	6144:oy0sr70Ekt+GGnQO2zC8bTfmJ80wljwKelDtNfz3Vm/NZfSCA.oNs0tAQPCbhwmwKepk/6
authentihash	5ca4aafb53a84e9370a715ff2968e5963879c75172906d1eafc28c4e1915c7d5
imphash	3da42dbe61b3b81acd989c34baaa9a87
File size	420.0 KB (430080 bytes)
File type	Win32 EXE
Magic literal	PE32 executable for MS Windows (GUI) Intel 80386 32-bit
TrID	Win32 Dynamic Link Library (generic) (43.5%) Win32 Executable (generic) (29.8%) Generic Win/DOS Executable (13.2%) DOS Executable Generic (13.2%) Autodesk FLIC Image File (extensions: flic, fli, cel) (0.0%)
Tags	peexe

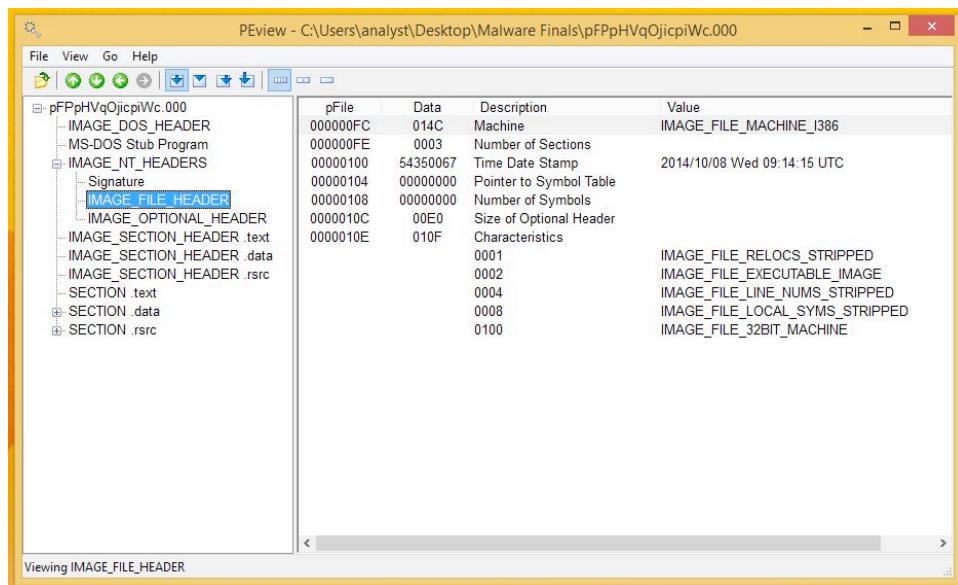
PEiD:

We next try to do another basic analysis on the malware using the PEiD tool to see if the file is packed or unpacked and to see the compiler used to compile the malware. And figure 7 below shows us that the file is unpacked and compiled using Microsoft Visual C++ 6.0.



PEview:

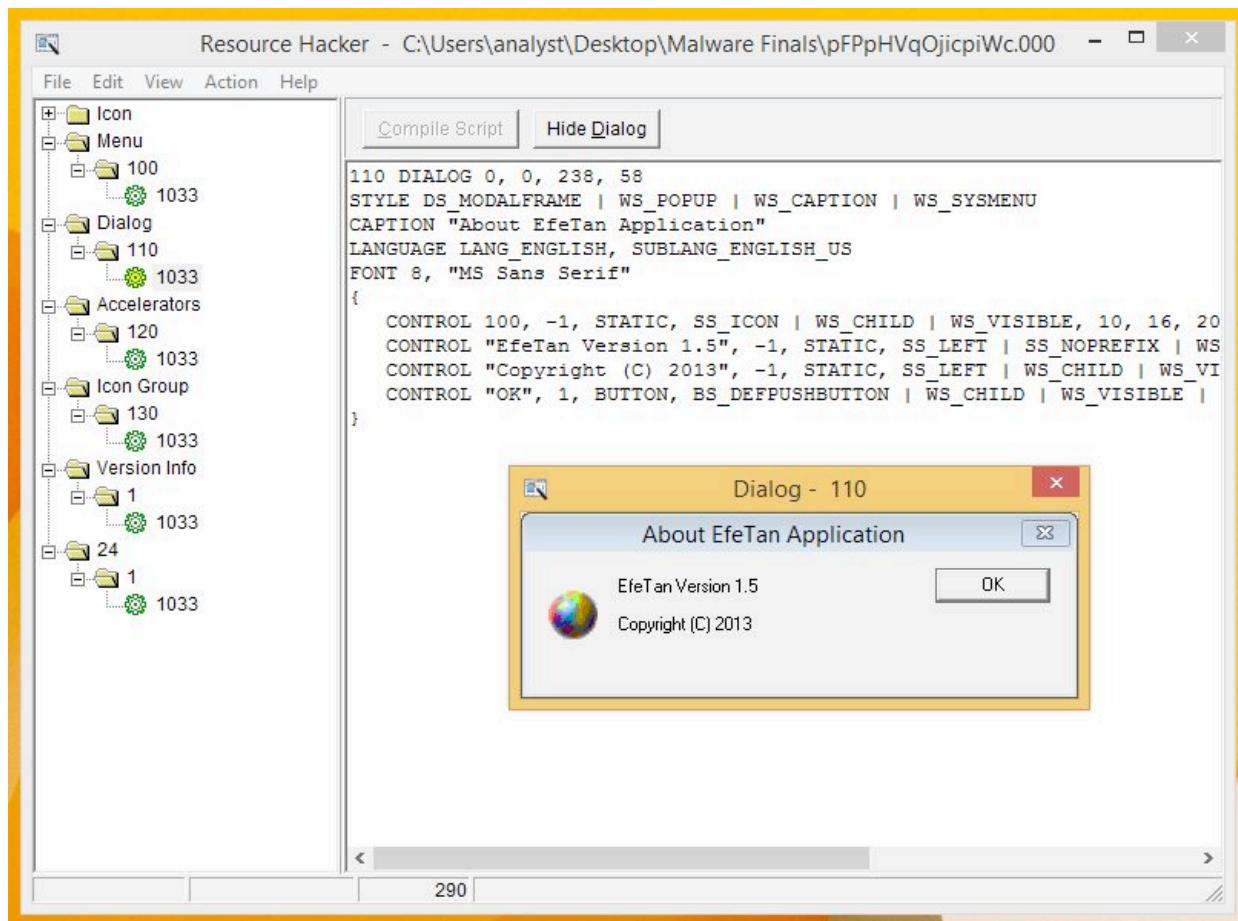
Using PEview, we open the file and navigate to the **IMAGE_FILE_HEADER**. Under this section we see elements like Machine, Number of Sections, Time Date Stamp which is **2014/10/08 Wed 09:14:15 UTC** as shown in Figure 8 below. We can use PEview to further explore and analyze the file and have a look at its imports and exports. However we are going to explore that using another Tool.



From PEview we can see **IMAGE_SECTION_HEADER.text**, **IMAGE_SECTION_HEADER.data** and **IMAGE_SECTION_HEADER.rsrc**. The **.text** sections contains instructions that the cpu executes. All other sections store data and supporting information. The **.data** section contains the program global data, which can be accessible from anywhere in the program. The **.rsrc** section includes the resources used by the executables that are not considered part of the executable such as icons, images, menus and strings.

Resource Hacker:

We will use the tool resource hacker to further analyze the malware. As shown in figure 9 below, resource hacker has several panels such as: **Icon**, **Menu**, **Dialog**, **Accelerators**, **Icon Group**, **Version Info** and **24**. For this file, the panel we are interested in viewing is the **Dialog** panel because it contains the programs dialog menus and what it is designed to do. However for this malware we really couldn't tell what it is designed to do. It appears that further analysis will be required to full investigate and understand what the malware is intended for. It only shows a dialog box, containing the creator information, version and copyright as shown below:

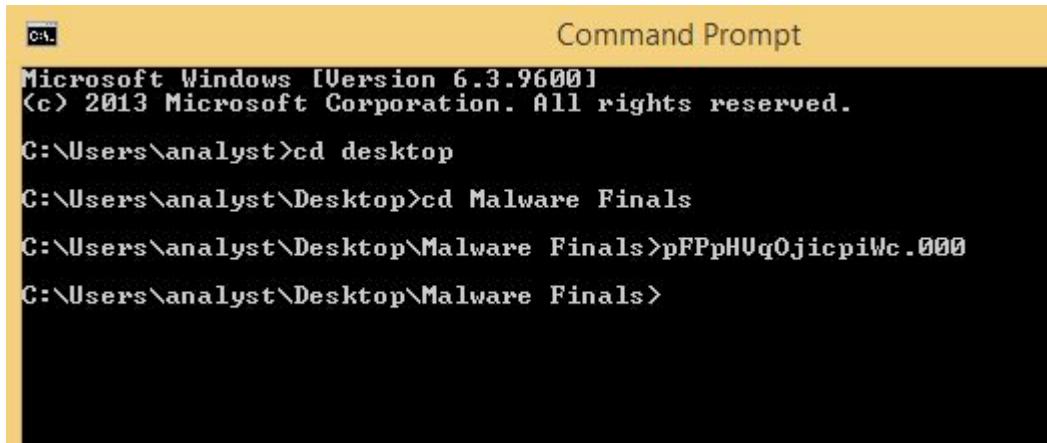


Basic Dynamic Analysis

Basic dynamic analysis is running the malware and observing its behavior on the system in order to remove the infection, produces effective signatures or both.

Analysis:

The first step is running the malware; using the command prompt we run the malware as shown in Figure 10 below:



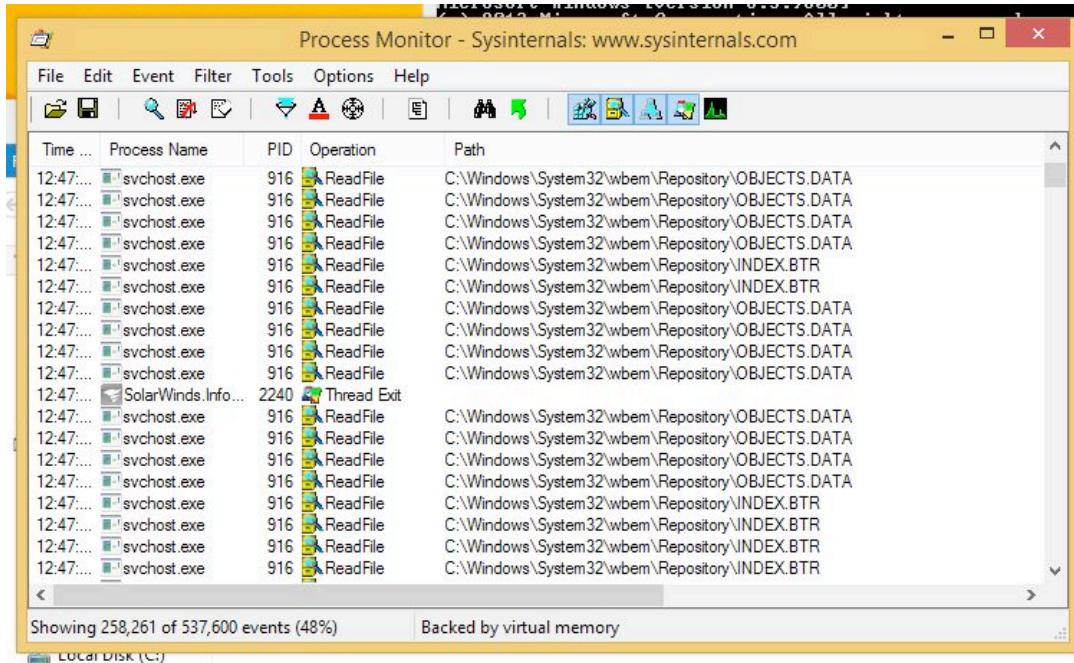
```
Command Prompt
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\analyst>cd desktop
C:\Users\analyst\Desktop>cd Malware Finals
C:\Users\analyst\Desktop\Malware Finals>pFPpHUq0jicpiWc.000
C:\Users\analyst\Desktop\Malware Finals>
```

Issue: Running the malware deletes the original final, however it seems like the process is running, we will use process monitor to explore that.

Process Monitor (Procmon):

We run process monitor to see if the malware is running and what it is doing. We decided to filter the processes using the Procmon filter as shown in Figure 11 below, but we couldn't find anything related to showing the malware running. Therefore we can assume that the malware self terminates and deletes itself ones it runs. It could be possible that it happens so that it doesn't leave any trails for what it does. However further analysis will reveal what the malware is intended and will provide more information as to why it deletes itself automatically.



In order to continue with the analysis we decided to use Process Explorer to see where the program is, or what is out of order.

Process Explorer:

Running the malware again and viewing the process explorer, we see the process **explorer.exe** which located in C:\Windows\System32. Another process dllhost.exe which is located under svchost.exe appears too as shown in figure 12 below. These processes happened to be part of the malware because they only appear when we start running the malware.

	Time	Virtual	Resident	Handle
svchost.exe		1,404 K	4,744 K	4260
dllhost.exe		3,176 K	6,236 K	4384
msdtc.exe		2,180 K	4,168 K	5032
svchost.exe		2,240 K	7,716 K	5064
SearchIndexer.exe	0.01	29,564 K	26,356 K	5612
lsass.exe	0.01	6,608 K	11,144 K	608
csrss.exe	0.05	1,996 K	7,292 K	516
winlogon.exe		1,296 K	4,700 K	572
dwm.exe	0.34	73,080 K	64,888 K	812
explorer.exe	1.54	104,000 K	145,816 K	5004

To further analyze this malware we do an Advance Static Analysis

Advanced Static Analysis

Advanced static analysis consists of reverse engineering the malware's internals by loading executable into a disassembler and looking at the program instruction in order to discover what the program does.

Analysis:

To start the advance static analysis, we upload the file into a special disassembler tool known as IDAPro.

IDAPro:

We successfully load the malware into IDAPro for analysis, and the first thing we will like to do is view the import and export because that will tell us about what the malware is designed for. As shown in fig 13 below. The imports appear to have a lot of gibberish imports. However few imports will be really helpful in telling us about the malware. These imports are: *GetModuleHandleA*, *GetStartupInfoA*, *exit*, and *SendMessageA*. We will go over all these imports soon enough to make understanding of each.

Imports			
Address	Ordinal	Name	Library
0045D0...	1842	?classCFrameWnd@CFrameWnd@@2UCRunti...	MFC42
0045D0...	1945	?classCView@CView@@2UCRuntimeClass@@B	MFC42
0045D3...	4234	?messageMap@CDialog@@1UAFX_MSGMAP...	MFC42
0045D3...	4238	?messageMap@CDocument@@1UAFX_MSG...	MFC42
0045D0...	4242	?messageMap@CFrameWnd@@1UAFX_MSG...	MFC42
0045D0...	4273	?messageMap@CView@@1UAFX_MSGMAP...	MFC42
0045D2...	4274	?messageMap@CWinApp@@1UAFX_MSGMA...	MFC42
0045D3...		EnableWindow	USER32
0045D0...		GetModuleHandleA	KERNEL32
0045D0...		GetStartupInfoA	KERNEL32
0045D3...		LoadIconA	USER32
0045D2...	6625	MFC42_6625	MFC42
0045D3...		SendMessageA	USER32
0045D3...		UpdateWindow	USER32
0045D3...		_XcptFilter	MSVCRT
0045D3...		__CxxFrameHandler	MSVCRT
0045D3...		__dolonexit	MSVCRT
0045D3...		__getmainargs	MSVCRT
0045D3...		__P__commode	MSVCRT
0045D3...		__P_fmode	MSVCRT
0045D3...		__set_app_type	MSVCRT
0045D3...		__setusermatherr	MSVCRT
0045D3...		_acmdln	MSVCRT
0045D3...		_adjust_fdiv	MSVCRT
0045D3...		_controlfp	MSVCRT
0045D3...		_except_handler3	MSVCRT
0045D3...		_exit	MSVCRT
0045D3...		_initerm	MSVCRT
0045D3...		_oneexit	MSVCRT
0045D3...		_setmbcp	MSVCRT
0045D3...		exit	MSVCRT

Line 207 of 229

Understanding the imports:

- GetModuleHandleA - From the string figure above we can see that this import is a KERNEL32 library. This import retrieves a module handle for a specified module. The module is usually loaded by a calling process. This tells us that somewhere there is a call to this function which we shall analyze soon. Next to the import, we can see the import address located at **0045D000**. We then go to the **IDA View-A** tab on IDAPro and click **G** on the keyboard, this will enable us to search for the address we are looking for. We can see that GetModuleHandleA is a call by the imports from **KERNEL32.dll** library as shown in figure 14 and 15 below:

```

.idata:0045D000 ; Imports from KERNEL32.dll
.idata:0045D000 ;
| .idata:0045D000 ; -----
| .idata:0045D000 ; Segment type: Externs
| .idata:0045D000 ; _idata
* .idata:0045D000 ; HMODULE __stdcall GetModuleHandleA(LPCSTR l|
| .idata:0045D000 ;                extrn GetModuleHandleA:dword
* .idata:0045D004 ; void __stdcall GetStartupInfoA(LPSTARTUPINFO|
| .idata:0045D004 ;                extrn GetStartupInfoA:dword ;
| .idata:0045D004 ;                                ; sub.
* .idata:0045D008

loc_41DF72:          ; nShowCmd
push    eax           ; lpCmdLine
push    esi           ; hPrevInstance
push    ebx           ; lpModuleName
call    GetModuleHandleA
push    eax           ; hInstance
call    _WinMain@16   ; WinMain(,,,)
mov     [ebp+var_68], eax ; int
push    eax           ; int
call    exit
start endp

```

We see the malware calling the `GetModuleHandleA` which definitely returns a module handle for the current process. We then see another import `SendMessageA` with address 0045D394. This malware is extracting resources and sending a message, but sending a message to where? We will know that we start viewing at the exports.

- `GetStartupInfoA`: This is also an import that is part of the KERNEL32.dll library. This import retrieves a structure containing details about how the current process was configured to run, such as where the standard handles are directed. So we can understand that this malware also collects the information about the processes that is running on the targeted computer and then sends it using the `SendMessageA` import. As shown in Figure 15 below, we can see a call to the import, which tells us it collect the Startup process information.



```

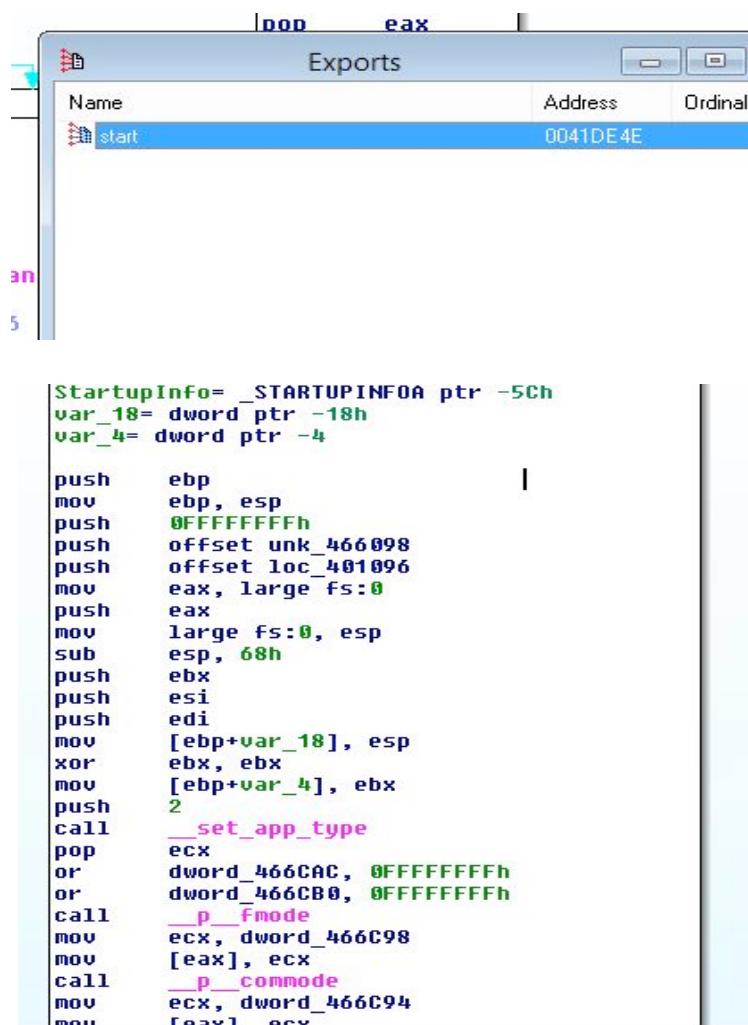
NUL
loc_41DF4B:
mov    [ebp+StartupInfo.dwFlags], ebx
lea    eax, [ebp+StartupInfo]
push   eax           ; lpStartupInfo
call   GetStartupInfoA
test  byte ptr [ebp+StartupInfo.dwFlags], 1
jz    short loc_41DF6F

```

- Exit – As the name implies, this is a self-destructive command that allows the malware to exit, and blind fold the target as if it never existed. That is the reason why the malware automatically deletes itself allowing no trace after it runs.

Understanding the Exports:

From figure 17 below, we see that the export has only one single file “**start**” with address **0041DE4E**. Double-clicking the address, we can see that the malware have memory operands like **eax**, **ebx**, **ecx**, **esi**, **ebp** and **esp** which refers to memory address that contains the value of interest as shown in figure 18 below. These are typically registers and can be either 32 or 16bits in assembly code.



The screenshot shows a debugger interface with two main windows. The top window is titled "Exports" and lists a single entry: "start" at address 0041DE4E. The bottom window displays assembly code:

```

StartupInfo=_STARTUPINFOA ptr -5Ch
var_18=dword ptr -18h
var_4=dword ptr -4

push    ebp
mov     ebp, esp
push    0FFFFFFFh
push    offset unk_466098
push    offset loc_401096
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
sub    esp, 68h
push    ebx
push    esi
push    edi
mov     [ebp+var_18], esp
xor    ebx, ebx
mov     [ebp+var_4], ebx
push    2
call   __set_app_type
pop    ecx
or     dword_466CAC, 0FFFFFFFh
or     dword_466CB0, 0FFFFFFFh
call   __p_Fmode
mov     ecx, dword_466C98
mov     [eax], ecx
call   __p_commode
mov     ecx, dword_466C94
mov     eax, ecx

```

From figure 18 above, we see **move eax** – this mov instruction copies a value into the eax register and stores it, therefore **move [eax], ecx** copies the memory location specified by the ECX register into the EAX register.

Another **mov** instruction we see is the **mov [ebp+var 18], esp** which copies the 18 bytes var characters located at the memory location specified by **ebp** into the **esp** register and then **mov [ebp+var 4], ebx** copies the 4 bytes characters located at the memory location specified by **ebp** into the **ebx** register.

Basically what these command and instructions tells us is that the malware copies content located at on the targeted computer registers and then stores in it other registers it has access to. Pretty much the malware manipulates data by copying it to other locations.

Understanding the strings:

The strings figure shown in 19 below has the same content as the imports which we already explained above. However we see a special string **MSVCRT.dll**. This is module library containing standard C library functions. This tells us that the malware uses this string for manipulation, memory allocation, C-style input/output calls etc.

Address	Length	Type	String
.data:0046504E	00000005	C	<^B.1
.data:00465284	00000006	C	Mn@`v\$.
.data:00465319	00000006	C	U'j] F
.data:0046533D	00000005	C	b-&J\$
.data:004653DD	00000005	C	A\`v?.T
.data:0046551C	00000006	C	\5<9.G
.data:00465598	00000005	C	4#N\`
.data:00465712	00000005	C	:?g#\r
.data:004670BE	0000000D	C	EnableWindow
.data:004670CE	0000000D	C	SendMessageA
.data:004670DE	0000000D	C	UpdateWindow
.data:004670E	0000000A	C	LoadIconA
.data:004670F8	00000008	C	USER32.dll
.data:00467108	00000010	C	GetStartupInfoA
.data:00467116	0000000D	C	KERNEL32.dll
.data:00467126	0000000D	C	?2@YAPAXI@Z
.data:00467136	00000012	C	_CxxFrameHandler
.data:0046714A	0000000D	C	?3@YAPAXK@Z
.data:00467158	0000000B	C	MSVCRT.dll
.data:00467166	0000000C	C	_dllonexit
.data:00467174	00000008	C	_oneexit
.data:0046717E	00000006	C	_exit
.data:00467186	0000000C	C	_XcptFilter
.data:00467194	00000005	C	exit

Understanding the functions:

Functions are portions of code within a program that perform a specific task and that are relatively independent of the remaining code. The main code calls and temporarily

transfers execution to functions before returning to the main code. As seen in figure 20 below, the function just shows the use, function and what the malware does. Functions like getmessage, translate message etc.

Function name	Segment	Start	Length	R	F	L	S	B	T
CView::OnDrop(COleDataObject *,ulong,CPoint)	.text	00401000	00000006	R
CCmdTarget::GetExtraConnectionPoints(CPtr)	.text	00401006	00000006	R
CWnd::CheckAutoCenter(void)	.text	0040100C	00000006	R
sub_401020	.text	00401020	0000002E	R	.	.	.	B	.
sub_401050	.text	00401050	00000025	R
CListView::DrawItem(tagDRAWITEMSTRUCT*)	.text	00401076	00000006	R
CListCtrl::GetMessageMap(void)	.text	0040107C	00000006	R
CWinApp::WinHelpA(ulong,uint)	.text	00401082	00000006	R
CWinThread::PreTranslateMessage(tagMSG *)	.text	00401088	00000006	R
CFrameWnd::EndModalState(void)	.text	0040109C	00000006	R
CWinApp::InitApplication(void)	.text	004010A2	00000006	R
CFrameWnd::BeginModalState(void)	.text	004010A8	00000006	R
sub_4010B0	.text	004010B0	0000009C	R	.	.	.	B	.
CFrameWnd::GetMenuBar(void)	.text	0040114C	00000006	R
operator new(uint)	.text	00401152	00000006	R
std::locale::facet::~facet(void)	.text	00404290	00000014	R	.	L	.	B	.
CComboBox::DeleteItem(tagDELETEITEMSTRUCT*)	.text	00404244	00000006	R
CWinThread::ProcessMessageFilter(int,tagMSG*)	.text	00404280	00000006	R
sub_4042C0	.text	004042C0	00000010	R	.	.	.	B	.
CButton::OnChildNotify(uint,uint,long,long*)	.text	004042D0	00000006	R
CWnd::OnCommand(uint,long)	.text	004042D6	00000006	R
CFrameWnd::SetActiveDocument(void)	.text	004042D8	00000006	R

We now have an idea of what the malware is designed to do, and we are going to analyze the malware again to examine its internal state. In order to do this we have to run the malware again to complete the analysis while it is running.

Advance Dynamic Analysis

Advanced dynamic analysis uses a debugger to examine the internal state of a running malicious executable. Advanced dynamic analysis techniques provide another way to extract detailed information from an executable.

In order to start the advance dynamic analysis, we first run the program again using the command prompt like we did in the basic dynamic analysis. This time around, the

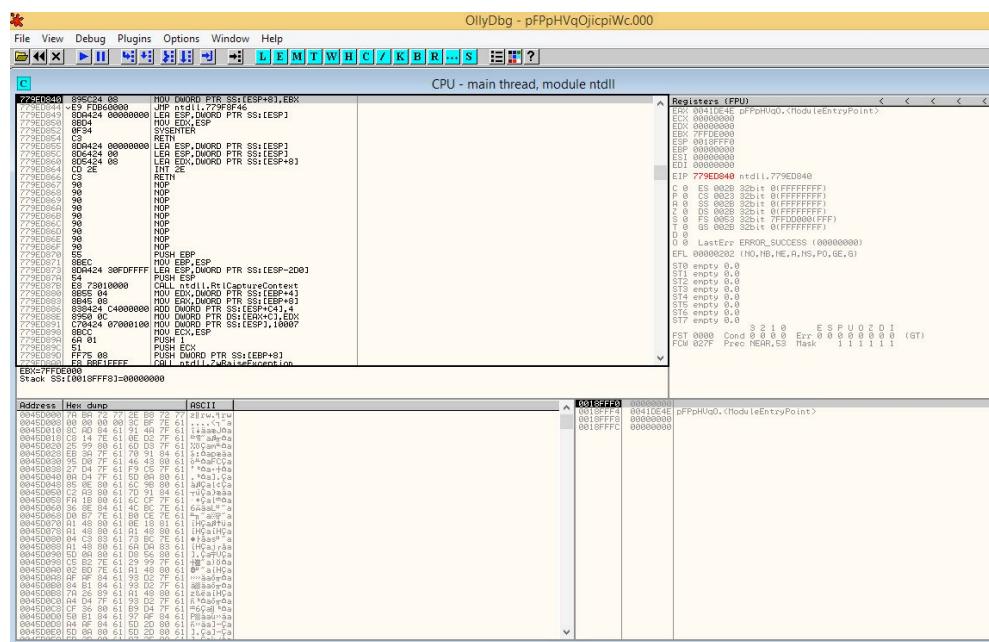
programmed runs successfully, however it replicates itself into other files: as shown in Figure 21 below:

Name	Date modified	Type	Size
pFPpHVqOjicpiWc	12/5/2014 1:20 PM	File folder	
malware.textClipping	11/19/2014 9:46 AM	TEXTCLIPPING File	0 KB
pFPpHVqOjicpiWc.000	10/8/2014 1:21 PM	000 File	420 KB
pFPpHVqOjicpiWc.id0	12/5/2014 5:58 PM	ID0 File	272 KB
pFPpHVqOjicpiWc.id1	12/5/2014 5:58 PM	ID1 File	1,656 KB
pFPpHVqOjicpiWc.nam	12/5/2014 5:58 PM	NAM File	16 KB
pFPpHVqOjicpiWc.til	12/5/2014 5:58 PM	TIL File	1 KB
pFPpHVqOjicpiWc	11/19/2014 9:46 AM	Compressed (zipp...)	273 KB
sample	11/19/2014 9:45 AM	Compressed (zipp...)	258 KB

Now we use a special tool designed to analyze malware while it is running. An x86 debugger known as OllyDbg.

OllyDbg

After running the malware, we load the file into OllyDbg as shown in Fig 22 below:



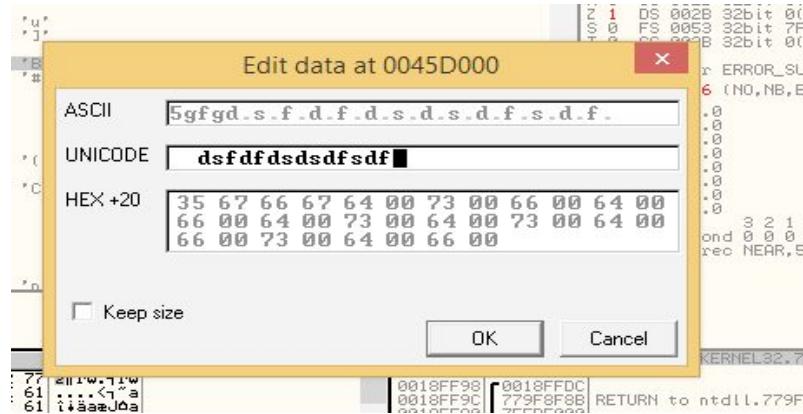
From the OllyDbg Disassembler's window above, we see the debugged malwares code, the registers window, the stack window, and the dump window. For this analysis, we shall go over each window and see what the content of the window means. From figure 15 above during advance basic analysis, we can see that the main call was a call made to GetModuleHandleA which has address 0045D000. Therefore we use the F8 key in OllyDbg to step-over until we arrive at the main address 0045D000. As shown in figure 23 below: The only thing we could find on the 00450000 address is the 'CHAR "B"' which doesn't make sense.

Address	OpCode	Mnemonic	Operands	Comments
0044FFEE	39	MOV	ECX, 442C6081	
0044FFEF	7E	MOV	ECX, 442C6081	
0044FFF0	21	MOV	ECX, 442C6081	
0044FFF1	64	MOV	ECX, 442C6081	
0044FFF2	60	MOV	ECX, 442C6081	
0044FFF3	C7	MOV	ECX, 442C6081	
0044FFF4	71	MOV	ECX, 442C6081	
0044FFF5	6A	MOV	ECX, 442C6081	
0044FFF6	AD	MOV	ECX, 442C6081	
0044FFF7	25	MOV	ECX, 442C6081	
0044FFF8	9E	MOV	ECX, 442C6081	
0044FFF9	57	MOV	ECX, 442C6081	
0044FFFA	14	MOV	ECX, 442C6081	
0044FFFB	A6	MOV	ECX, 442C6081	
0044FFFC	85	MOV	ECX, 442C6081	
0044FFFD	75	MOV	ECX, 442C6081	
0044FFFE	5D	MOV	ECX, 442C6081	
0044FFFF	C1	MOV	ECX, 442C6081	
00450000	42	MOV	ECX, 442C6081	CHAR 'B'
00450001	23	MOV	ECX, 442C6081	CHAR '#'
00450002	95	MOV	ECX, 442C6081	
00450003	A8	MOV	ECX, 442C6081	
00450004	8D	MOV	ECX, 442C6081	
00450005	CB	MOV	ECX, 442C6081	
00450006	FF	MOV	ECX, 442C6081	
00450007	28	MOV	ECX, 442C6081	
00450008	8E	MOV	ECX, 442C6081	CHAR '('
00450009	43	MOV	ECX, 442C6081	CHAR 'C'
0045000A	F6	MOV	ECX, 442C6081	
0045000B	E7	MOV	ECX, 442C6081	
0045000C	89	MOV	ECX, 442C6081	
0045000D	B9 31602C44	MOV ECX, 442C6081		
00450012	. 003F	ADD BYTE PTR DS:[EDI], BH		
00450014	. C3	RETN		
00450015	6F	DB FF		CHAR '\n'

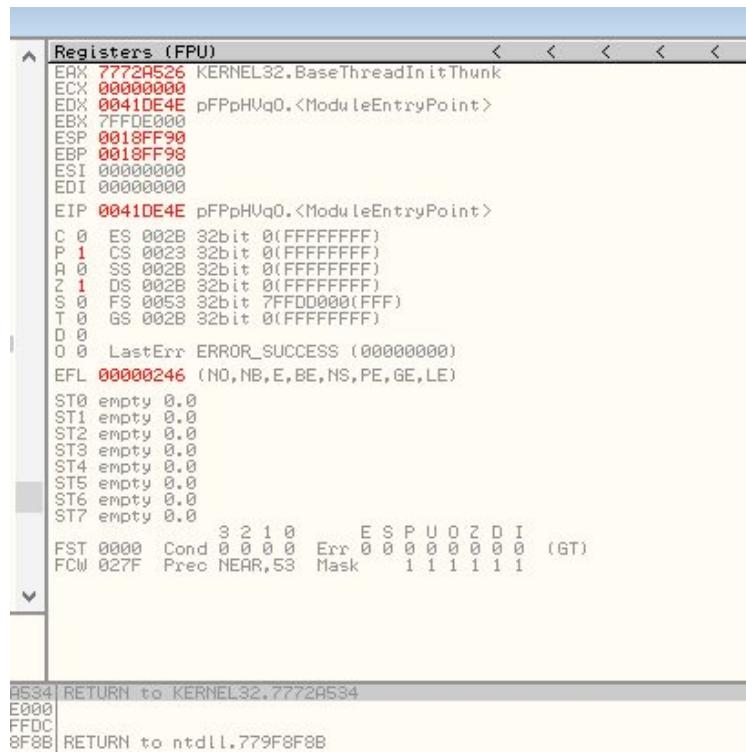
Now the register window shown below shows us the call to the main function.

Address	Hex dump	ASCII
\$ ==> 0045D000 <&KERNEL32.GetModuleHandleA>	7A BA 72 77 2E B8 72 77	z rw.7rw
\$+8 0045D008	00 00 00 00 3C BF 7E 61<"a
\$+10 0045D010 <&MFC42.#560>	8C AD 84 61 91 4A 7F 61	iääaaJoä
\$+18 0045D018 <&MFC42.#1945>	C8 14 7E 61 0E D2 7F 61	¶¶¶¶¶¶¶¶
\$+20 0045D020 <&MFC42.#652>	25 99 80 61 6D D3 7F 61	¶¶¶¶¶¶¶¶
\$+28 0045D028 <&MFC42.#4426>	EB 3A 7F 61 70 91 84 61	§:ðapzää

We then right-click the hex dump codes to see the binary, and it shows 8 continues strings, which contains some gibberish values in the ASCII, this could be another way to make the malware untraceable.



Also in OllyDbg we see the stacks of registers we saw in IDAPro as shown below:



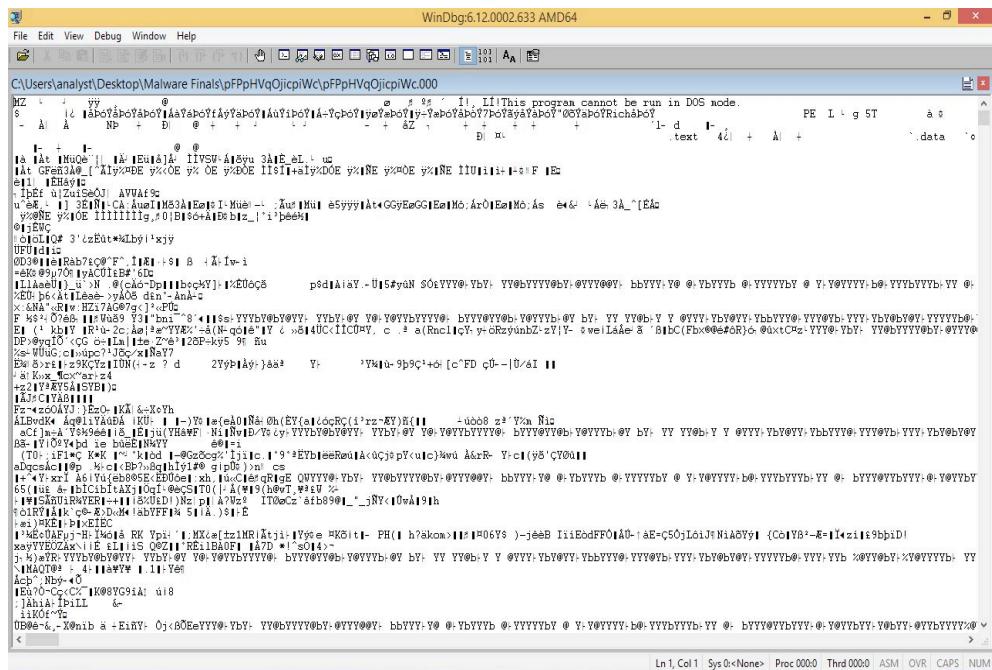
These are the same registers we analyzed using IDAPro. At the beginning of the advance dynamic analysis we see how the malware replicates itself into different files after we run it. Figure 27 below shows us the address of those replicate files.

0018F3E0	0033006D	
0018F3E4	005C0032	pFPpHVq0.0046004D
0018F3E8	0046004D	
0018F3EC	00340043	
0018F3F0	004C0032	
0018F3F4	0043004F	pFPpHVq0.0043004F
0018F3F8	0044002E	pFPpHVq0.0044002E
0018F3FC	004C004C	
0018F400	0044002E	pFPpHVq0.0044002E
0018F404	004C004C	
0018F408	00000000	
0018F40C	00000000	
0018F410	0018F4B8	
0018F414	77A0AC19	ntdll.77A0AC19
0018F418	00000000	
0018F41C	0018F438	
0018F420	00420040	pFPpHVq0.00420040
0018F424	006446F0	
0018F428	0018F640	

Next we are going to use another debugger known as Windbg to analyze the malware and see if we can reach the same result and conclusion like we did with OllyDbg.

Windbg:

We try to upload the malware into Windbg for further analysis, however we keeps receiving errors, and after further research an analysis, it's because this malware it is because this malware is not an executable. The file format for this malware is in **.000** whereas Windbg is designed for executables (.exe) and source files such as C, C++, C# and Assembly source files. Figure 28 and 29 below shows the error pages we encountered.



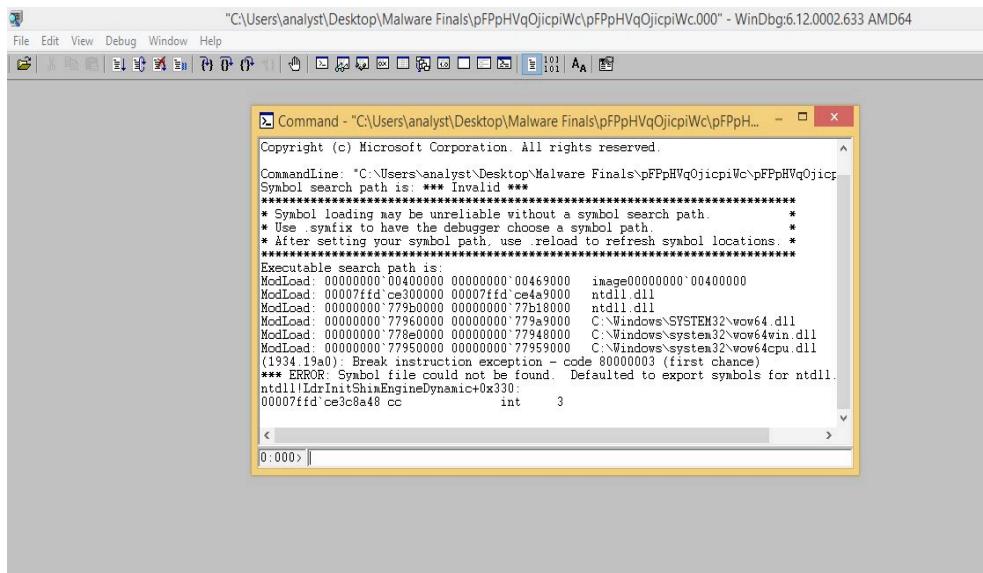
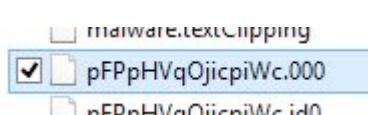
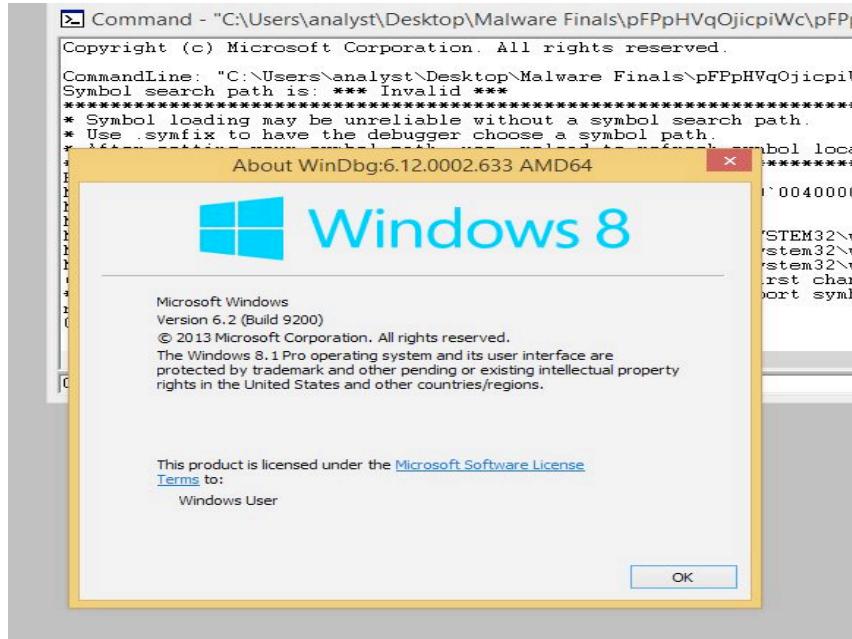


Figure 30 below shows the Windbg software license and figure 31 shows the malware file format (.000) and .000 is not a recognized windows format, which makes Windbg unsupportive in terms of this malware analysis.



Malware Behavior (Understanding the malware)

So far we have been analyzing the malware to understand its characteristics and categorize it as a malware and what type of malware it is. We have 2 types of malwares. The downloaders, which downloads other malwares and the launchers, which are executables that installs itself for covert executions.

Throughout our analysis, we haven't encountered any download characteristics or capability associated with this malware, however we see the way it can manipulates the system processes and copy it between registers. Therefore we categorize this malware as a **launcher**.

This malware monitors the processes running on the target computer and then copies it and store in the register it can access for retrieval. The processes running could be victim's credentials or other resourceful information depending on what is running.

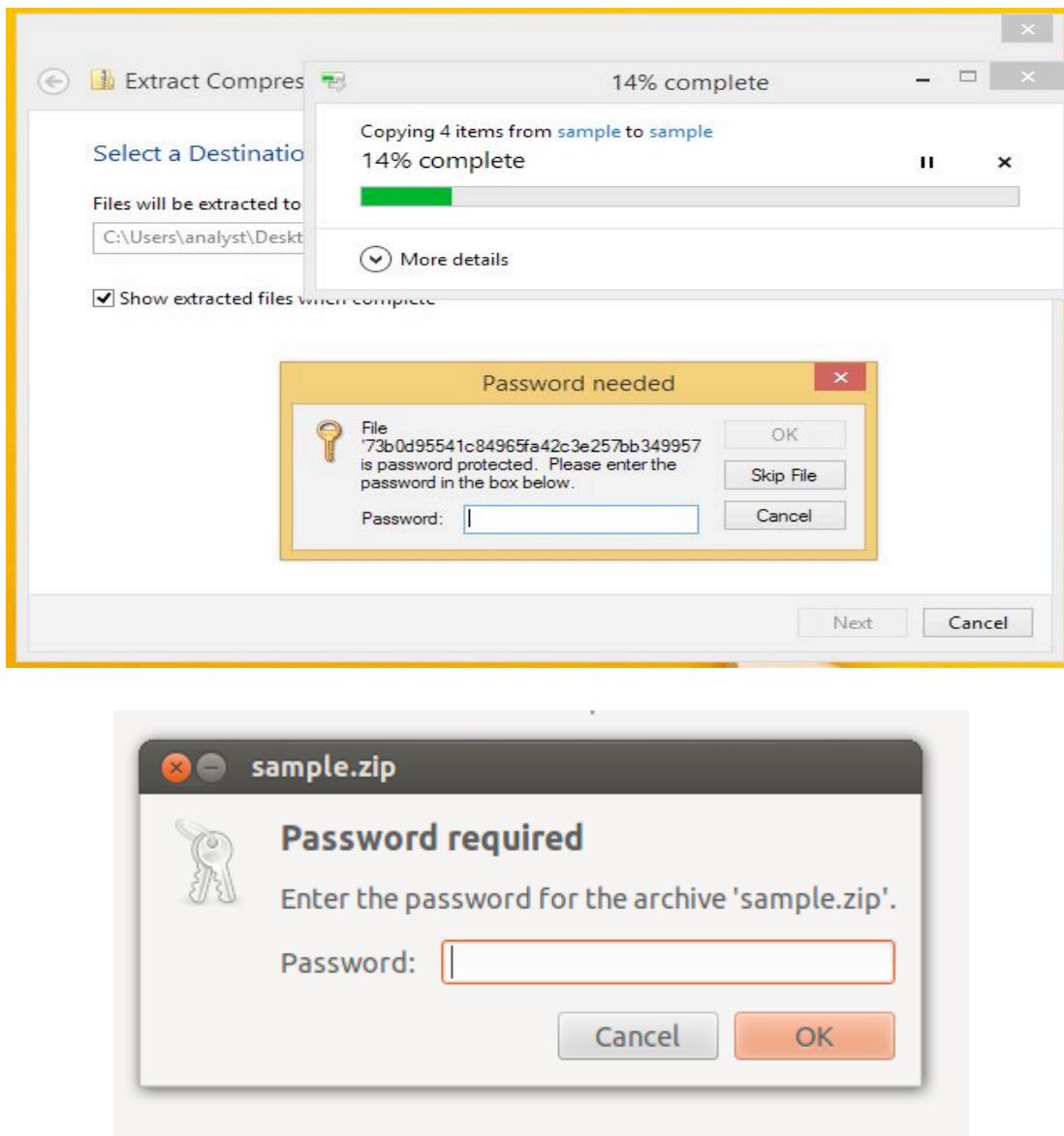
Part II

Malware 2 Analysis

File: **sample.zip**

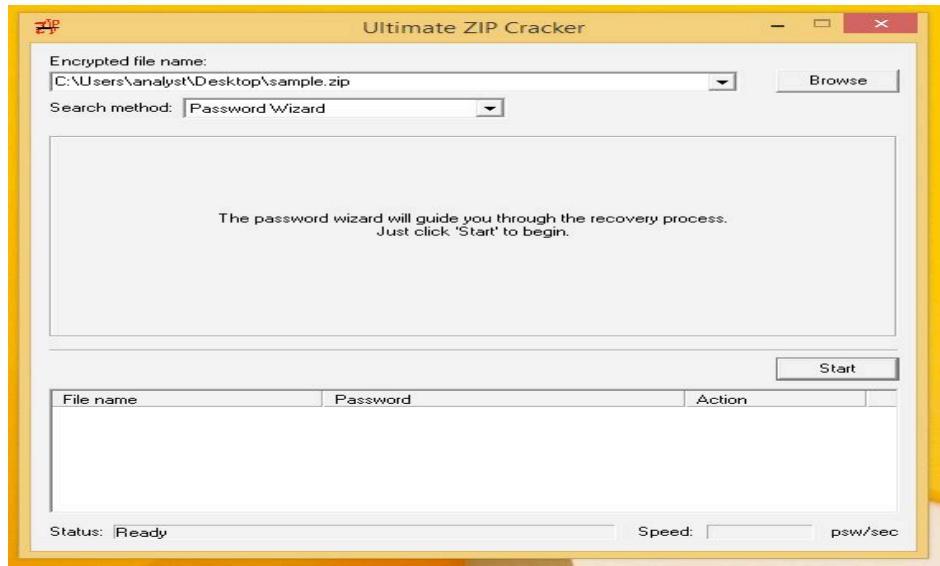
Extracting the file

The file is located in a zip folder, and has is supposedly not pass worded (as per the exams instructions). However it keeps asking for a password before it could be extracted. Trying it both in Windows & Linux proves that the file requires a password before extraction as shown in Figure 32 and 33 respectively below:



Now since we were not given any password for the zip file, we have to brute force it to see if we could crack the password. For easier use we decide to use a special program in

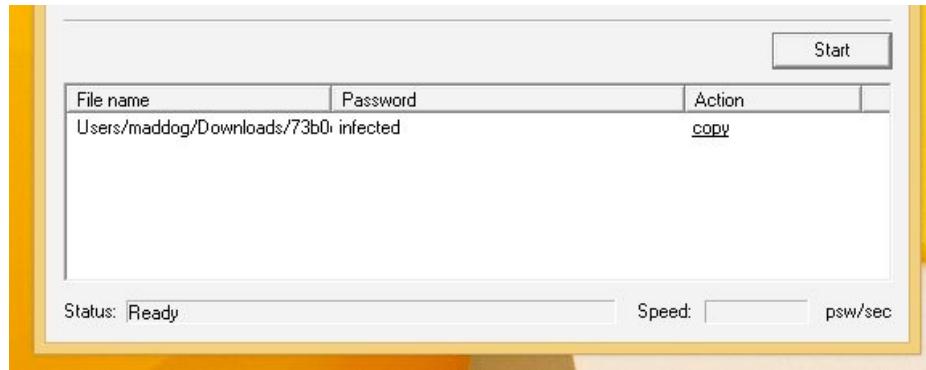
windows known as **UZC**. **UZC** also known as the Ultimate Zip Cracker is a software designed to crack password zip, word and excel files. The interface of the Zip Cracker is shown in figure 34 below.



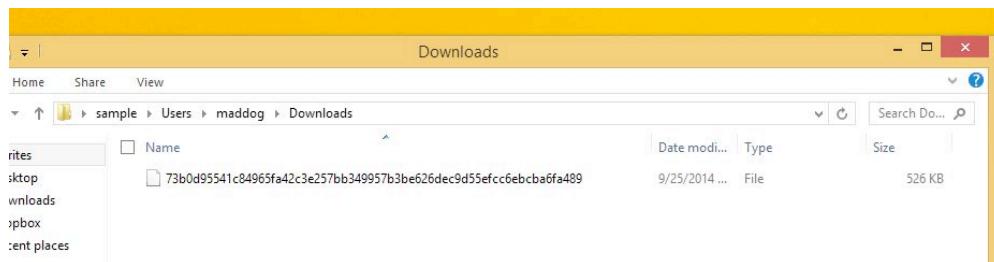
The way we used it is we located the zip files directory as shown above, then select **Password Wizard** as the search method and then click **Start**. We then follow the **Password Recovery Wizard** shown in figure 35 below:



As shown as the wizard is done, it shows you the password of the zip file, and below in figure 36, we can see that the password for this zip file has been identified as “infected”.



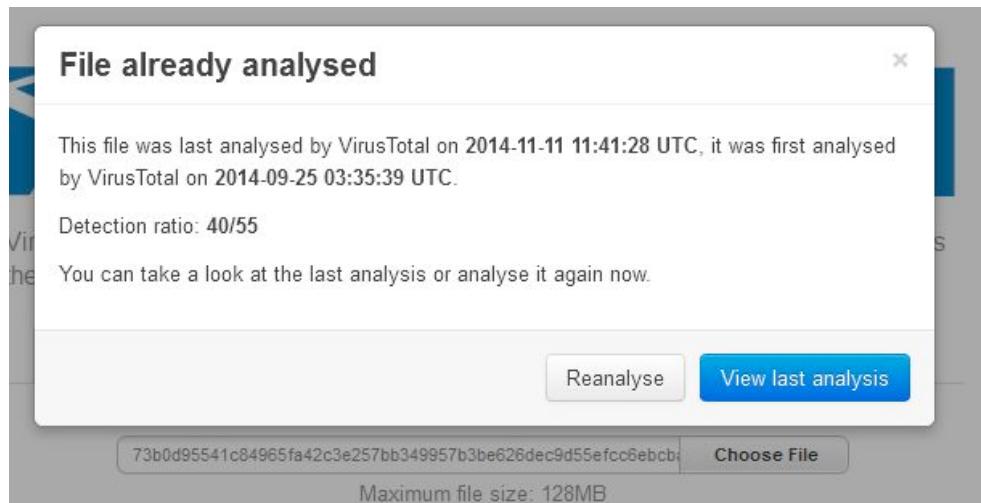
Now that we have the password we can use it to extract the zip file. Extracting the zip file, we see different directories before locating malware. These directory are **sample\users\maddog\downloads**. And the malware file name is a long alphanumeric characters that will be too long to write here. The malware file type is identified as “**File**” and the size is **526KB**. Hopefully after the analysis we will be able to know and understand more about the malware. Figure 37 below shows the extracted file.



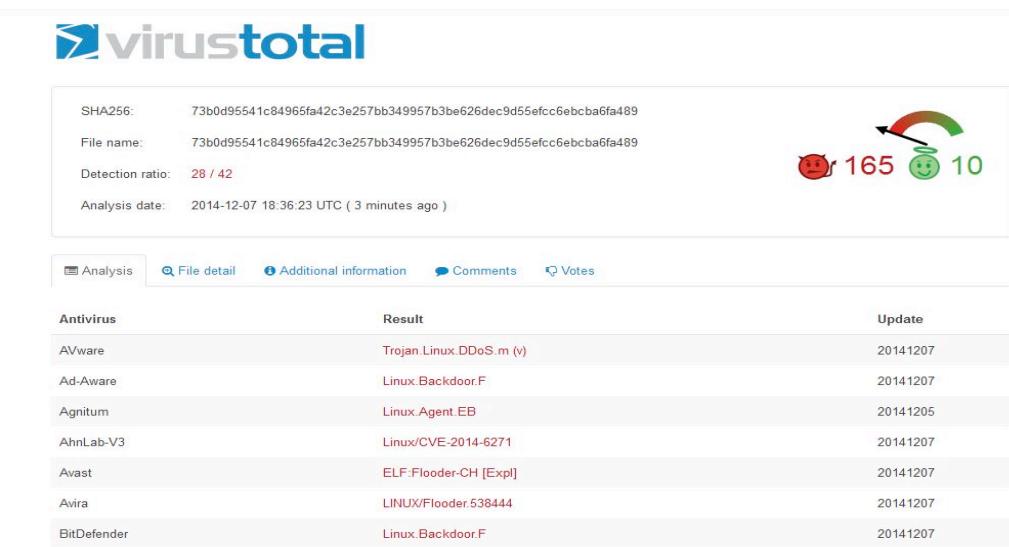
Basic Static Analysis

Analysis:

As usual we start the basic analysis with uploading the malware to see if it matches any existing virus signatures. As shown below, the file has been previously analyzed.

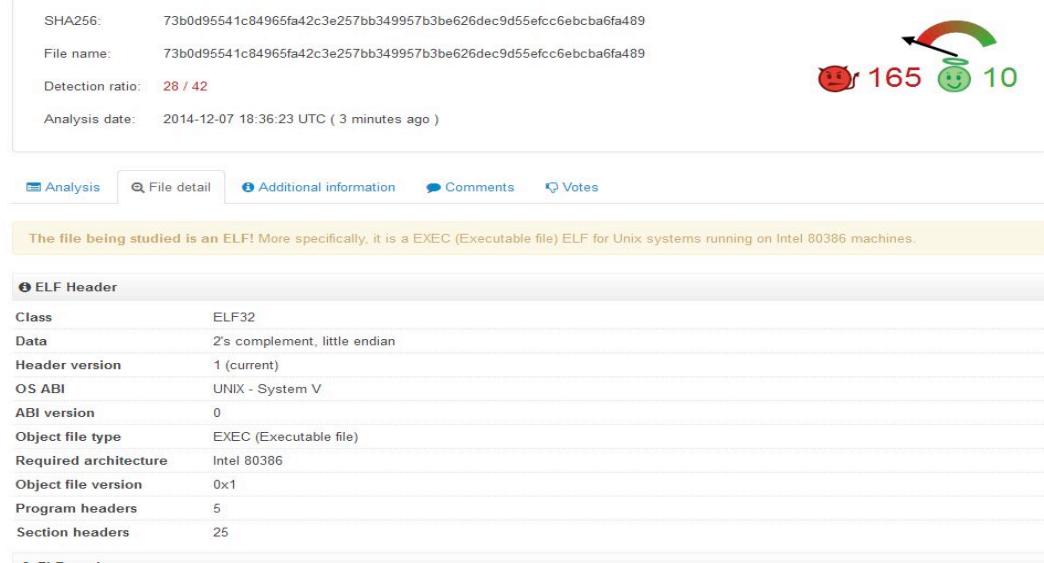


We then reanalyze the file again. We can see in figure 39 below that the malware matches 28 of 42 virus signatures.



From the **File Details** information provided in figure 40 and 41, we can see object file type is EXEC (Executable file), the required architecture for the malware is **Intel 80386**, and the malware class is **ELF32**. From the malware class we can understand that the malware is an **Executable and Linkable Format (ELF)** designed for **32bit** operating systems. However ELF files are the standard binary file formats for Unix and Unix-Like systems. From the architecture **Intel 80386** we see that the malware is only meant for systems with Intel Processors. Therefore before further analysis we can tell that the

malware is designed to target Unix systems (could be Apple OS X or Linux OS) with 32bit operating systems running with Intel Processor.



The screenshot shows a VirusShare analysis page for a file named 73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489. The analysis date is 2014-12-07 18:36:23 UTC (3 minutes ago). The detection ratio is 28 / 42, with 165 viruses flagged and 10 marked as safe. The file is identified as an ELF executable for Unix on Intel 80386 architecture.

ELF Header

Class	ELF32
Data	2's complement, little endian
Header version	1 (current)
OS ABI	UNIX - System V
ABI version	0
Object file type	EXEC (Executable file)
Required architecture	Intel 80386
Object file version	0x1
Program headers	5
Section headers	25

Sections

.text	PROGBITS	0x08048160	0x00000160	410028	A, X
__libc_freeers_fn	PROGBITS	0x080ac310	0x00064310	2807	A, X
.fini	PROGBITS	0x080ace08	0x00064e08	28	A, X
.rodata	PROGBITS	0x080ace40	0x00064e40	100444	A
__libc_atexit	PROGBITS	0x080c569c	0x0007d69c	4	A

Segments

- [+] LOAD
- [+] LOAD
- [+] NOTE
- [+] <unknown>
- [+] GNU_STACK

ExifTool file metadata

MIMEType	application/octet-stream
CPUByteOrder	Little endian
CPUArchitecture	32 bit
FileType	ELF executable
FileAccessDate	2014:12:07 19:36:39+01:00
ObjectFileType	Executable file
CPUType	i386
FileCreateDate	2014:12:07 19:36:39+01:00

We also see other information like the **FileCreateDate**, which is the day the file was created. It shows **2014:12:07 19:36:39** which really doesn't make any sense because it is the same with the **FileAccessDate** and that is the same day we are making the analysis.

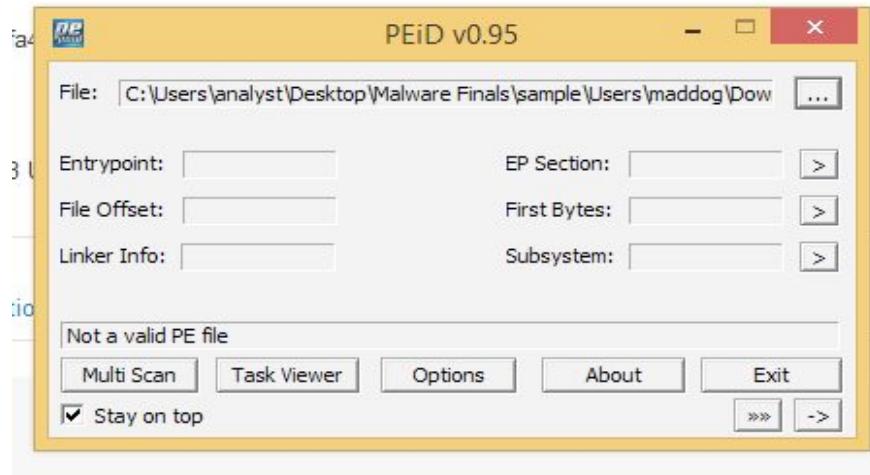
However we believe that further analysis will reveal more about the malware and why the **FileCreateDate** is being faked.

The **Additional Information** tab shown in figure 42 below didn't really tell us much besides the thing we already knew like file type and size etc. The new details we can see are the cryptographic hash functions **MD5, SHA1, and SHA256**. The knowledge of cryptographic hash functions are outside the scope of this class or the malware analysis and will therefore not be discussed. The comments section is the opinion of other analysts, however both of the analysts identified the file as **ShellShock** exploit.

File identification	
MD5	5924bcc045bb7039f55c6ce29234e29a
SHA1	0229e6fa359bce01954651df2cdbddcdf3e24776
SHA256	73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489
ssdeep	12288.79+DzT70zz3j6MI2xDsKO9qjCm7pQ2DP89HScvhQe/Q0FNc.7KXT70zztc15Dem7pQ2cSMhP
File size	525.8 KB (538444 bytes)
File type	ELF
Magic literal	ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, for GNU/Linux 2.6.18, stripped
TrID	ELF Executable and Linkable format (Linux) (50.1%) ELF Executable and Linkable format (generic) (49.8%)
Tags	cve-2014-6271 exploit elf
VirusTotal metadata	
First submission	2014-09-25 03:35:39 UTC (2 months, 1 week ago)
Last submission	2014-12-07 18:36:23 UTC (25 minutes ago)
File names	http_162.253.66.76_nginx 73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489 ss (2) nginx.1 5924bcc045bb7039f55c6ce29234e29a.exe 00000000 MD5 5924bcc045bb7039f55c6ce29234e29a.lst 5924bcc045bb7039f55c6ce29234e29a danger nginx

PEiD

To further the basic analysis we upload the file to **PEiD**. As shown in figure 43, we see that the malware is not a valid PE file. This will make our analysis a bit difficult considering we need to know if the file is packed or not, and also how the file is packed.



PEview

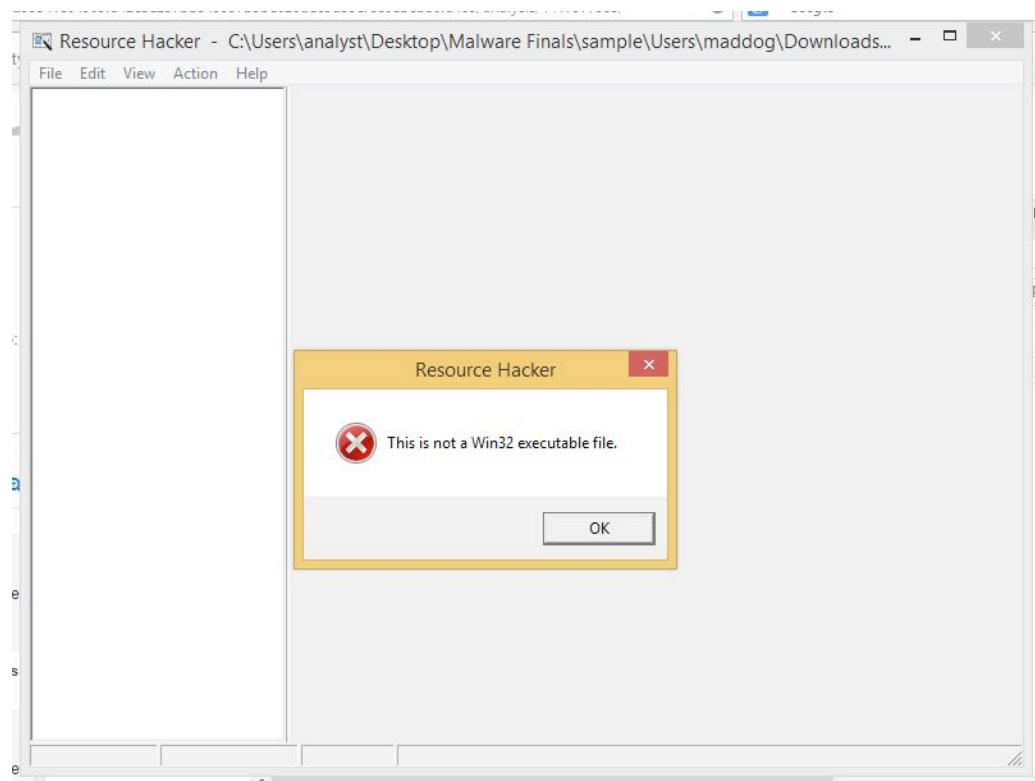
We use **PEview** with the hope of extracting more information about the malware, information such as the **Header and Imports**. However just like **PEiD**, the analysis using **PEview** doesn't show any results beside the file type **ELF** which we already know. Result of the **PEview** analysis is shown in figure 44 below:

pFile	Raw Data	Value
00000000	7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00	ELF.....
00000010	02 00 03 00 01 00 00 00 60 81 04 08 34 00 00 004..
00000020	64 33 08 00 00 00 00 00 34 00 20 00 05 00 28 00	d3.....4..(.
00000030	19 00 18 00 01 00 00 00 00 00 00 00 80 04 08)
00000040	00 80 04 08 CC 29 08 00 CC 29 08 00 05 00 00 00)
00000050	00 10 00 00 01 00 00 00 CC 29 08 00 CC B9 0C 08)
00000060	CC B9 0C 08 B4 08 00 00 08 65 00 00 06 00 00 00e..
00000070	00 10 00 00 04 00 00 00 D4 00 00 00 D4 80 04 08
00000080	D4 80 04 08 20 00 00 00 20 00 00 00 04 00 00 00
00000090	04 00 00 00 07 00 00 00 CC 29 08 00 CC B9 0C 08)
000000A0	CC B9 0C 08 10 00 00 00 3C 00 00 00 04 00 00 00<..
000000B0	04 00 00 00 51 E5 74 64 00 00 00 00 00 00 00 00	...Q.td..
000000C0	00 00 00 00 00 00 00 00 00 00 00 06 00 00 00 00
000000D0	04 00 00 00 04 00 00 00 10 00 00 00 01 00 00 00
000000E0	47 4E 55 00 00 00 00 00 02 00 00 00 06 00 00 00	GNU.....
000000F0	12 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000110	00 00 00 00 00 00 00 00 38 BA 0C 08 2A 00 00 008...*..
00000120	55 89 E5 53 83 EC 04 E8 00 00 00 00 5B 81 C3 00	U..S.....[..
00000130	39 08 00 8B 93 FC FF FF FF 85 D2 74 05 E8 BE 7E	9.....t...~
00000140	FB F7 E8 B9 00 00 00 E8 94 41 06 00 58 5B C9 C3A..X[..
00000150	FF 25 38 BA 0C 08 68 00 00 00 E9 00 00 00 00 %8 ..h..
00000160	31 ED 5E 89 E1 B3 E4 F0 50 54 52 68 00 BE 04 08 1PTRh....
00000170	EF 40 DF 01 00 E4 FF 00 CC B9 04 00 E9 2F 26 00

Viewing 73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcbaf489

Resource Hacker:

We conclude that extracting any basic static analysis information from this malware will be a hassle on windows considering this malware is designed for Unix OS. We use the tool Resource Hacker to extract some basic information about the malware. However it says that the file is not a Win32 executable file as shown in figure 45 below:



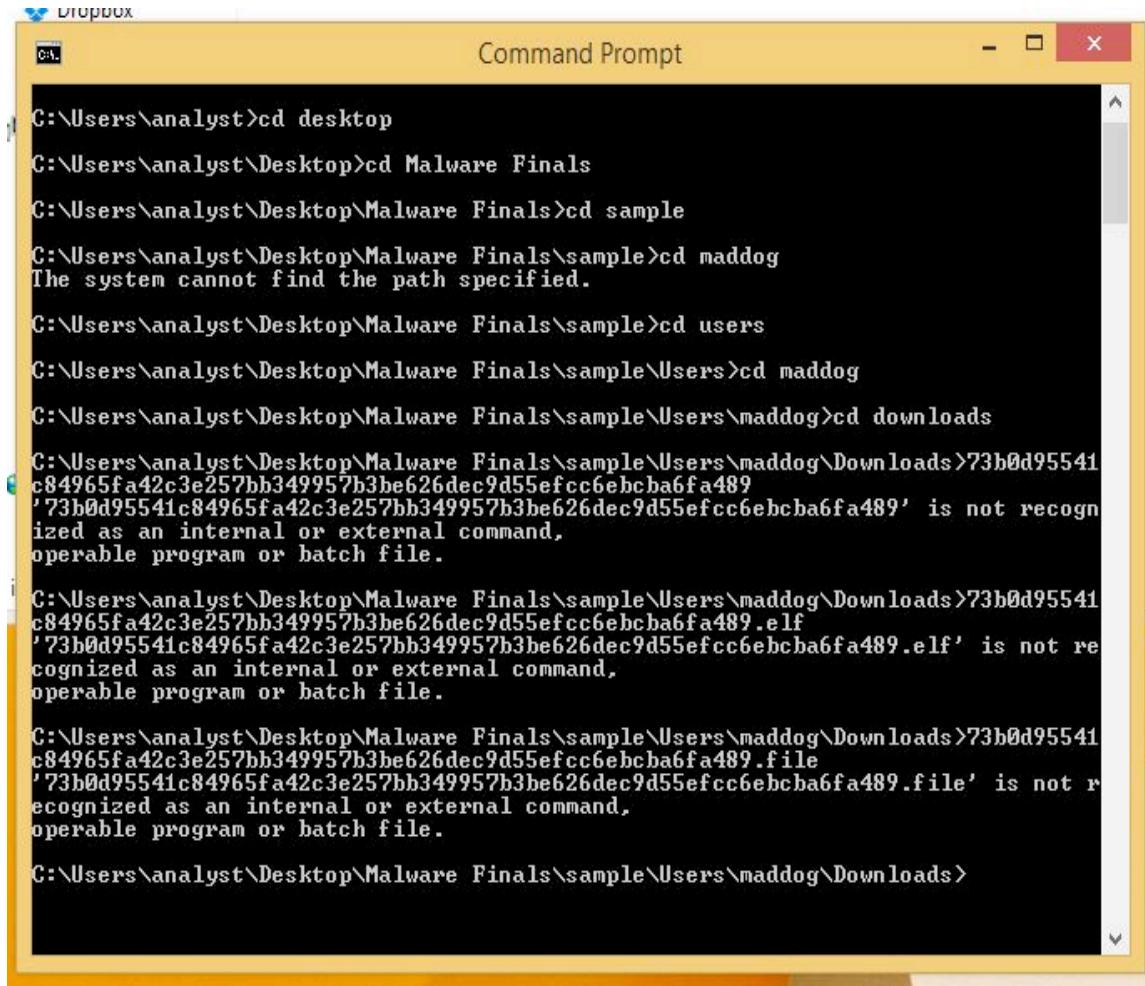
Basic Dynamic Analysis

It is time to run the malware and carryout a basic dynamic analysis in order to monitor its behavior.

Analysis:

We first run the malware (even though at this stage we know that the malware is designed for Unix systems. We will attempt to run it on Windows OS to see what happens!).

Considering the file is not designed for Windows systems and it does not have a file format. An attempt to run it using the command prompt failed as shown in figure 46 below:



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The user is navigating through a directory structure on the C:\Users\analyst\Desktop\Malware Finals\sample\Users\maddog\Downloads path. They are attempting to run several files, but each attempt results in an error message stating that the command or file is not recognized as an internal or external command, operable program or batch file. The files listed are 73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.elf and 73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.file.

```
C:\Users\analyst>cd desktop
C:\Users\analyst\Desktop>cd Malware Finals
C:\Users\analyst\Desktop\Malware Finals>cd sample
C:\Users\analyst\Desktop\Malware Finals\sample>cd maddog
The system cannot find the path specified.

C:\Users\analyst\Desktop\Malware Finals\sample>cd users
C:\Users\analyst\Desktop\Malware Finals\sample\Users>cd maddog
C:\Users\analyst\Desktop\Malware Finals\sample\Users\maddog>cd downloads
C:\Users\analyst\Desktop\Malware Finals\sample\Users\maddog\Downloads>73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489
'73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489' is not recognized as an internal or external command,
operable program or batch file.

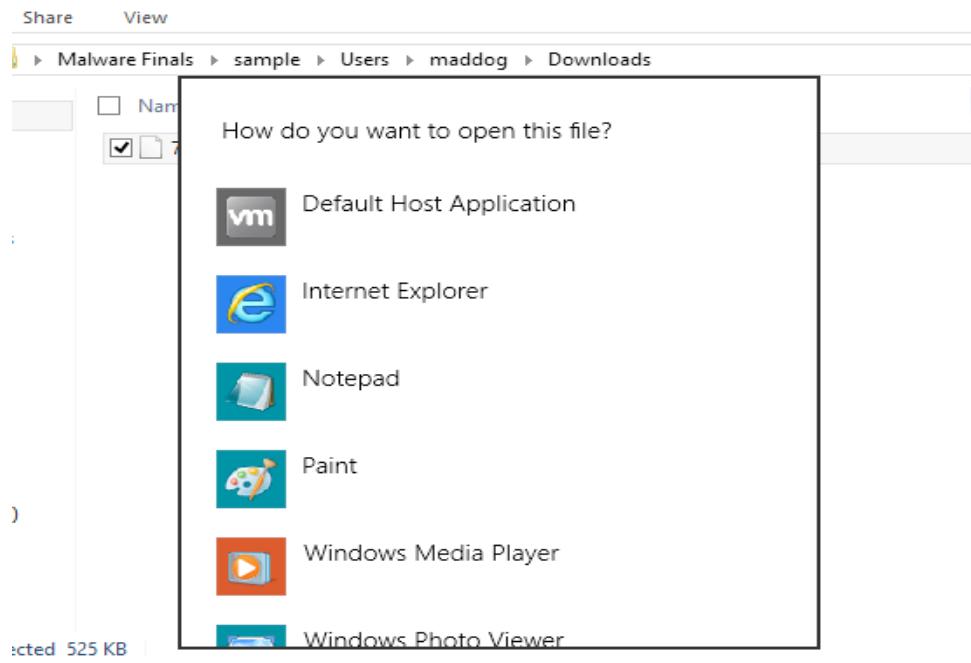
C:\Users\analyst\Desktop\Malware Finals\sample\Users\maddog\Downloads>73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.elf
'73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.elf' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\analyst\Desktop\Malware Finals\sample\Users\maddog\Downloads>73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.file
'73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.file' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\analyst\Desktop\Malware Finals\sample\Users\maddog\Downloads>
```

Therefore we are going to attempt running the malware by just double-clicking it to see what happens.

Doing so did not run the malware too; instead it asks how we want to open the file as shown in figure 47 below.



Without running the malware, there is no way we can carry out the basic dynamic analysis considering some of the tools used in basic dynamic analysis such as Process Monitor (Procmon) and Process Explorer will only yield results if the malware is running.

We will try and carry out an advance static analysis with the hope of understanding this malware designed for Unix.

Advanced Static Analysis

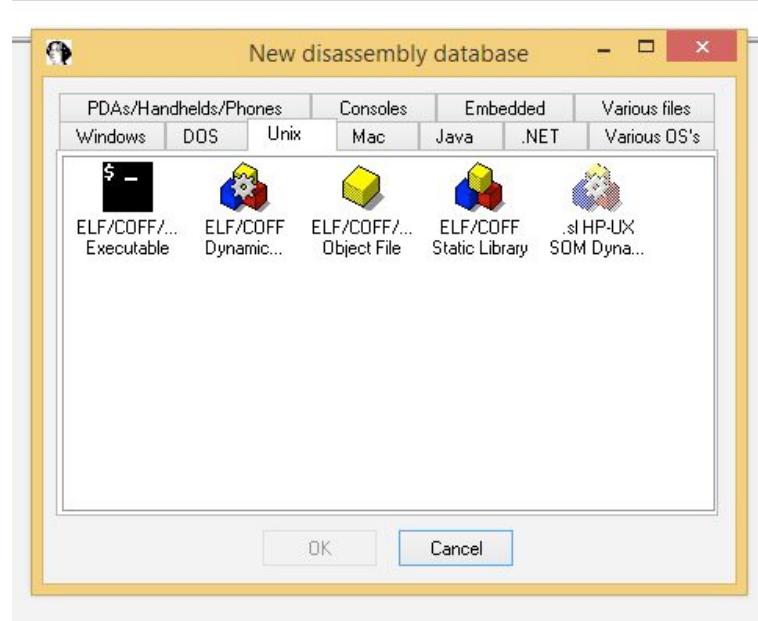
Here we will use an advanced tool IDAPro which is a disassembler designed to look at the programs instruction in order to discover what the program does.

Analysis:

IDAPro

The first step is to load the file into IDAPro. Using IDAPro we can decide on the file type on the “New disassembly database” page and what we did is go to the UNIX tab, and select the ELF Executable file type (considering we were able to find out that the file is an

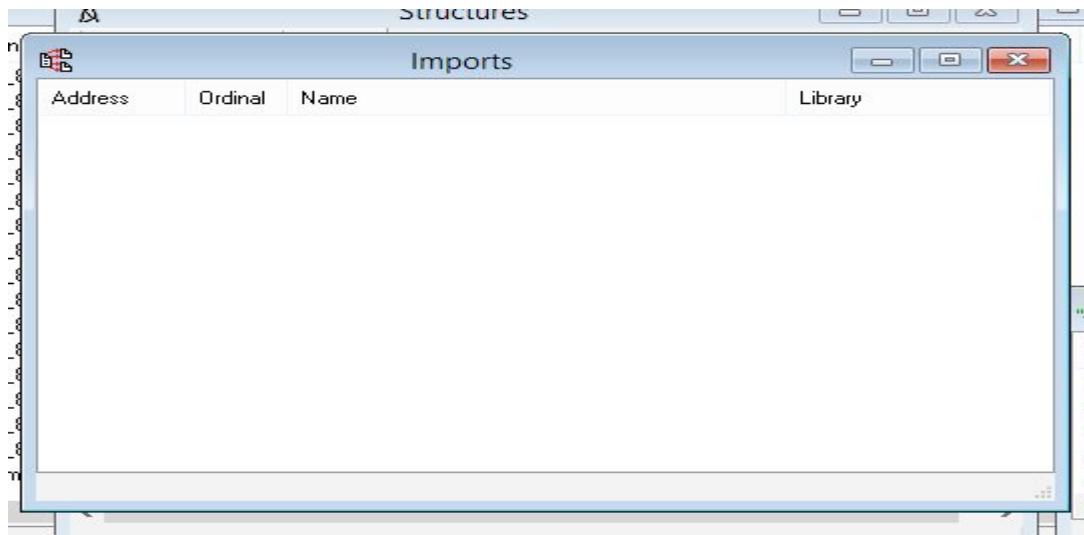
ELF file from basic static analysis). The new disassembly database page is as shown in figure 48 below.



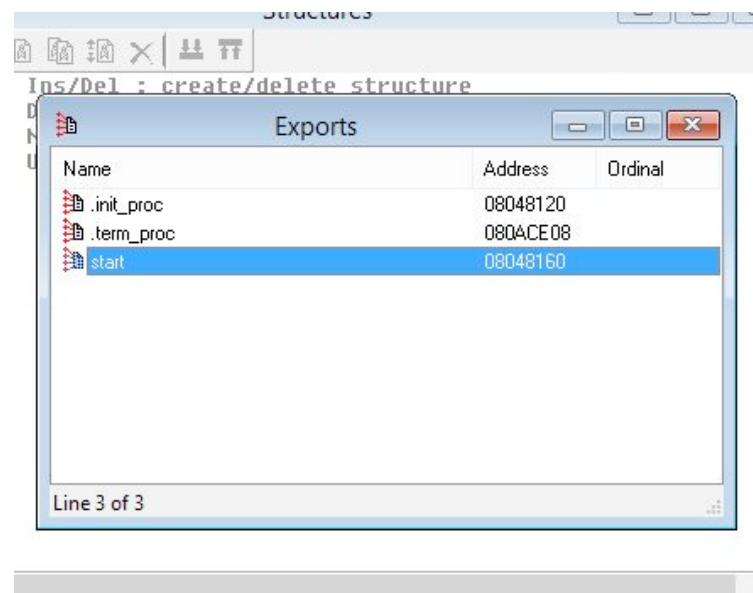
This time around the malware was successfully loaded into IDAPro considering IDAPro is a tool designed for both Windows & Unix systems. Loading the file into IDAPro results in the file being replicated as shown in figure 49 below. However we are going to analyze the file based on the information provided by IDAPro.

Name	Date modi...	Type	Size
73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489	9/25/2014 ...	File	526 KB
73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.id0	12/7/2014 ...	ID0 File	96 KB
73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.id1	12/7/2014 ...	ID1 File	2,240 KB
73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.nam	12/7/2014 ...	NAM File	16 KB
73b0d95541c84965fa42c3e257bb349957b3be626dec9d55efcc6ebcba6fa489.til	12/7/2014 ...	TIL File	1 KB

We see from IDAPro that the malware does not have any imports as shown in figure 50, this is interesting, it appears this malware will be tough to analyze without any imports.



We see only 3 exports associated with the file. **.init_proc**, **.term_proc** and **start**. However we can only make sense of the **start** export on address **08048160**. We believe it is the export responsible for starting the malware.



We then take a look at the strings. The strings showed in figure 52 below shows some interesting strings that we can put into consideration. We see strings like **/bin/sh**, **/dev/full**, **/etc/localtime** **/proc/cpuinfo** **/proc/meminfo** etc. All these are Unix paths that the malware uses.

Strings window			
Address	Length	Type	String
"..." .rodata:...	0000001E	C	./sysdeps/x86_64/cacheinfo.c
"..." .rodata:...	00000020	C	.lib section in a.out corrupted
"..." .rodata:...	00000032	C	/bin/busybox;echo -e "\\\147\\\141\\\171\\\146\\\147\\\164\\n\\n
"..." .rodata:...	00000008	C	/bin/sh
"..." .rodata:...	0000000D	C	/dev/console
"..." .rodata:...	0000000A	C	/dev/full
"..." .rodata:...	00000009	C	/dev/log
"..." .rodata:...	0000000A	C	/dev/null
"..." .rodata:...	00000009	C	/dev/tty
"..." .rodata:...	0000000D	C	/dev/urandom
"..." .rodata:...	00000011	C	/etc/ld.so.cache
"..." .rodata:...	00000013	C	/etc/ld.so.nohwcap
"..." .rodata:...	0000000F	C	/etc/localtime
"..." .rodata:...	00000010	C	/etc/suid-debug
"..." .rodata:...	00000015	C	/lib/i486-linux-gnu/
"..." .rodata:...	00000008	C	/lib32/
"..." .rodata:...	0000000E	C	/proc/cpuinfo
"..." .rodata:...	0000000E	C	/proc/meminfo
"..." .rodata:...	00000010	C	/proc/net/route
"..." .rodata:...	0000000F	C	/proc/self/exe
"..." .rodata:...	00000010	C	/proc/self/maps
"..." .rodata:...	0000000B	C	/proc/stat
"..." .rodata:...	0000001D	C	/proc/sys/kernel/ngroups_max
"..." .rodata:...	0000001B	C	/proc/sys/kernel/osrelease
"..." .rodata:...	0000001B	C	/proc/sys/kernel/rtsig-max
"..." .rodata:...	00000018	C	/sys/devices/system/cpu
"..." .rodata:...	00000011	C	/usr/lib/getconf
"..." .rodata:...	00000019	C	/usr/lib/i486-linux-gnu/
"..." .rodata:...	00000010	C	/usr/lib/locale
"..." .rodata:...	0000001F	C	/usr/lib/locale/locale-archive
"..." .rodata:...	0000000C	C	/usr/lib32/
"..." .rodata:...	00000025	C	/usr/lib32/gconv/gconv-modules.cache
"..." .rodata:...	00000012	C	/usr/share/locale
"..." .rodata:...	00000014	C	.../etc/.../etc/...

The path **/dev** is used to mount disk, **/bin** contains the variables used in single user mode, **/etc** contains configuration files, **/proc** contains access to process and kernel informations. From this string we can see that this malware pretty much hijacks the computer completely. We also see other strings like **MemFree** and **MemTotal**, and also **My IP.%s**. These are instructions used for memory count. We are talking about the physical memory allocation in the Unix kernel, not the hard disk storage. This tells us that the

malware is capable of controlling the physical memory of the hijacked computer. The **MyIP** string tells us that the malware have some networking functions attached to it. It could be masking the identity of the victim's computer so that the malware could not be detected. We see other string like **Networkisdown**, **Network is unreachable**. These are all part of the networking capabilities and functions associated with the malware.

The next interesting strings we see is **Print**, **processor** and **protected**. This are data processing instructions, it tells us that the malware stores some data or instructions using the command **print** and then controls the process while encrypting or protecting the data its prints.

Address	Length	Type	String
0000000000000024	00000004	C	MAC: %02X%02X%02X%02X%02X%02X\n
0000000000000007	00000007	C	MALLOC_
000000000000000D	0000000D	C	MALLOC_TRACE
000000000000000A	0000000A	C	MMAP_MAX_
0000000000000010	00000010	C	MMAP_THRESHOLD_
000000000000001E	0000001E	C	Machine is not on the network
0000000000000006	00000006	C	March
0000000000000010	00000010	C	MemFree: %ld kB
0000000000000011	00000011	C	MemTotal: %ld kB
0000000000000011	00000011	C	Message too long
0000000000000007	00000007	C	Monday
0000000000000013	00000013	C	Multihop attempted
000000000000000A	0000000A	C	My IP: %s
0000000000000009	00000009	C	NIS_PATH
0000000000000008	00000008	C	NLSPATH
000000000000001B	0000001B	C	Name not unique on network
0000000000000024	00000024	C	Network dropped connection on reset
0000000000000010	00000010	C	Network is down
0000000000000017	00000017	C	Network is unreachable
000000000000001B	0000001B	C	No CSI structure available
000000000000001E	0000001E	C	No XENIX semaphores available
0000000000000009	00000009	C	No anode
000000000000001A	0000001A	C	No buffer space available
0000000000000013	00000013	C	No child processes
0000000000000012	00000012	C	No data available
0000000000000013	00000013	C	No locks available
0000000000000010	00000010	C	No medium found
000000000000001B	0000001B	C	No message of desired type
0000000000000011	00000011	C	No route to host
0000000000000018	00000018	C	No space left on device
000000000000000F	0000000F	C	No such device
000000000000001A	0000001A	C	No such device or address
000000000000001A	0000001A	C	No such file or directory
0000000000000010	00000010	C	No such file or directory

Address	Length	Type	String
.... .rodata:...	00000006	C	print
.... .rodata:...	0000000A	C	processor
.... .rodata:...	0000000A	C	protected
.... .rodata:...	00000006	C	pse36
.... .rodata:...	00000006	C	punct
.... .rodata:...	00000015	C	r->r_state == RT_ADD
.... .rodata:...	00000019	C	ranges[cnt].from >= from
.... .rodata:...	0000001D	C	realloc(): invalid next size
.... .rodata:...	0000001C	C	realloc(): invalid old size
.... .rodata:...	0000001B	C	realloc(): invalid pointer
.... .rodata:...	0000000A	C	recv: %s\n
.... .rodata:...	00000011	C	relocation error
.... .rodata:...	00000010	C	remove_slotinfo
.... .rodata:...	00000022	C	result <= _dl_tls_max_dtv_idx + 1
.... .rodata:...	00000022	C	result == _dl_tls_max_dtv_idx + 1
.... .rodata:...	00000008	C	result > 0
.... .rodata:...	00000009	C	ret == 0
.... .rodata:...	00000005	C	root
.... .rodata:...	0000000F	C	s->_flags2 & 4
.... .rodata:...	00000004	C	s\SMALLC
.... .rodata:...	00000007	C	s\v\b@s\v\b
.... .rodata:...	0000003D	C	set_thread_area failed when setting up thread-local storage\n
.... .rodata:...	00000005	C	sh\v\n
.... .rodata:...	00000023	C	shared object cannot be dlopen()ed
.... .rodata:...	00000017	C	shared object not open
.... .rodata:...	00000006	C	space
.... .rodata:...	00000005	C	sse2
.... .rodata:...	000000A0	C	status == __GCONV_OK status == __GCONV_EMPTY_INPUT status ==...
.... .rodata:...	0000001C	C	status == __codecvt_partial
.... .rodata:...	0000001F	C	step->__end_fct == ((void *)0)
.... .rodata:...	00000024	C	step->__shlib_handle != ((void *)0)
.... .rodata:...	00000009	C	strops.c
.... .rodata:...	00000008	C	symbol
.... .data:...	00000014	C

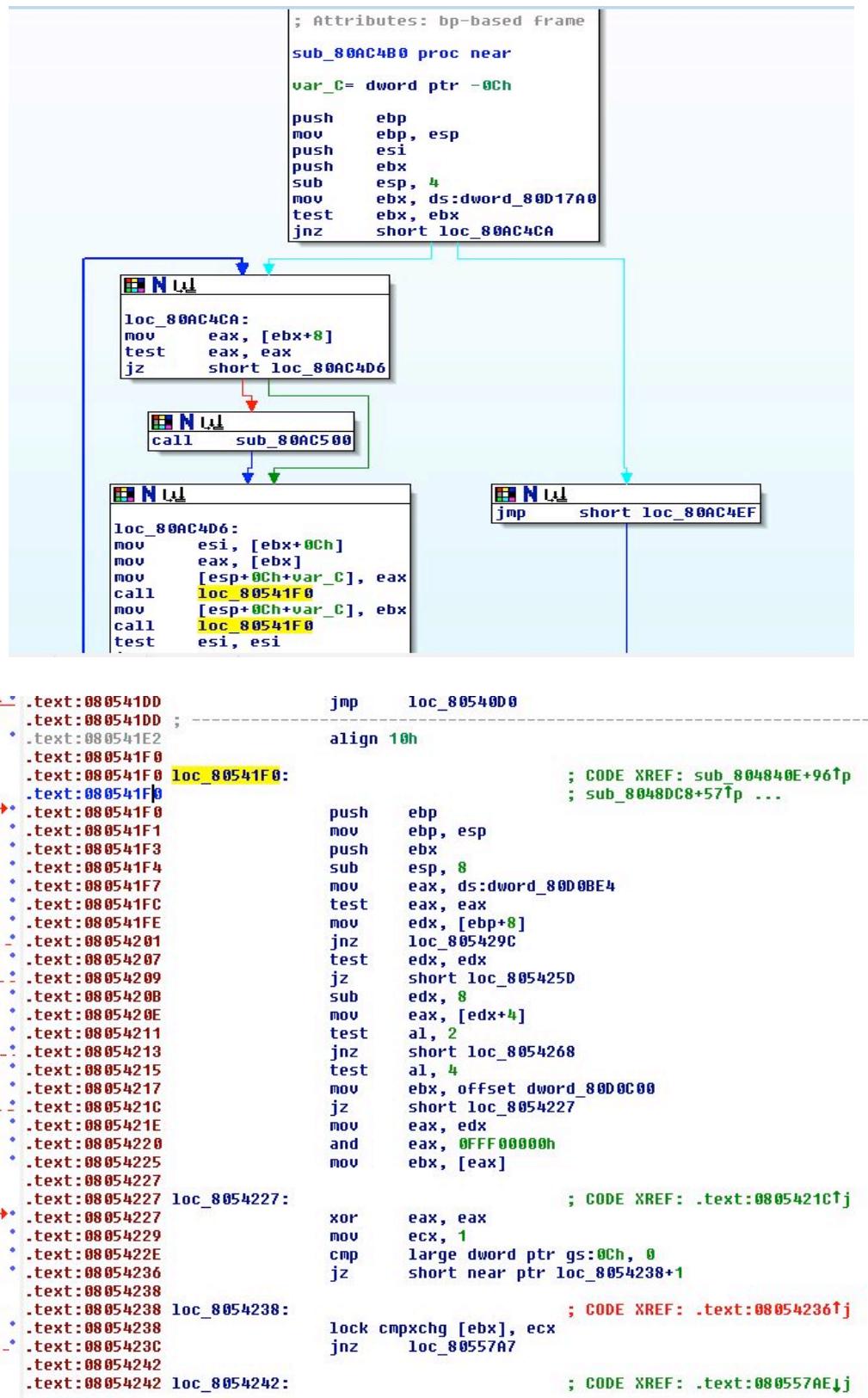
We check the functions shown in figure 55 below to see if we can find the call to the main function of the malware, however all we see is the functions name presented as address and then the length of the strings and where the functions starts.

Function name	Segment	Start	Length	R	F	L	S	B	T
sub_80A9F50	.text	080A9F50	00000070	R	.	.	.	B	.
sub_80A9FC0	.text	080A9FC0	00000080	R	.	.	.	B	.
sub_80AA040	.text	080AA040	00000190	R	.	.	.	B	.
sub_80AA1E0	.text	080AA1E0	000000D5	R	.	.	.	B	.
sub_80AA2C0	.text	080AA2C0	0000013E	R	.	.	.	B	.
sub_80AA400	.text	080AA400	00000117	R	.	.	.	B	.
sub_80AA630	.text	080AA630	00000143	R	.	.	.	B	.
sub_80AA780	.text	080AA780	00000563	R	.	.	.	B	.
sub_80AAD80	.text	080AAD80	00000640	R	.	.	.	B	.
sub_80AB3C0	.text	080AB3C0	000001DE	R	.	.	.	B	.
sub_80AB540	.text	080AB540	00000100	R	.	.	.	B	.
sub_80AB640	.text	080AB640	0000000C	R	.	.	.	B	.
sub_80AC0B0	.text	080AC0B0	0000017C	R	.	.	.	B	.
sub_80AC230	.text	080AC230	0000008E	R	.	.	.	B	.
sub_80AC2E0	.text	080AC2E0	0000002A	R	.	.	.	B	.
sub_80AC310	__libc_free...	080AC310	00000049	R	.	.	.	B	.
sub_80AC400	__libc_free...	080AC400	000000AF	R	.	.	.	B	.
sub_80AC4B0	__libc_free...	080AC4B0	00000046	R	.	.	.	B	.
sub_80AC500	__libc_free...	080AC500	000000E1	R	.	.	.	B	.
sub_80AC5F0	__libc_free...	080AC5F0	0000002C	R	.	.	.	B	.
sub_80AC620	__libc_free...	080AC620	00000042	R	.	.	.	B	.
sub_80AC670	__libc_free...	080AC670	0000004A	R	.	.	.	B	.
sub_80AC810	__libc_free...	080AC810	00000020	R	.	.	.	B	.
sub_80AC830	__libc_free...	080AC830	0000003F	R	.	.	.	B	.
sub_80AC870	__libc_free...	080AC870	00000027	R	.	.	.	B	.
sub_80AC8A0	__libc_free...	080AC8A0	00000026	R	.	.	.	B	.
sub_80AC8D0	__libc_free...	080AC8D0	00000091	R	.	.	.	B	.
sub_80ACC20	__libc_free...	080ACC20	00000050	R	.	.	.	B	.
sub_80ACC70	__libc_free...	080ACC70	0000009A	R	.	.	.	B	.
sub_80ACD10	__libc_free...	080ACD10	00000070	R	.	.	.	B	.
sub_80ACD80	__libc_free...	080ACD80	00000087	R	.	.	.	B	.
_term_proc	.fini	080ACE08	0000001C	R	.	.	.	B	.

The only information we could extract from the **names window** is the alphanumeric characters that appears like garbage, we couldn't really make sense of it, as it doesn't correlates to any information we have analyzed so far.

Name	Address	P.
aalso10646Utf8_3	080B4427	
aalso10646Utf8_4	080B4437	
aalso10646Utf8_5	080B4448	
aUcs2	080B4458	
aalso10646Ucs2_0	080B445F	
aUcs2_0	080B446F	
aalso10646Ucs2_1	080B4477	
aOsf00010100	080B4487	
aalso10646Ucs2_2	080B4495	
aOsf00010101	080B44A5	
aalso10646Ucs2_3	080B44B3	
aOsf00010102	080B44C3	
aalso10646Ucs2_4	080B44D1	
aAnsi_x3_4	080B44E1	
aAnsi_x3_41968	080B44ED	
aalso1r6	080B44FE	
aAnsi_x3_4196_0	080B4509	
aAnsi_x3_41986	080B451A	
aAnsi_x3_4196_1	080B452B	
aalso_646_irv199	080B453C	
aAnsi_x3_4196_2	080B454F	
aASCII	080B4560	
aAnsi_x3_4196_3	080B4568	
aalso646Us	080B4579	
aAnsi_x3_4196_4	080B4585	
aUsAscii	080B4596	
aAnsi_x3_4196_5	080B45A1	
aUs	080B45B2	
aAnsi_x3_4196_6	080B45B7	
albm367	080B45C8	
aAnsi_x3_4196_7	080B45D1	
aCp367	080B45E2	
... 4196_8 4196_9	080B45F4	

From the IDAview-A we see some **mov** functions between registers **eax**, **esp** and **esi**. We also call to address location **80541F0**. Following that address, we only see the move instructions between registers, and other few instructions, but nothing specific as to what and why these instructions are taking place.

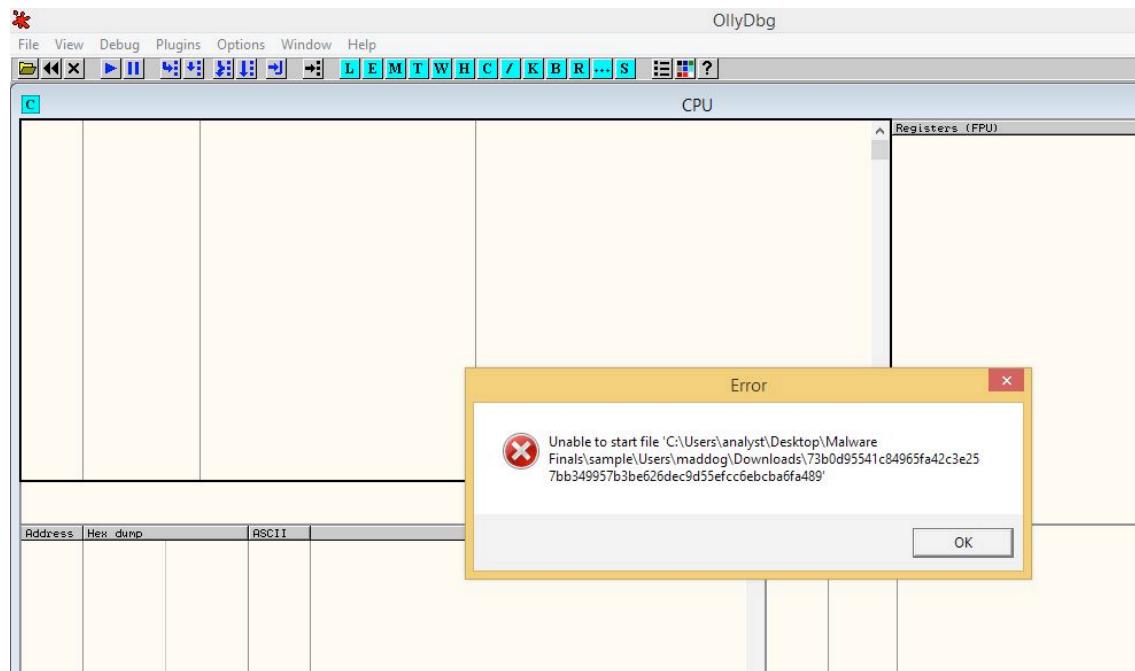


Even though we know we can't run the malware on a Windows system, we attempt to do an Advanced Dynamic Analysis using 2 special tools OllyDbg and Windbg.

Advanced Dynamic Analysis

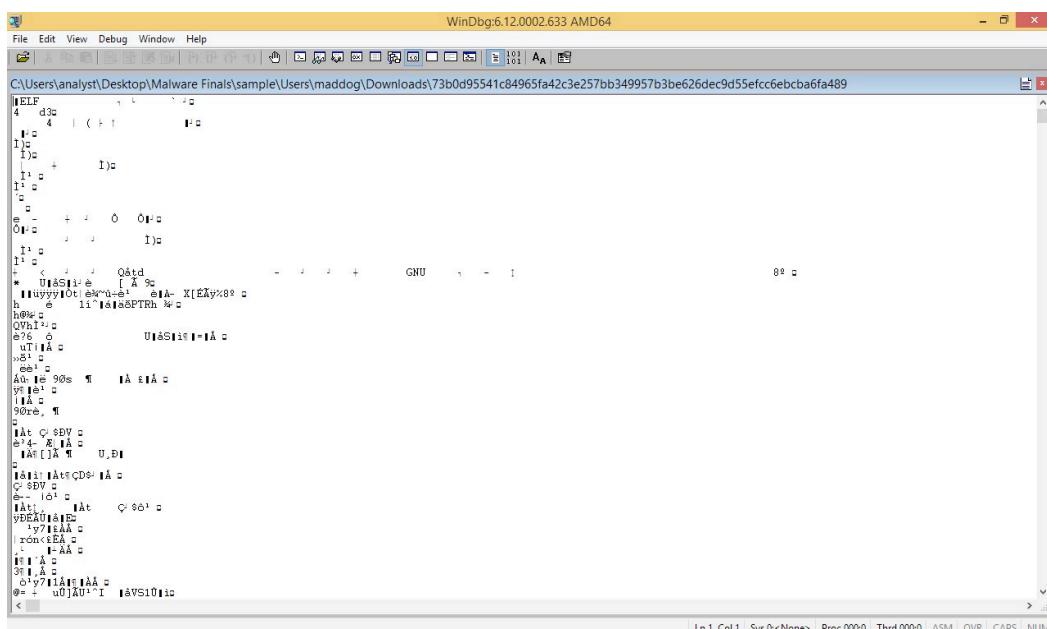
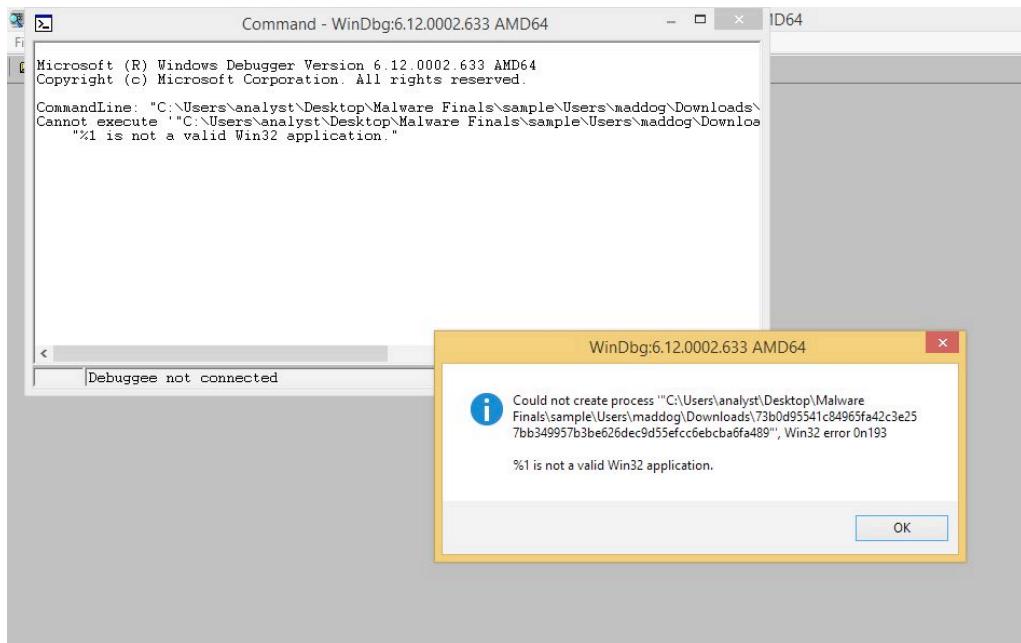
OllyDbg:

The malware was not successfully loaded to OllyDbg. It returns an error when an attempt was made to open the file.. Figure 59 below shows the error from OllyDbg.



WinDbg:

The file couldn't be loaded to Windbg, we tried opening it both as an executable file and also as an open source file. Figure 60 and 61 shows the result we get when we try to open it using both approaches.



Malware Behavior (Understanding the Malware).

Several approaches using different tools have been done to analyze this malware. However the malware is specifically designed for UNIX systems and we are attempting to analyze it on a Windows system. The reason of the major failures and drawbacks is that the malware needs to run in a UNIX environment and then it can only be analyzed using tools designed for UNIX malwares. This class and finals is aimed at analysis of malware designed for Windows system. As of this final we don't have the required knowledge to carryout analysis of UNIX based malwares.

From the attempts we made so far, we were able to program is a Unix Shellshock malware designed to be used for bash to process certain request using networking capabilities which allows an attacker to cause vulnerable bash commands so as to gain unauthorized administrative access to a computer system.

Conclusion

The first malware is identified as a Win32.exe file designed for Windows OS, while the second malware is an ELF file (Executable and Linkable Format), which is a standard binary format for UNIX systems. The first malware was successfully analyzed with no problems, but considering the last malware is designed for UNIX systems we ran into multiple problems.